

РАСПРЕДЕЛЕННЫЕ СИСТЕМЫ

ПРИНЦИПЫ И ПАРАДИГМЫ



Э. ТАНЕНБАУМ, М. ван СТЕЕН

С Е Р И Я

••• КЛАССИКА COMPUTER SCIENCE •••

DISTRIBUTED SYSTEMS

PRINCIPLES AND PARADIGMS

**Andrew S. Tanenbaum,
Maarten van Steen**



Prentice Hall PTR
Upper Saddle River, New Jersey 07458
www.phptr.com



Э. ТАНЕНБАУМ, М. ван СТЕЕН

РАСПРЕДЕЛЕННЫЕ СИСТЕМЫ

ПРИНЦИПЫ И ПАРАДИГМЫ



**Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара
Киев · Харьков · Минск**

2003

ББК 32.973.202

УДК 681.324

T18

T18 Распределенные системы. Принципы и парадигмы / Э. Таненбаум, М. ван Стеен. — СПб.: Питер, 2003. — 877 с.: ил. — (Серия «Классика computer science»).

ISBN 5-272-00053-6

Эта книга является фундаментальным курсом по распределенным системам. В ней подробно описаны принципы, концепции и технологии этих систем: связь, процессы, синхронизация, целостность и репликация, защита от сбоев и безопасность. Особое внимание в книге уделено World Wide Web, развитие которой и послужило толчком к резкому повышению интереса к распределенным системам. Как это характерно для всех книг Э.Таненбаума, последовательное и детальное изложение теории сопровождается примерами реально действующих систем.

Книга предназначена прежде всего студентам и преподавателям, но, безусловно, будет полезна и специалистам данной области.

ББК 32.973.202

УДК 681.324

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

© 2002 by Prentice-Hall, Inc.

ISBN 0-13-088893-1 (англ.)

© Перевод на русский язык, ЗАО Издательский дом «Питер», 2003

ISBN 5-272-00053-6

© Издание на русском языке, оформление, ЗАО Издательский дом «Питер», 2003

Краткое содержание

Об авторах.	14
Предисловие	16
Руководство по использованию книги	18
Глава 1. Введение	22
Глава 2. Связь.	81
Глава 3. Процессы	164
Глава 4. Именованние	214
Глава 5. Синхронизация	274
Глава 6. Непротиворечивость и репликация.	328
Глава 7. Отказоустойчивость	403
Глава 8. Защита	458
Глава 9. Распределенные системы объектов	539
Глава 10. Распределенные файловые системы.	623
Глава 11. Распределенные системы документов.	699
Глава 12. Распределенные системы согласования	752
Глава 13. Библиография	790
Список терминов	833
Алфавитный указатель.	855

Содержание

Об авторах.	14
Предисловие	16
От издательства.	17
Руководство по использованию книги	18
Курсы для студентов старших курсов и дипломников.	18
Семинары для профессионалов	19
Первый день.	20
Второй день	20
Самостоятельное изучение	20
Глава 1. Введение	22
1.1. Определение распределенной системы.	23
1.2. Задачи	25
1.2.1. Соединение пользователей с ресурсами	25
1.2.2. Прозрачность	26
1.2.3. Открытость	30
1.2.4. Масштабируемость	31
1.3. Концепции аппаратных решений	38
1.3.1. Мультипроцессоры	40
1.3.2. Гомогенные мультимикомпьютерные системы	42
1.3.3. Гетерогенные мультимикомпьютерные системы	44
1.4. Концепции программных решений	45
1.4.1. Распределенные операционные системы	46
1.4.2. Сетевые операционные системы.	57
1.4.3. Программное обеспечение промежуточного уровня	60
1.5. Модель клиент-сервер	67
1.5.1. Клиенты и серверы	67
1.5.2. Разделение приложений по уровням	72
1.5.3. Варианты архитектуры клиент-сервер	75
1.6. Итоги.	78
Вопросы и задания	79
Глава 2. Связь.	81
2.1. Уровни протоколов	82
2.1.1. Низкоуровневые протоколы	85
2.1.2. Транспортные протоколы	87
2.1.3. Протоколы верхнего уровня.	90
2.2. Удаленный вызов процедур	93
2.2.1. Базовые операции RPC	94
2.2.2. Передача параметров	98
2.2.3. Расширенные модели RPC.	103
2.2.4. Пример — DCE RPC	106
2.3. Обращение к удаленным объектам	111
2.3.1. Распределенные объекты	112
2.3.2. Привязка клиента к объекту	114
2.3.3. Статическое и динамическое удаленное обращение к методам	117
2.3.4. Передача параметров	118

2.3.5. Пример 1 — удаленные объекты DCE	120
2.3.6. Пример 2 — Java RMI	122
2.4. Связь посредством сообщений	126
2.4.1. Сохранность и синхронность во взаимодействиях	126
2.4.2. Нерезидентная связь на основе сообщений	131
2.4.3. Сохранная связь на основе сообщений	136
2.4.4. Пример — IBM MQSeries	143
2.5. Связь на основе потоков данных	148
2.5.1. Поддержка непрерывных сред	148
2.5.2. Потоки данных и качество обслуживания	151
2.5.3. Синхронизация потоков данных	156
2.6. Итоги	159
Вопросы и задания	160

Глава 3. Процессы 164

3.1. Потоки выполнения	165
3.1.1. Знакомство с потоками выполнения	165
3.1.2. Потоки выполнения в распределенных системах	171
3.2. Клиенты	175
3.2.1. Пользовательские интерфейсы	175
3.2.2. Клиентское программное обеспечение, обеспечивающее прозрачность распределения.	178
3.3. Серверы	179
3.3.1. Общие вопросы разработки	179
3.3.2. Серверы объектов	183
3.4. Перенос кода	189
3.4.1. Подходы к переносу кода	189
3.4.2. Перенос и локальные ресурсы	194
3.4.3. Перенос кода в гетерогенных системах	197
3.4.4. Пример — D'Agent	199
3.5. Программные агенты	204
3.5.1. Программные агенты в распределенных системах	205
3.5.2. Технология агентов	207
3.6. Итоги	210
Вопросы и задания	212

Глава 4. Именованное 214

4.1. Именованные сущности	215
4.1.1. Имена, идентификаторы и адреса	215
4.1.2. Разрешение имен	220
4.1.3. Реализация пространств имен	225
4.1.4. Пример — система доменных имен	232
4.1.5. Пример — X.500	237
4.2. Размещение мобильных сущностей	241
4.2.1. Именованное и локализация сущностей	241
4.2.2. Простые решения	244
4.2.3. Подходы на основе базовой точки	247
4.2.4. Иерархические подходы	249
4.3. Удаление сущностей, на которые нет ссылок	256
4.3.1. Проблема объектов, на которые нет ссылок	257
4.3.2. Подсчет ссылок	258
4.3.3. Организация списка ссылок	263
4.3.4. Идентификация сущностей, на которые нет ссылок	264

4.4. Итоги	270
Вопросы и задания	271
Глава 5. Синхронизация	274
5.1. Синхронизация часов	275
5.1.1. Физические часы	276
5.1.2. Алгоритмы синхронизации часов	280
5.1.3. Использование синхронизированных часов	284
5.2. Логические часы	285
5.2.1. Отметки времени Лампорта	286
5.2.2. Векторные отметки времени	290
5.3. Глобальное состояние.	292
5.4. Алгоритмы голосования.	296
5.4.1. Алгоритм записки	296
5.4.2. Кольцевой алгоритм	298
5.5. Взаимное исключение.	299
5.5.1. Централизованный алгоритм	299
5.5.2. Распределенный алгоритм	300
5.5.3. Алгоритм маркерного кольца	303
5.5.4. Сравнение трех алгоритмов	304
5.6. Распределенные транзакции.	305
5.6.1. Модель транзакций	306
5.6.2. Классификация транзакций	309
5.6.3. Реализация	312
5.6.4. Управление параллельным выполнением транзакций	315
5.7. Итоги	324
Вопросы и задания	325
Глава 6. Непротиворечивость и репликация.	328
6.1. Обзор.	329
6.1.1. Доводы в пользу репликации	329
6.1.2. Репликация объектов	331
6.1.3. Репликация как метод масштабирования	333
6.2. Модели непротворечивости, ориентированные на данные	335
6.2.1. Строгая непротворечивость	336
6.2.2. Линеаризуемость и последовательная непротворечивость	338
6.2.3. Причинная непротворечивость	343
6.2.4. Непротиворечивость FIFO	344
6.2.5. Слабая непротворечивость	346
6.2.6. Свободная непротворечивость	348
6.2.7. Поэлементная непротворечивость	351
6.2.8. Сравнение моделей непротворечивости	353
6.3. Модели непротворечивости, ориентированные на клиента	355
6.3.1. Потенциальная непротворечивость	356
6.3.2. Монотонное чтение	358
6.3.3. Монотонная запись	359
6.3.4. Чтение собственных записей	361
6.3.5. Запись за чтением.	362
6.3.6. Реализация	363
6.4. Протоколы распределения	365
6.4.1. Размещение реплик.	366
6.4.2. Распространение обновлений	370
6.4.3. Эпидемические протоколы	374

6.5. Протоколы непротиворечивости	377
6.5.1. Протоколы на базе первичной копии.	378
6.5.2. Протоколы реплицируемой записи.	382
6.5.3. Протоколы согласования кэшей	386
6.6. Примеры.	388
6.6.1. Orca	388
6.6.2. Слабая причинно непротиворечивая репликация	394
6.7. Итоги	399
Вопросы и задания	400
Глава 7. Отказоустойчивость	403
7.1. Понятие отказоустойчивости.	404
7.1.1. Основные концепции	404
7.1.2. Модели отказов	406
7.1.3. Маскирование ошибок при помощи избыточности	408
7.2. Отказоустойчивость процессов	410
7.2.1. Вопросы разработки	410
7.2.2. Маскировка ошибок и репликация	413
7.2.3. Соглашения в системах с ошибками	414
7.3. Надежная связь клиент-сервер	417
7.3.1. Сквозная передача	418
7.3.2. Семантика RPC при наличии ошибок	418
7.4. Надежная групповая рассылка.	424
7.4.1. Базовые схемы надежной групповой рассылки	424
7.4.2. Масштабируемость надежной групповой рассылки.	426
7.4.3. Атомарная групповая рассылка.	430
7.5. Распределенное подтверждение	437
7.5.1. Двухфазное подтверждение.	437
7.5.2. Трехфазное подтверждение.	442
7.6. Восстановление	445
7.6.1. Основные понятия.	445
7.6.2. Создание контрольных точек	448
7.6.3. Протоколирование сообщений	452
7.7. Итоги	455
Вопросы и задания	456
Глава 8. Защита	458
8.1. Общие вопросы защиты.	459
8.1.1. Угрозы, правила и механизмы.	459
8.1.2. Вопросы разработки	465
8.1.3. Криптография	470
8.2. Защищенные каналы	478
8.2.1. Аутентификация	478
8.2.2. Целостность и конфиденциальность сообщений	486
8.2.3. Защищенное групповое взаимодействие	490
8.3. Контроль доступа.	493
8.3.1. Общие вопросы контроля доступа	493
8.3.2. Брандмауэры.	497
8.3.3. Защита мобильного кода.	499
8.4. Управление защитой.	506
8.4.1. Управление ключами	507

8.4.2. Управление защищенными группами	511
8.4.3. Управление авторизацией	512
8.5. Пример — Kerberos	518
8.6. Пример — SESAME	520
8.6.1. Компоненты системы SESAME	521
8.6.2. Сертификаты атрибутов привилегий	523
8.7. Пример — электронные платежные системы	525
8.7.1. Электронные платежные системы	525
8.7.2. Защита в электронных платежных системах	527
8.7.3. Примеры протоколов	531
8.8. Итоги	535
Вопросы и задания	537
Глава 9. Распределенные системы объектов	539
9.1. CORBA	540
9.1.1. Обзор	541
9.1.2. Связь.	547
9.1.3. Процессы	554
9.1.4. Именованное	560
9.1.5. Синхронизация.	564
9.1.6. Кэширование и репликация	565
9.1.7. Отказоустойчивость.	567
9.1.8. Защита	569
9.2. DCOM	572
9.2.1. Обзор	572
9.2.2. Связь.	578
9.2.3. Процессы	581
9.2.4. Именованное	584
9.2.5. Синхронизация	588
9.2.6. Репликация	588
9.2.7. Отказоустойчивость.	588
9.2.8. Защита	589
9.3. Globe	592
9.3.1. Обзор	592
9.3.2. Связь.	600
9.3.3. Процессы	602
9.3.4. Именованное	604
9.3.5. Синхронизация.	607
9.3.6. Репликация	607
9.3.7. Отказоустойчивость.	611
9.3.8. Защита	611
9.4. Сравнение систем CORBA, DCOM и Globe	613
9.4.1. Философия	613
9.4.2. Связь.	615
9.4.3. Процессы	615
9.4.4. Именованное	616
9.4.5. Синхронизация.	617
9.4.6. Кэширование и репликация	617
9.4.7. Отказоустойчивость.	618
9.4.8. Защита	618
9.5. Итоги	620
Вопросы и задания	621

Глава 10. Распределенные файловые системы	623
10.1. Сетевая файловая система компании Sun	624
10.1.1. Обзор	624
10.1.2. Связь	629
10.1.3. Процессы	630
10.1.4. Именованье	631
10.1.5. Синхронизация	639
10.1.6. Кэширование и репликация	644
10.1.7. Отказоустойчивость	647
10.1.8. Защита	650
10.2. Файловая система Coda	654
10.2.1. Обзор	654
10.2.2. Связь	656
10.2.3. Процессы	658
10.2.4. Именованье	659
10.2.5. Синхронизация	661
10.2.6. Кэширование и репликация	665
10.2.7. Отказоустойчивость	668
10.2.8. Защита	671
10.3. Другие распределенные файловые системы	674
10.3.1. Plan 9 — ресурсы как файлы	674
10.3.2. xFS — файловая система без серверов	680
10.3.3. SFS — масштабируемая защита	686
10.4. Сравнение распределенных файловых систем	689
10.4.1. Философия	689
10.4.2. Связь	690
10.4.3. Процессы	690
10.4.4. Именованье	691
10.4.5. Синхронизация	692
10.4.6. Кэширование и репликация	692
10.4.7. Отказоустойчивость	693
10.4.8. Защита	693
10.5. Итоги	695
Вопросы и задания	696
Глава 11. Распределенные системы документов	699
11.1. World Wide Web	699
11.1.1. WWW	700
11.1.2. Связь	709
11.1.3. Процессы	715
11.1.4. Именованье	721
11.1.5. Синхронизация	723
11.1.6. Кэширование и репликация	724
11.1.7. Отказоустойчивость	728
11.1.8. Защита	729
11.2. Lotus Notes	730
11.2.1. Обзор	730
11.2.2. Связь	733
11.2.3. Процессы	734
11.2.4. Именованье	736
11.2.5. Синхронизация	738
11.2.6. Репликация	738

11.2.7. Отказоустойчивость	741
11.2.8. Защита	741
11.3. Сравнение WWW и Lotus Notes	745
11.4. Итоги	750
Вопросы и задания	750

Глава 12. Распределенные системы согласования. 752

12.1. Знакомство с моделями согласования	753
12.2. TIB/Rendezvous	755
12.2.1. Обзор	755
12.2.2. Связь	758
12.2.3. Процессы	762
12.2.4. Именованье	762
12.2.5. Синхронизация	763
12.2.6. Кэширование и репликация	765
12.2.7. Отказоустойчивость	766
12.2.8. Защита	768
12.3. Jini	770
12.3.1. Обзор	770
12.3.2. Связь	773
12.3.3. Процессы	774
12.3.4. Именованье	778
12.3.5. Синхронизация	780
12.3.6. Кэширование и репликация	782
12.3.7. Отказоустойчивость	782
12.3.8. Защита	782
12.4. Сравнение TIB/Rendezvous и Jini	783
12.5. Итоги	787
Вопросы и задания	788

Глава 13. Библиография 790

13.1. Литература для дополнительного чтения	790
13.1.1. Введение и общие вопросы	790
13.1.2. Связь	792
13.1.3. Процессы	792
13.1.4. Именованье	793
13.1.5. Синхронизация	794
13.1.6. Непротиворечивость и репликация	795
13.1.7. Отказоустойчивость	797
13.1.8. Защита	797
13.1.9. Распределенные системы объектов	799
13.1.10. Распределенные файловые системы	800
13.1.11. Распределенные системы документов	801
13.1.12. Распределенные системы согласования	802
13.2. Список литературы	803

Список терминов 833

Алфавитный указатель 855

*Сюзанне, Барбаре, Марвину и памяти Брэма и Свити п
Эндрю С. Тапенбаум*

*Мариэлле, Максу и Эльке
Маартен ван Стеен*

Об авторах

Эндрю С. Таненбаум (Andrew S. Tanenbaum) получил степень бакалавра в М.И.Т. и докторскую степень в университете штата Калифорния в Беркли. В настоящее время он является профессором вычислительной техники в университете Врие в Амстердаме, Голландия, где возглавляет отделение вычислительной техники. Кроме того, он является деканом высшей школы программирования и графики, института исследования по параллельным и распределенным вычислительным системам, а также системам обработки и формирования изображений. Тем не менее он всеми силами старается не превратиться в бюрократа.

В прошлом Эндрю С. Таненбаум занимался исследованиями по компиляторам, операционным системам, компьютерным сетям и локальным распределенным системам. В настоящее время он сосредоточился на разработке глобальной распределенной системы, допускающей масштабирование на миллионы пользователей. Эти исследования он выполняет вместе с доктором Маартеном ван Стееном. Их совместные исследовательские проекты привели к возникновению более чем 90 статей в журналах и материалах конференций и пяти книг.

Профессор Таненбаум разработал также значительный объем программного обеспечения. Он был главным архитектором «амстердамского пакета разработки компиляторов» (Amsterdam Compiler Kit), широко известного пакета для создания переносимых компиляторов, а также MINIX, миниатюрного клона UNIX, предназначенного для студенческих лабораторных работ по программированию. Вместе со своими аспирантами и программистами он способствовал созданию распределенной операционной системы Атмoeва, высокопроизводительной распределенной операционной системы на базе микроядра. Системы MINIX и Атмoeва бесплатно распространяются через Интернет.

Его аспирантов после получения степени ожидает блестящий успех. Он очень ими гордится. В этом отношении он очень напоминает курицу-наседку.

Профессор Таненбаум является членом ACM и IEEE, действительным членом Королевской академии наук и искусств Нидерландов, ему присуждены премии ACM «За заслуги в преподавательской деятельности Карла В. Кальстрема» 1994 года и ACM/SIGCSE «За впечатляющий вклад в преподавание компьютерных дисциплин» 1997 года. Он упомянут также в мировом справочнике «Кто есть кто». Его домашнюю страницу в Интернете можно найти по адресу <http://www.cs.vu.nl/~ast/>.

Маартен ван Стеен (Maarten van Steen) в настоящее время является доцентом университета Врие в Амстердаме, где ведет курсы по операционным системам, компьютерным сетям и распределенным системам. Он также успешно ведет

различные курсы по вопросам компьютеризации для специалистов ICT промышленности и государственных органов.

Доктор ван Стеен изучал прикладную математику в университете Твенте и получил степень доктора философии в Лейденском университете, защитив диссертацию в области методики разработки параллельных систем. После защиты он перешел на работу в промышленную исследовательскую лабораторию, где вырос до руководителя группы компьютерных систем, занимавшейся программной поддержкой параллельных приложений.

После пяти лет одновременных занятий исследованиями и руководства группой он решил вернуться в академическую науку, сначала в качестве ассистента факультета вычислительной техники в университете Эразма в Роттердаме, а позже — в качестве доцента в группе Эндрю Таненбаума в университете Врие в Амстердаме. Возвращение в университет было правильным решением, во всяком случае, его жена полагает именно так.

В настоящее время он занимается исследованиями крупномасштабных глобальных систем, сосредоточившись на вопросах локализации мобильных агентов, архитектуре систем и адаптивном распределении и репликации. Вместе с профессором Таненбаумом он руководит проектом Globe, в ходе которого группа из приблизительно дюжины исследователей совместно разрабатывают глобальную распределенную систему с таким же названием. Система Globe описана на сайте <http://www.cs.vu.nl/globe>.

Предисловие

Эта книга первоначально задумывалась как пересмотренный вариант книги *Distributed Operating Systems*, но вскоре мы поняли, что с 1995 года произошли такие изменения, что простым пересмотром здесь не обойтись. Нужна совершенно новая книга. Эта новая книга получила и новое название *Distributed Systems: Principles and Paradigms*. Изменения в названии отражают факт смещения акцентов. Раньше мы рассматривали вопросы, связанные с операционными системами, а эта книга посвящена распределенным системам в более широком смысле этого слова. Так, например, среда World Wide Web, которая считается самой большой из когда-либо построенных распределенных систем, в оригинальной книге даже не упоминалась, поскольку не является операционной системой. Теперь ей отведена едва ли не целая глава.

Книгу логически образуют две части — принципы и парадигмы. В первой главе дается введение в тему. Затем следует семь глав, описывающих отдельные принципы, которые мы сочли особенно важными: связь, процессы, именование, синхронизация, непротиворечивость и репликация, защита от сбоев и защита.

Реальные распределенные системы строятся обычно на основе некоей парадигмы, например, «все кругом — это файлы». В следующих четырех главах рассматриваются различные парадигмы и описываются построенные на их основе реальные системы. В качестве парадигм выбраны системы объектов, распределенные файловые системы, системы документов и системы согласования.

В последней главе имеется аннотация литературы, которая может быть использована в качестве отправной точки для дополнительного изучения темы, а также список работ, цитируемых в этой книге.

Книга предназначена для студентов старших курсов и дипломников по «компьютерным дисциплинам». Соответственно, у нее есть свой web-сайт с таблицами в формате PowerPoint и рисунками из книги в различных форматах. На него можно попасть, зайдя на страницу www.prenhall.com/nanenbaum и щелкнув на ссылке с названием этой книги. Для преподавателей, использующих эту книгу на занятиях, доступно руководство с решениями всех упражнений. За экземпляром руководства им следует обратиться к представителю издательства Prentice Hall. Разумеется, книга прекрасно подойдет и для тех, кто, хотя и никак не связан с получением образования, хотел бы получить дополнительные знания по этой нужной теме.

Множество людей внесли свой вклад в создание этой книги. Мы хотим особо поблагодарить Арно Беккера (Arno Bakker), Герко Баллентейна (Gerco Balintijn), Брента Саллахана (Brent Callaghan), Скотта Кеннона (Scott Cannon),

Сандру Комлиссен (Sandra Comelissen), Майка Далина (Mike Dahlin), Марка Дербишира (Mark Darbyshire), Гая Эддона (Guy Eddon), Амра эль Аббади (Amr el Abbadi), Винсента Фриха (Vincent Freeh), Чандана Гамаде (Chandana Gamage), Бена Граса (Ben Gras), Боба Грея (Bob Gray), Мишеля ван Хартскампа (Michael van Hartskamp), Филиппа Хомбурга (Philip Homburg), Эндрю Китчена (Andrew Kitchen), Ладислава Короута (Ladislav Kohout), Боба Каттера (Bob Kutter), Юссипекка Лейво (Jussipekka Leiwo), Леха Мак-Тэггера (Leah McTaggart), Эли Мессенгера (Eli Messenger), Дональда Миллера (Donald Miller), Шиваканта Мишру (Shivakant Mishra), Джима Муни (Jim Mooney), Мэтта Мутку (Matt Mutka), Роба Пайка (Rob Pike), Крити Рамамритама (Krithi Ramamritham), Шмуэля Ротенстрейча (Shmuel Rotenstreich), Сола Шатца (Sol Shatz), Гурдипа Сингха (Gurdip Singh), Адиту Шиврам (Aditya Shivram), Владимира Суконника (Vladimir Sukonnik), Болеслава Шимански (Boleslaw Szymanski), Лорена Теронда (Laurent Therond) и Леендерта ван Дума (Leendert van Doom) за то, что они прочли отдельные части рукописи и внесли полезные предложения.

И наконец, мы рады сказать «спасибо» членам наших семей.

Сюзанна проходила через этот кошмар уже дюжину раз и ни разу не сказала: «Хватит!», — хотя эта мысль, наверное, неоднократно приходила ей в голову. Спасибо. Барбара и Марвин теперь точно знают, что должен делать профессор, чтобы заработать себе на жизнь, и сумеют отличить хорошие учебники от плохих (меня они вдохновляют на создание хороших).

Эндрю С. Тапенбаум

Мариэль узнала, что ее ждет, когда я сказал ей, что снова решил заняться писательством. С самого начала она благосклонно отнеслась к этой идее, заметив также, что это занятие более приятное и не столь бесполезное, как предыдущие. «Спихнуть» на нее Эльке на все время создания книги — было не лучшей идеей, но позволило мне сконцентрироваться и правильно расставить приоритеты. В этом отношении прекрасно вел себя Макс, поскольку, будучи старше Эльке, хорошо понимал, когда лучше играть с кем-нибудь другим. Они — великолепные дети.

Маартен ван Стеен

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на web-сайте издательства <http://www.piter.com>.

Руководство по использованию книги

Много лет материалы, которые легли в основу этой книги, использовались для обучения студентов старших курсов и дипломников. Кроме того, на этих материалах строились одно-двухдневные семинары по распределенным системам и системам промежуточного уровня для аудитории, состоявшей из профессионалов-компьютерщиков.

Курсы для студентов старших курсов и дипломников

Студентам старших курсов и дипломникам материал этой книги обычно излагается в течение 12–15 недель. По нашим наблюдениям, для большинства студентов изучение распределенных систем заключается в изучении множества тем, тесно связанным друг с другом. Книга организована следующим образом: сначала мы представляем тему с различных позиций, а затем рассматриваем каждую позицию по отдельности. Это помогает поддерживать внимание студентов. В результате к концу первой части (главы 1–8) студенты оказываются хорошо подготовленными к восприятию общей картины.

Тем не менее такая сложная область, как распределенные системы, подразумевает множество различных аспектов. Некоторые из этих аспектов трудны для понимания, особенно когда изучаются впервые. Поэтому мы настойчиво рекомендуем студентам изучать главы в порядке, соответствующем курсу. Все документы, доступные на сайте соавторов (www.prenhall.com/tanenbaum), предназначены для того, чтобы студенты могли активнее работать на занятиях. Этот подход весьма успешен и высоко ценится студентами.

Весь материал можно уложить в 15-недельный курс. Большая часть времени уходит на изучение принципов распределенных систем, то есть на материал, изложенный в первых восьми главах. Наш опыт подсказывает, что когда настает время обсуждения парадигмы, остается изложить только самое главное. Детали проще изучать не на уроке, а непосредственно по книге. Так, например, в курсе мы отводим на системы объектов всего одну неделю, в то время как в книге они занимают приблизительно 80 страниц. Ниже приводится график курса, иллюстрирующий соответствие тем и отводимого на них лекционного времени.

Неделя	Тема	Глава	Разделы
1	Введение	1	Целиком
2	Связь	2	2.1–2.3
3	Связь	2	2.4–2.5
4	Процессы	3	Целиком
5	Именованье	4	4.1–4.2
6	Именованье	4	4.3
6	Синхронизация	5	5.1–5.2
7	Синхронизация	5	5.3–5.6
8	Непротиворечивость и репликация	6	6.1–6.4
9	Непротиворечивость и репликация	6	6.5–6.6
9	Отказоустойчивость	7	7.1–7.3
10	Отказоустойчивость	7	7.4–7.6
11	Защита	8	8.1–8.2
12	Защита	8	8.3–8.7
13	Распределенные системы объектов	9	Целиком
14	Распределенные файловые системы	10	Целиком
15	Распределенные системы документов	11	Целиком
15	Распределенные системы согласования	12	Целиком

Не все материалы предназначены для изучения на занятиях. Предполагается, что отдельные части студенты будут прорабатывать сами, в особенности это относится к деталям. Если на курс отводится менее 15 недель, мы рекомендуем пропустить главы, относящиеся к парадигме. Интересующиеся этой темой студенты могут прочесть их самостоятельно.

Для студентов младших курсов мы советуем разделить материал на два семестра и добавить к нему лабораторные работы. Так, студенты могут поработать с простой распределенной системой, модифицируя ее компоненты так, чтобы обеспечивать отказоустойчивость, обработку вызовов RPC при групповой рассылке и пр.

Семинары для профессионалов

На одно-двухдневных семинарах мы используем эту книгу как основной источник информации. Весь материал вполне можно изложить за два дня, отводимых на семинар, если пропускать все детали и сосредоточиться на основах распределенных систем. Кроме того, чтобы изложение было более живым, можно поменять порядок изложения. Это позволяет сразу же показать, как используются основные принципы. Студенты-дипломники перед тем, как перейти к вопросам применения распределенных систем, нуждаются в 10 неделях изложения принципов их строения (если они вообще заинтересованы в этих знаниях). Мотивация профессионалов возрастает, если они видят, как принципы используются на практике. Предварительный план двухдневного курса предполагает разбиение на логические блоки.

Первый день

Урок	Время, мин	Тема	Глава	Акцент
1	90	Введение	1	Архитектура клиент-сервер
2	60	Связь	2	RPC/RMI и сообщения
3	60	Системы согласования	12	Обмен сообщениями
4	60	Процессы	3	Мобильный код и агенты
5	30	Именованние	4	Трассировка мест размещения
6	90	Системы объектов	9	CORBA

Второй день

Урок	Время, мин	Тема	Глава	Акцент
1	90	Непротиворечивость и репликация	6	Модели и протоколы
2	60	Системы документов	11	Кэширование и репликация в Web
3	60	Отказоустойчивость	7	Группы процессов и протокол 2PC
4	90	Защита	8	Основные идеи
5	60	Распределенные файловые системы	10	NFS версий 3 и 4

Самостоятельное изучение

Эта книга с успехом может применяться и для самостоятельного изучения. Если имеются мотивация и свободное время, читателю можно посоветовать читать книгу подряд, от корки до корки.

Если времени на изучение всего материала не хватает, мы рекомендуем сосредоточиться на самых важных темах. В приведенной таблице указаны разделы, в которых, как мы полагаем, рассматриваются наиболее важные аспекты распределенных систем вместе с поясняющими примерами.

Глава	Тема	Разделы
1	Введение	1.1, 1.2, 1.4.3, 1.5
2	Связь	2.2, 2.3, 2.4
3	Процессы	3.3, 3.4, 3.5
4	Именованние	4.1, 4.2
5	Синхронизация	5.2, 5.3, 5.6
6	Непротиворечивость и репликация	6.1, 6.2.2, 6.2.5, 6.4, 6.5
7	Отказоустойчивость	7.1, 7.2.1, 7.2.2, 7.3, 7.4.1, 7.4.3, 7.5.1

Глава	Тема	Разделы
8	Защита	8.1, 8.2.1, 8.2.2, 8.3, 8.4
9	Распределенные системы объектов	9.1, 9.2, 9.4
10	Распределенные файловые системы	10.1, 10.4
11	Распределенные системы документов	11.1
12	Распределенные системы согласования	12.1, 12.2 или 12.3

Было бы неплохо, если бы можно было заранее оценить время, необходимое на усвоение рекомендуемого материала, однако это время определяется базовыми знаниями читателя. Тем не менее даже если этот материал читать по вечерам после работы, на него потребуется максимум несколько недель.

Глава 1

Введение

- 1.1. Определение распределенной системы
- 1.2. Задачи
- 1.3. Концепции аппаратных решений
- 1.4. Концепции программных решений
- 1.5. Модель клиент-сервер
- 1.6. Итоги

Компьютерные системы претерпевают революцию. С 1945 года, когда началась эпоха современных компьютеров, до приблизительно 1985 года компьютеры были большими и дорогими. Даже мини-компьютеры стоили сотни тысяч долларов. В результате большинство организаций имели в лучшем случае лишь несколько компьютеров, и, поскольку методы их соединения отсутствовали, эти компьютеры работали независимо друг от друга.

Однако в середине восьмидесятых под воздействием двух технологических новинок ситуация начала меняться. Первой из этих новинок была разработка мощных микропроцессоров. Изначально они были 8-битными, затем стали доступны 16-, 32- и 64-битные процессоры. Многие из них обладали вычислительной мощностью мэйнфреймов (то есть больших компьютеров), но лишь частью их цены.

Скорость роста, наблюдавшаяся в компьютерных технологиях в последние полвека, действительно потрясает. Ей нет прецедентов в других отраслях. От машин, стоивших 100 миллионов долларов и выполнявших одну команду в секунду, мы пришли к машинам, стоящим 1000 долларов и выполняющим 10 миллионов команд в секунду. Разница в соотношении цена/производительность достигла порядка 10^{12} . Если бы автомобили за этот период совершенствовались такими же темпами, «роллс-ройс» сейчас стоил бы один доллар и проходил миллиард миль на одном галлоне бензина (к сожалению, к нему потребовалось бы 200-страничное руководство по открыванию дверей).

Второй из новинок было изобретение высокоскоростных компьютерных сетей. *Локальные сети (Local-Area Networks, LAN)* соединяют сотни компьютеров, находящихся в здании, таким образом, что машины в состоянии обмениваться

небольшими порциями информации за несколько микросекунд. Большие массивы данных передаются с машины на машину со скоростью от 10 до 1000 Мбит/с. *Глобальные сети* (*Wide-Area Networks, WAN*) позволяют миллионам машин во всем мире обмениваться информацией со скоростями, варьирующимися от 64 кбит/с (килобит в секунду) до гигабит в секунду.

В результате развития этих технологий сегодня не просто возможно, но и достаточно легко можно собрать компьютерную систему, состоящую из множества компьютеров, соединенных высокоскоростной сетью. Она обычно называется компьютерной сетью, или *распределенной системой* (*distributed system*), в отличие от предшествовавших ей *централизованных* (*centralized systems*), или *однопроцессорных* (*single-processor systems*), систем, состоявших из одного компьютера, его периферии и, возможно, нескольких удаленных терминалов.

1.1. Определение распределенной системы

В литературе можно найти различные определения распределенных систем, причем ни одно из них не является удовлетворительным и не согласуется с остальными. Для наших задач хватит достаточно вольной характеристики.

Распределенная система — это набор независимых компьютеров, представляющий их пользователям единой объединенной системой.

В этом определении оговариваются два момента. Первый относится к аппаратуре: все машины автономны. Второй касается программного обеспечения: пользователи думают, что имеют дело с единой системой. Важны оба момента. Позже в этой главе мы к ним вернемся, но сначала рассмотрим некоторые базовые вопросы, касающиеся как аппаратного, так и программного обеспечения.

Возможно, вместо того чтобы рассматривать определения, разумнее будет сосредоточиться на важных характеристиках распределенных систем. Первая из таких характеристик состоит в том, что от пользователей скрыты различия между компьютерами и способы связи между ними. То же самое относится и к внешней организации распределенных систем. Другой важной характеристикой распределенных систем является способ, при помощи которого пользователи и приложения единообразно работают в распределенных системах, независимо от того, где и когда происходит их взаимодействие.

Распределенные системы должны также относительно легко поддаваться расширению, или масштабированию. Эта характеристика является прямым следствием наличия независимых компьютеров, но в то же время не указывает, каким образом эти компьютеры на самом деле объединяются в единую систему. Распределенные системы обычно существуют постоянно, однако некоторые их части могут временно выходить из строя. Пользователи и приложения не должны уведомляться о том, что эти части заменены или починены или что добав-

лены новые части для поддержки дополнительных пользователей или приложений.

Для того чтобы поддержать представление различных компьютеров и сетей в виде единой системы, организация распределенных систем часто включает в себя дополнительный уровень программного обеспечения, находящийся между верхним уровнем, на котором находятся пользователи и приложения, и нижним уровнем, состоящим из операционных систем, как показано на рис. 1.1. Соответственно, такая распределенная система обычно называется *системой промежуточного уровня (middleware)*.

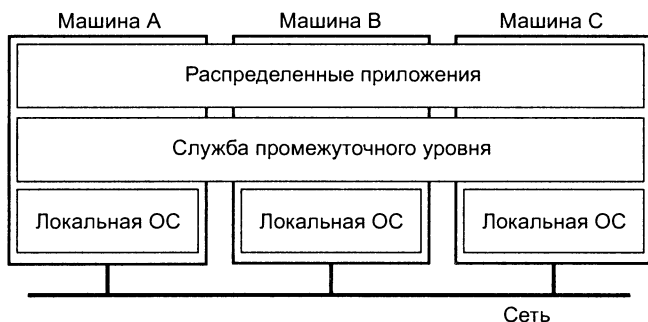


Рис. 1.1. Распределенная система организована в виде службы промежуточного уровня. Отметим, что промежуточный уровень распределен среди множества компьютеров

Взглянем теперь на некоторые примеры распределенных систем. В качестве первого примера рассмотрим сеть рабочих станций в университете или отделе компании. Вдобавок к персональной рабочей станции каждого из пользователей имеется пул процессоров машинного зала, не назначенных заранее ни одному из пользователей, но динамически выделяемых им при необходимости. Эта распределенная система может обладать единой файловой системой, в которой все файлы одинаково доступны со всех машин с использованием постоянного пути доступа. Кроме того, когда пользователь набирает команду, система может найти наилучшее место для выполнения запрашиваемого действия, возможно, на собственной рабочей станции пользователя, возможно, на простаивающей рабочей станции, принадлежащей кому-то другому, а может быть, и на одном из свободных процессоров машинного зала. Если система в целом выглядит и ведет себя как классическая однопроцессорная система с разделением времени (то есть многопользовательская), она считается распределенной системой. В качестве второго примера рассмотрим работу информационной системы, которая поддерживает автоматическую обработку заказов. Обычно подобные системы используются сотрудниками нескольких отделов, возможно в разных местах. Так, сотрудники отдела продаж могут быть разбросаны по обширному региону или даже по всей стране. Заказы передаются с переносных компьютеров, соединяемых с системой при помощи телефонной сети, а возможно, и при помощи сотовых телефонов. Приходящие заказы автоматически передаются в отдел планирования, превращаясь там во внутренние заказы на поставку, которые поступают в отдел достав-

ки, и в заявки на оплату, поступающие в бухгалтерию. Система автоматически пересылает эти документы имеющимся на месте сотрудникам, отвечающим за их обработку. Пользователи остаются в полном неведении о том, как заказы на самом деле курсируют внутри системы, для них все это представляется так, будто вся работа происходит в централизованной базе данных.

В качестве последнего примера рассмотрим World Wide Web. Web предоставляет простую, целостную и единообразную модель распределенных документов. Чтобы увидеть документ, пользователю достаточно активизировать ссылку. После этого документ появляется на экране. В теории (но определенно не в текущей практике) нет необходимости знать, с какого сервера доставляется документ, достаточно лишь информации о том, где он расположен. Публикация документа очень проста: вы должны только задать ему уникальное имя в форме *унифицированного указателя ресурса (Uniform Resource Locator, URL)*, которое ссылается на локальный файл с содержимым документа. Если бы Всемирная паутина представлялась своим пользователям гигантской централизованной системой документооборота, она также могла бы считаться распределенной системой. К сожалению, этот момент еще не наступил. Так, пользователи сознают, что документы находятся в различных местах и распределены по различным серверам.

1.2. Задачи

Возможность построения распределенных систем еще не означает полезность этого. Современная технология позволяет подключить к персональному компьютеру четыре дисководы. Это возможно, но бессмысленно. В этом разделе мы обсудим четыре важных задачи, решение которых делает построение распределенных систем осмысленным. Распределенные системы могут легко соединять пользователей с вычислительными ресурсами и успешно скрывать тот факт, что ресурсы разбросаны по сети и могут быть открытыми и масштабируемыми.

1.2.1. Соединение пользователей с ресурсами

Основная задача распределенных систем — облегчить пользователям доступ к удаленным ресурсам и обеспечить их совместное использование, регулируя этот процесс. Ресурсы могут быть виртуальными, однако традиционно они включают в себя принтеры, компьютеры, устройства хранения данных, файлы и данные. Web-страницы и сети также входят в этот список. Существует множество причин для совместного использования ресурсов. Одна из очевидных — это экономичность. Например, гораздо дешевле разрешить совместную работу с принтером нескольких пользователей, чем покупать и обслуживать отдельный принтер для каждого пользователя. Точно так же имеет смысл совместно использовать дорогие ресурсы, такие как суперкомпьютеры или высокопроизводительные хранилища данных.

Соединение пользователей и ресурсов также облегчает кооперацию и обмен информацией, что лучше всего иллюстрируется успехом Интернета с его простыми протоколами для обмена файлами, почтой, документами, аудио- и видеоинформацией. Связь через Интернет в настоящее время привела к появлению многочисленных виртуальных организаций, в которых географически удаленные друг от друга группы сотрудников работают вместе при помощи *систем групповой работы (groupware)* — программ для совместного редактирования документов, проведения телеконференций и т. п. Подобным же образом подключение к Интернету вызвало к жизни электронную коммерцию, позволяющую нам покупать и продавать любые виды товаров, обходясь без реального посещения магазина.

Однако по мере роста числа подключений и степени совместного использования ресурсов все более и более важными становятся вопросы безопасности. В современной практике системы имеют слабую защиту от подслушивания или вторжения по линиям связи. Пароли и другая особо важная информация часто пересылаются по сетям открытым текстом (то есть незашифрованными) или хранятся на серверах, надежность которых не подтверждена ничем, кроме нашей веры. Здесь имеется еще очень много возможностей для улучшения. Так, например, в настоящее время для заказа товаров необходимо просто сообщить номер своей кредитной карты. Редко требуется подтверждение того, что покупатель действительно владеет этой картой. В будущем заказ товара таким образом будет возможен только в том случае, если вы сможете физически подтвердить факт обладания этой картой при помощи считывателя карт.

Другая проблема безопасности состоит в том, что прослеживание коммуникаций позволяет построить профиль предпочтений конкретного пользователя [485]. Подобное отслеживание серьезно нарушает права личности, особенно если производится без уведомления пользователя. Связанная с этим проблема состоит в том, что рост подключений ведет к росту нежелательного общения, такого как получаемые по электронной почте бессмысленные письма, так называемый спам. Единственное, что мы можем сделать в этом случае, это защитить себя, используя специальные информационные фильтры, которые сортируют входящие сообщения на основании их содержимого.

1.2.2. Прозрачность

Важная задача распределенных систем состоит в том, чтобы скрыть тот факт, что процессы и ресурсы физически распределены по множеству компьютеров. Распределенные системы, которые представляются пользователям и приложениям в виде единой компьютерной системы, называются *прозрачными (transparent)*. Рассмотрим сначала, как прозрачность реализуется в распределенных системах, а затем зададимся вопросом, зачем вообще она нужна.

Прозрачность в распределенных системах

Концепция прозрачности, как видно из табл. 1.1, применима к различным аспектам распределенных систем [210].

Таблица 1.1. Различные формы прозрачности в распределенных системах

Прозрачность	Описание
Доступ	Скрывается разница в представлении данных и доступе к ресурсам
Местоположение	Скрывается местоположение ресурса
Перенос	Скрывается факт перемещения ресурса в другое место
Смена местоположения	Скрывается факт перемещения ресурса в процессе обработки в другое место
Репликация	Скрывается факт репликации ресурса
Параллельный доступ	Скрывается факт возможного совместного использования ресурса несколькими конкурирующими пользователями
Отказ	Скрывается отказ и восстановление ресурса
Сохранность	Скрывается, хранится ресурс (программный) на диске или находится в оперативной памяти

Прозрачность доступа (access transparency) призвана скрыть разницу в представлении данных и в способах доступа пользователя к ресурсам. Так, при пересылке целого числа с рабочей станции на базе процессора Intel на Sun SPARC необходимо принять во внимание, что процессоры Intel оперируют с числами формата «младший — последним» (то есть первым передается старший байт), а процессор SPARC использует формат «старший — последним» (то есть первым передается младший байт). Также в данных могут присутствовать и другие несоответствия. Например, распределенная система может содержать компьютеры с различными операционными системами, каждая из которых имеет собственные ограничения на способ представления имен файлов. Разница в ограничениях на способ представления имен файлов, так же как и собственно работа с ними, должны быть скрыты от пользователей и приложений.

Важная группа типов прозрачности связана с местоположением ресурсов. *Прозрачность местоположения (location transparency)* призвана скрыть от пользователя, где именно физически расположен в системе нужный ему ресурс. Важную роль в реализации прозрачности местоположения играет именование. Так, прозрачность местоположения может быть достигнута путем присвоения ресурсам только логических имен, то есть таких имен, в которых не содержится закодированных сведений о местоположении ресурса. Примером такого имени может быть URL: <http://~wiv.prenhall.com/index.html>, в котором не содержится никакой информации о реальном местоположении главного web-сервера издательства Prentice Hall. URL также не дает никакой информации о том, находился ли файл `index.html` в указанном месте постоянно или оказался там недавно. О распределенных системах, в которых смена местоположения ресурсов не влияет на доступ к ним, говорят как об обеспечивающих *прозрачность переноса (migration transparency)*. Более серьезна ситуация, когда местоположение ресурсов может измениться в процессе их использования, причем пользователь или приложение ничего не заметят. В этом случае говорят, что система поддерживает *прозрачность смены местоположения (relocation transparency)*. Примером могут

служить мобильные пользователи, работающие с беспроводным переносным компьютером и не отключающиеся (даже временно) от сети при перемещении с места на место.

Как мы увидим, репликация имеет важное значение в распределенных системах. Так, ресурсы могут быть реплицированы для их лучшей доступности или повышения их производительности путем помещения копии неподалеку от того места, из которого к ней осуществляется доступ. *Прозрачность репликации* (*replication transparency*) позволяет скрыть тот факт, что существует несколько копий ресурса. Для скрытия факта репликации от пользователей необходимо, чтобы все реплики имели одно и то же имя. Соответственно, система, которая поддерживает прозрачность репликации, должна поддерживать и прозрачность местоположения, поскольку иначе невозможно будет обращаться к репликам без указания их истинного местоположения.

Мы часто упоминаем, что главная цель распределенных систем — обеспечить совместное использование ресурсов. Во многих случаях совместное использование ресурсов достигается посредством кооперации, например в случае коммуникаций. Однако существует множество примеров настоящего совместного использования ресурсов. Например, два независимых пользователя могут сохранять свои файлы на одном файловом сервере или работать с одной и той же таблицей в совместно используемой базе данных. Следует отметить, что в таких случаях ни один из пользователей не имеет никакого понятия о том, что тот же ресурс задействован другим пользователем. Это явление называется *прозрачностью параллельного доступа* (*concurrency transparency*). Отметим, что подобный параллельный доступ к совместно используемому ресурсу сохраняет этот ресурс в непротиворечивом состоянии. Непротиворечивость может быть обеспечена механизмом блокировок, когда пользователи, каждый по очереди, получают исключительные права на запрашиваемый ресурс. Более изощренный вариант — использование транзакций, однако, как мы увидим в следующих главах, механизм транзакций в распределенных системах труднореализуем.

Популярное альтернативное определение распределенных систем, принадлежащее Лесли Лампорту (Leslie Lamport), выглядит так: «Вы понимаете, что у вас есть эта штука, поскольку при поломке компьютера вам никогда не предлагают приостановить работу». Это определение указывает еще на одну важную сторону распределенных систем: прозрачность отказов. *Прозрачность отказов* (*failure transparency*) означает, что пользователя никогда не уведомляют о том, что ресурс (о котором он мог никогда и не слышать) не в состоянии правильно работать и что система далее восстановилась после этого повреждения. Маскировка сбоев — это одна из сложнейших проблем в распределенных системах и столь же необходимая их часть. Определенные соображения по этому поводу будут приведены в главе 7. Основная трудность состоит в маскировке проблем, возникающих в связи с невозможностью отличить неработоспособные ресурсы от ресурсов с очень медленным доступом. Так, контактируя с перегруженным web-сервером, браузер выжидает положенное время, а затем сообщает о недоступности страницы. При этом пользователь не должен думать, что сервер и правда не работает.

Последний тип прозрачности, который обычно ассоциируется с распределенными системами, — это *прозрачность сохранности* (*persistence transparency*), маскирующая реальную (диск) или виртуальную (оперативная память) сохранность ресурсов. Так, например, многие объектно-ориентированные базы данных предоставляют возможность непосредственного вызова методов для сохраненных объектов. За сценой в этот момент происходит следующее: сервер баз данных сначала копирует состояние объекта с диска в оперативную память, затем выполняет операцию и, наконец, записывает состояние на устройство длительного хранения. Пользователь, однако, остается в неведении о том, что сервер перемещает данные между оперативной памятью и диском. Сохранность играет важную роль в распределенных системах, однако не менее важна она и для обычных (не распределенных) систем.

Степень прозрачности

Хотя прозрачность распределения в общем желательна для всякой распределенной системы, существуют ситуации, когда попытки полностью скрыть от пользователя всякую распределенность не слишком разумны. Это относится, например, к требованию присылать вам свежую электронную газету до 7 утра по местному времени, особенно если вы находитесь на другом конце света и живете в другом часовом поясе. Иначе ваша утренняя газета окажется совсем не той утренней газетой, которую вы ожидаете.

Точно так же в глобальной распределенной системе, которая соединяет процесс в Сан-Франциско с процессом в Амстердаме, вам не удастся скрыть тот факт, что мать-природа не позволяет пересылать сообщения от одного процесса к другому быстрее чем за примерно 35 мс. Практика показывает, что при использовании компьютерных сетей на это реально требуется несколько сотен миллисекунд. Скорость передачи сигнала ограничивается не столько скоростью света, сколько скоростью работы промежуточных переключателей.

Кроме того, существует равновесие между высокой степенью прозрачности и производительностью системы. Так, например, многие приложения, предназначенные для Интернета, многократно пытаются установить контакт с сервером, пока, наконец, не откажутся от этой затеи. Соответственно, попытки замаскировать сбой на промежуточном сервере, вместо того чтобы попытаться работать через другой сервер, замедляют всю систему. В данном случае было бы эффективнее как можно быстрее прекратить эти попытки или по крайней мере позволить пользователю прервать попытки установления контакта.

Еще один пример: мы нуждаемся в том, чтобы реплики, находящиеся на разных континентах, были в любой момент гарантированно идентичны. Другими словами, если одна копия изменилась, изменения должны распространиться на все системы до того, как они выполнят какую-либо операцию. Понятно, что одинокая операция обновления может в этом случае занимать до нескольких секунд и вряд ли возможно проделать ее незаметно для пользователей.

Вывод из этих рассуждений следующий. Достижение прозрачности распределения — это разумная цель при проектировании и разработке распределенных

систем, но она не должна рассматриваться в отрыве от других характеристик системы, например производительности.

1.2.3. Открытость

Другая важная характеристика распределенных систем — это открытость. *Открытая распределенная система (open distributed system)* — это система, предлагающая службы, вызов которых требует стандартные синтаксис и семантику. Например, в компьютерных сетях формат, содержимое и смысл посылаемых и принимаемых сообщений подчиняются типовым правилам. Эти правила формализованы в протоколах. В распределенных системах службы обычно определяются через *интерфейсы (interfaces)*, которые часто описываются при помощи *языка определения интерфейсов (Interface Definition Language, IDL)*. Описание интерфейса на IDL почти исключительно касается синтаксиса служб. Другими словами, оно точно отражает имена доступных функций, типы параметров, возвращаемых значений, исключительные ситуации, которые могут быть возбуждены службой и т. п. Наиболее сложно точно описать то, что делает эта служба, то есть семантику интерфейсов. На практике подобные спецификации задаются неформально, посредством естественного языка.

Будучи правильно описанным, определение интерфейса допускает возможность совместной работы произвольного процесса, нуждающегося в таком интерфейсе, с другим произвольным процессом, предоставляющим этот интерфейс. Определение интерфейса также позволяет двум независимым группам создать абсолютно разные реализации этого интерфейса для двух различных распределенных систем, которые будут работать абсолютно одинаково. Правильное определение самодостаточно и нейтрально. «Самодостаточно» означает, что в нем имеется все необходимое для реализации интерфейса. Однако многие определения интерфейсов сделаны самодостаточными не до конца, поскольку разработчикам необходимо включать в них специфические детали реализации. Важно отметить, что спецификация не определяет внешний вид реализации, она должна быть нейтральной. Самодостаточность и нейтральность необходимы для обеспечения переносимости и способности к взаимодействию [65]. *Способность к взаимодействию (interoperability)* характеризует, насколько две реализации систем или компонентов от разных производителей в состоянии совместно работать, полагаясь только на то, что службы каждой из них соответствуют общему стандарту. *Переносимость (portability)* характеризует то, насколько приложение, разработанное для распределенной системы *A*, может без изменений выполняться в распределенной системе *B*, реализуя те же, что и в *A* интерфейсы.

Следующая важная характеристика открытых распределенных систем — это гибкость. Под гибкостью мы понимаем легкость конфигурирования системы, состоящей из различных компонентов, возможно от разных производителей. Не должны вызывать затруднений добавление к системе новых компонентов или замена существующих, при этом прочие компоненты, с которыми не производилось никаких действий, должны оставаться неизменными. Другими словами, от-

крытая распределенная система должна быть расширяемой. Например, к гибкой системе должно быть относительно несложно добавить части, работающие под управлением другой операционной системы, или даже заменить всю файловую систему целиком. Насколько всем нам знакома сегодняшняя реальность, говорить о гибкости куда проще, чем ее осуществить.

Отделение правил от механизмов

В построении гибких открытых распределенных систем решающим фактором оказывается организация этих систем в виде наборов относительно небольших и легко заменяемых или адаптируемых компонентов. Это предполагает необходимость определения не только интерфейсов верхнего уровня, с которыми работают пользователи и приложения, но также и интерфейсов внутренних модулей системы и описания взаимодействия этих модулей. Этот подход относительно молод. Множество старых и современных систем создавались цельными так, что компоненты одной гигантской программы разделялись только логически. В случае использования этого подхода независимая замена или адаптация компонентов, не затрагивающая систему в целом, была почти невозможна. Монолитные системы вообще стремятся скорее к закрытости, чем к открытости.

Необходимость изменений в распределенных системах часто связана с тем, что компонент не оптимальным образом соответствует нуждам конкретного пользователя или приложения. Так, например, рассмотрим кэширование в World Wide Web. Браузеры обычно позволяют пользователям адаптировать правила кэширования под их нужды путем определения размера кэша, а также того, должен ли кэшируемый документ проверяться на соответствие постоянно или только один раз за сеанс. Однако пользователь не может воздействовать на другие параметры кэширования, такие как длительность сохранения документа в кэше или очередность удаления документов из кэша при его переполнении. Также невозможно создавать правила кэширования на основе *содержимого* документа. Так, например, пользователь может пожелать кэшировать железнодорожные расписания, которые редко изменяются, но никогда — информацию о пробках на улицах города.

Нам необходимо отделить правила от механизма. В случае кэширования в Web, например, браузер в идеале должен предоставлять только возможности для сохранения документов в кэше и одновременно давать пользователям возможность решать, какие документы и насколько долго там хранить. На практике это может быть реализовано предоставлением большого списка параметров, значения которых пользователь сможет (динамически) задавать. Еще лучше, если пользователь получит возможность сам устанавливать правила в виде подключаемых к браузеру компонентов. Разумеется, браузер должен понимать интерфейс этих компонентов, поскольку ему нужно будет, используя этот интерфейс, вызывать процедуры, содержащиеся в компонентах.

1.2.4. Масштабируемость

Повсеместная связь через Интернет быстро стала таким же обычным делом, как возможность послать кому угодно в мире письмо по почте. Помня это, мы гово-

рим, что масштабируемость — это одна из наиболее важных задач при проектировании распределенных систем.

Масштабируемость системы может измеряться по трем различным показателям [314]. Во-первых, система может быть масштабируемой по отношению к ее размеру, что означает легкость подключения к ней дополнительных пользователей и ресурсов. Во-вторых, система может масштабироваться географически, то есть пользователи и ресурсы могут быть разнесены в пространстве. В-третьих система может быть масштабируемой в административном смысле, то есть быть проста в управлении при работе во множестве административно независимых организаций. К сожалению, система, обладающая масштабируемостью по одному или нескольким из этих параметров, при масштабировании часто дает потерю производительности.

Проблемы масштабируемости

Если система нуждается в масштабировании, необходимо решить множество разнообразных проблем. Сначала рассмотрим масштабирование по размеру. Если возникает необходимость увеличить число пользователей или ресурсов, мы нередко сталкиваемся с ограничениями, связанными с централизацией служб, данных и алгоритмов (табл. 1.2). Так, например, многие службы централизуются потому, что при их реализации предполагалось наличие в распределенной системе только одного сервера, запущенного на конкретной машине. Проблемы такой схемы очевидны: при увеличении числа пользователей сервер легко может стать узким местом системы. Даже если мы обладаем фактически неограниченным запасом по мощности обработки и хранения данных, ресурсы связи с этим сервером в конце концов будут исчерпаны и не позволят нам расти дальше.

Таблица 1.2. Примеры ограничений масштабируемости

Концепция	Пример
Централизованные службы	Один сервер на всех пользователей
Централизованные данные	Единый телефонный справочник, доступный в режиме подключения
Централизованные алгоритмы	Организация маршрутизации на основе полной информации

К сожалению, использование единственного сервера время от времени неизбежно. Представьте себе службу управления особо конфиденциальной информацией, такой как истории болезни, банковские счета, кредиты и т. п. В подобных случаях необходимо реализовывать службы на одном сервере в отдельной хорошо защищенной комнате и отделять их от других частей распределенной системы посредством специальных сетевых устройств. Копирование информации, содержащейся на сервере, в другие места для повышения производительности даже не обсуждается, поскольку это делает службу менее стойкой к атакам злоумышленников.

Централизация данных так же вредна, как и централизация служб. Как вы будете отслеживать телефонные номера и адреса 50 миллионов человек? Пред-

положим, что каждая запись укладывается в 50 символов. Необходимой емкостью обладает один 2,5-гигабайтный диск. Но и в этом случае наличие единой базы данных несомненно вызовет перегрузку входящих и исходящих линий связи. Так, представим себе, как работал бы Интернет, если бы служба доменных имен (DNS) была бы реализована в виде одной таблицы. DNS обрабатывает информацию с миллионов компьютеров во всем мире и предоставляет службу, необходимую для определения местоположения web-серверов. Если бы каждый запрос на интерпретацию URL передавался бы на этот единственный DNS-сервер, воспользоваться Web не смог бы никто (кстати, предполагается, что эти проблемы придется решать снова).

И наконец, централизация алгоритмов — это тоже плохо. В больших распределенных системах гигантское число сообщений необходимо направлять по множеству каналов. Теоретически для вычисления оптимального пути необходимо получить полную информацию о загруженности всех машин и линий и по алгоритмам из теории графов вычислить все оптимальные маршруты. Эта информация затем должна быть раздана по системе для улучшения маршрутизации.

Проблема состоит в том, что сбор и транспортировка всей информации туда-сюда — не слишком хорошая идея, поскольку сообщения, несущие эту информацию, могут перегрузить часть сети. Фактически следует избегать любого алгоритма, который требует передачи информации, собираемой со всей сети, на одну из ее машин для обработки с последующей раздачей результатов. Использовать следует только децентрализованные алгоритмы. Эти алгоритмы обычно обладают следующими свойствами, отличающими их от централизованных алгоритмов:

- ◆ ни одна из машин не обладает полной информацией о состоянии системы;
- ◆ машины принимают решения на основе локальной информации;
- ◆ сбой на одной машине не вызывает нарушения алгоритма;
- ◆ не требуется предположения о существовании единого времени.

Первые три свойства поясняют то, о чем мы только что говорили. Последнее, вероятно, менее очевидно, но не менее важно. Любой алгоритм, начинающийся со слов: «Ровно в 12:00:00 все машины должны определить размер своих входных очередей», работать не будет, поскольку невозможно синхронизировать все часы на свете. Алгоритмы должны принимать во внимание отсутствие полной синхронизации таймеров. Чем больше система, тем большим будет и рассогласование. В одной локальной сети путем определенных усилий можно добиться, чтобы рассинхронизация всех часов не превышала нескольких миллисекунд, но сделать это в масштабе страны или множества стран? Вы, должно быть, шутите.

У географической масштабируемости имеются свои сложности. Одна из основных причин сложности масштабирования существующих распределенных систем, разработанных для локальных сетей, состоит в том, что в их основе лежит принцип *синхронной связи* (*synchronous communication*). В этом виде связи запрашивающий службу агент, которого принято называть *клиентом* (*client*), блокируется до получения ответа. Этот подход обычно успешно работает в локальных сетях, когда связь между двумя машинами продолжается максимум сотни микросекунд. Однако в глобальных системах мы должны принять во внимание

тот факт, что связь между процессами может продолжаться сотни миллисекунд, то есть на три порядка дольше. Построение интерактивных приложений с использованием синхронной связи в глобальных системах требует большой осторожности (и немалого терпения).

Другая проблема, препятствующая географическому масштабированию, состоит в том, что связь в глобальных сетях фактически всегда организуется от точки к точке и потому ненадежна. В противоположность глобальным, локальные сети обычно дают высоконадежную связь, основанную на широковещательной рассылке, что делает разработку распределенных систем для них значительно проще. Для примера рассмотрим проблему локализации службы. В локальной сети система просто рассылает сообщение всем машинам, опрашивая их на предмет предоставления нужной службы. Машины, предоставляющие службу, отвечают на это сообщение, указывая в ответном сообщении свои сетевые адреса. Невозможно представить себе подобную схему определения местоположения в глобальной сети. Вместо этого необходимо обеспечить специальные места для расположения служб, которые может потребоваться масштабировать на весь мир и обеспечить их мощностью для обслуживания миллионов пользователей. Мы вернемся к подобным службам в главе 4.

Географическая масштабируемость жестко завязана на проблемы централизованных решений, которые мешают масштабированию по размеру. Если у нас имеется система с множеством централизованных компонентов, то понятно, что географическая масштабируемость будет ограничиваться проблемами производительности и надежности, связанными с глобальной связью. Кроме того, централизованные компоненты в настоящее время легко способны вызвать перегрузку сети. Представьте себе, что в каждой стране существует всего одно почтовое отделение. Это будет означать, что для того, чтобы отправить письма родственникам, вам необходимо отправиться на центральный почтамт, расположенный, возможно, в сотнях миль от вашего дома. Ясно, что это не тот путь, которым следует идти.

И, наконец, нелегкий и во многих случаях открытый вопрос, как обеспечить масштабирование распределенной системы на множество административно независимых областей. Основная проблема, которую нужно при этом решить, состоит в конфликтах правил, относящихся к использованию ресурсов (и плате за них), управлению и безопасности.

Так, множество компонентов распределенных систем, находящихся в одной области, обычно может быть доверено пользователям, работающим в этой области. В этом случае системный администратор может тестировать и сертифицировать приложения, используя специальные инструменты для проверки того факта, что эти компоненты не могут ничего натворить. Проще говоря, пользователи доверяют своему системному администратору. Однако это доверие не распространяется естественным образом за границы области.

Если распределенные системы распространяются на другую область, могут потребоваться два типа проверок безопасности. Во-первых, распределенная система должна противостоять злонамеренным атакам из новой области. Так, например, пользователи новой области могут получить ограниченные права досту-

па к файловой службе системы в исходной области, скажем, только на чтение. Точно так же может быть закрыт доступ чужих пользователей и к аппаратуре, такой как дорогостоящие полноцветные устройства печати или высокопроизводительные компьютеры. Во-вторых, новая область сама должна быть защищена от злонамеренных атак из распределенной системы. Типичным примером является загрузка по сети программ, таких как апплеты в web-браузерах. Изначально новая область не знает, чего ожидать от чужого кода, и потому строго ограничивает ему права доступа. Проблема, как мы увидим в главе 8, состоит в том, как обеспечить соблюдение этих ограничений.

Технологии масштабирования

Обсуждение некоторых проблем масштабирования приводит нас к вопросу о том, а как же обычно решаются эти проблемы. Поскольку проблемы масштабируемости в распределенных системах, такие как проблемы производительности, вызываются ограниченной мощностью серверов и сетей, существуют три основные технологии масштабирования: сокрытие времени ожидания связи, распределение и репликация [314].

Сокрытие времени ожидания связи применяется в случае географического масштабирования. Основная идея проста: постараться по возможности избежать ожидания ответа на запрос от удаленного сервера. Например, если была запрошена служба удаленной машины, альтернативой ожиданию ответа от сервера будет осуществление на запрашивающей стороне других возможных действий. В сущности, это означает разработку запрашивающего приложения в расчете на использование исключительно *асинхронной связи* (*asynchronous communication*). Когда будет получен ответ, приложение прервет свою работу и вызовет специальный обработчик для завершения отправленного ранее запроса. Асинхронная связь часто используется в системах пакетной обработки и параллельных приложениях, в которых во время ожидания одной задачей завершения связи предполагается выполнение других более или менее независимых задач. Для осуществления запроса может быть запущен новый управляющий поток выполнения. Хотя он будет блокирован на время ожидания ответа, другие потоки процесса продолжат свое выполнение.

Однако многие приложения не в состоянии эффективно использовать асинхронную связь. Например, когда в интерактивном приложении пользователь посылает запрос, он обычно не в состоянии делать ничего более умного, чем просто ждать ответа. В этих случаях наилучшим решением будет сократить необходимый объем взаимодействия, например, переместив часть вычислений, обычно выполняемых на сервере, на клиента, процесс которого запрашивает службу. Стандартный случай применения этого подхода — доступ к базам данных с использованием форм. Обычно заполнение формы сопровождается посылкой отдельного сообщения на каждое поле и ожиданием подтверждения приема от сервера, как показано на рис. 1.2, а. Сервер, например, может перед приемом введенного значения проверить его на синтаксические ошибки. Более успешное решение состоит в том, чтобы перенести код для заполнения формы и, возможно, проверки введенных данных на клиента, чтобы он мог послать серверу целиком

заполненную форму (рис. 1.2, б). Такой подход — перенос кода на клиента — в настоящее время широко поддерживается в Web посредством Java-апплетов.

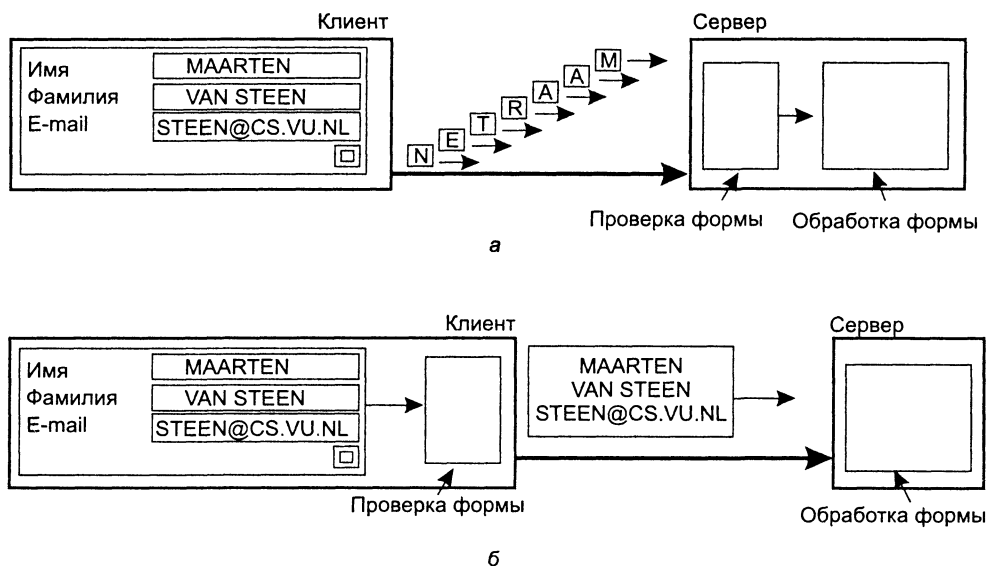


Рис. 1.2. Разница между проверкой формы по мере заполнения на сервере (а) и на клиенте (б)

Следующая важная технология масштабирования — *распределение* (*distribution*). Распределение предполагает разбиение компонентов на мелкие части и последующее разнесение этих частей по системе. Хорошим примером распределения является система доменных имен Интернета (DNS). Пространство DNS-имен организовано иерархически, в виде дерева *доменов* (*domains*), которые разбиты на неперекрывающиеся *зоны* (*zones*), как показано на рис. 1.3. Имена каждой зоны обрабатываются одним сервером имен. Не углубляясь чересчур в детали, можно считать, что каждое доменное имя является именем хоста в Интернете и ассоциируется с сетевым адресом этого хоста. В основном интерпретация имени означает получение сетевого адреса соответствующего хоста. Рассмотрим, к примеру, имя `nl.vu.cs.flits`. Для интерпретации этого имени оно сначала передается на сервер зоны *Z1* (рис. 1.3), который возвращает адрес сервера зоны *Z2*, который, вероятно, сможет обработать остаток имени, `vu.cs.flits`. Сервер зоны *Z2* вернет адрес сервера зоны *Z3*, который способен обработать последнюю часть имени и вернуть адрес соответствующего хоста.

Эти примеры демонстрируют, как *служба именования*, предоставляемая DNS, распределена по нескольким машинам и как это позволяет избежать обработки всех запросов на интерпретацию имен одним сервером.

В качестве другого примера рассмотрим World Wide Web. Для большинства пользователей Web представляется гигантской информационной системой документооборота, в которой каждый документ имеет свое уникальное имя — URL.

Концептуально можно предположить даже, что все документы размещаются на одном сервере. Однако среда Web физически разнесена по множеству серверов, каждый из которых содержит некоторое количество документов. Имя сервера, содержащего конкретный документ, определяется по URL-адресу документа. Только благодаря подобному распределению документов Всемирная паутина смогла вырасти до ее современных размеров.

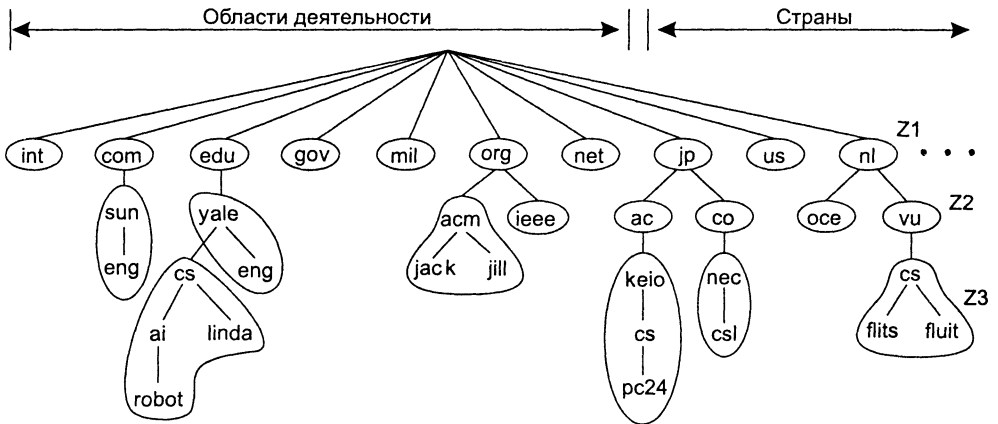


Рис. 1.3. Пример разделения пространства DNS-имен на зоны

При рассмотрении проблем масштабирования, часто проявляющихся в виде падения производительности, нередко хорошей идеей является *репликация* (*replication*) компонентов распределенной системы. Репликация не только повышает доступность, но и помогает выровнять загрузку компонентов, что ведет к повышению производительности. Кроме того, в сильно географически рассредоточенных системах наличие близко лежащей копии позволяет снизить остроту большей части ранее обсуждавшихся проблем ожидания завершения связи.

Кэширование (*caching*) представляет собой особую форму репликации, причем различия между ними нередко малозаметны или вообще искусственны. Как и в случае репликации, результатом кэширования является создание копии ресурса, обычно в непосредственной близости от клиента, использующего этот ресурс. Однако в противоположность репликации кэширование — это действие, предпринимаемое потребителем ресурса, а не его владельцем.

На масштабируемость может плохо повлиять один существенный недостаток кэширования и репликации. Поскольку мы получаем множество копий ресурса, модификация одной копии делает ее отличной от остальных. Соответственно, кэширование и репликация вызывают проблемы *непротиворечивости* (*consistency*).

Допустимая степень противоречивости зависит от степени загрузки ресурсов. Так, множество пользователей Web считают допустимым работу с кэшированным документом через несколько минут после его помещения в кэш без дополнительной проверки. Однако существует множество случаев, когда необходимо

гарантировать строгую непротиворечивость, например, при игре на электронной бирже. Проблема строгой непротиворечивости состоит в том, что изменение в одной из копий должно немедленно распространяться на все остальные. Кроме того, если два изменения происходят одновременно, часто бывает необходимо, чтобы эти изменения вносились в одном и том же порядке во все копии. Для обработки ситуаций такого типа обычно требуется механизм глобальной синхронизации. К сожалению, реализовать масштабирование подобных механизмов крайне трудно, а может быть и невозможно. Это означает, что масштабирование путем репликации может включать в себя отдельные немасштабируемые решения. Мы вернемся к вопросам репликации и непротиворечивости в главе 6.

1.3. Концепции аппаратных решений

Несмотря на то что все распределенные системы содержат по несколько процессоров, существуют различные способы их организации в систему. В особенности это относится к вариантам их соединения и организации взаимного обмена. В этом разделе мы кратко рассмотрим аппаратное обеспечение распределенных систем, в частности варианты соединения машин между собой. Предметом нашего обсуждения в следующем разделе будет программное обеспечение распределенных систем.

За прошедшие годы было предложено множество различных схем классификации компьютерных систем с несколькими процессорами, но ни одна из них не стала действительно популярной и широко распространенной. Нас интересуют исключительно системы, построенные из набора независимых компьютеров. На рис. 1.4 мы подразделяем все компьютеры на две группы. Системы, в которых компьютеры используют память совместно, обычно называются *мультипроцессорами* (*multiprocessors*), а работающие каждый со своей памятью — *мультикомпьютерами* (*multicomputers*). Основная разница между ними состоит в том, что мультипроцессоры имеют единое адресное пространство, совместно используемое всеми процессорами. Если один из процессоров записывает, например, значение 44 по адресу 1000, любой другой процессор, который после этого прочтет значение, лежащее по адресу 1000, получит 44. Все машины задействуют одну и ту же память.

В отличие от таких машин в мультикомпьютерах каждая машина использует свою собственную память. После того как один процессор запишет значение 44 по адресу 1000, другой процессор, прочитав значение, лежащее по адресу 1000, получит то значение, которое хранилось там раньше. Запись по этому адресу значения 44 другим процессором никак не скажется на содержимом его памяти. Типичный пример мультикомпьютера — несколько персональных компьютеров, объединенных в сеть.

Каждая из этих категорий может быть подразделена на дополнительные категории на основе архитектуры соединяющей их сети. На рис. 1.4 эти две архитектуры обозначены как *шинная* (*bus*) и *коммутруемая* (*switched*). Под шиной понимается одиночная сеть, плата, шина, кабель или другая среда, соединяющая

все машины между собой. Подобную схему использует кабельное телевидение: кабельная компания протягивает вдоль улицы кабель, а всем подписчикам делаются отводки от основного кабеля к их телевизорам.

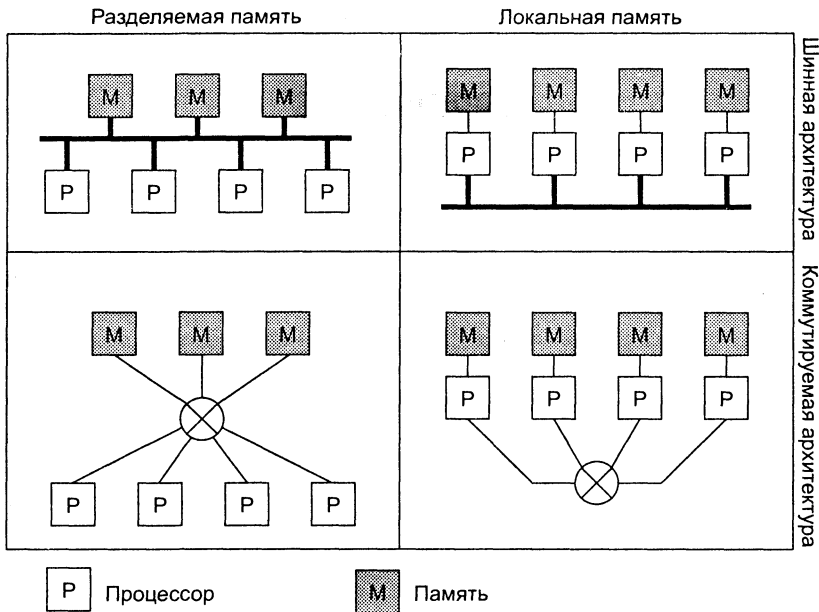


Рис. 1.4. Различные базовые архитектуры процессоров и памяти распределенных компьютерных систем

Коммутируемые системы, в отличие от шинных, не имеют единой магистрали, такой как у кабельного телевидения. Вместо нее от машины к машине тянутся отдельные каналы, выполненные с применением различных технологий связи. Сообщения передаются по каналам с принятием явного решения о коммутации с конкретным выходным каналом для каждого из них. Так организована глобальная телефонная сеть.

Мы проведем также разделение распределенных компьютерных систем на *гомогенные* (*homogeneous*) и *гетерогенные* (*heterogeneous*). Это разделение применяется исключительно к мультимашинным системам. Для гомогенных мультимашинных систем характерна одна соединяющая компьютеры сеть, использующая единую технологию. Одинаковы также и все процессоры, которые в основном имеют доступ к одинаковым объемам собственной памяти. Гомогенные мультимашинные системы нередко используются в качестве параллельных (работающих с одной задачей), в точности как мультипроцессорные.

В отличие от них гетерогенные мультимашинные системы могут содержать целую гамму независимых компьютеров, соединенных разнообразными сетями. Так, например, распределенная компьютерная система может быть построена из нескольких локальных компьютерных сетей, соединенных коммутируемой магистралью FDDI или ATM.

В следующих трех пунктах мы кратко рассмотрим мультипроцессорные, а также гомогенные и гетерогенные мультикомпьютерные системы. Несмотря на то, что эти вопросы не связаны напрямую с нашей основной темой, распределенными системами, они помогают лучше ее понять, поскольку организация распределенных систем часто зависит от входящей в их состав аппаратуры.

1.3.1. Мультипроцессоры

Мультипроцессорные системы обладают одной характерной особенностью: все процессоры имеют прямой доступ к общей памяти. Мультипроцессорные системы шинной архитектуры состоят из некоторого количества процессоров, подсоединенных к общей шине, а через нее — к модулям памяти. Простейшая конфигурация содержит плату с шиной или материнскую плату, в которую вставляются процессоры и модули памяти.

Поскольку используется единая память, когда процессор *A* записывает слово в память, а процессор *B* микросекундой позже считывает слово из памяти, процессор *B* получает информацию, записанную в память процессором *A*. Память, обладающая таким поведением, называется *согласованной (coherent)*. Проблема такой схемы состоит в том, что в случае уже 4 или 5 процессоров шина оказывается стабильно перегруженной и производительность резко падает. Решение состоит в размещении между процессором и шиной высокоскоростной *кэш-памяти (cache memory)*, как показано на рис. 1.5. В кэше сохраняются данные, обращение к которым происходит наиболее часто. Все запросы к памяти происходят через кэш. Если запрошенные данные находятся в кэш-памяти, то на запрос процессора реагирует она и обращения к шине не выполняются. Если размер кэш-памяти достаточно велик, вероятность успеха, называемая также *коэффициентом кэш-попаданий (hit rate)*, велика и шинный трафик в расчете на один процессор резко уменьшается, позволяя включить в систему значительно больше процессоров. Общепринятыми являются размеры кэша от 512 Кбайт до 1 Мбайт, коэффициент кэш-попаданий при этом обычно составляет 90 % и более.

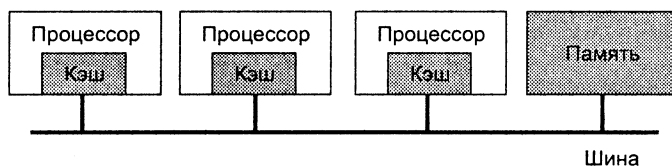


Рис. 1.5. Мультипроцессорная система с шинной архитектурой

Однако введение кэша создает серьезные проблемы само по себе. Предположим, что два процессора, *A* и *B*, читают одно и то же слово в свой внутренний кэш. Затем *A* перезаписывает это слово. Когда процессор *B* в следующий раз захочет воспользоваться этим словом, он считает старое значение из своего кэша, а не новое значение, записанное процессором *A*. Память стала несогласованной, и программирование системы осложнилось. Кэширование тем не менее активно

используется в распределенных системах, и здесь мы вновь сталкиваемся с проблемами несогласованной памяти. Мы вернемся к проблемам кэширования и согласованности в главе 6. Мультипроцессоры шинной архитектуры хорошо описаны в книге [265].

Проблема мультипроцессорных систем шинной архитектуры состоит в их ограниченной масштабируемости, даже в случае использования кэша. Для построения мультипроцессорной системы с более чем 256 процессорами для соединения процессоров с памятью необходимы другие методы. Один из вариантов — разделить общую память на модули и связать их с процессорами через *коммутирующую решетку (crossbar switch)*, как показано на рис. 1.6, а. Как видно из рисунка, с ее помощью каждый процессор может быть связан с любым модулем памяти. Каждое пересечение представляет собой маленький электронный *узловой коммутатор (crosspoint switch)*, который может открываться и закрываться аппаратно. Когда процессор желает получить доступ к конкретному модулю памяти, соединяющие их узловые коммутаторы мгновенно открываются, организуя запрошенный доступ. Достоинство узловых коммутаторов в том, что к памяти могут одновременно обращаться несколько процессоров, хотя если два процессора одновременно хотят получить доступ к одному и тому же участку памяти, то одному из них придется подождать.

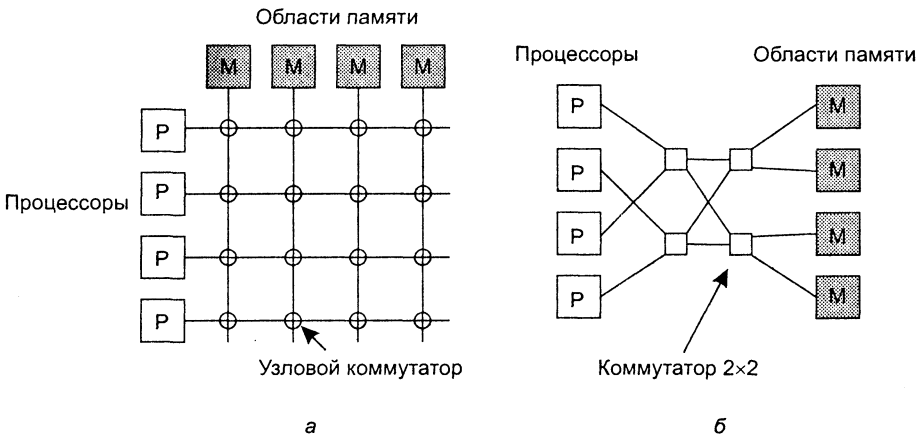


Рис. 1.6. Коммутирующая решетка (а). Коммутирующая омега-сеть (б)

Недостатком коммутирующей решетки является то, что при наличии n процессоров и n модулей памяти нам потребуется n^2 узловых коммутаторов. Для больших значений n это число может превысить наши возможности. Обнаружив это, человечество стало искать и нашло альтернативные коммутирующие сети, требующие меньшего количества коммутаторов. Один из примеров таких сетей — *омега-сеть (omega network)*, представленная на рис. 1.6, б. Эта сеть содержит четыре коммутатора 2×2 , то есть каждый из них имеет по два входа и два выхода. Каждый коммутатор может соединять любой вход с любым выходом. Если внимательно изучить возможные положения коммутаторов, становится яс-

но, что любой процессор может получить доступ к любому блоку памяти. Недостаток коммутирующих сетей состоит в том, что сигнал, идущий от процессора к памяти или обратно, вынужден проходить через несколько коммутаторов. Поэтому, чтобы снизить задержки между процессором и памятью, коммутаторы должны иметь очень высокое быстродействие, а дешево это не дается.

Люди пытаются уменьшить затраты на коммутацию путем перехода к иерархическим системам. В этом случае с каждым процессором ассоциируется некоторая область памяти. Каждый процессор может быстро получить доступ к своей области памяти. Доступ к другой области памяти происходит значительно медленнее. Эта идея была реализована в машине с *неунифицированным доступом к памяти* (*NonUniform Memory Access, NUMA*). Хотя машины NUMA имеют лучшее среднее время доступа к памяти, чем машины на базе омега-сетей, у них есть свои проблемы, связанные с тем, что размещение программ и данных необходимо производить так, чтобы большая часть обращений шла к локальной памяти.

1.3.2. Гомогенные мультикомпьютерные системы

В отличие от мультипроцессоров построить мультикомпьютерную систему относительно несложно. Каждый процессор напрямую связан со своей локальной памятью. Единственная оставшаяся проблема — это общение процессоров между собой. Понятно, что и тут необходима какая-то схема соединения, но поскольку нас интересует только связь между процессорами, объем трафика будет на несколько порядков ниже, чем при использовании сети для поддержания трафика между процессорами и памятью.

Сначала мы рассмотрим гомогенные мультикомпьютерные системы. В этих системах, известных под названием *системных сетей* (*System Area Networks, SAN*), узлы монтируются в большой стойке и соединяются единой, обычно высокоскоростной сетью. Как и в предыдущем случае, нам придется выбирать между системами на основе шинной архитектуры и системами на основе коммутации.

В мультикомпьютерных системах с шинной архитектурой процессоры соединяются при помощи разделяемой сети множественного доступа, например Fast Ethernet. Скорость передачи данных в сети обычно равна 100 Мбит/с. Как и в случае мультипроцессоров с шинной архитектурой, мультикомпьютерные системы с шинной архитектурой имеют ограниченную масштабируемость. В зависимости от того, сколько узлов в действительности нуждаются в обмене данными, обычно не следует ожидать высокой производительности при превышении системой предела в 25–100 узлов.

В коммутируемых мультикомпьютерных системах сообщения, передаваемые от процессора к процессору, *маршрутизируются* в соединительной сети в отличие от принятых в шинной архитектуре широковещательных рассылок. Было предложено и построено множество различных топологий. Две популярные топологии — квадратные решетки и гиперкубы — представлены на рис. 1.7. Решет-

ки просты для понимания и удобны для разработки на их основе печатных плат. Они прекрасно подходят для решения двухмерных задач, например задач теории графов или компьютерного зрения (глаза робота, анализ фотографий).

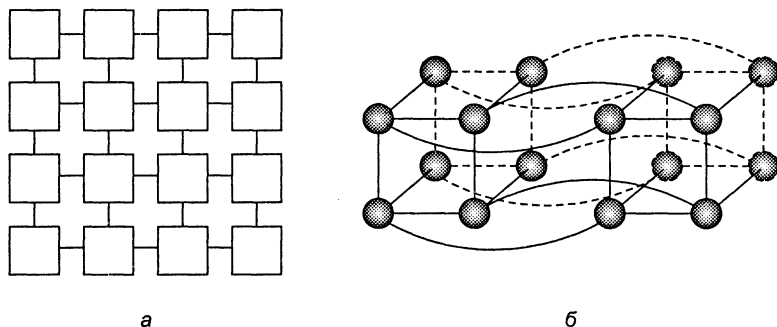


Рис. 1.7. Решетка (а). Гиперкуб (б)

Гиперкуб (hypercube) представляет собой куб размерности n . Гиперкуб, показанный на рис. 1.7, б, четырехмерен. Его можно представить в виде двух обычных кубов, с 8 вершинами и 12 ребрами каждый. Каждая вершина — это процессор. Каждое ребро — это связь между двумя процессорами. Соответствующие вершины обоих кубов соединены между собой. Для расширения гиперкуба в пятое измерение мы должны добавить к этой фигуре еще один комплект из двух связанных кубов, соединив соответствующие вершины двух половинок фигуры. Таким же образом можно создать шестимерный куб, семимерный и т. д.

Коммутируемые мультимикрокомпьютерные системы могут быть очень разнообразны. На одном конце спектра лежат *процессоры с массовым параллелизмом (Massively Parallel Processors, MPP)*, гигантские суперкомпьютеры стоимостью во много миллионов долларов, содержащие тысячи процессоров. Нередко они собираются из тех же процессоров, которые используются в рабочих станциях или персональных компьютерах. От других мультимикрокомпьютерных систем их отличает наличие патентованных высокоскоростных соединительных сетей. Эти сети проектируются в расчете на малое время задержки и высокую пропускную способность. Кроме того, предпринимаются специальные меры для защиты системы от сбоев. При наличии тысяч процессоров каждую неделю как минимум несколько будут выходить из строя. Нельзя допустить, чтобы поломка одного из них приводила к выводу из строя всей машины.

На другом конце спектра мы обнаруживаем популярный тип коммутируемых микрокомпьютеров, известных как *кластеры рабочих станций (Clusters Of Workstations, COW)*, основу которых составляют стандартные персональные компьютеры или рабочие станции, соединенные посредством коммерческих коммуникационных компонентов, таких как карты Myrinet [70]. Соединительные сети — вот то, что отличает COW от MPP. Кроме того, обычно не предпринимается никаких особых мер для повышения скорости ввода-вывода или защиты от сбоев в системе. Подобный подход делает COW проще и дешевле.

1.3.3. Гетерогенные мультимедийные системы

Наибольшее число существующих в настоящее время распределенных систем построено по схеме гетерогенных мультимедийных. Это означает, что компьютеры, являющиеся частями этой системы, могут быть крайне разнообразны, например, по типу процессора, размеру памяти и производительности каналов ввода-вывода. На практике роль некоторых из этих компьютеров могут исполнять высокопроизводительные параллельные системы, например мультипроцессорные или гомогенные мультимедийные.

Соединяющая их сеть также может быть сильно неоднородной. Так, например, авторы этой книги помогали разрабатывать самодельную распределенную компьютерную систему, названную DAS, состоящую из четырех кластеров мультимедийных систем, соединенных высокопроизводительными АТМ-коммутируемыми каналами. Фотографии этой системы и ссылки на исследования, проводимые на ней, можно найти по адресу <http://Www.cs.vu.nl-balldas.html>. Кластеры также были связаны между собой через стандартные Интернет-соединения. Каждый кластер содержал одинаковые процессоры (Pentium III) и соединяющую их сеть (Myrinet), но различался по числу процессоров (64–128).

Другим примером гетерогенности является создание крупных мультимедийных систем с использованием существующих сетей и каналов. Так, например, не является чем-то необычным существование кампусных университетских распределенных систем, состоящих из локальных сетей различных факультетов, соединенных между собой высокоскоростными каналами. В глобальных системах различные станции могут, в свою очередь, соединяться общедоступными сетями, например сетевыми службами, предлагаемыми коммерческими операторами связи, например SMDS или Frame relay.

В отличие от систем, обсуждавшихся в предыдущих пунктах, многие крупномасштабные гетерогенные мультимедийные системы нуждаются в глобальном подходе. Это означает, что приложение не может предполагать, что ему постоянно будет доступна определенная производительность или определенные службы. Так, в проекте I-way [147] несколько высокопроизводительных компьютерных центров были связаны через Интернет. Согласно общей модели системы предполагалось, что приложения будут резервировать и использовать ресурсы любого из центров, но полностью скрыть от приложений разницу между центрами оказалось невозможно.

Переходя к вопросам масштабирования, присущим гетерогенным системам, и учитывая необходимость глобального подхода, присущую большинству из них, заметим, что создание приложений для гетерогенных мультимедийных систем требует специализированного программного обеспечения. С этой проблемой распределенные системы справляются. Чтобы у разработчиков приложений не возникало необходимости волноваться об используемом аппаратном обеспечении, распределенные системы предоставляют программную оболочку, которая защищает приложения от того, что происходит на аппаратном уровне (то есть они обеспечивают прозрачность).

1.4. Концепции программных решений

Аппаратура важна для распределенных систем, однако от программного обеспечения значительно сильнее зависит, как такая система будет выглядеть на самом деле. Распределенные системы очень похожи на традиционные операционные системы. Прежде всего, они работают как *менеджеры ресурсов* (*resource managers*) существующего аппаратного обеспечения, которые помогают множеству пользователей и приложений совместно использовать такие ресурсы, как процессоры, память, периферийные устройства, сеть и данные всех видов. Во-вторых, что, вероятно, более важно, распределенная система скрывает сложность и гетерогенную природу аппаратного обеспечения, на базе которого она построена, представляя виртуальную машину для выполнения приложений.

Чтобы понять природу распределенной системы, рассмотрим сначала операционные системы с точки зрения распределенности. Операционные системы для распределенных компьютеров можно вчерне разделить на две категории — сильно связанные и слабо связанные системы. В сильно связанных системах операционная система в основном старается работать с одним, глобальным представлением ресурсов, которыми она управляет. Слабо связанные системы могут представляться несведущему человеку набором операционных систем, каждая из которых работает на собственном компьютере. Однако эти операционные системы функционируют совместно, делая собственные службы доступными другим.

Это деление на сильно и слабо связанные системы связано с классификацией аппаратного обеспечения, приведенной в предыдущем разделе. Сильно связанные операционные системы обычно называются *распределенными операционными системами* (*Distributed Operating System, DOS*) и используются для управления мультипроцессорными и гомогенными мультикомпьютерными системами. Как и у традиционных однопроцессорных операционных систем, основная цель распределенной операционной системы состоит в сокрытии тонкостей управления аппаратным обеспечением, которое одновременно используется множеством процессов.

Слабо связанные *сетевые операционные системы* (*Network Operating Systems, NOS*) используются для управления гетерогенными мультикомпьютерными системами. Хотя управление аппаратным обеспечением и является основной задачей сетевых операционных систем, они отличаются от традиционных. Это отличие вытекает из того факта, что локальные службы должны быть доступными для удаленных клиентов. В следующих пунктах мы рассмотрим в первом приближении те и другие.

Чтобы действительно составить распределенную систему, служб сетевой операционной системы недостаточно. Необходимо добавить к ним дополнительные компоненты, чтобы организовать лучшую поддержку прозрачности распределения. Этими дополнительными компонентами будут средства, известные как *системы промежуточного уровня* (*middleware*), которые и лежат в основе современных распределенных систем. Средства промежуточного уровня также обсуждаются в этой главе. В табл. 1.3 представлены основные данные по распределенным и сетевым операционным системам, а также средствам промежуточного уровня.

Таблица 1.3. Краткое описание распределенных и сетевых операционных систем, а также средств промежуточного уровня

Система	Описание	Основное назначение
Распределенные операционные системы	Сильно связанные операционные системы для мультипроцессоров и гомогенных мультикомпьютерных систем	Соккрытие и управление аппаратным обеспечением
Сетевые операционные системы	Слабо связанные операционные системы для гетерогенных мультикомпьютерных систем (локальных или глобальных сетей)	Предоставление локальных служб удаленным клиентам
Средства промежуточного уровня	Дополнительный уровень поверх сетевых операционных систем, реализующий службы общего назначения	Обеспечение прозрачности распределения

1.4.1. Распределенные операционные системы

Существует два типа распределенных операционных систем. *Мультипроцессорная операционная система (multiprocessor operating system)* управляет ресурсами мультипроцессора. *Мультикомпьютерная операционная система (multicomputer operating system)* разрабатывается для гомогенных мультикомпьютеров. Функциональность распределенных операционных систем в основном не отличается от функциональности традиционных операционных систем, предназначенных для компьютеров с одним процессором за исключением того, что она поддерживает функционирование нескольких процессоров. Поэтому давайте кратко обсудим операционные системы, предназначенные для обыкновенных компьютеров с одним процессором. Введение в операционные системы для одного и нескольких процессоров можно отыскать в [447].

Операционные системы для однопроцессорных компьютеров

Операционные системы традиционно строились для управления компьютерами с одним процессором. Основной задачей этих систем была организация легкого доступа пользователей и приложений к разделяемым устройствам, таким как процессор, память, диски и периферийные устройства. Говоря о разделении ресурсов, мы имеем в виду возможность использования одного и того же аппаратного обеспечения различными приложениями изолированно друг от друга. Для приложения это выглядит так, словно эти ресурсы находятся в его полном распоряжении, при этом в одной системе может выполняться одновременно несколько приложений, каждое со своим собственным набором ресурсов. В этом смысле говорят, что операционная система реализует *виртуальную машину (virtual machine)*, предоставляя приложениям средства мультизадачности.

Важным аспектом совместного использования ресурсов в такой виртуальной машине является то, что приложения отделены друг от друга. Так, невозможна ситуация, когда при одновременном исполнении двух приложений, *A* и *B*, приложение *A* может изменить данные приложения *B*, просто работая с той частью общей памяти, где эти данные хранятся. Также требуется гарантировать, что

приложения смогут использовать предоставленные им средства только так, как предписано операционной системой. Например, приложениям обычно запрещено копировать сообщения прямо в сетевой интерфейс. Взамен операционная система предоставляет первичные операции связи, которые можно использовать для пересылки сообщений между приложениями на различных машинах.

Следовательно, операционная система должна полностью контролировать использование и распределение аппаратных ресурсов. Поэтому большинство процессоров поддерживают как минимум два режима работы. В *режиме ядра (kernel mode)* выполняются все разрешенные инструкции, а в ходе выполнения доступна вся имеющаяся память и любые регистры. Напротив, в *пользовательском режиме (user mode)* доступ к регистрам и памяти ограничен. Так, приложению не будет позволено работать с памятью за пределами набора адресов, установленного для него операционной системой, или обращаться напрямую к регистрам устройств. На время выполнения кода операционной системы процессор переключается в режим ядра. Однако единственный способ перейти из пользовательского режима в режим ядра — это сделать системный вызов, реализуемый через операционную систему. Поскольку системные вызовы — это лишь базовые службы, предоставляемые операционной системой, и поскольку ограничение доступа к памяти и регистрам нередко реализуется аппаратно, операционная система в состоянии полностью их контролировать.

Существование двух режимов работы привело к такой организации операционных систем, при которой практически весь их код выполняется в режиме ядра. Результатом часто становятся гигантские монолитные программы, работающие в едином адресном пространстве. Обратная сторона такого подхода состоит в том, что перенастроить систему часто бывает нелегко. Другими словами, заменить или адаптировать компоненты операционной системы без полной перезагрузки, а возможно и полной перекомпиляции и новой установки очень трудно. С точки зрения открытости, проектирования программ, надежности или легкости обслуживания монолитные операционные системы — это не самая лучшая из идей.

Более удобен вариант с организацией операционной системы в виде двух частей. Одна часть содержит набор модулей для управления аппаратным обеспечением, которые прекрасно могут выполняться в пользовательском режиме. Например, управление памятью состоит в основном из отслеживания, какие блоки памяти выделены под процессы, а какие свободны. Единственный момент, когда мы нуждаемся в работе в режиме ядра, — это установка регистров блока управления памятью.

Вторая часть операционной системы содержит небольшое *микроядро (microkernel)*, содержащее исключительно код, который выполняется в режиме ядра. На практике микроядро должно содержать только код для установки регистров устройств, переключения процессора с процесса на процесс, работы с блоком управления памятью и перехвата аппаратных прерываний. Кроме того, в нем обычно содержится код, преобразующий вызовы соответствующих модулей пользовательского уровня операционной системы в системные вызовы и возвращающий результаты. Такой подход приводит к организации, показанной на рис. 1.8.

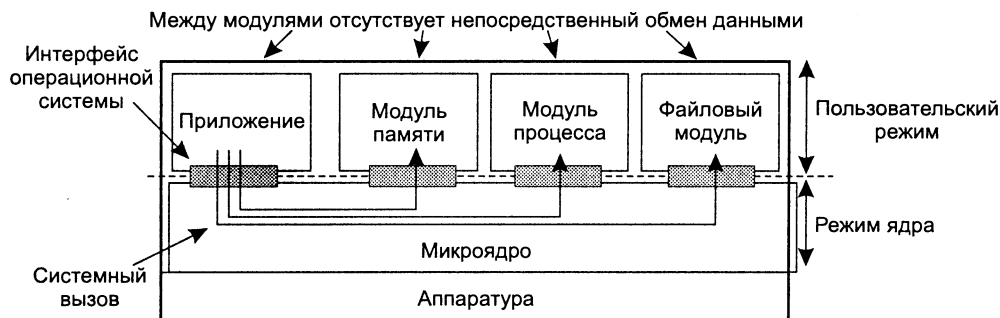


Рис. 1.8. Разделение приложений в операционной системе посредством микроядра

Использование микроядра дает нам разнообразные преимущества. Наиболее важное из них состоит в гибкости: поскольку большая часть операционной системы выполняется в пользовательском режиме, относительно несложно заменить один из модулей без повторной компиляции или повторной установки всей системы. Другой серьезный плюс заключается в том, что модули пользовательского уровня могут в принципе размещаться на разных машинах. Так, мы можем установить модуль управления файлами не на той машине, на которой он управляет службой каталогов. Другими словами, подход с использованием микроядра отлично подходит для переноса однопроцессорных операционных систем на распределенные компьютеры.

У микроядер имеется два существенных недостатка. Во-первых, они работают иначе, чем существующие операционные системы, а попытки поменять сложившееся «статус-кво» всегда встречают активное сопротивление («если эта операционная система подходила для моего деда — она подойдет и для меня»). Во-вторых, микроядро требует дополнительного обмена, что слегка снижает производительность. Однако, зная как быстры современные процессоры, снижение производительности в 20 % вряд ли можно считать фатальным.

Мультипроцессорные операционные системы

Важным, но часто не слишком очевидным расширением однопроцессорных операционных систем является возможность поддержки нескольких процессоров, имеющих доступ к совместно используемой памяти. Концептуально это расширение несложно. Все структуры данных, необходимые операционной системе для поддержки аппаратуры, включая поддержку нескольких процессоров, размещаются в памяти. Основная разница заключается в том, что теперь эти данные доступны нескольким процессорам и должны быть защищены от параллельного доступа для обеспечения их целостности.

Однако многие операционные системы, особенно предназначенные для персональных компьютеров и рабочих станций, не могут с легкостью поддерживать несколько процессоров. Основная причина такого поведения состоит в том, что они были разработаны как монолитные программы, которые могут выполняться только в одном потоке управления. Адаптация таких операционных систем под

мультипроцессорные означает повторное проектирование и новую реализацию всего ядра. Современные операционные системы изначально разрабатываются с учетом возможности работы в мультипроцессорных системах.

Многопроцессорные операционные системы нацелены на поддержание высокой производительности конфигураций с несколькими процессорами. Основная их задача — обеспечить прозрачность числа процессоров для приложения. Сделать это достаточно легко, поскольку сообщение между различными приложениями или их частями требует тех же примитивов, что и в многозадачных однопроцессорных операционных системах. Идея состоит в том, что все сообщение происходит путем работы с данными в специальной совместно используемой области данных, и все что нам нужно — это защитить данные от одновременного доступа к ним. Защита осуществляется посредством примитивов синхронизации. Два наиболее важных (и эквивалентных) примитива — это семафоры и мониторы.

Семафор (semaphore) может быть представлен в виде целого числа, поддерживающего две операции: *up* (увеличить) и *down* (уменьшить). При уменьшении сначала проверяется, превышает ли значение семафора 0. Если это так, его значение уменьшается и выполнение процесса продолжается. Если же значение семафора нулевое, вызывающий процесс блокируется. Оператор увеличения совершает противоположное действие. Сначала он проверяет все заблокированные в настоящее время процессы, которые были неспособны завершиться в ходе предыдущей операции уменьшения. Если таковые существуют, он разблокирует один из них и продолжает работу. В противном случае он просто увеличивает счетчик семафора. Разблокированный процесс выполняется до вызова операции уменьшения. Важным свойством операций с семафорами является то, что они *атомарны (atomic)*, то есть в случае запуска операции уменьшения или увеличения до момента ее завершения (или до момента блокировки процесса) никакой другой процесс не может получить доступ к семафору.

Известно, что программирование с использованием семафоров для синхронизации процесса вызывает множество ошибок, кроме, разве что, случаев простой защиты разделяемых данных. Основная проблема состоит в том, что наличие семафоров приводит к неструктурированному коду. Похожая ситуация возникает при частом использовании печально известной инструкции *goto*. В качестве альтернативы семафорам многие современные системы, поддерживающие параллельное программирование, предоставляют библиотеки для реализации мониторов.

Формально *монитор (monitor)* представляет собой конструкцию языка программирования, такую же, как объект в объектно-ориентированном программировании [200]. Монитор может рассматриваться как модуль, содержащий переменные и процедуры. Доступ к переменным можно получить только путем вызова одной из процедур монитора. В этом смысле монитор очень похож на объект. Объект также имеет свои защищенные данные, доступ к которым можно получить только через методы, реализованные в этом объекте. Разница между мониторами и объектами состоит в том, что монитор разрешает выполнение процедуры только одному процессу в каждый момент времени. Другими словами,

если процедура, содержащаяся в мониторе, выполняется процессом *A* (мы говорим, что *A* вошел в монитор) и процесс *B* также вызывает одну из процедур монитора, *B* будет блокирован до завершения выполнения *A* (то есть до тех пор, пока *A* не покинет монитор).

В качестве примера рассмотрим простой монитор для защиты целой переменной (листинг 1.1). Монитор содержит одну закрытую (*private*) переменную *count*, доступ к которой можно получить только через три открытых (*public*) процедуры — чтения текущего значения, увеличения на единицу и уменьшения. Конструкция монитора гарантирует, что любой процесс, который вызывает одну из этих процедур, получит атомарный доступ к внутренним данным монитора.

Листинг 1.1. Монитор, предохраняющий целое число от параллельного доступа

```
monitor Counter {
private:
    int count = 0;
public:
    int value() { return count; }
    void incr() { count = count + 1;}
    void decr() { count = count - 1;}
}
```

Итак, мониторы пригодны для простой защиты совместно используемых данных. Однако для условной блокировки процесса необходимо большее. Например, предположим, нам нужно заблокировать процесс при вызове операции уменьшения, если обнаруживается, что значение *count* равно нулю. Для этой цели в мониторах используются *условные переменные* (*condition variables*). Это специальные переменные с двумя доступными операциями: *wait* (ждать) и *signal* (сигнализировать). Когда процесс *A* находится в мониторе и вызывает для условной переменной, хранящейся в мониторе, операцию *wait*, процесс *A* будет заблокирован и откажется от своего исключительного доступа к монитору. Соответственно, процесс *B*, ожидавший получения исключительного доступа к монитору, сможет продолжить свою работу. В определенный момент времени *B* может разблокировать процесс *A* вызовом операции *signal* для условной переменной, которого ожидает *A*. Чтобы предотвратить наличие двух активных процессов внутри монитора, мы доработаем схему так, чтобы процесс, подавший сигнал, покидал монитор. Теперь мы можем переделать наш предыдущий пример. Монитор из листинга 1.2 — это новая реализация обсуждавшегося ранее семафора.

Листинг 1.2. Монитор, предохраняющий целое число от параллельного доступа и блокирующий процесс

```
monitor Counter {
private:
    int count = 0;
    int blocked_procs = 0;
    condition unblocked;
public:
    int value() { return count;}
```

```
void incr() {
    if (blocked_procs == 0)
        count = count + 1;
    else
        signal( unblocked );
}

void decr() {
    if (count == 0) {
        blocked_procs = blocked_procs + 1;
        wait( unblocked );
        blocked_procs = blocked_procs - 1;
    }
    else
        count = count - 1;
}
```

Оборотная сторона мониторов состоит в том, что они являются конструкцией языка программирования. Так, Java поддерживает мониторы, просто разрешая каждому объекту предохранять себя от параллельного доступа путем использования в нем инструкции `synchronized` и операций `wait` и `notify`. Библиотечная поддержка мониторов обычно реализуется на базе простых семафоров, которые могут принимать только значения 0 и 1. Такие семафоры часто называются *переменными-мьютексами* (*mutex variables*), или просто *мьютексами*. С мьютексами ассоциируются операции `lock` (блокировать) и `unlock` (разблокировать). Захват мьютекса возможен только в том случае, если его значение равно единице, в противном случае вызывающий процесс будет блокирован. Соответственно, освобождение мьютекса означает установку его значения в 1, если нет необходимости разблокировать какой-нибудь из ожидающих процессов. Условные переменные и соответствующие им операции также поставляются в виде библиотечных процедур. Дополнительную информацию по примитивам синхронизации можно найти в [17].

Мультикомпьютерные операционные системы

Мультикомпьютерные операционные системы обладают гораздо более разнообразной структурой и значительно сложнее, чем мультипроцессорные. Эта разница проистекает из того факта, что структуры данных, необходимые для управления системными ресурсами, не должны больше отвечать условию легкости совместного использования, поскольку их не нужно помещать в физически общую память. Единственным возможным видом связи является *передача сообщений* (*message passing*). Мультикомпьютерные операционные системы в основном организованы так, как показано на рис. 1.9.

Каждый узел имеет свое ядро, которое содержит модули для управления локальными ресурсами — памятью, локальным процессором, локальными дисками и т. д. Кроме того, каждый узел имеет отдельный модуль для межпроцессорного взаимодействия, то есть посылки сообщений на другие узлы и приема сообщений от них.

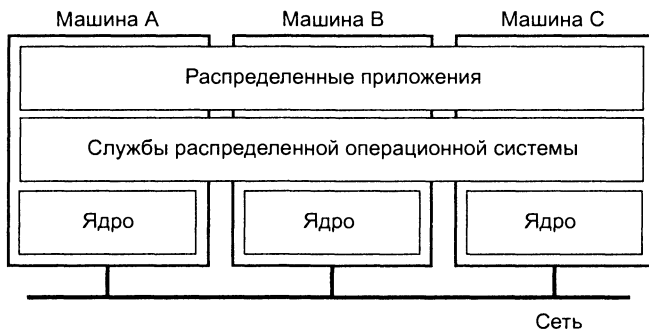


Рис. 1.9. Общая структура мультимашинных операционных систем

Поверх каждого локального ядра лежит уровень программного обеспечения общего назначения, реализующий операционную систему в виде виртуальной машины, поддерживающей параллельную работу над различными задачами. На деле, как мы сейчас кратко рассмотрим, этот уровень может даже предоставлять абстракцию мультипроцессорной машины. Другими словами, он предоставляет полную программную реализацию совместно используемой памяти. Дополнительные средства, обычно реализуемые на этом уровне, предназначены, например, для назначения задач процессорам, маскировки сбоев аппаратуры, обеспечения прозрачности сохранения и общего обмена между процессами. Другими словами, эти средства абсолютно типичны для операционных систем вообще.

Мультимашинные операционные системы, не предоставляющие средств для совместного использования памяти, могут предложить приложениям только средства для обмена сообщениями. К сожалению, семантика примитивов обмена сообщениями в значительной степени разная для разных систем. Понять эти различия проще, если отмечать, буферизуются сообщения или нет. Кроме того, мы нуждаемся в учете того, блокируется ли посылающий или принимающий процесс. На рис. 1.10 продемонстрирован вариант с буферизацией и блокировкой.

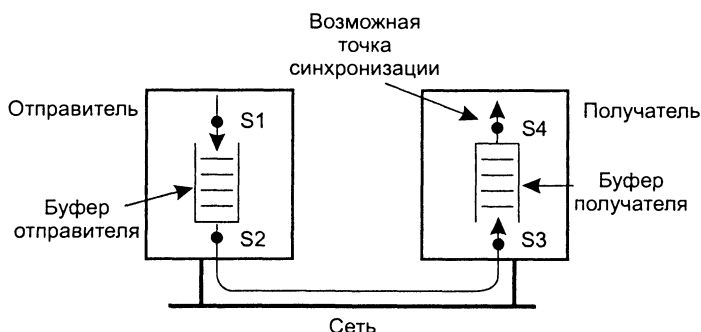


Рис. 1.10. Возможности блокировки и буферизации при пересылке сообщений

Существует всего два возможных места буферизации сообщений — на стороне отправителя или на стороне получателя. Это приводит к четырем возможным точкам синхронизации, то есть точкам возможной блокировки отправителя или получателя. Если буферизация происходит на стороне отправителя, это дает возможность заблокировать отправителя, только если его буфер полон, что показано точкой синхронизации $S1$ на рисунке. С другой стороны, процедура помещения сообщения в буфер может возвращать состояние, показывающее, что операция успешно выполнена. Это позволяет отправителю избежать блокировки по причине переполнения буфера. Если же отправитель не имеет буфера, существует три альтернативных точки блокировки отправителя: отправление сообщения (точка $S2$), поступление сообщения к получателю (точка $S3$), принятие сообщения получателем (точка $S4$). Отметим, что если блокировка происходит в точке $S2$, $S3$ или $S4$, наличие или отсутствие буфера на стороне отправителя не имеет никакого значения.

Блокировка получателя имеет смысл только в точке синхронизации $S3$ и может производиться, только если у получателя нет буфера или если буфер пуст. Альтернативой может быть опрос получателем наличия входящих сообщений. Однако эти действия часто ведут к пустой трате процессорного времени или слишком запоздалой реакции на пришедшее сообщение, что, в свою очередь, приводит к переполнению буфера входящими сообщениями и их потере [52].

Другой момент, важный для понимания семантики обмена сообщениями, — надежность связи. Отличительной чертой надежной связи является получение отправителем гарантии приема сообщения. На рис. 1.10 надежность связи означает, что все сообщения гарантированно достигают точки синхронизации $S3$. При ненадежной связи всякие гарантии отсутствуют. Если буферизация производится на стороне отправителя, о надежности связи ничего определенного сказать нельзя. Также операционная система не нуждается в гарантированно надежной связи в случае блокировки отправителя в точке $S2$.

С другой стороны, если операционная система блокирует отправителя до достижения сообщением точки $S3$ или $S4$, она должна иметь гарантированно надежную связь. В противном случае мы можем оказаться в ситуации, когда отправитель ждет подтверждения получения, а сообщение было потеряно при передаче. Отношение между блокировкой, буферизацией и гарантиями относительно надежности связи суммированы в табл. 1.4.

Таблица 1.4. Соотношение между блокировкой, буферизацией и надежностью связи

Точка синхронизации	Буферизация отправителя	Гарантия надежной связи
Блокировка отправителя до наличия свободного места в буфере	Да	Нет необходимости
Блокировка отправителя до отправки сообщения	Нет	Нет необходимости
Блокировка отправителя до приема сообщения	Нет	Необходима
Блокировка отправителя до обработки сообщения	Нет	Необходима

Множество аспектов проектирования мультимикомпьютерных операционных систем одинаково важны для любой распределенной системы. Основная разница между мультимикомпьютерными операционными системами и распределенными системами состоит в том, что в первом случае обычно подразумевается, что аппаратное обеспечение гомогенно и полностью управляемо. Множество распределенных систем, однако, строится на базе существующих операционных систем. Далее мы обсудим этот вопрос.

Системы с распределенной разделяемой памятью

Практика показывает, что программировать мультимикомпьютерные системы значительно сложнее, чем мультипроцессорные. Разница объясняется тем, что связь посредством процессов, имеющих доступ к совместно используемой памяти, и простых примитивов синхронизации, таких как семафоры и мониторы, значительно проще, чем работа с одним только механизмом обмена сообщениями. Такие вопросы, как буферизация, блокировка и надежность связи, только усложняют положение.

По этой причине проводились впечатляющие исследования по вопросу эмуляции совместно используемой памяти на мультимикомпьютерных системах. Их целью было создание виртуальных машин с разделяемой памятью, работающих на мультимикомпьютерных системах, для которых можно было бы писать приложения, рассчитанные на модель совместно используемой памяти, даже если физически она отсутствует. Главную роль в этом играет мультимикомпьютерная операционная система.

Один из распространенных подходов — задействовать виртуальную память каждого отдельного узла для поддержки общего виртуального адресного пространства. Это приводит нас к *распределенной разделяемой памяти (Distributed Shared Memory, DSM)* со страничной организацией. Принцип работы этой памяти следующий. В системе с DSM адресное пространство разделено на страницы (обычно по 4 или по 8 Кбайт), распределенные по всем процессорам системы. Когда процессор адресуется к памяти, которая не является локальной, происходит внутреннее прерывание, операционная система считывает в локальную память страницу, содержащую указанный адрес, и перезапускает выполнение вызвавшей прерывание инструкции, которая теперь успешно выполняется. Этот подход продемонстрирован на рис. 1.11, а для адресного пространства из 16 страниц и четырех процессоров. Это вполне нормальная страничная организация, если не считать того, что в качестве временного хранилища информации используется не диск, а удаленная оперативная память.

В этом примере при обращении процессора 1 к коду или данным со страницы 0, 2, 5 или 9 обращение происходит локально. Ссылки на другие страницы вызывают внутреннее прерывание. Так, например, ссылка на адрес со страницы 10 вызывает внутреннее прерывание операционной системы, и она перемещает страницу 10 с машины 2 на машину 1, как показано на рис. 1.11, б.

Одно из улучшений базовой системы, часто позволяющее значительно повысить ее производительность, — это репликация страниц, которые объявляются

закрытыми на запись, например, страниц, содержащих текст программы, константы «только для чтения» или другие закрытые на запись структуры. Например, если страница 10 — это секция текста программы, ее использование процессором 1 приведет к пересылке процессору 1 ее копии, а оригинал в памяти процессора 2 будет продолжать спокойно храниться, как показано на рис. 1.11, в. В этом случае процессоры 1 и 2 оба смогут обращаться к странице 10 так часто, как им понадобится, не вызывая при этом никаких внутренних прерываний для выборки памяти.

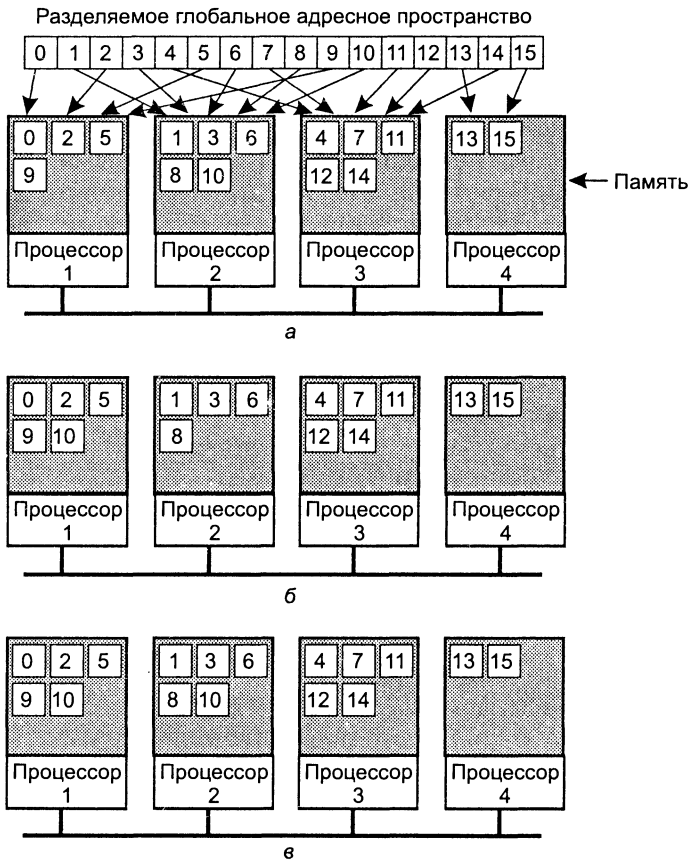


Рис. 1.11. Страницы адресного пространства распределены по четырем машинам (а). Ситуация после обращения процессора 1 к странице 10 (б). Ситуация, когда запись в страницу 10 невозможна и необходима репликация (в)

Другая возможность — это репликация также и не закрытых на запись страниц, то есть любых страниц. Пока производится только чтение, никакой разницы между репликацией закрытых и незакрытых на запись страниц нет. Однако если реплицированная страница внезапно изменяется, необходимо предпринимать специальные действия для предотвращения появления множества несовмес-

тимых копий. Обычно все копии, кроме одной, перед проведением записи объявляются неверными.

Дополнительного увеличения производительности можно добиться путем ухода от строгого соответствия между реплицируемыми страницами. Другими словами, мы позволяем отдельной копии временно отличаться от других. Практика показывает, что этот подход действительно может помочь, но, к сожалению, может также сильно осложнить жизнь программиста, вынужденного в этом случае отслеживать возможную несовместимость. Поскольку основной причиной разработки DSM была простота программирования, ослабление соответствия не находит реального применения. Мы вернемся к проблемам соответствия в главе 6.

Другой проблемой при разработке эффективных систем DSM является вопрос о размере страниц. Мы сталкивались с подобным выбором, когда рассматривали вопрос размера страниц в однопроцессорной системе с виртуальной памятью. Так, затраты на передачу страницы по сети в первую очередь определяются затратами на подготовку к передаче, а не объемом передаваемых данных. Соответственно, большой размер страниц может снизить общее число сеансов передачи при необходимости доступа к большому количеству последовательных элементов данных. С другой стороны, если страница содержит данные двух независимых процессов, выполняющихся на разных процессорах, операционная система будет вынуждена постоянно пересылать эту страницу от одного процессора к другому, как показано на рис. 1.12. Размещение данных двух независимых процессов на одной странице называется *ошибочным разделением* (*false sharing*).

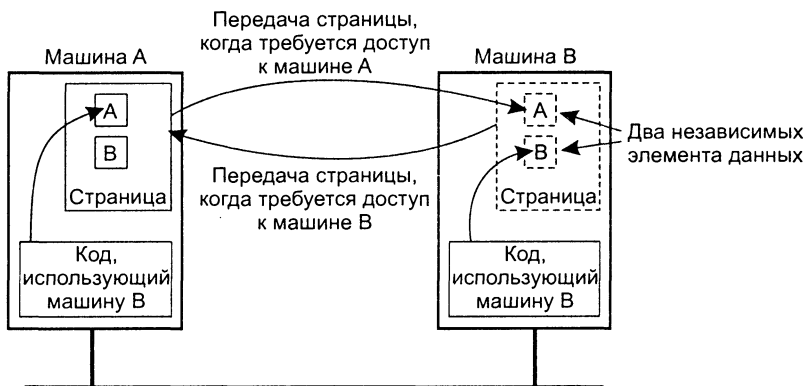


Рис. 1.12. Ошибочное разделение страницы двумя независимыми процессами

После почти 15 лет исследований распределенной памяти совместного использования разработчики DSM продолжают добиваться сочетания эффективности и легкости программирования. Для достижения высокой производительности крупномасштабных мультимашинных систем программисты прибегают к пересылке сообщений, невзирая на ее высокую, по сравнению с программированием систем (виртуальной) памяти совместного использования, сложность. Это позволяет нам сделать вывод о том, что DSM не оправдывает наших ожиданий

для высокопроизводительного параллельного программирования. Дополнительную информацию по DSM можно найти в книге [363].

1.4.2. Сетевые операционные системы

В противоположность распределенным операционным системам сетевые операционные системы не нуждаются в том, чтобы аппаратное обеспечение, на котором они функционируют, было гомогенно и управлялось как единая система. Напротив, обычно они строятся для набора однопроцессорных систем, каждая из которых имеет собственную операционную систему, как показано на рис. 1.13. Машины и их операционные системы могут быть разными, но все они соединены в сеть. Кроме того, сетевая операционная система позволяет пользователям использовать службы, расположенные на конкретной машине. Возможно, будет проще описать сетевую операционную систему, кратко рассмотрев службы, которые она обычно предоставляет.

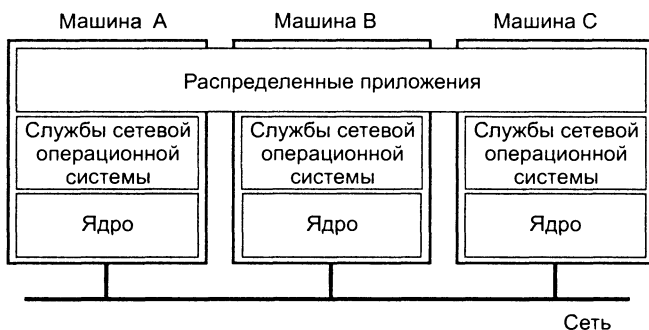


Рис. 1.13. Общая структура сетевой операционной системы

Служба, обычно предоставляемая сетевыми операционными системами, должна обеспечивать удаленное соединение пользователя с другой машиной путем применения команды типа:

```
rlogin machine
```

В результате выполнения этой команды происходит переключение рабочей станции пользователя в режим удаленного терминала, подключенного к удаленной машине. Это означает, что пользователь сидит у графической рабочей станции, набирая команды на клавиатуре. Команды передаются на удаленную машину, а результаты с удаленной машины отображаются в окне на экране пользователя. Для того чтобы переключиться на другую удаленную машину, необходимо открыть новое окно и воспользоваться командой `rlogin` для соединения с другой машиной. Выбор удаленной машины производится вручную.

Сетевые операционные системы также имеют в своем составе команду удаленного копирования для копирования файлов с одной машины на другую. Например:

```
rcp machine1:file1 machine2:file2
```

Эта команда приведет к копированию файла `file1` с машины `machine1` на `machine2` и присвоению ему там имени `file2`. При этом перемещение файлов задается в явном виде, и пользователю необходимо точно знать, где находятся файлы и как выполняются команды.

Такая форма связи хотя и лучше чем ничего, но все же крайне примитивна. Это подвигло проектировщиков систем на поиски более удобных вариантов связи и совместного использования информации. Один из подходов предполагает создание глобальной общей файловой системы, доступной со всех рабочих станций. Файловая система поддерживается одной или несколькими машинами, которые называются *файловыми серверами* (*file servers*). Файловые серверы принимают запросы от программ пользователей, запускаемых на других машинах (не на серверах), которые называются *клиентами* (*clients*), на чтение и запись файлов. Каждый пришедший запрос проверяется и выполняется, а результат пересылается назад, как показано на рис. 1.14.

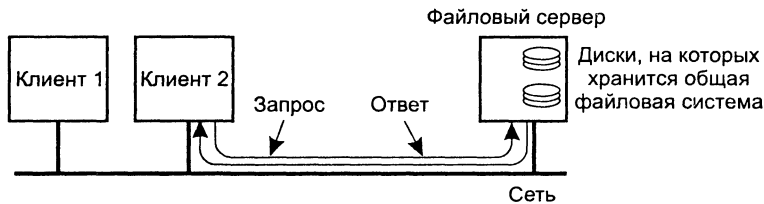


Рис. 1.14. Два клиента и сервер в сетевой операционной системе

Файловые серверы обычно поддерживают иерархические файловые системы, каждая с корневым каталогом, содержащим вложенные каталоги и файлы. Рабочие станции могут импортировать или монтировать эти файловые системы, увеличивая свою локальную файловую систему за счет файловой системы сервера. Так, например, на рис. 1.15 показаны два файловых сервера. На одном из них имеется каталог под названием **games**, а на другом — каталог под названием **work** (имена каталогов на рисунке выделены жирным шрифтом). Каждый из этих каталогов содержит некоторые файлы. На обоих клиентах смонтированы файловые системы обоих серверов, но в разных местах файловых систем клиентов. Клиент 1 смонтировал их в свой корневой каталог и имеет к ним доступ по путям `/games` и `/work` соответственно. Клиент 2, подобно Клиенту 1, смонтировал каталог **work** в свой корневой каталог, но решил, что игры (**games**) должны быть его частным делом. Поэтому он создал каталог, который назвал `/private`, и смонтировал каталог **games** туда. Соответственно, он получит доступ к файлу `racwoman` через путь `/private/games/racwoman`, а не `/games/racwoman`.

Хотя обычно не имеет значения, в какое место своей иерархии каталогов клиент смонтировал сервер, важно помнить, что различные клиенты могут иметь различное представление файловой системы. Имя файла зависит от того, как организуется доступ к нему и как выглядит файловая система на самой машине. Поскольку каждая клиентская машина работает относительно независимо от других, невозможно дать какие-то гарантии, что они обладают одинаковой иерархией каталогов для своих программ.

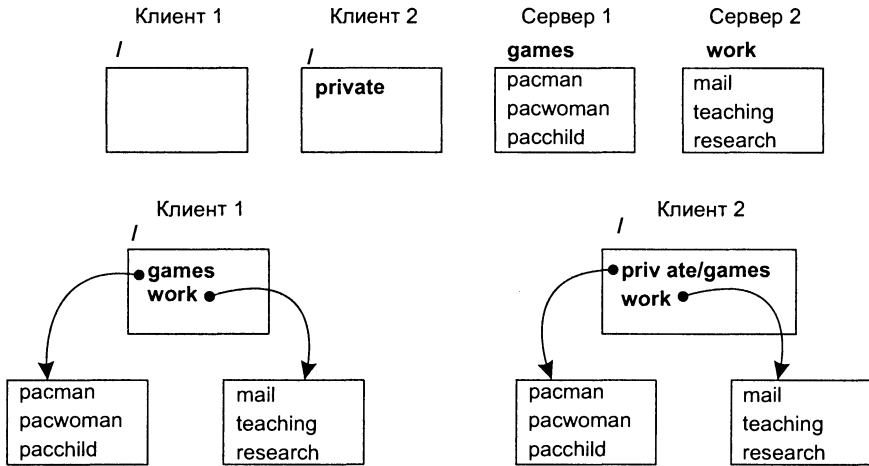


Рис. 1.15. Различные клиенты могут монтировать файловые системы серверов по-разному

Сетевые операционные системы выглядят значительно примитивнее распределенных. Основная разница между этими двумя типами операционных систем состоит в том, что в распределенных операционных системах делается серьезная попытка добиться полной прозрачности, то есть создать представление единой системы.

«Нехватка» прозрачности в сетевых операционных системах имеет некоторые очевидные обратные стороны. Например, с ними часто сложно работать, поскольку пользователь вынужден явно подсоединяться к удаленным машинам или копировать файлы с одной машины на другую. Это также проблемы с управлением. Поскольку все машины под управлением сетевой операционной системы независимы, часто и управлять ими можно исключительно независимо. В результате пользователь может получить удаленное соединение с машиной X, только имея на ней регистрацию. Таким образом, если пользователь хочет использовать один пароль на «все случаи жизни», то для смены пароля он вынужден будет явно сменить его на каждой машине. Рассуждая далее, мы видим, что в основном все права доступа относятся к конкретной машине. Нет простого метода сменить права доступа, поскольку всюду они свои. Такой децентрализованный подход к безопасности нередко затрудняет защиту сетевой операционной системы от атак злоумышленников.

Имеются также и преимущества по сравнению с распределенными операционными системами. Поскольку узлы сетевых операционных систем в значительной степени независимы друг от друга, добавить или удалить машину очень легко. В некоторых случаях все, что надо сделать, чтобы добавить узел, — это подсоединить соответствующую машину к общей сети и поставить в известность о ее существовании остальные машины сети. В Интернете, например, добавление нового сервера происходит именно так. Чтобы сведения о машине попали в Интернет, мы должны просто дать ей сетевой адрес, а лучше символическое имя, которое затем будет внесено в DNS вместе с ее сетевым адресом.

1.4.3. Программное обеспечение промежуточного уровня

Ни распределенные, ни сетевые операционные системы не соответствуют нашему определению распределенных систем, данному в разделе 1.1. Распределенные операционные системы не предназначены для управления набором *независимых* компьютеров, а сетевые операционные системы не дают представления *одной согласованной системы*. На ум приходит вопрос: а возможно ли вообще разработать распределенную систему, которая объединяла бы в себе преимущества двух «миров» — масштабируемость и открытость сетевых операционных систем и прозрачность и относительную простоту в использовании распределенных операционных систем? Решение было найдено в виде дополнительного уровня программного обеспечения, который в сетевых операционных системах позволяет более или менее скрыть от пользователя разнородность набора аппаратных платформ и повысить прозрачность распределения. Многие современные распределенные системы построены в расчете на этот дополнительный уровень, который получил название программного обеспечения промежуточного уровня. В этом пункте мы кратко рассмотрим, как устроено программное обеспечение промежуточного уровня, чтобы понять его особенности.

Позиционирование программного обеспечения промежуточного уровня

Многие распределенные приложения допускают непосредственное использование программного интерфейса, предлагаемого сетевыми операционными системами. Так, связь часто реализуется через операции с сокетами, которые позволяют процессам на разных машинах обмениваться сообщениями [438]. Кроме того, приложения часто пользуются интерфейсами локальных файловых систем. Как мы понимаем, проблема такого подхода состоит в том, что наличие распределения слишком очевидно. Решение заключается в том, чтобы поместить между приложением и сетевой операционной системой промежуточный уровень программной поддержки, обеспечивающий дополнительное абстрагирование. Потому этот уровень и называется *промежуточным*. Он находится посередине между приложением и сетевой операционной системой, как показано на рис. 1.16.

Каждая локальная система, составляющая часть базовой сетевой операционной системы, предоставляет управление локальными ресурсами и простейшие коммуникационные средства для связи с другими компьютерами. Другими словами, программное обеспечение промежуточного уровня не управляет каждым узлом, эта работа по-прежнему приходится на локальные операционные системы.

Основная наша задача — скрыть разнообразие базовых платформ от приложений. Для решения этой задачи многие системы промежуточного уровня предоставляют более или менее полные наборы служб и «не одобряют» желания использовать что-то еще для доступа к этим службам, кроме своих интерфейсов. Другими словами, обход промежуточного уровня и непосредственный вызов служб одной из базовых операционных систем не приветствуется. Мы кратко рассмотрим службы промежуточного уровня.

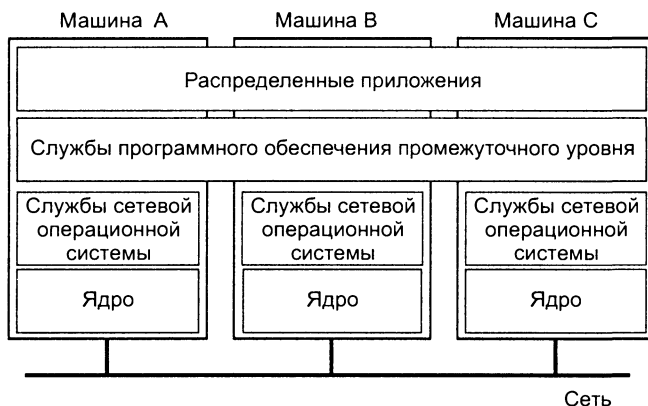


Рис. 1.16. Общая структура распределенных систем с промежуточным уровнем

Интересно отметить, что промежуточный уровень не был изобретен в академических условиях в попытке обеспечить прозрачность распределения. После появления и широкого распространения сетевых операционных систем многие организации обнаружили, что у них накопилась масса сетевых приложений, которые невозможно легко интегрировать в единую систему [45]. Именно тогда производители и начали создавать независимые от приложений службы верхнего уровня для этих систем. Типичные примеры обеспечивали поддержку распределенного взаимодействия и улучшенные коммуникационные возможности.

Разумеется, дорога к правильному промежуточному уровню была непростой. Была создана организация, призванная определить общий стандарт для решений на базе промежуточного уровня. К настоящему времени таких стандартов существует множество. Стандарты в основном несовместимы друг с другом, и что еще хуже, продукты разных производителей, реализующие один и тот же стандарт, редко способны работать вместе. Вероятно, все это ненадолго. Кто-нибудь непременно предложит «программное обеспечение следующего уровня», которое исправит этот недостаток.

Модели промежуточного уровня

Чтобы сделать разработку и интеграцию распределенных приложений как можно более простой, основная часть программного обеспечения промежуточного уровня базируется на некоторой модели, или *парадигме*, определяющей распределение и связь. Относительно простой моделью является представление всех наблюдаемых объектов в виде файлов. Этот подход был изначально введен в UNIX и строго соблюдался в Plan 9 [353]. В Plan 9 все ресурсы, включая устройства ввода-вывода, такие как клавиатура, мышь, диск, сетевые интерфейсы и т. д., рассматривались как файлы. Важно, что удаленные и локальные файлы ничем не отличались. Приложение открывало файлы, читало и записывало в них байты и закрывало их. Поскольку файлы могли совместно использоваться несколькими процессами, связь сокращалась до простого обращения к одному и тому же файлу.

Подобный же подход, но менее строгий, чем в Plan 9, применяется в программном обеспечении промежуточного уровня, построенном по принципу *распределенной файловой системы (distributed file system)*. Во многих случаях это программное обеспечение всего на один шаг ушло от сетевых операционных систем в том смысле, что прозрачность распределения поддерживается только для стандартных файлов (то есть файлов, предназначенных только для хранения данных). Процессы, например, часто должны запускаться исключительно на определенных машинах. Программное обеспечение промежуточного уровня, основанное на модели распределенной файловой системы, оказалось достаточно легко масштабируемым, что способствовало его популярности.

Другая важная ранняя модель программного обеспечения промежуточного уровня основана на *удаленных вызовах процедур (Remote Procedure Calls, RPC)*. В этой модели акцент делается на сокрытии сетевого обмена за счет того, что процессу разрешается вызывать процедуры, реализация которых находится на удаленной машине. При вызове такой процедуры параметры прозрачно передаются на удаленную машину, где, собственно, и выполняется процедура, после чего результат выполнения возвращается в точку вызова процедуры. За исключением, вероятно, некоторой потери производительности, все это выглядит как локальное исполнение вызванной процедуры: вызывающий процесс не уведомляется об имевшем место факте сетевого обмена. Мы вернемся к вызовам удаленных процедур в следующей главе.

По мере того как все более входит в моду ориентированность на объекты, становится ясно, что если вызов процедуры проходит через границы отдельных машин, он может быть представлен в виде прозрачного обращения к объекту, находящемуся на удаленной машине. Это привело к появлению разнообразных систем промежуточного уровня, реализующих представление о *распределенных объектах (distributed objects)*. Идея распределенных объектов состоит в том, что каждый объект реализует интерфейс, который скрывает все внутренние детали объекта от его пользователя. Интерфейс содержит методы, реализуемые объектом, не больше и не меньше. Все, что видит процесс, — это интерфейс.

Распределенные объекты часто реализуются путем размещения объекта на одной из машин и открытия доступа к его интерфейсу с множества других. Когда процесс вызывает метод, реализация интерфейса на машине с процессом просто преобразует вызов метода в сообщение, пересылаемое объекту. Объект выполняет запрашиваемый метод и отправляет назад результаты. Затем реализация интерфейса преобразует ответное сообщение в возвращаемое значение, которое передается вызвавшему процессу. Как и в случае с RPC, процесс может оказаться не осведомленным об этом обмене.

Как модели могут упростить использование сетевых систем, вероятно, наилучшим образом видно на примере World Wide Web. Успех среды Web в основном определяется тем, что она построена на базе потрясающе простой, но высокоэффективной модели *распределенных документов (distributed documents)*. В модели, принятой в Web, информация организована в виде документов, каждый из которых размещен на машине, расположение которой абсолютно прозрачно. Документы содержат ссылки, связывающие текущий документ с други-

ми. Если следовать по ссылке, то документ, с которым связана эта ссылка, будет извлечен из места его хранения и выведен на экран пользователя. Концепция документа не ограничивается исключительно текстовой информацией. Например, в Web поддерживаются аудио- и видеодокументы, а также различные виды документов на основе интерактивной графики.

Позже мы еще вернемся к парадигмам промежуточного уровня.

Службы промежуточного уровня

Существует некоторое количество стандартных для систем промежуточного уровня служб. Все программное обеспечение промежуточного уровня неизменно должно тем или иным образом реализовывать *прозрачность доступа* путем предоставления высокоуровневых *средств связи* (*communication facilities*), скрывающих низкоуровневую пересылку сообщений по компьютерной сети. Интерфейс программирования транспортного уровня, предоставляемый сетевой операционной системой, полностью заменяется другими средствами. Способ, которым поддерживается связь, в значительной степени зависит от модели распределения, предлагаемой программным обеспечением промежуточного уровня пользователям и приложениям. Мы уже упоминали удаленный вызов процедур и обращение к распределенным объектам. Кроме того, многие системы промежуточного уровня предоставляют средства для прозрачного доступа к удаленным данным, такие как распределенные файловые системы или распределенные базы данных. Прозрачная доставка документов, реализуемая в Web, — это еще один пример коммуникаций высокого уровня (однонаправленных).

Важная служба, общая для всех систем промежуточного уровня, — это *именование* (*naming*). Службы именования сравнимы с телефонными книгами или справочниками типа «Желтых страниц». Они позволяют совместно использовать и искать сущности (как в каталогах). Хотя присвоение имен на первый взгляд кажется простым делом, при масштабировании возникают серьезные трудности. Проблема состоит в том, что для эффективного поиска имени в большой системе местоположение разыскиваемой сущности должно считаться фиксированным. Такое допущение, в частности, принято в среде World Wide Web, в которой любой документ поименован посредством URL. URL содержит имя сервера, на котором находится документ с данным URL-адресом. Таким образом, если документ переносится на другой сервер, его URL перестает работать.

Многие системы промежуточного уровня предоставляют специальные средства хранения данных, также именуемые средствами *сохранности* (*persistence*). В своей простейшей форме сохранность обеспечивается распределенными файловыми системами, но более совершенное программное обеспечение промежуточного уровня содержит интегрированные базы данных или предоставляет средства для связи приложений с базами данных.

Если хранение данных играет важную роль для оболочки, то обычно предоставляются и средства для *распределенных транзакций* (*distributed transactions*). Важным свойством транзакций является возможность множества операций чтения и записи в ходе одной атомарной операции. Под атомарностью мы понимаем тот факт, что транзакция может быть либо успешной (когда все операции записи

завершаются успешно), либо неудачной, что оставляет все задействованные данные не измененными. Распределенные транзакции работают с данными, которые, возможно, разбросаны по нескольким машинам.

Предоставление таких служб, как распределенные транзакции, особенно важно в свете того, что маскировка сбоев для распределенных систем нередко затруднена. К сожалению, транзакции легче масштабировать на нескольких географически удаленных машинах, чем на множестве локальных.

Ну и, наконец, практически все системы промежуточного уровня, используемые не только для экспериментов, предоставляют средства обеспечения *защиты* (*security*). По сравнению с сетевыми операционными системами проблема защиты в системах промежуточного уровня состоит в том, что они распределены. Промежуточный уровень в принципе не может «надеяться» на то, что базовые локальные операционные системы будут адекватно обеспечивать защиту всей сети. Соответственно, защита отчасти ложится на программное обеспечение промежуточного уровня. В сочетании с требованием расширяемости защита превращается в одни из наиболее трудно реализуемых в распределенных системах служб.

Промежуточный уровень и открытость

Современные распределенные системы обычно создаются в виде систем промежуточного уровня для нескольких платформ. При этом приложения создаются для конкретной распределенной системы и не зависят от платформы (операционной системы). К сожалению, эта независимость часто заменяется жесткой зависимостью от конкретной системы промежуточного уровня. Проблема заключается в том, что системы промежуточного уровня часто значительно менее открыты, чем утверждается.

Как мы обсуждали ранее, истинно открытая распределенная система определяется полнотой (завершенностью) ее интерфейса. Полнота означает реальное наличие всех необходимых для создания систем описаний. Неполнота описания интерфейса приводит к тому, что разработчики систем вынуждены добавлять свои собственные интерфейсы. Таким образом, мы можем прийти к ситуации, когда разными командами разработчиков в соответствии с одним и тем же стандартом создаются разные системы промежуточного уровня и приложения, написанные под одну из систем, не могут быть перенесены под другую без значительных усилий.

Не менее неприятна ситуация, когда неполнота приводит к невозможности совместной работы двух реализаций, несмотря на то, что они поддерживают абсолютно одинаковый набор интерфейсов, но различные базовые протоколы. Так, если две реализации основаны на несовместимых коммуникационных протоколах, поддерживаемых сетевой операционной системой, маловероятно, что удастся с легкостью добиться их совместной работы. Необходимо, чтобы и протоколы промежуточного уровня, и его интерфейсы были одинаковы, как показано на рис. 1.17.

Еще один пример. Для гарантии совместной работы различных реализаций необходимо, чтобы к сущностям разных систем можно было одинаково обра-

щаться. Если к сущностям в одной системе обращение идет через URL, а в другой системе — через сетевой адрес, понятно, что перекрестные обращения приведут к проблемам. В подобных случаях определения интерфейсов должны точно предписывать вид ссылок.

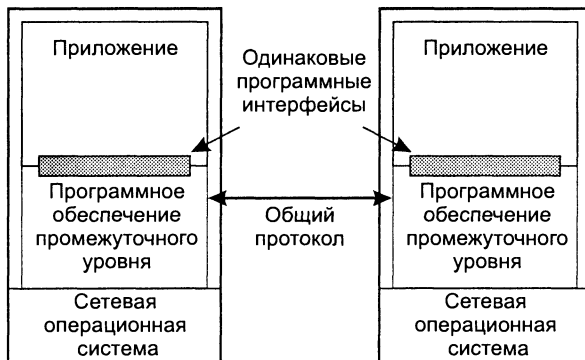


Рис. 1.17. В открытых распределенных системах должны быть одинаковыми как протоколы, используемые промежуточными уровнями каждой из систем, так и интерфейсы, предоставляемые приложениям

Сравнение систем

Краткое сравнение распределенных операционных систем, сетевых операционных систем и распределенных систем промежуточного уровня приводится в табл. 1.5.

Таблица 1.5. Сравнение распределенных операционных систем, сетевых операционных систем и распределенных систем промежуточного уровня

Характеристика	Распределенная операционная система		Сетевая операционная система	Распределенная система промежуточного уровня
	мульти-процессорная	мульти-компьютерная		
Степень прозрачности	Очень высокая	Высокая	Низкая	Высокая
Идентичность операционной системы на всех узлах	Поддерживается	Поддерживается	Не поддерживается	Не поддерживается
Число копий ОС	1	N	N	N
Коммуникации на основе	Совместно используемой памяти	Сообщений	Файлов	В зависимости от модели
Управление ресурсами	Глобальное, централизованное	Глобальное, распределенное	Отдельно на узле	Отдельно на узле

продолжение ➤

Таблица 1.5 (продолжение)

Характеристика	Распределенная операционная система		Сетевая операционная система	Распределенная система промежуточного уровня
	мульти-процессорная	мульти-компьютерная		
Масштабируемость	Отсутствует	Умеренная	Да	Различная
Открытость	Закрытая	Закрытая	Открытая	Открытая

Что касается прозрачности, ясно, что распределенные операционные системы работают лучше, чем сетевые. В мультипроцессорных системах нам нужно скрывать только общее число процессоров, а это относительно несложно. Сложнее скрыть физическое распределение памяти, поэтому не просто создать мультикомпьютерную операционную систему с полностью прозрачным распределением. Распределенные системы часто повышают прозрачность путем использования специальных моделей распределения и связи. Так, например, распределенные файловые системы обычно хорошо скрывают местоположение файлов и доступ к ним. Однако они несколько теряют в общности, и пользователи, решившие выразить все, что можно, в рамках конкретной модели, могут получить взамен проблемы с некоторыми приложениями.

Распределенные операционные системы гомогенны, то есть каждый узел имеет собственную операционную систему (ядро). В мультипроцессорных системах нет необходимости копировать данные — таблицы и пр., поскольку все они находятся в общей памяти и могут использоваться совместно. В этом случае вся связь также осуществляется через общую память, в то время как в мультикомпьютерных системах требуются сообщения. Мы обсуждали, что в сетевых операционных системах связь чаще всего базируется на файлах. Так, например, в Интернете большая часть обмена осуществляется путем передачи файлов. Кроме того, однако, интенсивно используется обмен сообщениями высокого уровня в виде систем электронной почты и досок объявлений. Связь в распределенных системах промежуточного уровня зависит от модели, на которой основана система.

Ресурсы в сетевых операционных системах и распределенных системах промежуточного уровня управляются на каждом узле, что делает масштабирование этих систем относительно простым. Однако практика показывает, что реализация в распределенных системах программного обеспечения промежуточного уровня часто приводит к ограниченности масштабирования. Распределенные операционные системы осуществляют глобальное управление ресурсами, что усложняет их масштабирование. В связи с централизованным подходом (когда все данные находятся в общей памяти) в мультипроцессорных системах они также масштабируются с трудом.

И, наконец, сетевые операционные системы и распределенные системы промежуточного уровня выигрывают с точки зрения открытости. В основном узлы поддерживают стандартный коммуникационный протокол типа TCP/IP, что делает несложной организацию их совместной работы. Однако здесь могут встретиться трудности с переносом приложений под разные платформы. Распределенные операционные системы в основном рассчитаны не на открытость, а на

максимальную производительность, в результате на дороге к открытым системам у них стоит множество запатентованных решений.

1.5. Модель клиент-сервер

До этого момента мы вряд ли сказали что-то о действительной организации распределенных систем, более интересуясь тем, как в этих системах организованы *процессы*. Несмотря на то что достичь согласия по вопросам, связанным с распределенными системами, было нелегко, по одному из вопросов исследователи и разработчики все же договорились. Они пришли к выводу о том, что мышление в понятиях клиентов, запрашивающих службы с серверов, помогает понять сложность распределенных систем и управляться с ней. В этом разделе мы кратко рассмотрим модель клиент-сервер.

1.5.1. Клиенты и серверы

В базовой модели клиент-сервер все процессы в распределенных системах делятся на две возможно перекрывающиеся группы. Процессы, реализующие некоторую службу, например службу файловой системы или базы данных, называются *серверами* (*servers*). Процессы, запрашивающие службы у серверов путем отправки запроса и последующего ожидания ответа от сервера, называются *клиентами* (*clients*). Взаимодействие клиента и сервера, известное также под названием *режим работы запрос-ответ* (*request-reply behavior*), иллюстрирует рис. 1.18.

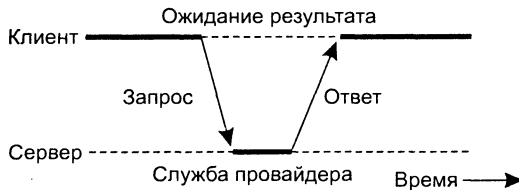


Рис. 1.18. Обобщенное взаимодействие между клиентом и сервером

Если базовая сеть так же надежна, как локальные сети, взаимодействие между клиентом и сервером может быть реализовано посредством простого протокола, не требующего установления соединения. В этом случае клиент, запрашивая службу, облекает свой запрос в форму сообщения с указанием в нем службы, которой он желает воспользоваться, и необходимых для этого исходных данных. Затем сообщение посылается серверу. Последний, в свою очередь, постоянно ожидает входящего сообщения, получив его, обрабатывает, упаковывает результат обработки в ответное сообщение и отправляет его клиенту.

Использование не требующего соединения протокола дает существенный выигрыш в эффективности. До тех пор пока сообщения не начнут пропадать или повреждаться, можно вполне успешно применять протокол типа запрос-ответ. К сожалению, создать протокол, устойчивый к случайным сбоям связи, — нетри-

виальная задача. Все, что мы можем сделать, — это дать клиенту возможность повторно послать запрос, на который не был получен ответ. Проблема, однако, состоит в том, что клиент не может определить, действительно ли первоначальное сообщение с запросом было потеряно или ошибка произошла при передаче ответа. Если потерялся ответ, повторная посылка запроса может привести к повторному выполнению операции. Если операция представляла собой что-то вроде «снять 10 000 долларов с моего банковского счета», понятно, что было бы гораздо лучше, если бы вместо повторного выполнения операции вас просто уведомили о произошедшей ошибке. С другой стороны, если операция была «сообщите мне, сколько денег у меня осталось», запрос прекрасно можно было бы послать повторно. Нетрудно заметить, что у этой проблемы нет единого решения. Мы отложим детальное рассмотрение этого вопроса до главы 7.

В качестве альтернативы во многих системах клиент-сервер используется надежный протокол с установкой соединения. Хотя это решение в связи с его относительно низкой производительностью не слишком хорошо подходит для локальных сетей, оно великолепно работает в глобальных системах, для которых ненадежность является «врожденным» свойством соединений. Так, практически все прикладные протоколы Интернета основаны на надежных соединениях по протоколу TCP/IP. В этих случаях всякий раз, когда клиент запрашивает службу, до посылки запроса серверу он должен установить с ним соединение. Сервер обычно использует для посылки ответного сообщения то же самое соединение, после чего оно разрывается. Проблема состоит в том, что установка и разрыв соединения в смысле затрачиваемого времени и ресурсов относительно дороги, особенно если сообщения с запросом и ответом невелики. Мы обсудим альтернативные решения, в которых управление соединением объединяется с передачей данных, в следующей главе.

Примеры клиента и сервера

Чтобы внести большую ясность в то, как работают клиент и сервер, в этом пункте мы рассмотрим описание клиента и файлового сервера на языке C. И клиент, и сервер должны совместно использовать некоторые определения, которые мы соберем вместе в файл под названием `header.h`, текст которого приведен в листинге 1.3. Эти определения затем включаются в тексты программ клиента и сервера следующей строкой:

```
#include <header.h>
```

Эта инструкция вызовет активность препроцессора, который посимвольно вставит все содержимое файла `header.h` в текст программы до того, как начнется ее компиляция.

Листинг 1.3. Файл `header.h`, используемый клиентом и сервером

```
/* Определения, необходимые и клиентам, и серверам */
#define TRUE 1
/* Максимальная длина имени файла */
#define MAX_PATH 255
/* Максимальное количество данных, передаваемое за один раз */
```

```

#define BUF_SIZE          1024
/* Сетевой адрес файлового сервера */
#define FILE_SERVER       243

/* Определения разрешенных операций */
/* создать новый файл */
#define CREATE            1
/* считать данные из файла и вернуть их */
#define READ              2
/* записать данные в файл */
#define WRITE             3
/* удалить существующий файл */
#define DELETE            4

/* Коды ошибок */
/* операция прошла успешно */
#define OK                0
/* запрос неизвестной операции */
#define E_BAD_OPER        -1
/* ошибка в параметре */
#define E_BAD_PARAM       -2
/* ошибка диска или другая ошибка чтения-записи */
#define E_IO               -3

/* Определение формата сообщения */
struct message {
    long source;           /* идентификатор источника */
    long dest;             /* идентификатор приемника */
    long opcode;           /* запрашиваемая операция */
    long count;            /* число передаваемых байт */
    long offset;           /* позиция в файле, с которой начинается
                           ввод-вывод */
    long result;           /* результат операции */
    char name[MAX_PATHN];  /* имя файла, с которым производятся
                           операции */
    char data[BUF_SIZE];   /* данные, которые будут считаны или
                           записаны */
};

```

Итак, перед нами текст файла `header.h`. Он начинается с определения двух констант, `MAX_PATHN` и `BUF_SIZE`, которые определяют размер двух массивов, используемых в сообщении. Первая задает число символов, которое содержится в имени файла (то есть в строке с путем типа `/usr/ast/books/opsys/chapter1.t`). Вторая задает размер блока данных, который может быть прочитан или записан за одну операцию путем установки размера буфера. Следующая константа, `FILE_SERVER`, задает сетевой адрес файлового сервера, на который клиенты могут посылать сообщения.

Вторая группа констант задает номера операций. Они необходимы для того, чтобы и клиент, и сервер знали, какой код представляет чтение, какой код — запись и т. д. Мы приводим здесь только четыре константы, в реальных системах их обычно больше.

Каждый ответ содержит код результата. Если операция завершена успешно, код результата обычно содержит полезную информацию (например, реальное число считанных байтов). Если нет необходимости возвращать значение (например, при создании файла), используется значение OK. Если операция по каким-либо причинам окончилась неудачей, код результата (E_BAD_OPER, E_BAD_PARAM и др.) сообщает нам, почему это произошло.

Наконец, мы добрались до наиболее важной части файла `header.h` — определения формата сообщения. В нашем примере это структура с 8 полями. Этот формат используется во всех запросах клиентов к серверу и ответах сервера. В реальных системах, вероятно, фиксированного формата у сообщений не будет (поскольку не во всех случаях нужны все поля), но здесь это упростит объяснение. Поля `source` и `dest` определяют, соответственно, отправителя и получателя. Поле `opcode` — это одна из ранее определенных операций, то есть `create`, `read`, `write` или `delete`. Поля `count` и `offset` требуются для передачи параметров. Поле `result` в запросах от клиента к серверу не используется, а при ответах сервера клиенту содержит значение результата. В конце структуры имеются два массива. Первый из них, `name`, содержит имя файла, к которому мы хотим получить доступ. Во втором, `data`, находятся возвращаемые сервером при чтении или передаваемые на сервер при записи данные.

Давайте теперь рассмотрим код в листингах 1.4 и 1.5. Листинг 1.4 — это программа сервера, листинг 1.5 — программа клиента. Программа сервера достаточно элементарна. Основной цикл начинается вызовом `receive` в ответ на сообщение с запросом. Первый параметр определяет отправителя запроса, поскольку в нем указан его адрес, а второй указывает на буфер сообщений, идентифицируя, где должно быть сохранено пришедшее сообщение. Процедура `receive` блокирует сервер, пока не будет получено сообщение. Когда оно наконец приходит, сервер продолжает свою работу и определяет тип кода операции. Для каждого кода операции вызывается своя процедура. Входящее сообщение и буфер для исходящих сообщений заданы в параметрах. Процедура проверяет входящее сообщение в параметре `m1` и строит исходящее в параметре `m2`. Она также возвращает значение функции, которое передается через поле `result`. После отправки ответа сервер возвращается к началу цикла, выполняет вызов `receive` и ожидает следующего сообщения.

В листинге 1.5 находится процедура, копирующая файлы с использованием сервера. Тело процедуры содержит цикл чтения блока из исходного файла и записи его в файл-приемник. Цикл повторяется до тех пор, пока исходный файл не будет полностью скопирован, что определяется по коду возврата операции чтения — должен быть ноль или отрицательное число.

Первая часть цикла состоит из создания сообщения для операции чтения и пересылки его на сервер. После получения ответа запускается вторая часть цикла, в ходе выполнения которой полученные данные посылаются обратно на сервер для записи в файл-приемник. Программы из листингов 1.4 и 1.5 — это только набросок кода. В них опущено множество деталей. Так, например, не приводятся процедуры `do_xxx` (которые на самом деле выполняют работу), отсутствует также обработка ошибок. Однако общая идея взаимодействия клиента и сервера вполне

понятна. В следующих пунктах мы ближе рассмотрим некоторые дополнительные аспекты модели клиент-сервер.

Листинг 1.4. Пример сервера

```
#include <header.h>
void main(void) {
    struct message m1, m2;          /* входящее и исходящее сообщения */
    int r;                          /* код результата */
    while(TRUE) {                  /* сервер работает непрерывно */
        receive(FILE_SERVER, &m1); /* блок ожидания сообщения */
        switch(mi.opcode) {        /* в зависимости от типа запроса */
            case CREATE:           r = do_create(&m1, &m2); break;
            case READ:             r = do_read(&m1, &m2) break;
            case WRITE:            r = do_write(&m1, &m2); break;
            case DELETE:           r = do_delete(&m1, &m2); break;
            default:                r = E_BAD_OPER;
        }
        m2.result = r;             /* вернуть результат клиенту */
        send(mi.source, &m2);      /* послать ответ */
    }
}
```

Листинг 1.5. Клиент, использующий сервер из листинга 1.4 для копирования файла

```
#include <header.h>
/* процедура копирования файла через сервер */
int copy(char *src, char *dst){
    struct message m1;             /* буфер сообщения */
    long position;                 /* текущая позиция в файле */
    long client = 110;             /* адрес клиента */
    initialize();                  /* подготовиться к выполнению */
    position = 0;
    do {
        m1.opcode = READ;          /* операция чтения */
        m1.offset = position;      /* текущая позиция в файле */
        m1.count = BUF_SIZE;      /* сколько байт прочитать */
        strcpy(&m1.name, src);     /* скопировать имя читаемого
                                   файла*/
        send(FILESERVER, &m1);     /* послать сообщение на файловый сервер*/
        receive(client, &m1);      /* блок ожидания ответа */
        /* Записать полученные данные в файл-приемник. */
        m1.opcode = WRITE;         /* операция: запись */
        m1.offset = position;      /* текущая позиция в файле */
        m1.count = m1.result;      /* сколько байт записать */
        strcpy(&m1.name, clst);    /* скопировать имя записываемого
                                   файла*/
        send(FILE_SERVER, &m1);    /* послать сообщение на файловый сервер */
        receive(client, &m1);      /* блок ожидания ответа */
        position += m1.result;     /* в m1.result содержится
                                   количество записанных байтов */
    } while( m1.result > 0 );      /* повторять до окончания */
    /* вернуть OK или код ошибки */
    return(m1.result >= 0 ? OK : m1.result);
}
```

1.5.2. Разделение приложений по уровням

Модель клиент-сервер была предметом множества дебатов и споров. Один из главных вопросов состоял в том, как точно разделить клиента и сервера. Понятно, что обычно четкого различия нет. Например, сервер распределенной базы данных может постоянно выступать клиентом, передающим запросы на различные файловые серверы, отвечающие за реализацию таблиц этой базы данных. В этом случае сервер баз данных сам по себе не делает ничего, кроме обработки запросов.

Однако рассматривая множество приложений типа клиент-сервер, предназначенных для организации доступа пользователей к базам данных, многие рекомендовали разделять их на три уровня.

- ♦ уровень пользовательского интерфейса;
- ♦ уровень обработки;
- ♦ уровень данных.

Уровень пользовательского интерфейса содержит все необходимое для непосредственного общения с пользователем, например для управление дисплеем. Уровень обработки обычно содержит приложения, а уровень данных — собственно данные, с которыми происходит работа. В следующих пунктах мы обсудим каждый из этих уровней.

Уровень пользовательского интерфейса

Уровень пользовательского интерфейса обычно реализуется на клиентах. Этот уровень содержит программы, посредством которых пользователь может взаимодействовать с приложением. Сложность программ, входящих в пользовательский интерфейс, весьма различна.

Простейший вариант программы пользовательского интерфейса не содержит ничего, кроме символьного (не графического) дисплея. Такие интерфейсы обычно используются при работе с мэйнфреймами. В том случае, когда мэйнфрейм контролирует все взаимодействия, включая работу с клавиатурой и монитором, мы вряд ли можем говорить о модели клиент-сервер. Однако во многих случаях терминалы пользователей производят некоторую локальную обработку, осуществляя, например, эхо-печать вводимых строк или предоставляя интерфейс форм, в котором можно отредактировать введенные данные до их пересылки на главный компьютер.

В наше время даже в среде мэйнфреймов наблюдаются более совершенные пользовательские интерфейсы. Обычно на клиентских машинах имеется как минимум графический дисплей, на котором можно задействовать всплывающие или выпадающие меню и множество управляющих элементов, доступных для мыши или клавиатуры. Типичные примеры таких интерфейсов — надстройка X-Windows, используемая во многих UNIX-системах, и более ранние интерфейсы, разработанные для персональных компьютеров, работающих под управлением MS-DOS и Apple Macintosh.

Современные пользовательские интерфейсы значительно более функциональны. Они поддерживают совместную работу приложений через единственное графическое окно и в ходе действий пользователя обеспечивают через это окно обмен данными. Например, для удаления файла часто достаточно перенести значок, соответствующий этому файлу, на значок мусорной корзины. Аналогичным образом многие текстовые процессоры позволяют пользователю перемещать текст документа в другое место, пользуясь только мышью. Мы вернемся к пользовательским интерфейсам в главе 3.

Уровень обработки

Многие приложения модели клиент-сервер построены как бы из трех различных частей: части, которая занимается взаимодействием с пользователем, части, которая отвечает за работу с базой данных или файловой системой, и средней части, реализующей основную функциональность приложения. Эта средняя часть логически располагается на уровне обработки. В противоположность пользовательским интерфейсам или базам данных на уровне обработки трудно выделить общие закономерности. Однако мы приведем несколько примеров для разъяснения вопросов, связанных с этим уровнем.

В качестве первого примера рассмотрим поисковую машину в Интернете. Если отбросить все анимированные баннеры, картинки и прочие оконные украшения, пользовательский интерфейс поисковой машины очень прост: пользователь вводит строку, состоящую из ключевых слов, и получает список заголовков web-страниц. Результат формируется из гигантской базы просмотренных и проиндексированных web-страниц. Ядром поисковой машины является программа, трансформирующая введенную пользователем строку в один или несколько запросов к базе данных. Затем она помещает результаты запроса в список и преобразует этот список в набор HTML-страниц. В рамках модели клиент-сервер часть, которая отвечает за выборку информации, обычно находится на уровне обработки. Эта структура показана на рис. 1.19.

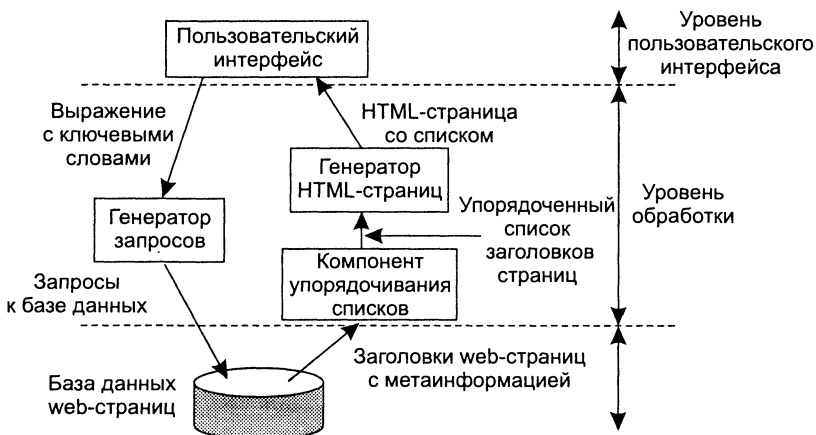


Рис. 1.19. Обобщенная организация трехуровневой поисковой машины для Интернета

В качестве второго примера рассмотрим систему поддержки принятия решений для фондового рынка. Так же как и в поисковой машине, эту систему можно разделить на внешний интерфейс, реализующий работу с пользователем, внутреннюю часть, отвечающую за доступ к базе с финансовой информацией, и промежуточную программу анализа. Анализ финансовых данных может потребовать замысловатых методик и технологий на основе статистических методов и искусственного интеллекта. В некоторых случаях для того, чтобы обеспечить требуемые производительность и время отклика, ядро системы поддержки финансовых решений должно выполняться на высокопроизводительных компьютерах.

Нашим последним примером будет типичный офисный пакет, состоящий из текстового процессора, приложения для работы с электронными таблицами, коммуникационных утилит и т. д. Подобные офисные пакеты обычно поддерживают обобщенный пользовательский интерфейс, возможность создания составных документов и работу с файлами в домашнем каталоге пользователя. В этом случае уровень обработки будет включать в себя относительно большой набор программ, каждая из которых призвана поддерживать какую-то из функций обработки.

Уровень данных

Уровень данных в модели клиент-сервер содержит программы, которые предоставляют данные обрабатывающим их приложениям. Специфическим свойством этого уровня является требование *сохранности (persistence)*. Это означает, что когда приложение не работает, данные должны сохраняться в определенном месте в расчете на дальнейшее использование. В простейшем варианте уровень данных реализуется файловой системой, но чаще для его реализации задействуется полномасштабная база данных. В модели клиент-сервер уровень данных обычно находится на стороне сервера.

Кроме простого хранения информации уровень данных обычно также отвечает за поддержание целостности данных для различных приложений. Для базы данных поддержание целостности означает, что метаданные, такие как описания таблиц, ограничения и специфические метаданные приложений, также хранятся на этом уровне. Например, в приложении для банка мы можем пожелать сформировать уведомление, если долг клиента по кредитной карте достигнет определенного значения. Это может быть сделано при помощи триггера базы данных, который в нужный момент активизирует процедуру, связанную с этим триггером.

Обычно в деловой среде уровень данных организуется в форме реляционной базы данных. Ключевым здесь является независимость данных. Данные организуются независимо от приложений так, чтобы изменения в организации данных не влияли на приложения, а приложения не оказывали влияния на организацию данных. Использование реляционных баз данных в модели клиент-сервер помогает нам отделить уровень обработки от уровня данных, рассматривая обработку и данные независимо друг от друга.

Однако существует обширный класс приложений, для которых реляционные базы данных не являются наилучшим выбором. Характерной чертой таких приложений является работа со сложными типами данных, которые проще модели-

ровать в понятиях объектов, а не отношений. Примеры таких типов данных — от простых наборов прямоугольников и окружностей до проекта самолета в случае систем автоматизированного проектирования. Также и мультимедийным системам значительно проще работать с видео- и аудиопотоками, используя специфичные для них операции, чем с моделями этих потоков в виде реляционных таблиц.

В тех случаях, когда операции с данными значительно проще выразить в понятиях работы с объектами, имеет смысл реализовать уровень данных средствами объектно-ориентированных баз данных. Подобные базы данных не только поддерживают организацию сложных данных в форме объектов, но и хранят реализации операций над этими объектами. Таким образом, часть функциональности, приходившейся на уровень обработки, мигрирует в этом случае на уровень данных.

1.5.3. Варианты архитектуры клиент-сервер

Разделение на три логических уровня, обсуждавшееся в предыдущем пункте, приводит на мысль о множестве вариантов физического распределения по отдельным компьютерам приложений в модели клиент-сервер. Простейшая организация предполагает наличие всего двух типов машин.

- ◆ Клиентские машины, на которых имеются программы, реализующие только пользовательский интерфейс или его часть.
- ◆ Серверы, реализующие все остальное, то есть уровни обработки и данных.

Проблема подобной организации состоит в том, что на самом деле система не является распределенной: все происходит на сервере, а клиент представляет собой не что иное, как простой терминал. Существует также множество других возможностей, наиболее употребительные из них мы сейчас рассмотрим.

Многослойные архитектуры

Один из подходов к организации клиентов и серверов — это распределение программ, находящихся на уровне приложений, описанном в предыдущем пункте, по различным машинам, как показано на рис. 1.20 [216, 467]. В качестве первого шага мы рассмотрим разделение на два типа машин: на клиенты и на серверы, что приведет нас к *физически двухзвенной архитектуре* (*physically two-tiered architecture*).

Один из возможных вариантов организации — поместить на клиентскую сторону только терминальную часть пользовательского интерфейса, как показано на рис. 1.20, *а*, позволив приложению удаленно контролировать представление данных. Альтернативой этому подходу будет передача клиенту всей работы с пользовательским интерфейсом (рис. 1.20, *б*). В обоих случаях мы отделяем от приложения графический внешний интерфейс, связанный с остальной частью приложения (находящейся на сервере) посредством специфичного для данного приложения протокола. В подобной модели внешний интерфейс делает только то, что необходимо для предоставления интерфейса приложения.

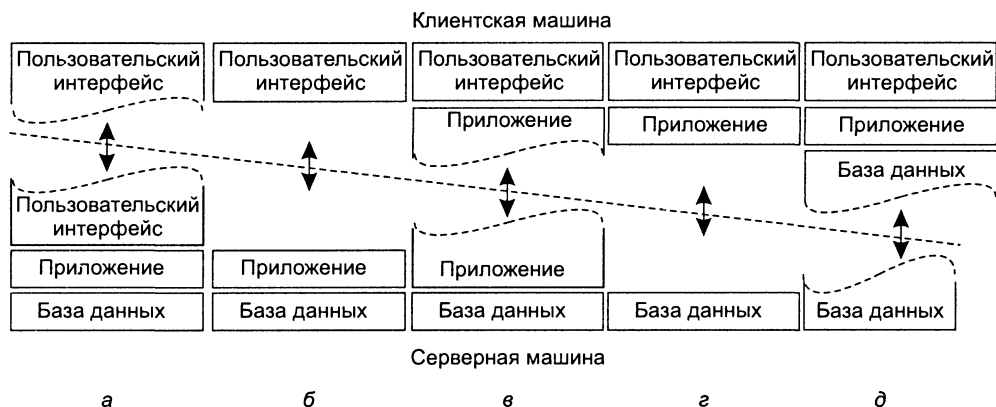


Рис. 1.20. Альтернативные формы организации архитектуры клиент-сервер

Продолжим линию наших рассуждений. Мы можем перенести во внешний интерфейс часть нашего приложения, как показано на рис. 1.20, в. Примером может быть вариант, когда приложение создает форму непосредственно перед ее заполнением. Внешний интерфейс затем проверяет правильность и полноту заполнения формы и при необходимости взаимодействует с пользователем. Другим примером организации системы по образцу, представленному на рис. 1.20, в, может служить текстовый процессор, в котором базовые функции редактирования осуществляются на стороне клиента с локально кэшируемыми или находящимися в памяти данными, а специальная обработка, такая как проверка орфографии или грамматики, выполняется на стороне сервера.

Во многих системах клиент-сервер популярна организация, представленная на рис. 1.20, г и д. Эти типы организации применяются в том случае, когда клиентская машина — персональный компьютер или рабочая станция — соединена сетью с распределенной файловой системой или базой данных. Большая часть приложения работает на клиентской машине, а все операции с файлами или базой данных передаются на сервер. Рисунок 1.20, д отражает ситуацию, когда часть данных содержится на локальном диске клиента. Так, например, при работе в Интернете клиент может постепенно создать на локальном диске огромный кэш наиболее часто посещаемых web-страниц.

Рассматривая только клиенты и серверы, мы упускаем тот момент, что серверу иногда может понадобиться работать в качестве клиента. Такая ситуация, отраженная на рис. 1.21, приводит нас к *физически трехзвенной архитектуре (physically three-tiered architecture)*.

В подобной архитектуре программы, составляющие часть уровня обработки, выносятся на отдельный сервер, но дополнительно могут частично находиться и на машинах клиентов и серверов. Типичный пример трехзвенной архитектуры — обработка транзакций. В этом случае отдельный процесс — монитор транзакций — координирует все транзакции, возможно, на нескольких серверах данных. Мы вернемся к обработке транзакций в следующих главах.

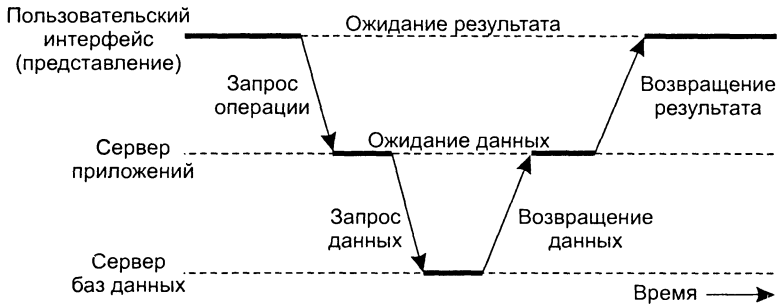


Рис. 1.21. Пример сервера, действующего как клиент

Современные варианты архитектуры

Многозвенные архитектуры клиент-сервер являются прямым продолжением разделения приложений на уровни пользовательского интерфейса, компонентов обработки и данных. Различные звенья взаимодействуют в соответствии с логической организацией приложения. Во множестве бизнес-приложений распределенная обработка эквивалентна организации многозвенной архитектуры приложений клиент-сервер. Мы будем называть такой тип распределения *вертикальным распределением* (*vertical distribution*). Характеристической особенностью вертикального распределения является то, что оно достигается размещением логически различных компонентов на разных машинах. Это понятие связано с концепцией *вертикального разбиения* (*vertical fragmentation*), используемой в распределенных реляционных базах данных, где под этим термином понимается разбиение по столбцам таблиц для их хранения на различных машинах [337].

Однако вертикальное распределение — это лишь один из возможных способов организации приложений клиент-сервер, причем во многих случаях наименее интересный. В современных архитектурах распределение на клиенты и серверы происходит способом, известным как *горизонтальное распределение* (*horizontal distribution*). При таком типе распределения клиент или сервер может содержать физически разделенные части логически однородного модуля, причем работа с каждой из частей может происходить независимо. Это делается для выравнивания загрузки.

В качестве распространенного примера горизонтального распределения рассмотрим web-сервер, реплицированный на несколько машин локальной сети, как показано на рис. 1.22. На каждом из серверов содержится один и тот же набор web-страниц, и всякий раз, когда одна из web-страниц обновляется, ее копии незамедлительно рассылаются на все серверы. Сервер, которому будет передан приходящий запрос, выбирается по правилу «карусели» (*round-robin*). Эта форма горизонтального распределения весьма успешно используется для выравнивания нагрузки на серверы популярных web-сайтов.

Таким же образом, хотя и менее очевидно, могут быть распределены и клиенты. Для несложного приложения, предназначенного для коллективной работы, мы можем не иметь сервера вообще. В этом случае мы обычно говорим об *однопарном распределении* (*peer-to-peer distribution*). Подобное происходит, напри-

мер, если пользователь хочет связаться с другим пользователем. Оба они должны запустить одно и то же приложение, чтобы начать сеанс. Третий клиент может общаться с одним из них или обоими, для чего ему нужно запустить то же самое приложение.

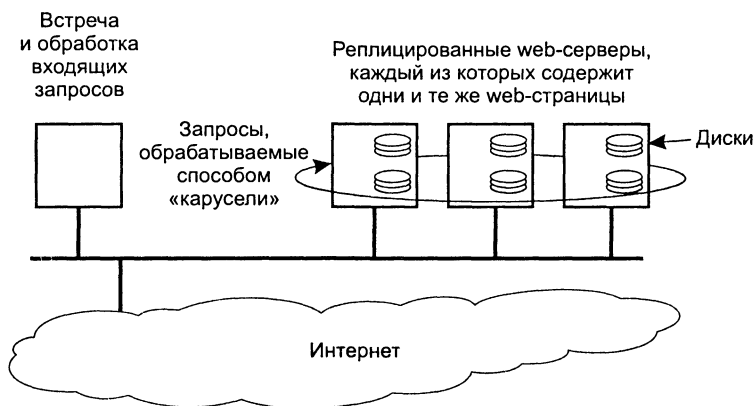


Рис. 1.22. Пример горизонтального распределения web-службы

Число альтернативных организаций архитектуры клиент-сервер обсуждается в книге [3]. В следующих главах мы рассмотрим множество других вариантов организации распределенных систем, в частности мы познакомимся с системами, распределенными одновременно и по вертикали, и по горизонтали.

1.6. Итоги

Распределенные системы состоят из автономных компьютеров, которые работают совместно, представляя в виде единой связной системы. Их важное преимущество состоит в том, что они упрощают интеграцию различных приложений, работающих на разных компьютерах, в единую систему. Еще одно их преимущество — при правильном проектировании распределенные системы хорошо масштабируются. Их размер ограничивается только размером базовой сети. Платой за эти преимущества часто является очень сложное программное обеспечение, падение производительности и особенно проблемы с безопасностью. Тем не менее заинтересованность в построении и внедрении распределенных систем наблюдается повсеместно.

Существуют различные типы распределенных систем. Распределенные операционные системы используются для управления аппаратным обеспечением взаимосвязанных компьютерных систем, к которым относятся мультипроцессорные и гомогенные мультикомпьютерные системы. Эти распределенные системы на самом деле не состоят из автономных компьютеров, но успешно воспринимаются в виде единой системы. Сетевые операционные системы, с другой стороны, с успехом объединяют различные компьютеры, работающие под управлением

своих операционных систем, так что пользователи с легкостью могут получать доступ к локальным службам каждого из узлов. Однако сетевые операционные системы не создают ощущения работы с единой системой, которое характерно для распределенных операционных систем.

Современные распределенные системы обычно содержат поверх сетевой операционной системы дополнительный уровень программного обеспечения. Этот уровень, называемый промежуточным, предназначен для того, чтобы скрыть гетерогенность и распределенную природу базового набора компьютеров. Распределенные системы с промежуточным уровнем обычно требуют специфическую модель распределения и связи. Известные модели основаны на удаленном вызове процедур, а также на распределенных объектах, файлах или документах.

Для каждой распределенной системы важна схема ее внутренней организации. Широко применяется модель, в которой процессы клиента запрашивают службы у процессов сервера. Клиент посылает на сервер сообщение и ожидает, пока тот вернет ответ. Эта модель тесно связана с традиционным программированием, в котором службы реализуются в виде процедур в отдельных модулях. Дальнейшее уточнение обычно состоит в подразделении на уровень пользовательского интерфейса, уровень обработки и уровень данных. Сервер обычно отвечает за уровень данных, а уровень пользовательского интерфейса реализуется на стороне клиента. Уровень обработки может быть реализован на клиенте, на сервере или поделен между ними.

В современных распределенных системах для построения крупномасштабных систем такой вертикальной организации приложений модели клиент-сервер недостаточно. Необходимо горизонтальное распределение, при котором клиенты и серверы физически распределены и реплицируются на несколько компьютеров. Типичным примером успешного применения горизонтального распределения является World Wide Web.

Вопросы и задания

1. Какова роль программного обеспечения промежуточного уровня в распределенных системах?
2. Объясните, что такое прозрачность (распределения) и приведите примеры различных видов прозрачности.
3. Почему иногда так трудно скрыть наличие в распределенной системе сбоя и восстановление после него?
4. Почему реализация максимально возможной степени прозрачности — это не всегда хорошо?
5. Что такое открытая распределенная система и какие преимущества дает открытость?
6. Опишите точно, что такое масштабируемая система.
7. Масштабируемости можно добиться, используя различные методики. Что это за методики?

8. Чем мультипроцессорная система отличается от мультикомпьютерной?
9. Мультикомпьютерная система с 256 процессорами организована в виде решетки 16×16 . Какая в такой системе может быть максимальная задержка (в хопх)?
10. Рассмотрим 256-процессорный гиперкуб. Какая максимальная задержка может наблюдаться в нем (снова в хопх)?
11. В чем состоит разница между распределенными и сетевыми операционными системами?
12. Расскажите, как можно использовать микроядро для организации операционной системы, работающей в режиме клиент-сервер.
13. Опишите основные принципы работы распределенной системы с совместно используемой памятью страничной организации.
14. Какова причина разработки распределенных систем с совместно используемой памятью? В чем, по-вашему, состоит главная трудность их эффективной реализации?
15. Расскажите, что такое ошибочное разделение в распределенных системах с совместно используемой памятью. Какие решения этой проблемы вы можете предложить?
16. Экспериментальный файловый сервер $3/4$ времени работает, а $1/4$ времени «лежит» по причине ошибок. Сколько реплик этого сервера должно быть сделано, чтобы его доступность составляла хотя бы 99 %?
17. Что такое трехзвенная архитектура клиент-сервер?
18. В чем состоит разница между горизонтальным и вертикальным распределениями?
19. Рассмотрим цепочку процессов P_1, P_2, \dots, P_n , которая реализована в многозвенной архитектуре клиент-сервер. Процесс P_i является клиентом процесса P_{i+1} , возвращая, в свою очередь, результат процессу P_{i-1} только после того, как сам получит результат от процесса P_{i+1} . Какова будет главная проблема подобной организации, когда мы станем рассматривать производительность запроса-ответа для процесса P_1 ?

Глава 2

Связь

- 2.1. Уровни протоколов
- 2.2. Удаленный вызов процедур
- 2.3. Обращение к удаленным объектам
- 2.4. Связь посредством сообщений
- 2.5. Связь на основе потоков данных
- 2.6. Итоги

Связь между процессами — это суть распределенных систем. Нет смысла изучать распределенные системы, не рассматривая при этом во всех подробностях способы обмена информацией между различными процессами, выполняющимися на разных машинах. Взаимодействие в распределенных системах всегда базируется на низкоуровневом механизме передачи сообщений, предоставляемом базовой сетью. Как мы обсуждали в предыдущей главе, реализация взаимодействия через передачу сообщений сложнее, чем использование примитивов на базе разделяемой памяти. Современные распределенные системы часто включают в себя тысячи или даже миллионы процессов, разбросанных по ненадежной сети, такой как Интернет. Если не заменить простейшие средства взаимодействия в компьютерных сетях чем-то иным, разработка масштабных приложений будет достаточно сложной.

Мы начнем эту главу с обсуждения правил, которых придерживаются общающиеся между собой процессы. Их обычно называют протоколами. Мы сосредоточимся на структурировании этих протоколов в виде уровней. Затем мы рассмотрим четыре широко распространенные модели взаимодействия: удаленный вызов процедур (Remote Procedure Call, RPC), удаленное обращение к методам (Remote Method Invocation, RMI), ориентированный на сообщения промежуточный уровень (Message-Oriented Middleware, MOM) и потоки данных (streams).

Нашей первой моделью взаимодействия в распределенных системах станет удаленный вызов процедур. Механизм RPC нацелен на сокрытие большей части проблем передачи сообщений и идеален для приложений архитектуры клиент-сервер. Усовершенствованный вариант модели RPC имеет вид удаленного обра-

щения к методам, которое основано на представлении распределенных объектов. Механизмы RPC и RMI рассматриваются в отдельных разделах.

Во многих распределенных приложениях связь не ограничивается слегка урезанным шаблоном взаимодействий клиента и сервера. В подобных случаях мыслить категориями сообщений оказывается предпочтительнее. Однако применение разнообразных низкоуровневых средств связи компьютерных сетей приведет к серьезным нарушениям прозрачности распределения. Альтернативой им является высокоуровневая модель очереди сообщений, связь в которой очень напоминает системы электронной почты. Ориентированный на сообщения средний уровень — это достаточно важная тема, чтобы отвести отдельный раздел и на нее.

Что касается мультимедиа в распределенных системах, понемногу становится очевидным, что таким системам недостает поддержки передачи непрерывных потоков, таких как аудио или видео. Им необходимо понятие потока, который позволяет поддерживать непрерывно идущие сообщения в соответствии с различными ограничениями по синхронизации. Потоки обсуждаются в последнем разделе этой главы.

2.1. Уровни протоколов

В условиях отсутствия совместно используемой памяти вся связь в распределенных системах основана на обмене (низкоуровневыми) сообщениями. Если процесс *A* хочет пообщаться с процессом *B*, он должен сначала построить сообщение в своем собственном адресном пространстве. Затем он выполняет системный вызов, который пересылает сообщение по сети процессу *B*. Хотя основная идея выглядит несложной, во избежание хаоса *A* и *B* должны договориться о смысле пересылаемых нулей и единиц. Если *A* посылает потрясающий новый роман, написанный по-французски, в кодировке IBM EBCDIC, а *B* ожидает результаты переучета в супермаркете, на английском языке и в кодировке ASCII, их взаимодействие будет не слишком успешным.

Необходимо множество различных договоренностей. Сколько вольт следует использовать для передачи нуля, а сколько для передачи единицы? Как получатель узнает, что этот бит сообщения — последний? Как ему определить, что сообщение было повреждено или утеряно, и что ему делать в этом случае? Какую длину имеют числа, строки и другие элементы данных и как они отображаются? Короче говоря, необходимы соглашения различного уровня, от низкоуровневых подробностей передачи битов до высокоуровневых деталей отображения информации.

Чтобы упростить работу с множеством уровней и понятий, используемых в передаче данных, *Международная организация по стандартам (International Standards Organization, ISO)* разработала эталонную модель, которая ясно определяет различные уровни, дает им стандартные имена и указывает, какой уровень за что отвечает. Эта модель получила название *Эталонной модели взаимодействия открытых систем (Open Systems Interconnection Reference Model)* [120].

Это название обычно заменяется сокращением *модель ISO OSI*, или просто *модель OSI*. Следует заметить, что протоколы, которые должны были реализовывать части модели OSI, никогда не получали широкого распространения. Однако сама по себе базовая модель оказалась вполне пригодной для исследования компьютерных сетей. Несмотря на то что мы не собираемся приводить здесь полное описание этой модели и всех ее дополнений, небольшое введение в нее будет нам полезно. Дополнительные детали можно почерпнуть в [446].

Модель OSI разрабатывалась для того, чтобы предоставить открытым системам возможность взаимодействовать друг с другом. Открытая система — это система, которая способна взаимодействовать с любой другой открытой системой по стандартным правилам, определяющим формат, содержимое и смысл отправляемых и принимаемых сообщений. Эти правила зафиксированы в том, что называется *протоколами (protocols)*. Для того чтобы группа компьютеров могла поддерживать связь по сети, они должны договориться об используемых протоколах. Все протоколы делятся на два основных типа. В протоколах с *установлением соединения (connection-oriented)* перед началом обмена данными отправитель и получатель должны установить соединение и, возможно, договориться о том, какой протокол они будут использовать. После завершения обмена они должны разорвать соединение. Системой с установлением соединения является, например, телефон. В случае протоколов без *установления соединения (connectionless)* никакой подготовки не нужно. Отправитель посылает первое сообщение, как только он готов это сделать. Письмо, опущенное в почтовый ящик, — пример связи без установления соединения. В компьютерных технологиях широко применяется как связь с установлением соединения, так и связь без установления соединения.

В модели OSI взаимодействие подразделяется на семь уровней, как показано на рис. 2.1. Каждый уровень отвечает за один специфический аспект взаимодействия. Таким образом, проблема может быть разделена на поддающиеся решению части, каждая из которых может разбираться независимо от других. Каждый из уровней предоставляет интерфейс для работы с вышестоящим уровнем. Интерфейс состоит из набора операций, которые совместно определяют интерфейс, предоставляемый уровнем тем, кто им пользуется.

Когда процесс *A* на машине 1 хочет пообщаться с процессом *B* на машине 2, он строит сообщение и посылает его прикладному уровню своей машины. Этот уровень может представлять собой, например, библиотечную процедуру или реализовываться как-то иначе (например, внутри операционной системы или внешнего сетевого процессора). Программное обеспечение прикладного уровня добавляет в начало сообщения свой *заголовок (header)* и передает получившееся сообщение через интерфейс с уровня 7 на уровень 6, уровень представления. Уровень представления, в свою очередь, добавляет в начало сообщения свой заголовок и передает результат вниз, на сеансовый уровень и т. д. Некоторые уровни добавляют не только заголовок в начало, но и завершение в конец. Когда сообщение дойдет до физического уровня, он осуществит его реальную передачу, как это показано на рис. 2.2.

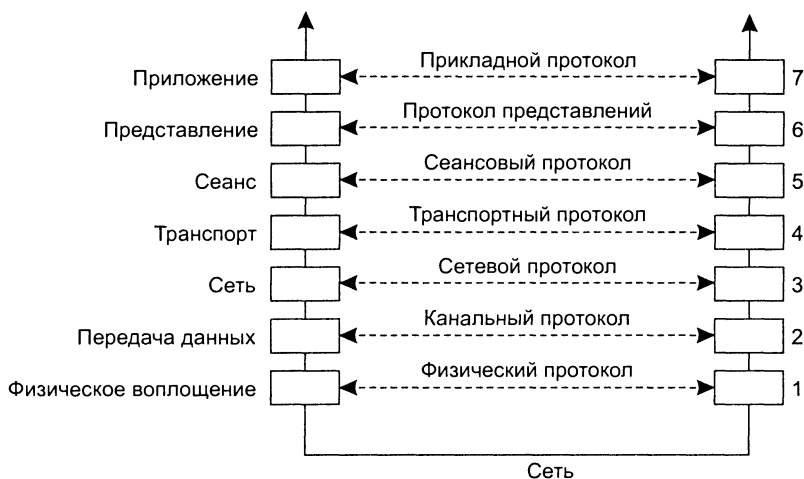


Рис. 2.1. Уровни, интерфейсы и протоколы модели OSI



Рис. 2.2. Передача по сети типового сообщения

Когда сообщение приходит на машину 2, оно передается вверх, при этом на каждом уровне считывается и проверяется соответствующий заголовок. В конце концов сообщение достигает получателя, процесса *B*, который может ответить на него, при этом вся история повторяется в обратном направлении. Информация из заголовка уровня *n* используется протоколом уровня *n*.

В качестве примера важности многоуровневых протоколов рассмотрим обмен информацией между двумя компаниями, авиакомпанией Zippy Airlines и поставщиком продуктов Mushy Meals, Inc. Каждый месяц начальник отдела обслуживания пассажиров Zippy просит своего секретаршу связаться с секретаршей менеджера по продажам Mushy и заказать 100 000 коробок «резиновых» цыплят. Обычно заказы пересылались почтой. Однако из-за постепенного ухудшения качества почтовых услуг в один прекрасный момент секретарши решают больше не писать друг другу письма, а связываться по факсу. Они могут делать это, не беспокоя своих боссов, поскольку протокол касается физической передачи заказов, а не их содержания.

Точно так же начальник отдела обслуживания пассажиров может решить отказать от цыплят и перейти на новый деликатес от Mushy, превосходные козы ребра, это решение никак не скажется на работе секретарш. Это происходит потому, что у нас есть два уровня — боссы и их секретарши. Каждый уровень имеет свой собственный протокол (темы для обсуждения и технологию), который можно изменить независимо от другого. Эта независимость делает многоуровневые протоколы привлекательными. Каждый уровень при появлении новых технологий может быть изменен независимо от других.

В модели OSI, как было показано на рис. 2.1, не два уровня, а семь. Набор протоколов, используемых в конкретной системе, называется *комплексом*, или *стеком протоколов*. Важно отличать эталонную модель от реальных протоколов. Как мы отмечали, протоколы OSI никогда не были популярны. В противоположность им протоколы, разрабатывавшиеся для Интернета, такие как TCP и IP, используются повсеместно. В последующих пунктах мы кратко рассмотрим каждый из уровней модели OSI, начиная с нижнего. Однако вместо того, чтобы приводить примеры соответствующих протоколов OSI, мы рассмотрим протоколы Интернета, используемые на каждом из этих уровней.

2.1.1. Низкоуровневые протоколы

Мы начнем с обсуждения трех нижних уровней комплекта протоколов OSI. Эти три уровня совместно реализуют основные функции компьютерной сети.

Физический уровень

Физический уровень ответственен за передачу нулей и единиц. Сколько вольт использовать для передачи нуля или единицы, сколько бит в секунду можно передать и можно ли осуществлять передачу одновременно в двух направлениях — вот основные проблемы физического уровня. Кроме того, к физическому уровню относится размер и форма сетевых коннекторов (разъемов), а также число выводов и назначение каждого из них.

Протоколы физического уровня отвечают за стандартизацию электрических, механических и сигнальных интерфейсов, чтобы, если одна машина посылает ноль, другая приняла его как ноль, а не как единицу. Было разработано множество стандартов физического уровня для различных носителей, например стандарт RS-232-C для последовательных линий связи.

Канальный уровень

Физический уровень только пересылает биты. Пока нет ошибок, все хорошо. Однако в реальных сетях происходят ошибки, и их нужно как-то находить и исправлять. Это и является главной задачей канального уровня. Он группирует биты в модули, обычно называемые *кадрами* (*frames*), и следит за тем, чтобы каждый кадр был передан правильно.

Канальный уровень делает это путем помещения специальной битовой маски в начало и конец каждого кадра для их маркировки, а также путем вычисления *контрольной суммы* (*checksum*), то есть суммирования всех байтов кадра опреде-

ленным образом. Канальный уровень добавляет контрольную сумму к кадру. Когда кадр принимается, приемник повторно вычисляет контрольную сумму данных и сравнивает результат с контрольной суммой, пришедшей вместе с кадром. Если они совпадают, кадр считается верным и принимается. Если они различны, получатель просит отправителя снова отправить этот кадр. Кадры последовательно нумеруются с указанием номеров в заголовке, так что все понимают, где какой кадр.

На рис. 2.3 мы видим случай (отчасти патологический) послышки двух сообщений 1 и 2 с машины А на машину В. Сообщение с данными 0 посылается с машины А в момент времени 0. Когда в момент времени 1 это сообщение достигает машины В, обнаруживается, что оно повреждено помехами в линии передачи и контрольная сумма не соответствует действительности. Машина В обнаруживает это и в момент времени 2 запрашивает повторную посылку, отправляя соответствующее контрольное сообщение 0. К сожалению, в тот же самый момент машина А посылает сообщение 1. В ответ на полученный машиной А запрос на повторную посылку отправляется сообщение 0. Когда машина В получает сообщение с данными 1 (а ведь она запрашивала сообщение с данными 0!), она посылает машине А новое контрольное сообщение 1, настаивая на том, что она хочет получить сообщение с данными 0, а не 1. Когда контрольное сообщение 1 доходит до машины А, та посылает сообщение с данными 0 в третий раз.

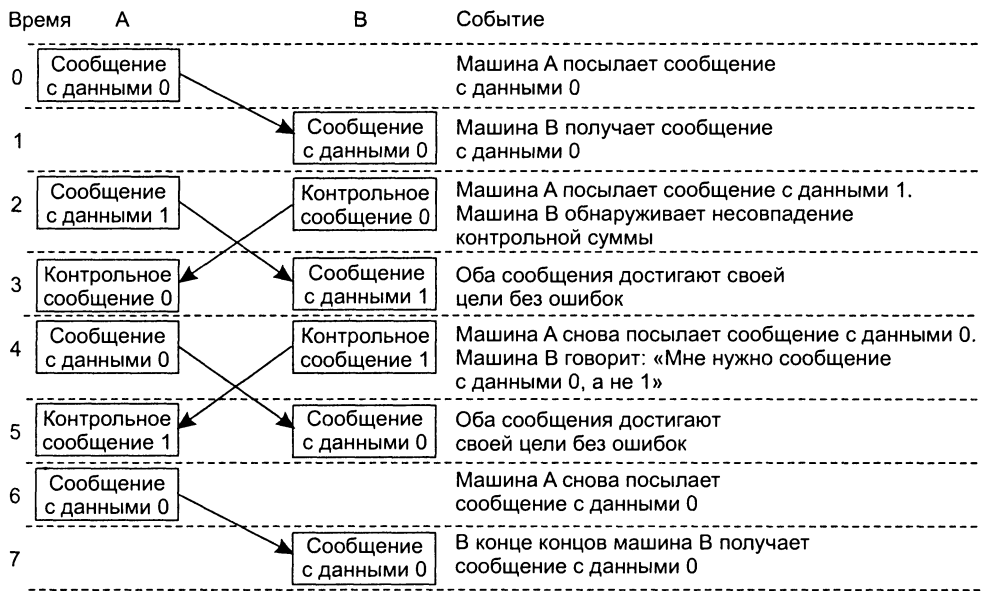


Рис. 2.3. Обмен между получателем и отправителем канального уровня

Задача этого обсуждения — не показать, насколько хорош описанный протокол (он очень плох), а объяснить, что на каждом уровне существует необходимость во взаимодействии между отправителем и получателем. Типичные сообщения: «Пошли, пожалуйста, сообщение *n* заново» — «Уже посылал дважды» —

«Нет, ты этого не делал» — «А я говорю, делал» — «Ну ладно, пускай делал, все равно пошли еще раз» и т. д. Это взаимодействие происходит в полях заголовка, где определены различные запросы и ответы и могут поддерживаться параметры (например, номера кадров).

Сетевой уровень

В локальных сетях у отправителя обычно нет необходимости находить местоположение получателя. Он просто бросает сообщение в локальную сеть, а получатель забирает его оттуда. Глобальные сети, однако, содержат множество машин, каждая из которых имеет собственные линии связи с другими машинами. Так на крупномасштабной карте показано множество городов и соединяющих их дорог. Сообщение, посылаемое от отправителя к получателю, должно пройти массу сетевых сегментов, на каждом из которых происходит выбор исходящей линии. Задача выбора наилучшего пути называется *маршрутизацией (routing)* и является основной задачей сетевого уровня.

Проблема усложняется тем, что наиболее короткий путь не всегда является наилучшим. На самом деле важна величина задержки на выбранном маршруте. Она, в свою очередь, зависит от объема трафика и числа сообщений, стоящих в очереди на отправку по различным линиям. С течением времени задержка может меняться. Некоторые алгоритмы маршрутизации могут подстраиваться под изменения загруженности линий, некоторые же удовлетворяются тем, что принимают решение на основе усредненных значений.

В настоящее время, вероятно, наиболее широко распространенным сетевым протоколом является не требующий установки соединения *протокол Интернета (Internet protocol, IP)*, входящий в комплект протоколов Интернета. На сетевом уровне сообщение именуется термином *пакет (packet)*. IP-пакет может быть послан без какой-либо предварительной подготовки. Маршрут каждого из IP-пакетов до места назначения выбирается независимо от других пакетов. Никакие внутренние пути не выбираются заранее и не запоминаются.

Протоколом с соединением, приобретающим популярность в настоящее время, является *виртуальный канал (virtual channel)* на базе сетей АТМ. Виртуальный канал в АТМ — это не прямое соединение, устанавливаемое от источника к приемнику, возможно, проходящее через несколько промежуточных АТМ-коммутаторов. Чтобы между двумя компьютерами не устанавливать каждый из виртуальных каналов по отдельности, набор виртуальных каналов может быть сгруппирован в *виртуальный путь (virtual path)*. Виртуальный путь сравним с предопределенным маршрутом между двумя узлами, вдоль которого выстраиваются все его виртуальные каналы. Изменение маршрута виртуального пути означает автоматическую переключку всех ассоциированных с ним каналов [194].

2.1.2. Транспортные протоколы

Транспортный уровень — это последняя часть того, что называют базовым стеком сетевых протоколов, поскольку в нем реализованы все службы, которые необходимы для построения сетевых приложений и которые не вошли в интерфейс се-

тевого уровня. Другими словами, транспортный уровень дает возможность разработчикам приложений использовать базовую сеть, лежащую в его основе.

Функции транспортного уровня

На пути от отправителя к получателю пакеты могут теряться. В то время как одни приложения задействуют собственные процедуры исправления ошибок, другие предпочитают надежную связь. Предоставить им эту службу — дело транспортного уровня. Идея состоит в том, что приложение должно быть в состоянии передать сообщение транспортному уровню и ожидать того, что оно будет доставлено без потерь.

После получения сообщения с прикладного уровня транспортный уровень разбивает его для успешной передачи на достаточно мелкие части, присваивает им последовательные номера и пересылает их. Взаимодействие на уровне заголовка транспортного уровня сводится к обсуждению того, какой пакет был послан, какой — принят, сколько места есть у адресата для приема дальнейших сообщений, что следует послать повторно и тому подобным вопросам.

Надежное транспортное соединение (которое по определению представляет собой связь с установкой соединения) можно построить поверх сетевых служб как с соединениями, так и без соединений. В первом случае все пакеты будут доставлены в правильной последовательности (если они посылаются одновременно), а в последнем возможно, что один из пакетов пойдет по другому маршруту и придет раньше, чем пакет, посланный до него. Это побуждает программное обеспечение транспортного уровня складывать пакеты в правильной последовательности, чтобы поддержать представление о транспортном соединении как о большой трубе — вы кладете в него сообщения на одном конце, и они добираются до другого неповрежденными, в том же порядке, в котором и отправлялись.

Транспортный протокол для Интернета называется *протоколом управления передачей* (*Transmission Control Protocol, TCP*). Он детально разобран в книге [109]. Комбинация TCP/IP в настоящее время является стандартом де-факто при сетевых взаимодействиях. Комплект протоколов Интернета также включает в себя не требующий соединения транспортный протокол под названием *UDP* (*Universal Datagram Protocol — универсальный протокол датаграмм*), который, по сути, представляет собой IP с некоторыми небольшими дополнениями. Пользовательские программы, не нуждающиеся в протоколе с соединениями, обычно используют UDP.

Официальный транспортный протокол ISO имеет пять разновидностей — от TP0 до TP4. Различия относятся к обработке ошибок и к возможности работать с несколькими транспортными соединениями на базе одного соединения низкого уровня (особенно X.25). Выбор того, какой из них использовать, зависит от свойств лежащего ниже сетевого уровня. Никогда ни один из них не должен перегружаться.

Время от времени предлагаются дополнительные транспортные протоколы. Так, например, для поддержки передачи данных в реальном времени был определен *транспортный протокол реального времени* (*Real-time Transport Protocol, RTP*). RTP — это кадровый протокол, который определяет формат пакета для

данных реального времени, ничего не говоря о механизмах гарантированной доставки этих данных. Кроме того, в нем определен протокол для мониторинга и управления передачей RTP-пакетов [406].

ТСР в архитектуре клиент-сервер

Взаимодействие клиент-сервер в распределенных системах часто осуществляется путем использования транспортного протокола базовой сети. С ростом популярности Интернета нередко можно наблюдать построение приложений клиент-сервер и систем на базе ТСР. Преимущества протокола ТСР по сравнению с UDP состоят в том, что он надежно работает в любой сети. Очевидная обратная сторона — протокол ТСР создает значительную дополнительную нагрузку на сеть, особенно в тех случаях, когда базовая сеть высоконадежна, например, в локальных сетях.

Когда на карту поставлены производительность и надежность, альтернативой всегда является протокол UDP в сочетании с дополнительными процедурами контроля ошибок и потоков, оптимизированными для конкретного приложения. Обратная сторона этого подхода состоит в том, что приходится проделывать массу дополнительной работы, а также в том, что полученное решение будет частным, снижающим открытость системы.

Протокол ТСР во многих случаях непривлекателен из-за невозможности приспособить его для поддержки синхронного поведения запрос-ответ, используемого во многих взаимодействиях клиент-сервер. На рис. 2.4, а показано применение протокола ТСР для поддержки взаимодействия клиент-сервер в нормальных условиях (когда сообщения не пропадают). Сначала клиент инициирует установление соединения, которое происходит по трехэтапному протоколу установления связи (первые три сообщения на рисунке). Этот протокол необходим для достижения договоренности о порядке нумерации пакетов, пересылаемых через соединение (детали можно найти в [446]). Когда соединение установлено, клиент посылает свой запрос (сообщение 4), сопровождаемый пакетом, требующим от сервера разорвать соединение (сообщение 5).

Сервер отвечает немедленным подтверждением приема запроса от клиента, которое скомпоновано с подтверждением разрыва соединения (сообщение 6). Затем сервер выполняет требуемую работу и посылает клиенту ответ (сообщение 7), также сопровождаемый требованием разорвать соединение (сообщение 8). Клиент должен только послать подтверждение разрыва связи на сервер (сообщение 9).

Ясно, что большая часть дополнительной нагрузки на сеть связана с управлением соединением. При использовании ТСР для управления взаимодействием клиент-сервер гораздо дешевле будет скомбинировать установление соединения с немедленной посылкой запроса, а посылку ответа — с разрывом соединения. Получившийся протокол носит название *ТСР для транзакций* (*TCP for Transactions*), сокращаемое до *T/TCP*. То, как функционирует этот протокол в нормальных условиях, можно увидеть на рис. 2.4, б.

В нормальной обстановке происходит следующее: клиент посылает одно сообщение (сообщение 1), содержащее три порции информации: запрос на уста-

новление соединения, собственно запрос к серверу и запрос, сообщающий серверу, что после этого он может немедленно разорвать соединение.

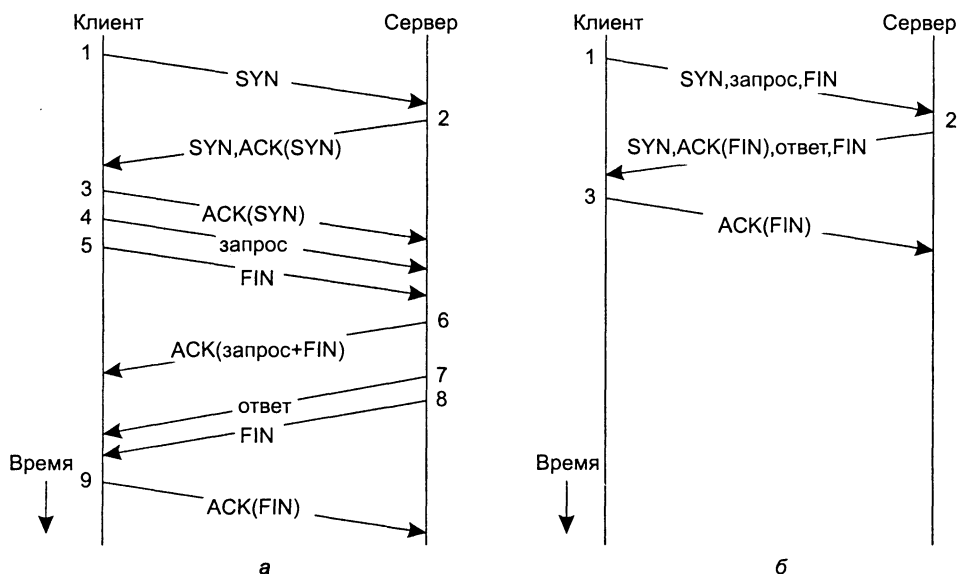


Рис. 2.4. Нормальное функционирование протокола TCP (а).
Функционирование протокола TCP для транзакций (б)

Сервер отвечает только после того, как обработает запрос. Он может послать данные, для передачи которых и создавалось соединение, и немедленно потребовать его разрыва, что иллюстрирует сообщение 2. После этого клиенту остается только подтвердить разрыв соединения (сообщение 3).

Протокол T/TCP, созданный как расширение TCP, автоматически преобразуется в нормальный протокол TCP, если другая сторона не поддерживает это расширение. Дополнительную информацию по TCP/IP можно найти в [437].

2.1.3. Протоколы верхнего уровня

Поверх транспортного уровня OSI указывает на наличие трех дополнительных уровней. На практике используется только прикладной уровень. На самом деле в комплекте протоколов Интернета все, что находится выше транспортного уровня, собирается в одну «кучу». В этом пункте мы увидим, почему с точки зрения систем промежуточного уровня нас не устраивает ни подход OSI, ни подход Интернета.

Сеансовые протоколы и протоколы представления

Сеансовый уровень представляет собой фактически расширенную версию транспортного уровня. Он обеспечивает управление диалогом, отслеживая и запоминая, какая сторона говорит в настоящий момент, и предоставляет средства син-

хронизации. Последние требуются для создания пользователями контрольных точек при длинных сеансах передачи данных, а также уведомления их о сбое в ходе такого сеанса. При этом необходимо сделать откат только до последней контрольной точки и не нужно проходить весь путь сначала. На практике сеансовый уровень нужен немногим приложениям и поддерживается редко. Он даже не входит в комплект протоколов Интернета.

В отличие от предыдущих уровней, на которых мы заботились о точной и эффективной пересылке битов от отправителя к получателю, уровень представления занимается смыслом этих битов. Большинство сообщений содержат не случайные последовательности битов, а структурированную информацию типа фамилий, адресов, денежных сумм и т. п. На уровне представления можно определить записи, содержащие подобного рода поля, и потребовать у отправителя уведомлять получателя, что сообщение содержит отдельные записи соответствующего формата. Это упрощает взаимодействие между машинами с различным внутренним представлением данных.

Прикладные протоколы

Прикладной уровень модели OSI изначально должен был содержать набор стандартных сетевых приложений, например для работы с электронной почтой, передачи файлов и эмуляции терминала. В настоящее время он стал местом собрания всех приложений и протоколов, которые не удалось пристроить ни на один из более низких уровней. В свете эталонной модели OSI все распределенные системы являются просто приложениями.

Чего в этой модели нет — так это четкого разграничения приложений, специальных протоколов приложений и протоколов общего назначения. Так, например, популярный в Интернете *протокол передачи файлов (File Transfer Protocol, FTP)* [203, 361] определяет передачу файлов между клиентской машиной и сервером. Этот протокол не следует путать с программой ftp, которая представляет собой законченное приложение для передачи файлов и совпадает (но не полностью) с реализацией протокола FTP для Интернета.

Другим примером сугубо специального прикладного протокола [142] может служить *протокол передачи гипертекста (Hypertext Transfer Protocol, HTTP)*, разработанный для удаленного управления и загрузки web-страниц. Протокол реализован в таких приложениях, как web-браузеры и web-серверы. Однако сейчас этот протокол используется и в системах, связь которых с Web не предполагается. Так, например, в RMI в языке Java протокол HTTP используется для обращения к удаленному объекту, защищенному брандмауэром [441].

Кроме того, существует множество протоколов общего назначения, используемых в различных приложениях. Они попали в прикладные протоколы, потому что их нельзя было отнести к транспортным. Во многих случаях они могут считаться протоколами промежуточного уровня, которые мы сейчас рассмотрим.

Протоколы промежуточного уровня

К промежуточному уровню относятся приложения, логически помещаемые на прикладной уровень, но содержащие множество протоколов общего назначения,

что дает им право на их собственный уровень, независимый от других, более специализированных приложений. Можно отделить высокоуровневые протоколы взаимодействия от протоколов для предоставления различных служб промежуточного уровня.

Существует множество протоколов, которые поддерживают разнообразные службы промежуточного уровня. Так, например, в главе 8 мы обсуждаем разнообразные методы аутентификации, то есть методы удостоверения личности. Протоколы аутентификации не привязаны к какому-либо конкретному приложению. Вместо этого они встраиваются в системы промежуточного уровня на правах общедоступной службы. Протоколы авторизации, согласно которым подтвердившие свой статус пользователи и процессы получают доступ только к тем ресурсам, на которые они имеют право, также имеют тенденцию к переходу на независимый от приложений уровень.

В качестве другого примера рассмотрим множество протоколов распределенного подтверждения (commit) из главы 7. Протоколы подтверждения делают так, чтобы в группе процессов либо все процессы прошли через определенную операцию, либо операция не была применена ни к одному из них. Это явление, известное также как *атомарность*, широко используется при транзакциях. Как мы увидим, не только транзакции, но и другие приложения, особенно рассчитанные на устойчивость к сбоям, также могут нуждаться в протоколах распределенного подтверждения.

В качестве последнего примера рассмотрим протоколы распределенной блокировки, при помощи которых может быть предотвращен одновременный доступ к ресурсам нескольких процессов, распределенных по множеству машин. Мы рассмотрим некоторые из этих протоколов в главе 5. Это еще один пример протоколов, которые могут быть реализованы в виде общедоступной службы промежуточного уровня, который в то же время не зависит от какого-либо конкретного приложения.

Коммуникационные протоколы промежуточного уровня поддерживают высокоуровневые коммуникационные службы. Так, например, в следующих двух пунктах мы обсудим протоколы, которые позволяют процессам прозрачно вызывать процедуры или обращаться к объектам на удаленных машинах. Существуют также коммуникационные службы высокого уровня для запуска и синхронизации потоков данных в реальном времени, что характерно, например, для мультимедийных приложений. В качестве последнего примера приведем предоставляемые некоторыми системами промежуточного уровня надежные службы групповой рассылки, способные поддерживать тысячи получателей, разбросанных по глобальной сети.

Некоторые из коммуникационных протоколов промежуточного уровня могут быть помещены и на транспортный уровень, но для их «повышения» имеются особые причины. Так, например, службы надежной групповой рассылки с гарантированной масштабируемостью могут быть реализованы, только если принимаются во внимание требования к приложению. Соответственно, системы промежуточного уровня могут предоставлять те или иные протоколы (настраиваемые), каждый из которых реализует различные транспортные протоколы, но предоставляет одинаковый интерфейс.

Такой подход к подразделению на уровни приводит нас к слегка измененной эталонной модели взаимодействия (рис. 2.5). По сравнению с моделью OSI сеансовый уровень и уровень представления заменены одним промежуточным уровнем, который содержит не зависящие от приложений протоколы. Как мы говорили, эти протоколы нельзя поместить на более низкие уровни. Истинные транспортные службы также могут быть представлены в виде служб промежуточного уровня. Для этого не потребуются даже модификации. Этот подход аналогичен переносу UDP на транспортный уровень. Точно так же службы промежуточного уровня могут включать в себя службы пересылки сообщений, схожие с теми, которые предоставляются на транспортном уровне.

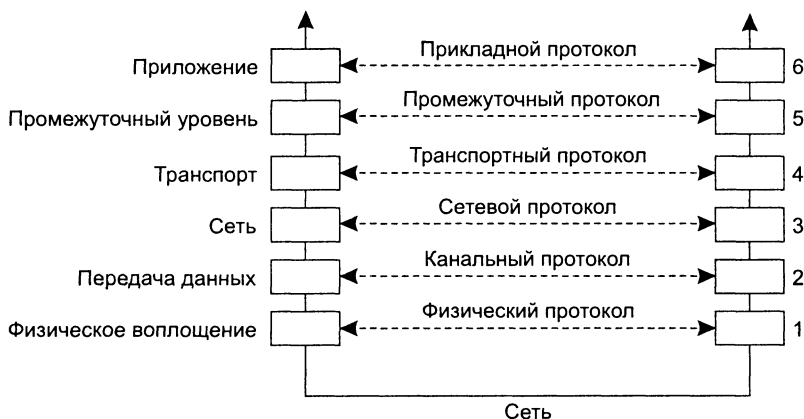


Рис. 2.5. Измененная эталонная модель сетевого взаимодействия

В оставшейся части этой главы мы сосредоточимся на четырех высокоуровневых коммуникационных службах промежуточного уровня: удаленном вызове процедур, удаленном обращении к объектам, очередям сообщений и потокам данных.

2.2. Удаленный вызов процедур

Основой множества распределенных систем является явный обмен сообщениями между процессами. Однако процедуры `send` и `receive` не скрывают взаимодействия, что необходимо для обеспечения прозрачности доступа. Эта проблема была известна давно, но по ней мало что было сделано до появления в 1980 году статьи [62], в которой предлагался абсолютно новый способ взаимодействия. Хотя идея была совершенно простой (естественно, после того как кто-то все придумал), ее реализация часто оказывается весьма хитроумной. В этом разделе мы рассмотрим саму концепцию, ее реализацию, сильные и слабые стороны.

Если не вдаваться в подробности, в упомянутой статье было предложено позволить программам вызывать процедуры, находящиеся на других машинах. Когда процесс, запущенный на машине *A*, вызывает процедуру с машины *B*, вызываю-

щий процесс на машине *A* приостанавливается, а выполнение вызванной процедуры происходит на машине *B*. Информация может быть передана от вызывающего процесса к вызываемой процедуре через параметры и возвращена процессу в виде результата выполнения процедуры. Программист абсолютно ничего не заметит. Этот метод известен под названием *удаленный вызов процедур (Remote Procedure Call, RPC)*.

Базовая идея проста и элегантна, сложности возникают при реализации. Для начала, поскольку вызывающий процесс и вызываемая процедура находятся на разных машинах, они выполняются в различных адресных пространствах, что тут же рождает проблемы. Параметры и результаты также передаются от машины к машине, что может вызвать свои затруднения, особенно если машины не одинаковы. Наконец, обе машины могут сбоить, и любой возможный сбой вызовет разнообразные сложности. Однако с большинством из этих проблем можно справиться, и RPC является широко распространенной технологией, на которой базируются многие распределенные системы.

2.2.1. Базовые операции RPC

Мы начнем с обсуждения общепринятых вызовов процедур, а затем рассмотрим, как вызов может быть распределен между клиентской и серверной частями системы, каждая из которых выполняется на различных машинах.

Общепринятые вызовы процедур

Для того чтобы понять, как работает RPC, важно сначала разобраться, как осуществляются общепринятые (в пределах одной машины) вызовы процедур. Рассмотрим вызов в языке типа C:

```
count = read(fd, buf, bytes);
```

Здесь *fd* — целое, указывающее на файл; *buf* — массив символов, в который будут считываться данные; *bytes* — другое целое, говорящее, сколько байт следует считать. Если вызов производится из главной программы, стек до вызова выглядит так, как показано на рис. 2.6, *а*. Делая вызов, вызывающая программа помещает параметры в стек в порядке «последний первым», как показано на рис. 2.6, *б*. Причина, по которой компилятор C помещает параметры в стек в обратном порядке, заключается в функциях типа *printf*. Чтобы выполняться, функция *printf* всегда должна знать, где искать свой первый аргумент — строку формата. После завершения *read* система помещает возвращаемое значение в регистр, удаляет адрес возврата и возвращает управление назад, в вызвавшую программу. Затем вызвавшая программа удаляет из стека параметры, приводя его в исходное состояние.

Следует особо отметить некоторые моменты. Во-первых, в C параметры могут передаваться как по значению, так и по ссылке. Параметр-значение, такой как *fd* или *bytes*, просто копируется в стек, что и показано на рис. 2.6, *б*. Для вызываемой процедуры параметр-значение представляет собой просто инициализированную локальную переменную. Вызываемая процедура может ее изменить,

но все эти изменения не могут повлиять на ее оригинальное значение на вызывающей стороне.

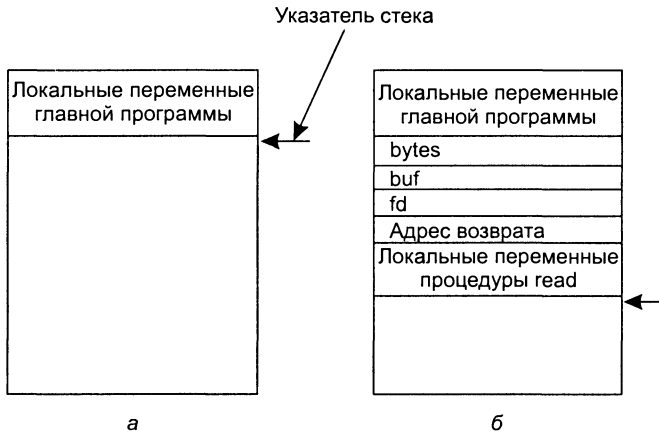


Рис. 2.6. Передача параметров при локальном вызове процедуры: стек до вызова функции read (а). Стек во время работы вызванной процедуры (б)

Параметр-ссылка в С — это указатель на переменную (то есть адрес переменной), а не ее значение. В вызове read второй параметр является параметром-ссылкой, поскольку массивы в С всегда передаются через ссылку. В стек на самом деле будет помещен адрес символьного массива. Если вызванная процедура использует этот параметр для помещения чего-либо в символьный массив, это приведет к изменению массива в вызывающей программе. Мы увидим, что разница между параметрами-значениями и параметрами-ссылками крайне важна для RPC.

Существует еще один механизм передачи параметров, не применяющийся в С. Он называется *вызов через копирование/восстановление* (*call-by-copy/restore*). В этом случае до вызова вызывающей программой производится копирование переменной в стек, как при вызове по значению, а после завершения вызова — копирование этой переменной из стека назад, с удалением исходного значения этой переменной. В большинстве случаев это дает тот же эффект, что и при вызове по ссылке. Однако иногда, например, если один и тот же параметр присутствует в списке аргументов многократно, их семантика различается. Во многих языках вызов через копирование/восстановление отсутствует.

Решение об использовании того или иного механизма передачи параметров принимается обычно разработчиками языка и является его фиксированным свойством. Раньше это решение зависело от типов передаваемых данных. В С, например, как мы видели, целые и другие скалярные типы всегда передаются по значению, а массивы — по ссылке. Некоторые компиляторы языка Ада поддерживают вызов через копирование/восстановление для *изменяемых* (*in out*) параметров, передавая остальные по ссылке. Определение языка позволяет выбрать любой вариант, который хоть немного упрощает семантику.

Заглушки для клиента и сервера

Идея, стоящая за RPC, состоит в том, чтобы удаленный вызов процедур выглядел точно так же, как и локальный. Другими словами, мы ждем от RPC прозрачности — вызываемая процедура не должна уведомляться о том, что вызываемая процедура выполняется на другой машине, и наоборот. Предположим, программа хочет считать некоторые данные из файла. Для чтения из файла необходимых данных программист помещает в код вызов `read`. В традиционной (однопроцессорной) системе процедура `read` извлекается компоновщиком из библиотеки и вставляется в объектный код программы. Это короткая процедура, которая обычно реализуется путем системного вызова `read`. Другими словами, процедура `read` — это интерфейс между кодом пользователя и локальной операционной системой.

Даже если `read` — это системный вызов, он производится аналогичным образом, путем помещения параметров в стек, как показано на рис. 2.6, б. Таким образом, программист так и не узнает, что `read` делает что-то хитрое.

RPC организует свою прозрачность аналогичным образом. Если `read` является удаленной процедурой (то есть будет исполняться на машине файлового сервера), в библиотеку помещается специальная версия `read`, называемая *клиентской заглушкой* (*client stub*). Как и оригинальная функция, она также вызывается в соответствии с последовательностью, показанной на рис. 2.6, б. Как и оригинал, она также производит вызов локальной операционной системы, только в отличие от оригинальной функции клиентская заглушка не запрашивает данные у операционной системы. Вместо этого она упаковывает параметры в сообщение и путем вызова процедуры `send` требует переслать это сообщение на сервер, как показано на рис. 2.7. После вызова процедуры `send` клиентская заглушка вызывает процедуру `receive`, блокируясь до получения ответа.

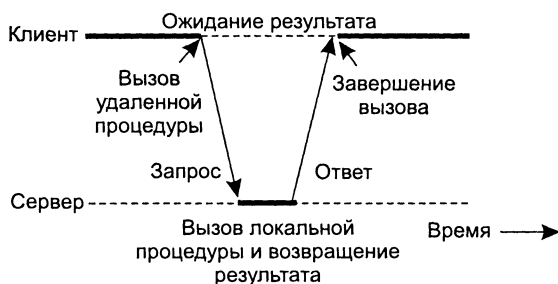


Рис. 2.7. Схема RPC между программами клиента и сервера

Когда сообщение приходит на сервер, операционная система сервера передает его *серверной заглушке* (*server stub*). Серверная заглушка эквивалентна клиентской, но работает на стороне сервера. Это фрагмент кода, который преобразует приходящие по сети запросы в вызовы локальных процедур. Обычно серверная заглушка запускает процедуру `receive` и блокируется, ожидая входящих сообщений. После получения сообщения серверная заглушка распаковывает его, извлекая параметры, и традиционным способом (то есть так, как показано на рис. 2.6)

вызывает процедуру сервера. С точки зрения сервера это воспринимается как прямой вызов с клиента — параметры и адрес возврата находятся в стеке, где они и должны находиться, и ничего необычного в этом нет. Сервер делает свое дело и обычным порядком возвращает результат вызвавшей процедуре. Так, например, в случае чтения файла сервер заполняет данными буфер, на который указывает второй параметр. Этот буфер — внутренний буфер серверной заглушки.

Когда серверная заглушка после окончания обработки вызова возвращает управление вызвавшей программе, она запаковывает результаты выполнения (буфер) в сообщение и вызывает процедуру `send`, чтобы вернуть их клиенту. После этого серверная заглушка вновь вызывает процедуру `receive`, переходя в режим ожидания следующего сообщения.

Когда на клиентскую машину приходит ответное сообщение, операционная система клиента обнаруживает, что оно адресовано клиентскому процессу (на самом деле клиентской заглушке, но операционная система их не различает). Сообщение копируется в буфер ожидания, и клиентский процесс разблокируется. Клиентская заглушка рассматривает сообщение, распаковывает его, извлекая результаты, копирует их в память вызвавшей программы и, завершая работу, передает в нее код возврата. Когда вызвавшая программа получает управление после вызова `read`, все, что она знает, — это то, что запрошенные данные находятся там, где и предполагалось, то есть в буфере. У нее нет никакого представления о том, как осуществлялся вызов — удаленно или в рамках локальной операционной системы.

В блаженстве неведения, царящем на стороне клиента, и состоит прелесть такой схемы. Как мы видели, доступ к удаленным службам осуществляется посредством вызова обычных (то есть локальных) процедур, без всяких там `send` и `receive`. Все детали пересылки сообщений скрыты в двух библиотечных процедурах, так же как в традиционных библиотеках скрыты детали реально производимых системных вызовов.

Подведем итоги. При удаленном вызове процедур происходят следующие действия.

1. Процедура клиента обычным образом вызывает клиентскую заглушку.
2. Клиентская заглушка создает сообщение и вызывает локальную операционную систему.
3. Операционная система клиента пересылает сообщение удаленной операционной системе.
4. Удаленная операционная система передает сообщение серверной заглушке.
5. Серверная заглушка извлекает из сообщения параметры и вызывает сервер.
6. Сервер выполняет вызов и возвращает результаты заглушке.
7. Серверная заглушка запаковывает результаты в сообщение и вызывает свою локальную операционную систему.
8. Операционная система сервера пересылает сообщение операционной системе клиента.

9. Операционная система клиента принимает сообщение и передает его клиентской заглушке.
10. Заглушка извлекает результаты из сообщения и передает их клиенту.

Сетевые эффекты этих шагов состоят в том, что клиентская заглушка превращает локальный вызов процедуры клиента в локальный вызов процедуры сервера, причем ни клиент, ни сервер ничего не знают о промежуточных действиях.

2.2.2. Передача параметров

Назначение клиентской заглушки состоит в том, чтобы получить параметры, упаковать их в сообщение и послать его серверной заглушке. Хотя эти действия выглядят несложно, они не так просты, как кажется на первый взгляд. В этом пункте мы рассмотрим некоторые вопросы, связанные с передачей параметров в системах RPC.

Передача параметров по значению

Упаковка параметров в сообщение носит название *маршалинга параметров* (*parameter marshaling*). В качестве простейшего примера рассмотрим удаленную процедуру, `add(i, j)`, которая использует два целых параметра, `i` и `j`, и возвращает в результате их арифметическую сумму. (На практике так не делается, поскольку удаленная реализация столь простой процедуры крайне невыгодна, но для примера сойдет.) Вызов `add` иллюстрируется левой частью рис. 2.8 (в клиентском процессе). Клиентская заглушка извлекает два ее параметра и, как показано на рисунке, упаковывает их в сообщение. Она также помещает туда имя или номер вызываемой в сообщении процедуры, поскольку сервер может поддерживать несколько разных процедур и ему следует указать, какая из них потребовалась в данном случае.

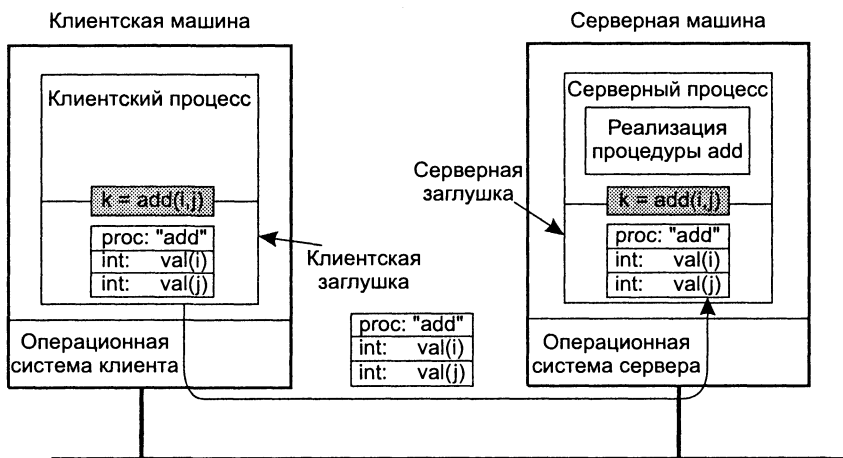


Рис. 2.8. Процесс удаленных вычислений с использованием RPC

Когда сообщение приходит на сервер, заглушка исследует сообщение в поисках указания на то, какую процедуру следует вызвать, а затем делает соответствующий вызов. Если сервер поддерживает и другие удаленные процедуры, серверная заглушка должна содержать инструкцию типа `switch` для выбора вызываемой процедуры в зависимости от первого поля сообщения. Реальный вызов процедуры сервера из серверной заглушки выглядит почти как первоначальный клиентский вызов, если не считать того, что параметрами являются переменные, инициализированные значениями, взятыми из сообщения.

Таким образом, имеет место следующая пошаговая процедура.

1. Клиент вызывает процедуру `add`.
2. Клиентская заглушка строит сообщение.
3. Сообщение отправляется по сети на сервер.
4. Операционная система сервера передает сообщение серверной заглушке.
5. Серверная заглушка распаковывает сообщение.
6. Серверная заглушка выполняет локальный вызов процедуры `add`.

Когда сервер заканчивает работу, управление вновь передается серверной заглушке. Она получает результат, переданный сервером, и запаковывает его в сообщение. Это сообщение отправляется назад, к клиентской заглушке, которая распаковывает его и возвращает полученное значение клиентской процедуре.

До тех пор пока машины клиента и сервера идентичны, а все параметры и результаты имеют скалярный тип (то есть целый, символьный или логический), эта модель работает абсолютно правильно. Однако в больших распределенных системах обычно присутствуют машины разных типов. Каждая из машин часто имеет собственное представление чисел, символов и других элементов данных. Так, в мэйнфреймах IBM используется кодовая таблица EBCDIC, а в персональных компьютерах той же фирмы — ASCII. Вследствие этого, если передать символьный параметр с клиента на базе IBM PC на мэйнфрейм IBM, используемый в качестве сервера, по простейшей схеме, показанной на рис. 2.8, сервер поймет эти символы неправильно.

Сходные проблемы могут обнаружиться при передаче целых чисел (знаковый или значащий старший бит) и чисел с плавающей точкой. Вдобавок существует значительно более серьезная проблема, состоящая в том, что в некоторых машинах, таких как Intel Pentium, байты нумеруются справа налево, а в других, например в Sun SPARC, — в обратном направлении. Формат компании Intel называется *остроконечным* (*little endian*), а формат SPARC — *тупоконечным* (*big endian*), по аналогии с названиями политических партий из книги «Путешествия Гулливера», которые в спорах о том, с какой стороны разбивать яйца, дошли до войны [108]. Для примера рассмотрим процедуру с двумя параметрами, целым числом и строкой из четырех символов. Для размещения каждого из параметров требуется одно 32-битное слово. На рис. 2.9, *a* показано, как будет выглядеть содержащий параметры фрагмент сообщения, построенного клиентской заглушкой, когда клиент работает на компьютере Intel Pentium. Первое слово содержит целый параметр, в данном случае 5, а второе слово — строку *JILL*.

0	3	0	2	0	1	0		0	5	0	2	0	3	0		0	0	0	2	3	5		
L	7	L	6	I	5	4	J	J	4	I	5	L	6	7	L	L	4	L	5	I	6	J	
а								б								в							

Рис. 2.9. Исходное сообщение, подготовленное на Pentium (а). Сообщение после получения на SPARC (б). Сообщение после инверсии (в). Цифры в квадратах показывают адрес каждого байта

Поскольку сообщение передается по сети байт за байтом (на самом деле бит за битом), первый посланный байт будет и первым принятым. На рис. 2.9, б мы видим, как будет выглядеть сообщение с рис. 2.9, а, принятое на компьютере SPARC. Нумерация байтов здесь такова, что нулевым байтом считается левый (верхний байт), а не правый (нижний байт), как в процессорах Intel. После того как серверная заглушка прочитает параметры по адресам 0 и 4, сервер получит, соответственно, целое число, равное $83\ 886\ 080\ (5 \times 2^{24})$, и строку *JILL*.

Очевидное, но, к сожалению, неверное решение — просто инвертировать байты каждого слова после того, как оно будет принято (рис. 2.9, в). Теперь целое число стало правильным, а строка превратилась в *LLIJ*. Проблема состоит в том, что целые нужно приводить к другому порядку следования байтов, а строки — нет. Не имея дополнительной информации о том, где строка, а где целое, мы не в состоянии исправить положение дел.

Передача параметров по ссылке

Теперь мы подошли к сложной проблеме: как передавать указатели или, в общем случае, ссылки? Общий ответ таков: с величайшим трудом. Мы помним, что указатель имеет смысл только в адресном пространстве того процесса, в котором он используется. Возвращаясь к нашему примеру с процедурой *read*, который мы обсуждали ранее, второй параметр (адрес буфера) для клиента может быть равен, например, 1000, но нельзя же просто передать на сервер число 1000 и ожидать, что это сработает. На сервере адрес 1000 вполне может прийти на середину текста программы.

Одно из решений состоит в том, чтобы вообще забыть про указатели и ссылки в качестве параметров. Однако важность таких параметров делает такое решение абсолютно неподходящим. На самом деле в нем нет особой необходимости. В примере с процедурой *read* клиентской заглушке известно, что второй параметр указывает на массив символов. Предположим на минуту, что заглушка также знает и величину этого массива. После этого вырисовывается следующая стратегия: скопировать этот массив в сообщение и передать его на сервер. Серверная заглушка может после этого вызвать сервер, передав ему указатель на этот массив, даже если числовое значение этого указателя будет отличаться от переданного во втором параметре вызова процедуры *read*. Изменения, которые с помощью указателя сделает сервер (то есть по указанному адресу запишет данные), прямо отразятся на буфере сообщения серверной заглушки. Когда сервер закон-

чит работу, оригинальное сообщение будет отослано назад, клиентской заглушке, которая скопирует буфер клиенту. В результате вызов по ссылке будет подменен копированием/восстановлением. Несмотря на то что это не одно и то же, часто такой замены вполне достаточно.

Небольшая оптимизация позволяет сделать этот механизм вдвое эффективнее. Если обеим заглушкам известно, входящим или исходящим параметром является буфер для сервера, то одну из копий можно удалить. Если массив используется сервером в качестве исходных данных (то есть при вызове `write`), то копировать его обратно не нужно. Если это результат, то нет необходимости изначально передавать его серверу.

В качестве последнего комментария отметим, что нет ничего особенно хорошего в том, что мы можем работать с указателями на простые массивы и структуры, если нам по-прежнему недоступна работа с более общими вариантами указателей — с указателями на произвольные структуры данных, например на сложные графы. В некоторых системах делается попытка решить эту проблему путем передачи серверной заглушке реальных указателей с последующей генерацией специального кода в процедурах сервера для работы с этими указателями. Так, для получения данных, которые соответствуют указателю, сервер может сделать специальный запрос.

Спецификация параметров и генерация заглушек

После всех этих объяснений становится ясно, что сокрытие механизма удаленного вызова процедур требует, чтобы вызывающая и вызываемая системы договорились о формате сообщений, которыми они обмениваются, и при необходимости пересылки, например, данных сложной структуры, следовали определенному порядку действий. Другими словами, при совершении RPC обе стороны должны следовать одному протоколу.

В качестве простого примера рассмотрим процедуру, показанную на рис. 2.10, а. Она имеет три параметра: символ, число с плавающей точкой и массив из пяти целых чисел. Предполагая, что длина слова составляет четыре байта, протокол RPC может предписать передачу символа в крайнем правом байте слова (оставляя последующие три пустыми), числа с плавающей точкой — в целом слове, а массива — в виде последовательности слов с общей длиной, равной длине массива, и предшествующим словом, содержащим длину последовательности (рис. 2.10, б). Если ввести подобные правила, то клиентская заглушка будет знать, что для процедуры `foobar` необходимо использовать тот формат, который представлен на рис. 2.10, б, а серверная заглушка — что именно такой формат будет иметь входящее сообщение для вызова процедуры `foobar`.

Определение формата сообщения — это только одна сторона протокола RPC. Этого недостаточно. Также нам необходимо, чтобы клиент и сервер пришли к договоренности по вопросу представления простых типов данных, таких как целые числа, символы, логические значения и т. д. Так, протокол может предписать, чтобы целые передавались без знака, символы в 16-битной кодировке Unicode, а числа с плавающей точкой — в формате стандарта IEEE 754, и все это — в окончательном формате. Только при наличии такой дополнительной информации сообщение может быть однозначно интерпретировано.

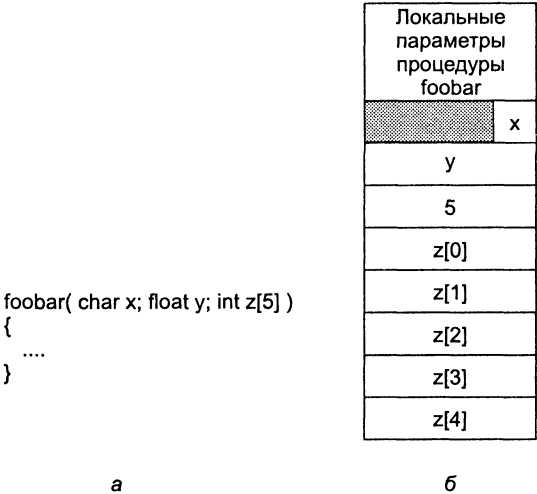


Рис. 2.10. Процедура (а) и соответствующее процедуре сообщение (б)

После того как все биты до последнего выстроены в ряд по согласованным правилам кодирования, осталось сделать только одно. Вызывающая и вызываемая системы должны договориться между собой об обмене реальными сообщениями. Например, они могут решить использовать транспортный протокол с соединениями, такой как TCP/IP. Альтернативой ему будет ненадежная служба дейтаграмм, в этом случае клиент и сервер должны включить реализацию схемы контроля ошибок в RPC. На практике возможны различные варианты.

После завершения определения протокола RPC необходимо реализовать заглушки — клиентскую и серверную. К счастью, заглушки, работающие по одному протоколу, для разных процедур различаются лишь интерфейсом с приложениями. Интерфейс состоит из набора процедур, которые могут быть вызваны клиентом, но реализуются на сервере. Доступ к интерфейсу осуществляется обычно из определенного языка программирования, одного из тех, на которых написан клиент или сервер (хотя, строго говоря, это не обязательно). Для упрощения работы интерфейсы часто описываются с использованием *языка определения интерфейсов (Interface Definition Language, IDL)*. Интерфейс, определенный на чем-то вроде IDL, компилируется затем в заглушки клиента и сервера, а также в соответствующие интерфейсы времени компиляции и времени выполнения.

Практика показывает, что использование языка определения интерфейсов делает приложения клиент-сервер, базирующиеся на RPC, существенно проще. Поскольку клиентская и серверная заглушки очень легко сгенерировать полностью автоматически, все системы промежуточного уровня, основанные на RPC, используют IDL для поддержки разработки программного обеспечения. В некоторых случаях применение IDL просто обязательно. Мы рассмотрим подобные случаи в следующих главах.

2.2.3. Расширенные модели RPC

Удаленные вызовы процедур стали фактическим стандартом для связи в распределенных системах. Популярность этой модели объясняется ее несомненной простотой. В этом пункте мы рассмотрим два расширения базовой модели RPC, созданные для разрешения некоторых ее недостатков.

Входы

Базовая модель RPC предполагает, что вызывающая и вызываемая системы могут связываться друг с другом для обмена сообщениями по сети. В общем случае это предположение истинно. Однако рассмотрим вариант, когда клиент и сервер установлены на одной машине. В стандартном случае мы должны использовать средства локального *межпроцессного взаимодействия* (*InterProcess Communication, IPC*), которые базовая операционная система предоставляет процессам, запущенным на одной машине. Так, например, в UNIX соответствующие средства включают в себя совместно используемую память, каналы и очереди сообщений (детальное обсуждение IPC в UNIX-системах можно найти в [439]).

Локальные средства IPC обычно значительно более эффективны, чем сетевые, даже если последние используются для связи между процессами на одной машине. Соответственно, если важна производительность, следует совмещать различные механизмы межпроцессного взаимодействия, руководствуясь тем, подходят ли процессы, в которых мы заинтересованы, на одной машине или нет.

В качестве компромисса некоторые операционные системы предоставляют процессам, размещенным на одной машине, эквивалент RPC под названием *входов* (*doors*). Вход — это обобщенное имя процедур, существующих в адресном пространстве процессов сервера, которые могут вызываться процессами, размещенными на одной с сервером машине. Входы впервые появились в операционной системе Spring [297] и были хорошо описаны в [193]. Сходный механизм, под названием *упрощенный вызов RPC* (*lightweight RPC*), описан в [49].

Вызов входов требует поддержки локальной операционной системы, как показано на рис. 2.11. Так, для того чтобы появилась возможность вызвать вход, процесс сервера должен зарегистрировать его. При регистрации входа возвращается его идентификатор, который впоследствии можно будет использовать в качестве символического имени входа. Регистрация заканчивается вызовом `door_create`. Доступ других процессов к зарегистрированному входу может осуществляться просто по тому идентификатору, который мы получили при регистрации входа. Так, например, в Solaris каждый вход имеет файловое имя, которое можно получить через идентификатор простым вызовом `fattach`. Клиент вызывает вход через системный вызов `door_call`, в который идентификатор входа передается так же, как и любой другой обязательный параметр. Затем операционная система производит вызов того процесса, который зарегистрировал вход. Результатом этого вызова будет вызов входа сервера. Результаты вызова входа будут возвращены в процесс клиента через системный вызов `door_return`.

Главное преимущество входов состоит в том, что они позволяют использовать для связи в распределенных системах единый механизм — вызовы процедур.

К сожалению, разработчики приложений часто нуждаются в сведениях о том, выполняется ли данный вызов в текущем процессе, в другом процессе на этой же машине или в удаленном процессе.

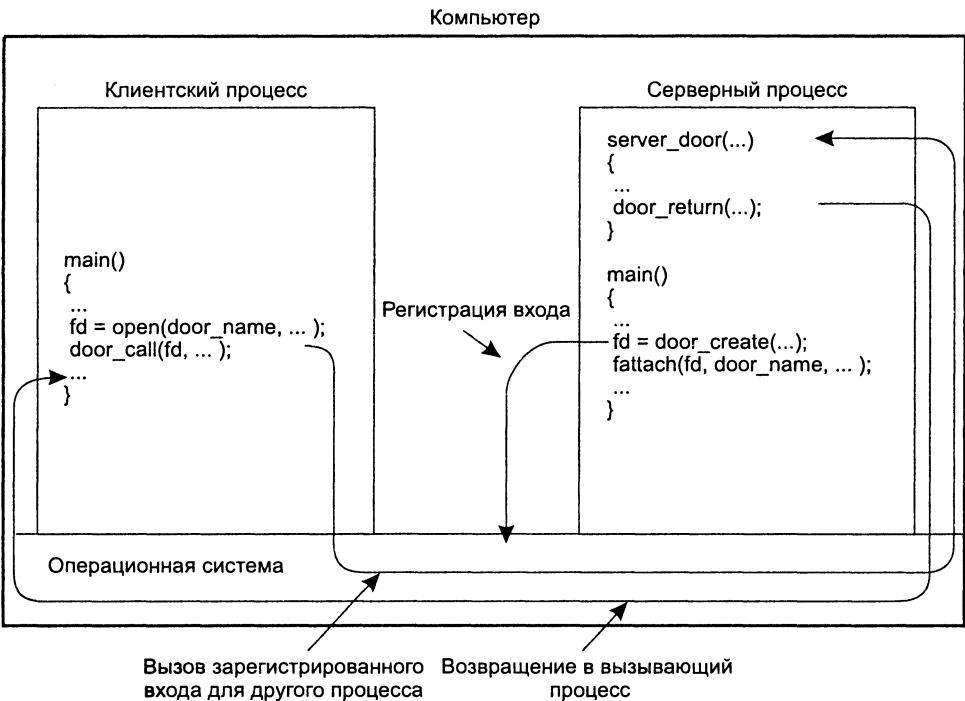


Рис. 2.11. Принципы использования входов в качестве механизма IPC

Асинхронный вызов RPC

В стандартном варианте вызова клиентом удаленной процедуры его работа приостанавливается до получения ответа. Когда ответ не нужен, этот жесткий алгоритм «запрос-ответ» не является необходимым, приводя только к блокированию клиента с невозможностью производить работу до получения ответа от удаленной процедуры. Примеры действий, при которых обычно нет необходимости в ожидании ответа: перечисление денег с одного банковского счета на другой, добавление записей в базу данных, запуск удаленной службы, пакетная обработка и множество других.

Для обработки подобных случаев системы RPC могут предоставлять средства для так называемого *асинхронного вызова RPC (asynchronous RPC)*. При помощи этих средств клиент получает возможность продолжить свою работу сразу после выполнения запроса RPC. При асинхронном вызове RPC сервер немедленно по приходу запроса отправляет клиенту ответ, после чего вызывает запрошенную процедуру. Ответ служит подтверждением того, что сервер приступил к обработке RPC. Клиент продолжает работу, снимая блокировку, сразу после получения

от сервера этого подтверждения. На рис. 2.12, *а* приведен стандартный алгоритм взаимодействия «запрос-ответ», а на рис. 2.12, *б* — алгоритм взаимодействия клиента и сервера в случае асинхронного вызова RPC.

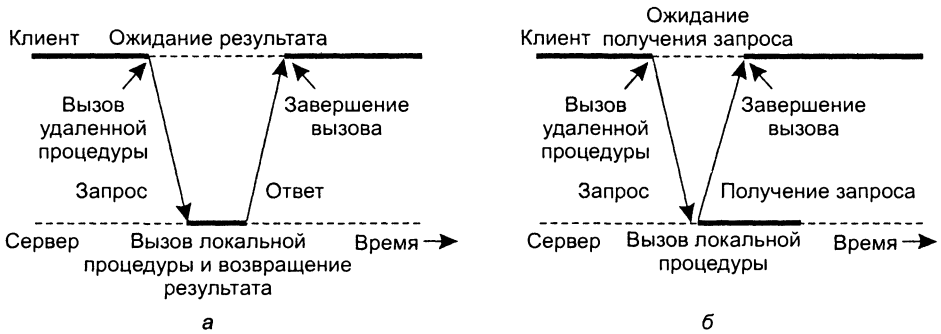


Рис. 2.12. Взаимодействие между клиентом и сервером в RPC традиционной схемы (а). Взаимодействие с использованием асинхронного вызова RPC (б)

Асинхронные вызовы RPC также могут быть полезны в тех случаях, когда ответ будет послан, но клиент не готов просто ждать его, ничего не делая. Например, клиент может пожелать заранее выбрать стевые адреса из набора хостов, с которыми вскоре будет связываться. В то время, пока служба именования соберет эти адреса, клиент может заняться другими вещами. В подобных случаях имеет смысл организовать сообщение между клиентом и сервером через два асинхронных вызова RPC, как это показано на рис. 2.13. Сначала клиент вызывает сервер, чтобы передать ему список имен хостов, который следует подготовить, и продолжает свою работу, когда сервер подтверждает получение этого списка. Второй вызов делает сервер, который вызывает клиента, чтобы передать ему найденные адреса. Комбинация из двух асинхронных вызовов RPC иногда называется также *отложенным синхронным вызовом RPC (deferred synchronous RPC)*.

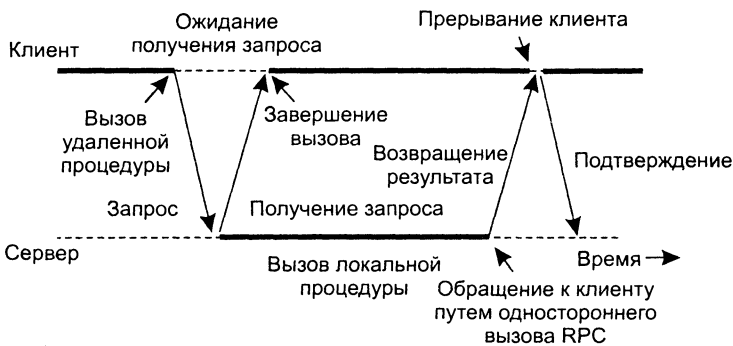


Рис. 2.13. Взаимодействие клиента и сервера посредством двух асинхронных вызовов RPC

Следует отдельно отметить вариант асинхронного вызова RPC, реализующегося в тех случаях, когда клиент продолжает работу немедленно после отправки запроса на сервер. Другими словами, клиент не ожидает от сервера подтверждения в получении запроса. Мы будем называть такие вызовы *односторонними вызовами RPC (one-way RPC)*. Проблема такого подхода состоит в том, что при отсутствии гарантий надежности клиент не может быть точно уверен, что его запрос будет выполнен. Мы вернемся к этому вопросу в главе 7.

2.2.4. Пример — DCE RPC

Механизм удаленных вызовов процедур был тщательно адаптирован для использования в качестве основы систем промежуточного уровня и вообще распределенных систем. В этом пункте мы рассмотрим одну из специальных систем RPC: *среду распределенных вычислений (Distributed Computing Environment, DCE)*, разработанную организацией OSF (Open Software Foundation), которая сейчас переименована в Open Group. Система DCE RPC не настолько популярна, как некоторые другие системы RPC, например Sun RPC. Однако DCE RPC — прекрасный представитель систем RPC. Спецификация DCE RPC адаптирована к системе распределенных вычислений на базе решений Microsoft. Кроме того, как мы увидим в следующем пункте, DCE RPC вдобавок иллюстрирует отношения между системами RPC и распределенными объектами. Мы начнем с краткого введения в DCE, за которым последует обсуждение принципов работы DCE RPC.

Знакомство с DCE

DCE — настоящая система промежуточного уровня, разработанная, чтобы абстрагировать существующие (сетевые) операционные системы от распределенных приложений. Изначально она была разработана под UNIX, однако в настоящее время существуют версии DCE для всех распространенных операционных систем, включая VMS и Windows NT, а также операционных систем настольных компьютеров. Идея состоит в том, что покупатель может взять набор компьютеров, поставить программное обеспечение DCE и начать запускать распределенные приложения, и все это без каких-либо неполадок в работе существующих (нераспределенных) приложений. Хотя большая часть пакета DCE работает в пространстве пользователя, в некоторых конфигурациях часть (отвечающая за распределенную файловую систему) может быть добавлена и к ядру. Сама по себе организация Open Group только продает исходные тексты, а поставщики встраивают их в свои системы.

Модель программирования, лежащая в основе всей системы DCE, — это модель клиент-сервер, широко обсуждавшаяся в предыдущей главе. Процессы пользователей действуют как клиенты, вызывающие удаленные службы, предоставляемые серверными процессами. Некоторые из этих служб являются составными частями DCE, другие же принадлежат к приложениям и написаны прикладными программистами. Вся связь между клиентами и серверами осуществляется посредством RPC.

Существуют службы, которые сами по себе образуют часть DCE. *Служба распределенных файлов (distributed file service)* представляет собой всемирную файловую систему, предоставляющую прозрачные методы доступа к любому файлу системы одинаковым образом. Она может быть построена поверх базовых файловых систем хостов или работать независимо от них. *Служба каталогов (directory service)* используется для отслеживания местонахождения любого из ресурсов системы. В число этих ресурсов входят машины, принтеры, серверы, данные и многое другое. Географически они могут быть распределены по всему миру. Служба каталогов позволяет процессу запрашивать ресурсы, не задумываясь о том, где они находятся, если это не необходимо для процесса. *Служба защиты (security service)* позволяет защищать ресурсы любого типа, кроме того, получение некоторых данных может быть открыто только тем, кому это разрешено. И наконец, *служба распределенного времени (distributed time service)* позволяет поддерживать глобальную синхронизацию часов различных машин. Как мы увидим в следующей главе, существование некоторого представления о глобальном времени сильно упрощает гарантию целостности при параллельной работе в распределенных системах.

Задачи DCE RPC

Задачи систем DCE RPC вполне традиционны. Они позволяют клиенту получить доступ к удаленной службе простым вызовом локальной процедуры. Этот интерфейс дает возможность писать клиентские (то есть прикладные) программы простым, хорошо знакомым большинству программистов способом. Также он упрощает запуск в распределенной среде больших объемов существующего кода с минимальными изменениями или без них.

Самое важное для системы RPC — это скрыть все возможные детали от клиента и до некоторой степени от сервера. Для начала система RPC может автоматически определить необходимый сервер и установить связь между клиентом и сервером. Обычно это называется *привязкой (binding)*. Кроме того, она может управлять транспортировкой сообщений в обе стороны, а также, если в этом есть необходимость, их дроблением и последующей сборкой, например, если один из параметров сообщения является большим массивом. И наконец, система RPC может автоматически отслеживать преобразование типов данных между клиентом и сервером, даже если они работают на системах с разной архитектурой, которые имеют различный порядок следования байт.

В заключение скажем несколько слов о способности систем RPC скрывать детали. Клиент и сервер могут быть почти независимыми друг от друга. Клиент может быть написан на Java, а сервер на C или наоборот. Клиент и сервер могут работать на разных платформах и использовать различные операционные системы. Поддерживается также многообразие сетевых протоколов и представлений данных, все это — без какого-либо вмешательства в клиент или сервер.

Написание клиента и сервера

Система DCE RPC состоит из множества компонентов. В нее входят, в частности, языки, библиотеки, программы-демоны и утилиты. Все это делает возмож-

ным создание разнообразных клиентов и серверов. В этом пункте мы опишем части этих программ и то, как они стыкуются друг с другом. Общий процесс написания и использования клиента и сервера суммирован на рис. 2.14.

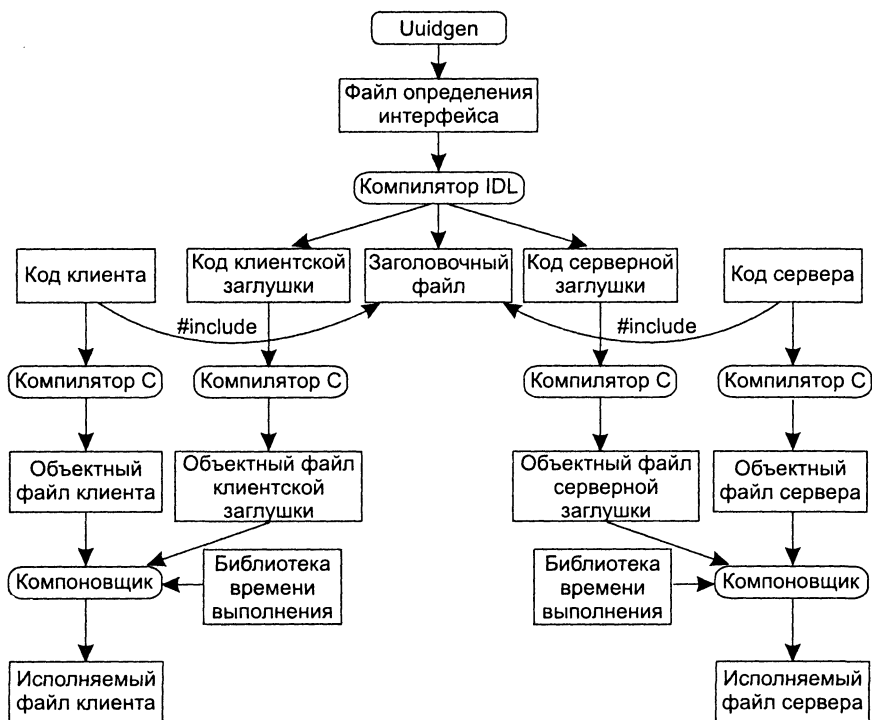


Рис. 2.14. Этапы написания клиента и сервера согласно DCE RPC

В системе клиент-сервер клеем, соединяющим все в единую систему, является описание интерфейса, которое создается с помощью языка *определения интерфейсов* (*Interface Definition Language, IDL*). Он позволяет описать процедуры в виде, очень похожем на прототипы функций в ANSI C. Файлы IDL могут также содержать определения типов, описания констант и другую информацию, необходимую для правильного маршалинга параметров и демаршалинга результатов. В идеале описание интерфейсов содержит также формальное определение действий, осуществляемых процедурой, но это выходит за рамки современных возможностей программирования, так что определение интерфейса включает в себя только его синтаксис, но не семантику. В лучшем случае программист может лишь добавить комментарии, описывающие, что делает та или иная функция.

Важнейшим элементом каждого файла IDL является глобальный уникальный идентификатор описываемого интерфейса. Клиент пересылает этот идентификатор в первом сообщении RPC, а сервер проверяет его правильность. В этом случае если клиент по ошибке пытается выполнить привязку не к тому серверу

или к старой версией правильного сервера, сервер обнаружит ошибку и привязки не произойдет.

Описания интерфейсов и уникальные идентификаторы в DCE в значительной степени взаимосвязаны. Как показано на рис. 2.14, первым шагом при написании приложения клиент-сервер является запуск программы *Uuidgen*, от которой мы хотим создания прототипа файла IDL, содержащего идентификатор интерфейса, гарантированно не использовавшийся ни в одном интерфейсе, созданном при помощи программы *Uuidgen*. Уникальность обеспечивается путем кодирования идентификатора машины и времени создания. Идентификатор представляет собой 128-битное число, представляемое в файле IDL в шестнадцатеричном формате в виде строки ASCII.

Следующим шагом является редактирование файла IDL, задание в нем имен удаленных процедур и их параметров. Несмотря на то что RPC не является полностью прозрачной системой (например, клиент и сервер не могут совместно использовать глобальные переменные), правила IDL делают описание неподдерживаемых конструкций невозможным.

После того как файл IDL будет закончен, для его обработки вызывается компилятор IDL. В результате работы компилятора мы получаем три файла:

- ◆ заголовочный файл (то есть `interface.h`, в терминологии C);
- ◆ файл клиентской заглушки;
- ◆ файл серверной заглушки.

Заголовочный файл содержит уникальный идентификатор, определения типов, констант и описания функций. Он может быть включен (с помощью директивы `#include`) в код сервера и клиента. Клиентская заглушка клиента содержит те процедуры, которые будет непосредственно вызывать клиентская программа. Эти процедуры отвечают за подбор параметров и упаковку их в исходящие сообщения с последующими обращениями к системе для их отправки. Клиентская заглушка также занимается распаковкой ответов, приходящих от сервера, и передачей значений, содержащихся в этих ответах, клиенту. Серверная заглушка содержит процедуры, вызываемые системой на машине сервера по приходе на нее сообщений. Они, в свою очередь, вызывают процедуры сервера, непосредственно выполняющие необходимую работу.

Следующим шагом программиста является написание кода клиента и сервера. После этого они оба, а также обе заглушки, компилируются. Полученные объектные файлы клиента и клиентской заглушки компоуются с библиотеками времени выполнения, что дает в результате исполняемый файл клиента. Таким же точно образом из файлов сервера и серверной заглушки после компиляции и компоновки получается исполняемый файл сервера. Во время исполнения клиент и сервер будут запущены, и приложение начнет свою работу.

Привязка клиента к серверу

Чтобы позволить клиенту вызывать сервер, необходимо, чтобы сервер был зарегистрирован и готов к приему входящих вызовов. Регистрация сервера дает кли-

енту возможность реально обнаружить сервер и выполнить привязку к нему. Обнаружение сервера происходит в два этапа.

1. Обнаружение машины сервера.
2. Обнаружение сервера (то есть нужный процесс) на этой машине.

Второй шаг немного непонятен. В общем случае для того, чтобы связаться с сервером, клиенту нужно знать *конечную точку (endpoint)* машины сервера, которой он может посылать сообщения. Конечная точка (более известная под названием *port*) используется операционной системой сервера для получения входящих сообщений от различных внешних процессов. В DCE на каждой из серверных машин процессом, известным под названием *DCE-демон (DCE daemon)*, поддерживается таблица пар сервер — конечная точка. Перед тем как сервер станет доступным для входящих запросов, он должен запросить у операционной системы конечную точку. Далее сервер регистрирует эту конечную точку у DCE-демона. DCE-демон записывает эту информацию (включая и протоколы, по которым может осуществляться обмен информацией с сервером) в таблицу конечных точек для последующего использования.

Сервер также регистрирует (с помощью службы каталогов) предоставленные серверной машине сетевой адрес и имя, под которым сервер будет доступен. Затем происходит привязка клиента к серверу, как показано на рис. 2.15.

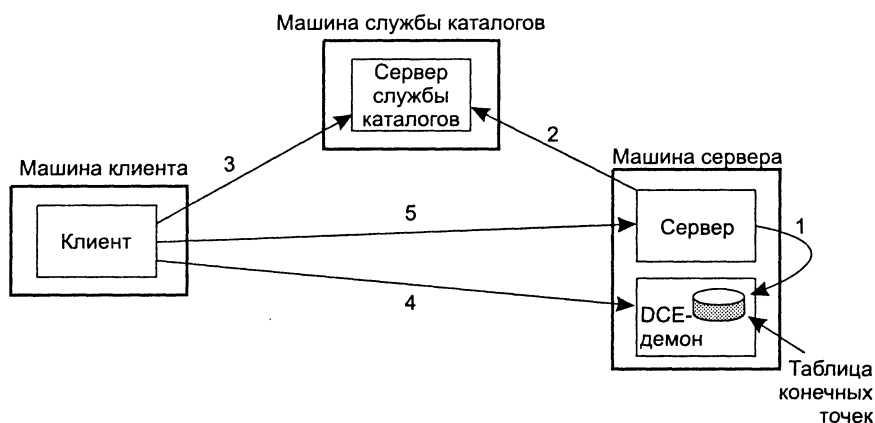


Рис. 2.15. Привязка клиента к серверу в DCE

Как показано на рисунке, привязка выполняется в несколько этапов.

1. Регистрация конечной точки.
2. Регистрация службы.
3. Поиск сервера службы каталогов.
4. Запрос конечной точки.
5. Выполнение вызова RPC.

Предположим, клиенту требуется привязка к серверу видеoinформации, локально доступному под именем `/local/multimedia/video/movies`. Он передает это

имя серверу службы каталогов. Последний возвращает сетевой адрес машины, на которой работает сервер видеоданных. После этого клиент обращается к DCE-демону этой машины (имеющему общеизвестную конечную точку) и просит его найти в его таблице конечных точек конечную точку сервера видеоинформации. Теперь, вооружившись полученными данными, мы можем выполнить вызов RPC. В ходе последующих вызовов RPC нам нет нужды проделывать всю процедуру поиска заново.

При необходимости система DCE дает клиенту возможность усложненного поиска необходимого сервера. Безопасность RPC также входит в ее задачи.

Выполнение вызова RPC

Реальный вызов RPC происходит прозрачно и обычным образом. Клиентская заглушка выполняет маршалинг параметров в том порядке, который необходим для библиотечных функций, осуществляющих передачу с использованием выбранного при привязке протокола. Когда сообщение приходит на машину с серверами, оно передается нужному серверу в соответствии с содержащейся в сообщении конечной точкой. Библиотека времени выполнения передает сообщение серверной заглушке, которая выполняет демаршалинг параметров и вызывает сервер. Ответ отправляется назад по тому же маршруту.

DCE предоставляет программистам некоторые семантические возможности. По умолчанию поддерживается *одноразовая операция* (*at-most-once operation*), в соответствие с которой ни один вызов не может осуществляться более одного раза, даже в случае краха системы. На практике это означает, что если сервер в ходе вызова RPC «рухнул», а затем был быстро восстановлен, клиент не должен повторять операцию, поскольку она, возможно, уже выполнена.

С другой стороны, можно пометить (в файле IDL) удаленную процедуру как *идемпотентную* (*idempotent*), в этом случае не возбраняются многочисленные повторы запросов. Так, например, чтение некоторого блока из файла можно повторять снова и снова, пока оно не будет успешно закончено. Если выполнение идемпотентного блока из-за сбоя сервера срывается, клиент может подождать перезагрузки сервера и сделать новую попытку. Также имеется и другая (редко используемая) семантика, включающая в себя широковещательные рассылки вызовов RPC всем машинам текущей локальной сети. Мы вернемся к семантике RPC в главе 7 при рассмотрении работы RPC в условиях сбоев.

2.3. Обращение к удаленным объектам

Объектно-ориентированная технология показала свое значение при разработке нераспределенных приложений. Одним из наиболее важных свойств объекта является то, что он скрывает свое внутреннее строение от внешнего мира посредством строго определенного интерфейса. Такой подход позволяет легко заменять или изменять объекты, оставляя интерфейс неизменным.

По мере того как механизм RPC постепенно становился фактическим стандартом осуществления взаимодействия в распределенных системах, люди начали

понимать, что принципы RPC могут быть равно применены и к объектам. В этом разделе мы распространим идею RPC на обращения к удаленным объектам и рассмотрим, как подобный подход может повысить прозрачность распределения по сравнению с вызовами RPC. Мы сосредоточимся на относительно простых удаленных объектах. В главе 10 мы коснемся нескольких объектных распределенных систем, включая CORBA и DCOM, каждая из которых поддерживает более серьезную, совершенную объектную модель, чем та, которую мы будем рассматривать сейчас.

2.3.1. Распределенные объекты

Ключевая особенность объекта состоит в том, что он инкапсулирует данные, называемые *состоянием* (*state*), и операции над этими данными, называемые *методами* (*methods*). Доступ к методам можно получить через интерфейс. Важно понять, что единственно правильным способом доступа или манипулирования состоянием объекта является использование методов, доступ к которым осуществляется через интерфейс этого объекта. Объект может реализовывать множество интерфейсов. Точно так же для данного описания интерфейса может существовать несколько объектов, предоставляющих его реализацию.

Это подразделение на интерфейсы и объекты, реализующие их, очень важно для распределенных систем. Четкое разделение позволяет нам помещать интерфейс на одну машину при том, что сам объект находится на другой. Структура, показанная на рис. 2.16, обычно и называется *распределенным объектом* (*distributed object*).

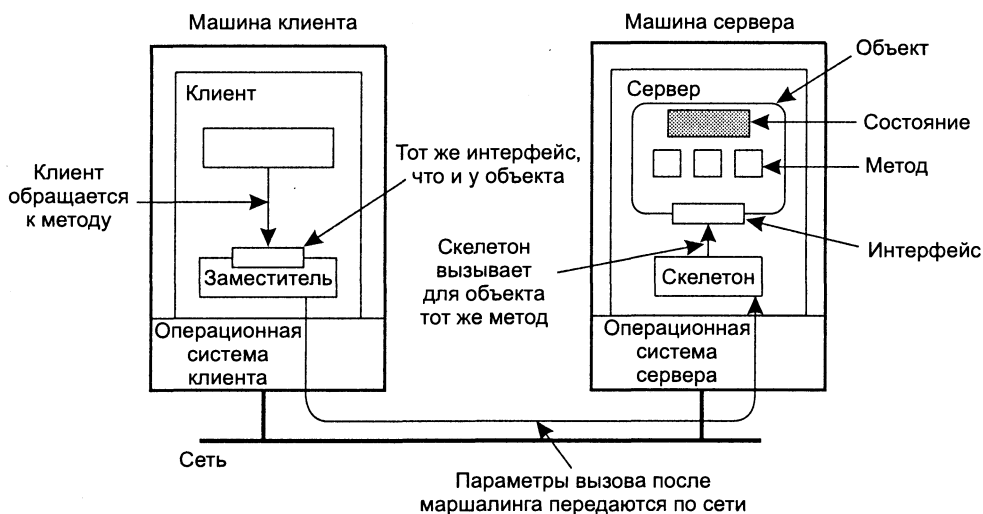


Рис. 2.16. Обобщенная организация удаленных объектов с использованием заместителя клиента

Когда клиент выполняет привязку к распределенному объекту, в адресное пространство клиента загружается реализация интерфейса объекта, называемая *заместителем (proxy)*. Заместитель клиента аналогичен клиентской заглушке в системах RPC. Единственное, что он делает, — выполняет маршалинг параметров в сообщениях при обращении к методам и демаршалинг данных из ответных сообщений, содержащих результаты обращения к методам, передавая их клиенту. Сами объекты находятся на сервере и предоставляют необходимые клиентской машине интерфейсы. Входящий запрос на обращение к методу сначала попадает на серверную заглушку, часто именуемую *скелетоном (skeleton)*. Скелетон преобразует его в правильное обращение к методу через интерфейс объекта, находящегося на сервере. Серверная заглушка также отвечает за маршалинг параметров в ответных сообщениях и их пересылку заместителю клиента.

Характерной, но немного противоречащей интуитивному представлению особенностью большинства распределенных объектов является то, что их состояние (данные) не распределяется — оно локализовано на одной машине. С других машин доступны только интерфейсы, реализованные в объекте. Такие объекты еще называют *удаленными (remote object)*. Как мы увидим в последующих главах при общем знакомстве с распределенными объектами, их состояние может быть физически распределено по нескольким машинам, но это распределение также скрывается от клиентов за интерфейсами объектов.

Объекты времени компиляции против объектов времени выполнения

Объекты в распределенных системах существуют в различных формах. В наиболее распространенном варианте они соответствуют объектам выбранного языка программирования, например Java, C++ или другого объектно-ориентированного языка, и представляют собой объекты времени компиляции. В этих случаях объект является экземпляром класса. *Класс* — это описание абстрактного типа в виде модуля, содержащего элементы данных и операций над этими данными [291].

Использование объектов времени компиляции в распределенных системах обычно значительно упрощает создание распределенных приложений. Так, в языке Java объект может быть полностью описан в рамках своего класса и интерфейсов, которые этот класс реализует. Компиляция определения класса порождает код, позволяющий создавать экземпляры объектов языка Java. Интерфейсы можно скомпилировать в клиентские и серверные заглушки, позволяющие обращаться к объектам Java с удаленных машин. Разработчик программы на Java чаще всего может не беспокоиться по поводу распределения объектов: он занимается только текстом программы на языке Java.

Очевидная обратная сторона использования объектов времени компиляции состоит в зависимости от конкретного языка программирования. Существует и альтернативный способ создания распределенных объектов — непосредственно во время выполнения. Такой подход характерен для множества объектных распределенных систем, поскольку распределенные приложения, созданные в соответствии с ним, не зависят от конкретного языка программирования. В част-

ности, приложение может быть создано из объектов, написанных на различных языках программирования.

При работе с объектами времени исполнения тот способ, которым они будут реализованы, обычно остается открытым. Так, например, разработчик может решить написать на С библиотеку, содержащую набор функций, которые смогут работать с общим файлом данных. Главный вопрос состоит в том, как превратить эту реализацию в объект, методы которого будут доступны с удаленной машины. Традиционный способ состоит в том, чтобы использовать *адаптер объектов* (*object adapter*), который послужит *оболочкой* (*wrapper*) реализации с единственной задачей — придать реализации видимость объекта. Сам термин «адаптер» взят из шаблона проектирования, описанного в [157], который предоставляет интерфейс, преобразуемый в то, что ожидает клиент. Примером адаптера объектов может быть некая сущность, динамически привязываемая к описанной ранее библиотеке на С и открывающая файл данных, соответствующий текущему состоянию объекта.

Адаптеры объектов играют важную роль в объектных распределенных системах. Чтобы сделать оболочку как можно проще, объекты определяются исключительно в понятиях интерфейсов, которые они реализуют. Реализация интерфейса регистрируется в адаптере, который, в свою очередь, создает интерфейс для удаленных обращений. Адаптер будет принимать приходящие обращения и создавать для клиентов образ удаленного объекта. Мы вернемся к вопросам организации серверов и адаптеров объектов в следующей главе.

Сохранные и нерезидентные объекты

Помимо деления на объекты, зависящие от языка программирования, и объекты времени выполнения существует также деление на сохранные и нерезидентные объекты. *Сохранный объект* (*persistent object*) — это объект, который продолжает существовать, даже не находясь постоянно в адресном пространстве серверного процесса. Другими словами, сохранный объект не зависит от своего текущего сервера. На практике это означает, что сервер, обычно управляющий таким объектом, может сохранить состояние объекта во вспомогательном запоминающем устройстве и завершить свою работу. Позже вновь запущенный сервер может считать состояние объекта из запоминающего устройства в свое адресное пространство и обработать запрос на обращение к объекту. В противоположность ему, *нерезидентный объект* (*transient object*) — это объект, который существует, только пока сервер управляет им. Когда сервер завершает работу, этот объект прекращает существовать. Использовать сохранные объекты или нет — это спорный вопрос. Некоторые полагают, что нерезидентных объектов вполне достаточно. Сейчас мы не будем вдаваться в детали и вернемся к этому вопросу, когда будем обсуждать объектные распределенные системы в главе 9.

2.3.2. Привязка клиента к объекту

Интересная разница между традиционными системами RPC и системами, поддерживающими распределенные объекты, состоит в том, что последние обычно

предоставляют ссылки на объекты, уникальные в пределах системы. Такие ссылки могут свободно передаваться между процессами, запущенными на различных машинах, например как параметры обращения к методу. Путем сокрытия истинной реализации ссылок на объекты (то есть обеспечения их непрозрачности) и может быть даже использования их в качестве единственного средства обращения к объектам прозрачность распределения по сравнению с традиционным механизмом RPC повышается.

Когда процесс хранит ссылку на объект, перед обращением к любому из методов объекта процесс должен в первую очередь выполнить привязку к этому объекту. Результатом привязки будет заместитель, размещаемый в адресном пространстве процесса и реализующий интерфейс с методами, к которым обращается процесс. Во многих случаях привязка осуществляется автоматически. Когда базовая система получает ссылку на объект, ей требуется способ отыскать сервер, управляющий этим объектом, и поместить заместителя в адресное пространство клиента.

При *неявной привязке* (*implicit binding*) клиенту предоставляется простой механизм, позволяющий напрямую запрашивать методы, используя только ссылку на объект. Так, например, в C++ можно переопределить унарный оператор выбора (->) для класса так, чтобы он позволял обращаться со ссылками на объекты как с простыми указателями (листинг 2.1). В случае неявной привязки клиент прозрачно связывается с объектом в момент разрешения ссылки и получения этого объекта в действительности. С другой стороны, в случае *явной привязки* (*explicit binding*) клиент должен до обращения к методам вызвать специальную функцию для привязки к объекту. При явной привязке обычно возвращается указатель на локально доступный заместитель, как показано в листинге 2.2.

Листинг 2.1. Пример неявной привязки с использованием только глобальных переменных

```
// Определить внутрисистемную ссылку на объект
Distr_object* obj_ref;
// Инициализировать ссылку на распределенный объект
obj_ref = ... ;
// Неявно провести привязку и обратиться к методу
obj_ref->do_something();
```

Листинг 2.2. Пример явной привязки с использованием как глобальных, так и локальных переменных

```
// Определить внутрисистемную ссылку на объект
Distr_object obj_ref;
// Определить указатель на локальные объекты
Local_object* obj_ptr;
// Инициализировать ссылку на распределенный объект
obj_ref = ... ;
// Явно провести привязку и получить указатель
// на локальный заместитель
obj_ptr = bind(obj_ref);
// Обратиться к методу локального заместителя
obj_ptr->do_something();
```


Реализация ссылок на объекты

Очевидно, что ссылки на объекты должны содержать достаточно информации, чтобы обеспечить клиентам привязку к объекту. Простая ссылка на объект должна содержать сетевой адрес машины, на которой размещен реальный объект, вместе с конечной точкой, определяющей сервер, который управляет объектом, и указанием на то, что это за объект. Последний обычно представляется сервером, например, в форме 16-битного числа. Конечная точка в данном случае абсолютно такая же, как и та, что обсуждалась при рассмотрении системы DCE RPC. На практике она соответствует локальному порту, который динамически выделяется локальной операционной системой сервера. Однако эта схема имеет множество недостатков.

Во-первых, если в сервере произойдет сбой и после восстановления сервер получит другую конечную точку, все ссылки на объекты станут неправильными. Эту проблему можно решить так же, как это сделано в DCE: создать на машине локального демона, который будет опрашивать известные конечные точки и отслеживать назначения конечных точек серверам в таблице конечных точек. После привязки клиента к объекту мы первым делом запросим у демона текущую конечную точку сервера. Такой подход требует, чтобы мы кодировали идентификатор сервера в ссылке на объект, которая может использоваться в качестве индекса в таблице конечных точек. Сервер, в свою очередь, всегда должен регистрироваться локальным демоном.

Однако кодирование сетевого адреса машины сервера в ссылке на объект — вообще говоря, плохая идея. Проблема подобного подхода — в том, что сервер невозможно будет перенести на другую машину, не объявив недействительными все ссылки на объекты, которыми он управлял. Традиционное решение этой проблемы состоит в распространении идеи локальных демонов, поддерживающих таблицы конечных точек, на *сервер локализации* (*location server*), следящий за постоянной работой серверов, на которых расположены объекты. Ссылка на объект в результате должна содержать сетевой адрес сервера локализации, а также действующий в системе идентификатор сервера. Как мы увидим в главе 4, это решение также имеет множество недостатков, особенно по части масштабируемости.

До настоящего момента мы в тактических целях предполагали, что клиент и сервер уже были каким-то образом сконфигурированы под единый стек протоколов. При этом предполагается, что они используют не только общий транспортный протокол, такой как TCP, но также и общий протокол маршallingа и демаршallingа параметров сообщения. Они должны также пользоваться общим протоколом для установки исходного соединения, обработки ошибок, контроля потока и пр.

Мы можем осторожно отбросить это допущение, если предположим, что к ссылке на объект добавляется дополнительная информация. Такая информация может включать указание на используемый для привязки к объекту протокол, который поддерживается сервером объекта. Так, например, некоторый сервер может одновременно поддерживать прием данных по протоколу TCP и дейта-

граммы UDP. При этом на клиенте лежит ответственность за реализацию заместителя как минимум для одного протокола, указанного в ссылке на объект.

Мы можем пойти еще дальше, включив в ссылку на объект *дескриптор реализации* (*implementation handle*), указывающий на полную реализацию заместителя. Эту реализацию клиент может динамически загрузить при привязке к объекту. Например, дескриптор реализации может принимать вид URL-адреса, указывающего на файл архива, типа `ftp://ftp.clientware.org/proxies/java/proxy-v1.1a.zip`. Протокол привязки в этом случае будет нужен только для указания на то, что данный файл следует динамически загрузить, разархивировать, установить, а впоследствии создать экземпляр. Плюс такого подхода состоит в том, что клиент не должен заботиться о том, доступна ли реализация данного протокола. Кроме того, это дает разработчику объекта полную свободу в разработке специфических для объекта заместителей. Однако, как мы увидим в главе 8, чтобы гарантировать клиенту, что он может доверять загруженному откуда-то коду, нам нужно предпринимать специальные действия по обеспечению защиты.

2.3.3. Статическое и динамическое удаленное обращение к методам

После того как клиент свяжется с объектом, он может через заместителя обратиться к методам объекта. Подобное *удаленное обращение к методам* (*Remote Method Invocation, RMI*) в части маршалинга и передачи параметров очень напоминает RPC. Основное различие между RMI и RPC состоит в том, что RMI, как говорилось ранее, в основном поддерживает внутрисистемные ссылки на объекты. Кроме того, отпадает необходимость в клиентских и серверных заглушках общего назначения. Вместо них мы можем использовать значительно более удобные в работе и специфические для объектов заглушки, которые мы также обсуждали.

Стандартный способ поддержки RMI — описать интерфейсы объектов на языке определения интерфейсов, так же как в RPC. Однако с тем же успехом мы можем использовать объектный язык, например Java, который обеспечивает автоматическую генерацию заглушек. Такой подход к применению предопределенных определений интерфейсов часто называют *статическим обращением* (*static invocation*). Статическое обращение требует, чтобы интерфейсы объекта при разработке клиентского приложения были известны. Также оно предполагает, что при изменении интерфейса клиентское приложение перед использованием новых интерфейсов будет перекомпилировано.

В качестве альтернативы обращение к методам может осуществляться более динамичным образом. В частности, иногда удобнее собрать параметры обращения к методу во время исполнения. Этот процесс известен под названием *динамического обращения* (*dynamic invocation*). Основное его отличие от статического обращения состоит в том, что во время выполнения приложение выбирает, какой метод удаленного объекта будет вызван. Динамическое обращение обычно выглядит следующим образом:

```
invoke(object, method, input_parameters, output_parameters);
```

Здесь `object` идентифицирует распределенный объект; `method` — параметр, точно задающий вызываемый метод; `input_parameters` — структура данных, в которой содержатся значения входных параметров метода; `output_parameters` — структура данных, в которой хранятся возвращаемые значения.

В качестве примера рассмотрим добавление целого числа `int` к объекту `fobject` файла. Для этого действия объект предоставляет метод `append`. В этом случае статическое обращение будет иметь вид:

```
fobject.append (int)
```

Динамическое обращение будет выглядеть так:

```
invoke(fobject, id(append). int)
```

Здесь операция `id(append)` возвращает идентификатор метода `append`. Для иллюстрации динамического обращения рассмотрим браузер объектов, используемый для просмотра наборов объектов. Предположим, что этот браузер поддерживает удаленное обращение к объектам. Это будет означать, что браузер в состоянии выполнить привязку к распределенному объекту и предоставить пользователю интерфейс с объектом. Пользователю после этого может быть предложено выбрать метод и ввести значения его параметров, после чего браузер сможет произвести действительное обращение. Обычно подобные браузеры объектов разрабатываются так, чтобы они поддерживали любые возможные интерфейсы. Такой подход требует исследования интерфейсов во время выполнения и динамического создания обращений к методам.

Другая область применения динамических обращений — службы пакетной обработки, для которых запросы на обращение могут обрабатываться в течение всего того времени, пока обращение ожидает выполнения. Служба может быть реализована в виде очереди запросов на обращение, упорядоченных по времени поступления. Основной цикл службы просто ожидает назначения очередного запроса, удаляет его из очереди и, как это было показано ранее, вызывает процедуру `invoke`.

2.3.4. Передача параметров

Поскольку большинство систем RMI поддерживает ссылки на объекты в пределах системы, возможности по передаче параметров при обращениях к методам обычно не столь ограничены, как в случае RPC. Однако существуют определенные тонкости, которые могут усложнить обращения RMI по сравнению с первоначальным представлением. Ниже мы кратко обсудим этот вопрос.

Сперва рассмотрим ситуацию, когда все объекты — распределенные. Другими словами, все объекты в системе доступны с удаленных машин. В этом случае при обращениях к методам мы постоянно используем ссылки на объекты как параметры. Ссылки передаются по значению и копируются с одной машины на другую. Если процесс получает ссылку на объект в качестве результата обращения к методу, он легко может выполнить привязку к объекту, на который указывает ссылка, позже, если ему это понадобится.

К сожалению, использование исключительно распределенных объектов обычно слишком неэффективно, особенно если эти объекты малы и просты, как, на-

пример, целые числа и логические значения. Каждое обращение клиента, если он не находится на том же сервере, что и объект, порождает запрос между различными адресными пространствами или, что еще хуже, между различными машинами. Поэтому работа со ссылками на удаленные и локальные объекты обычно происходит по-разному.

При обращении к методу с использованием ссылки на объект в качестве параметра эта ссылка копируется и передается как параметр-значение только тогда, когда она относится к удаленному объекту. Именно в этом случае происходит передача объекта по ссылке. Однако если ссылка относится к локальному объекту, то есть к объекту в адресном пространстве клиента, то объект, на который указывает ссылка, целиком копируется и в процессе обращения передается клиенту. Другими словами, объект передается по значению.

Эти две ситуации показаны на рис. 2.17, на котором изображены клиентская программа, выполняемая на машине А, и программа-сервер, выполняемая на машине С. Клиент обладает ссылкой на локальный объект О1, который используется в качестве параметра при вызове серверной программы на машине С. Кроме того, он обладает ссылкой на находящийся на машине В удаленный объект О2, который также используется в качестве параметра. При вызове сервера на машину С передается копия всего объекта О1 и копия ссылки на объект О2.

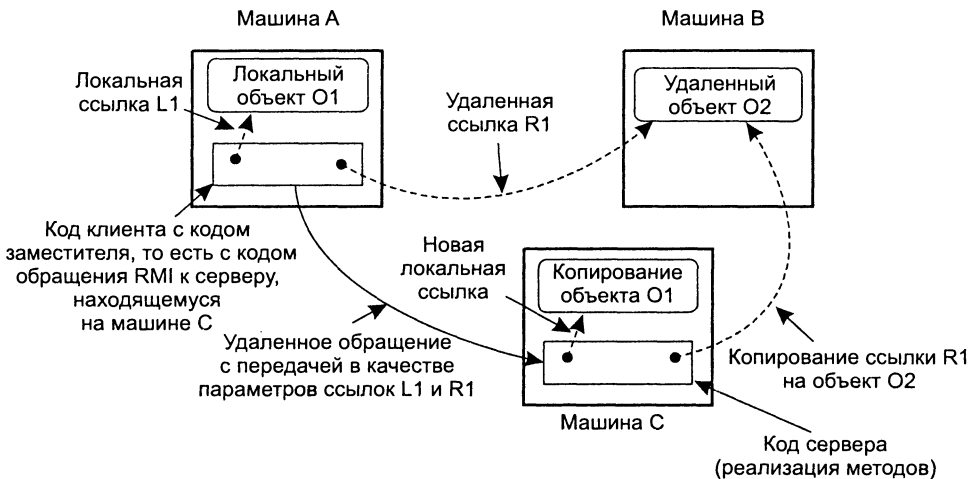


Рис. 2.17. Передача объекта по ссылке и по значению

Отметим, что независимо от того, работаем ли мы со ссылкой на локальный или глобальный объект, эта работа может быть весьма прозрачна, как, например, в Java. В Java разница будет заметна только в том случае, если локальные объекты будут иметь типы данных, отличные от типов данных удаленных объектов. В противном случае оба типа ссылок будут трактоваться почти одинаково [494]. С другой стороны, в традиционных языках программирования, таких как С, ссылками на локальные объекты могут быть простые указатели, использовать которые в качестве ссылок на удаленные объекты будет невозможно.

Дополнительный эффект обращения к методам с использованием в качестве параметра ссылки на объект состоит в возможности *копирования* объектов. Скрыть это невозможно, и в результате мы вынуждены явно указывать на различие между локальными и распределенными объектами. Понятно, что это различие не только разрушает прозрачность распределения — оно также затрудняет написание распределенных приложений.

2.3.5. Пример 1 — удаленные объекты DCE

DCE — это пример распределенной системы, появившейся хотя и в правильном месте, но не совсем в нужное время. Она была одной из первых распределенных систем, реализованных в виде промежуточного уровня поверх существующих операционных систем, и достаточно долго вынуждена была доказывать свою состоятельность. К сожалению, время доказательств совпало с «рождением» удаленных объектов и объявлением их панацеей распределенных систем. Поскольку среда DCE была традиционной, основанной на RPC системой, для нее наступили тяжелые времена. Она была признана устаревшей еще до того, как потребители успели установить достаточное число копий. Нет объектов — нет и разговора, а среда DCE их никогда не имела.

Команда, создававшая среду DCE, долго убеждала адвокатов объектных технологий, что в ней *реализована* поддержка объектов. Так, они утверждали, что системы RPC по определению основаны на объектах, поскольку все вопросы реализации и распределения скрыты за интерфейсами. Однако эти аргументы не котировались, и DCE была вынуждена перейти на более явную объектную технологию. В этом пункте мы рассмотрим, как DCE поддерживает распределенные объекты. Объекты DCE интересны для нас тем, что они являются прямым продолжением модели клиент-сервер, основанной на RPC, и представляют собой мост от вызовов удаленных процедур к обращениям к удаленным объектам.

Модель распределенных объектов DCE

Распределенные объекты были добавлены в DCE в форме расширений языка определения интерфейсов (IDL) вместе с привязками на языке C++. Другими словами, распределенные объекты в DCE описываются на IDL и реализуются на C++. Распределенные объекты имеют вид удаленных объектов, реализация которых находится на сервере. Сервер отвечает за локальное создание объектов C++ и обеспечение доступа к их методам с удаленных клиентских машин. Других способов создания распределенных объектов не существует.

Поддерживаются два типа распределенных объектов. *Динамические распределенные объекты* (*distributed dynamic objects*) — это объекты, которые создаются сервером локально по требованию клиента и к которым в принципе имеет доступ только один клиент. Для создания такого объекта сервер должен получить запрос от клиента. Это означает, что каждый класс, чтобы иметь возможность создавать динамические объекты, должен содержать процедуру *create*, которую можно вызвать, используя стандартный механизм RPC. После создания динами-

ческого объекта управление им переходит к исполняющей системе DCE, связывающей его с тем клиентом, по требованию которого он был создан.

В противоположность динамическим объектам, *именованные распределенные объекты* (*distributed named objects*) не предназначены для работы с единственным клиентом. Они создаются сервером для совместного использования несколькими клиентами. Именованные объекты регистрируются службой каталогов, так что клиент может найти объект и выполнить привязку к нему. Регистрация означает сохранение уникального идентификатора объекта, а также информации о том, как соединиться с сервером объекта. Разницу между динамическими и именованными объектами иллюстрирует рис. 2.18.

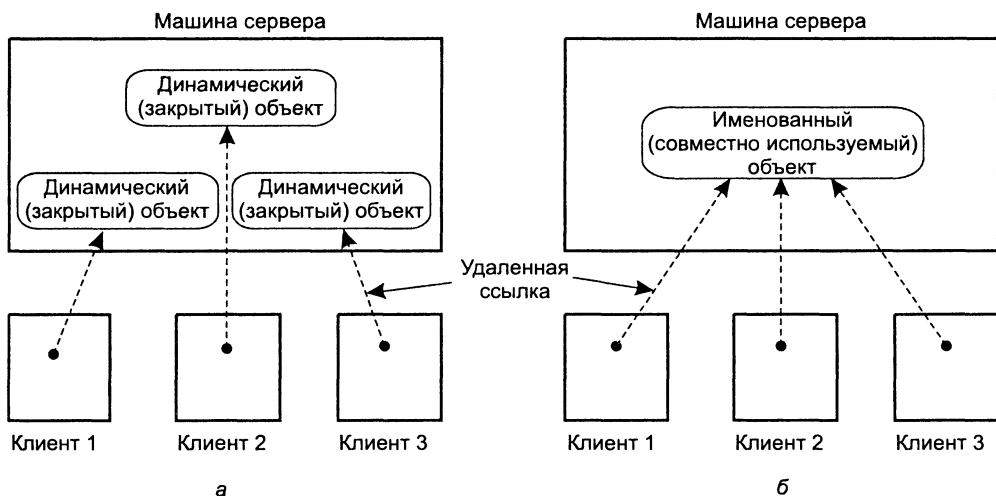


Рис. 2.18. Распределенные динамические объекты в DCE (а).
Распределенные именованные объекты (б)

Обращение к удаленным объектам в DCE

Как можно заметить, все обращения к удаленным объектам в DCE производятся средствами RPC. Клиент, обращаясь к методу, передает серверу идентификатор объекта, идентификатор интерфейса, содержащего метод, идентификацию самого метода и параметры. Сервер поддерживает таблицу объектов. С помощью этой таблицы сервер, получив идентификатор объекта и идентификатор интерфейса, идентифицирует объект, к которому обратился клиент. Затем он выбирает запрошенный метод и передает ему параметры.

Поскольку сервер может поддерживать тысячи объектов, DCE предоставляет возможность не держать все объекты в памяти, а помещать их при необходимости во вспомогательное хранилище данных. Когда к серверу приходит запрос на обращение к объекту, отсутствующему в таблице объектов, исполняющая система может вызвать специализированную функцию поиска, которая извлечет объект из хранилища и поместит его в адресное пространство сервера. Обращение к объекту произойдет после его помещения в оперативную память.

У распределенных объектов в DCE имеется одна проблема, связанная с их чрезвычайной близостью с RPC: не существует механизма прозрачных ссылок на объекты. Клиент имеет в лучшем случае *дескриптор привязки* (*binding handle*), ассоциированный с именованным объектом. Дескриптор привязки содержит идентификатор интерфейса, транспортный протокол, используемый для связи с сервером объектов, а также адрес и конечную точку хоста сервера. Дескриптор привязки может быть преобразован в строку и в таком виде передан другому процессу.

Отсутствие приемлемого механизма ссылок на объекты в пределах системы делает передачу параметров в DCE более сложной, чем в большинстве объектных систем. Разработчик приложений для RPC должен сам придумывать механизм передачи параметров. На деле это означает, что необходимо явно производить маршалинг объектов для передачи их по значению и самостоятельно создавать для этого соответствующие процедуры.

В качестве альтернативы разработчик может использовать делегирование. Для этого из спецификации интерфейса объекта генерируется специальная заглушка. Заглушка работает как оболочка нужного объекта и содержит только те методы, которые будут вызываться удаленными процессами. Заглушка может быть скомпонована с любым процессом, который хочет использовать этот объект. Преимущества такого подхода становятся понятны, если вспомнить, что DCE разрешает передавать удаленные ссылки на заглушки в качестве параметров вызовов RPC. Соответственно, появляется возможность ссылаться на объекты системы при помощи ссылок на заглушки.

Дополнительную информацию по объектному программированию в DCE можно найти в [335, 476].

2.3.6. Пример 2 — Java RMI

В DCE распределенные объекты были добавлены в качестве расширения вызовов удаленных процедур. Вместо того чтобы указывать удаленную процедуру на сервере, клиент указывал удаленную процедуру для объекта на сервере. Отсутствие подобающего механизма ссылок на объекты системы подчеркивало, что мы действительно имеем дело с простым расширением RPC.

Взглянем теперь на распределенные объекты с абсолютно другой точки зрения. В Java распределенные объекты интегрированы с языком. Основная цель этого — сохранить, насколько это возможно, семантику нераспределенных объектов. Другими словами, создатели языка Java стремились к высокому уровню прозрачности распределения. Однако, как мы увидим, создатели Java также решили, что в тех случаях, когда высокая степень прозрачности может быть неэффективна, затруднительна или нереализуема, распределенность может быть явной.

Модель распределенных объектов Java

Язык Java также поддерживает распределенные объекты исключительно в форме удаленных объектов. Напомним, что удаленный объект — это распределенный

объект, тело которого постоянно находится на одной и той же машине, а интерфейсы доступны удаленным процессам. Интерфейсы реализованы обычным образом через заместителя, который предоставляет те же интерфейсы, что и удаленный объект. Сам заместитель имеет вид локального объекта, находящегося в адресном пространстве клиента.

Между удаленными и локальными объектами существует лишь несколько различий, но различия эти тонки и важны. Во-первых, значительно различается клонирование локальных и удаленных объектов. Клонирование локального объекта *O* приводит к появлению нового объекта такого же типа, что и *O*, в точно таком же состоянии. Процедура клонирования возвращает точную копию клонированного объекта. Подобная семантика слабо применима к удаленным объектам. Если мы попытаемся создать точную копию удаленного объекта, нам потребуется клонировать не только сам объект на его сервере, но и заместитель на каждом из клиентов, которые в этот момент привязаны к удаленному объекту. Поэтому операция клонирования удаленного объекта производится только на сервере. Она приводит к созданию точной копии объекта в адресном пространстве сервера. Заместитель объекта не клонируется. Если клиент на удаленной машине хочет получить доступ к клону объекта на сервере, он должен сначала выполнить повторную привязку к этому объекту.

Более существенное различие между локальными и удаленными объектами в Java заключается в семантике блокировки объектов. Java позволяет построить любой объект в виде *монитора*. Для этого достаточно объявить один из методов *синхронизируемым* (*synchronized*). Если два процесса одновременно вызовут синхронизируемый метод, работать будет только один из них, в то время как второй окажется заблокированным. Таким образом, мы можем гарантировать, что доступ к внутренним данным объекта будет осуществляться только последовательно. Как и в мониторе, процесс может быть заблокирован и изнутри объекта в ожидании выполнения некоторого условия, как мы описывали в главе 1.

Рассуждая логически, блокировка в удаленных объектах — дело несложное. Предположим, что клиент *A* вызвал синхронизируемый метод удаленного объекта. Чтобы получить доступ к удаленному объекту, который всегда выглядит так же, как локальный, необходимо заблокировать *A* на уровне клиентской заглушки, которая реализует интерфейс объекта и к которой *A* имеет прямой доступ. Точно так же и другой клиент на другой машине должен быть заблокирован до пересылки запроса на сервер. Это означает, что нам необходимо блокировать разных клиентов на различных машинах. Как мы увидим в главе 5, распределенная синхронизация может оказаться довольно сложным делом.

Альтернативный подход состоит в том, чтобы производить блокировку только на сервере. В принципе это неплохо работает, но возникают проблемы с нарушениями работы клиентов в момент обработки сервером их обращений. Как мы увидим в главе 7, обработка подобных ситуаций требует достаточно хитроумного протокола и способна значительно снизить общую производительность обращений к удаленным методам.

Поэтому разработчики Java RMI ограничились блокировкой удаленных объектов блокировкой заместителей [494]. На практике это означает, что просто путем

использования синхронизированных методов удаленные объекты невозможно защитить от одновременного доступа процессов, работающих через разные заместители. Вместо этого следует использовать явные методы распределенной блокировки.

Обращение к удаленным объектам в Java

Поскольку разница между локальными и удаленными объектами на уровне языка слабо заметна, Java может в ходе обращений к удаленным методам скрывать большую часть различий между ними. Так, в ходе обращения RMI в качестве параметра может быть передан любой простой или объектный тип, что предполагает возможность маршалинга типов. В терминологии Java это означает, что типы *сериализуемы* (*serializable*). Хотя в принципе сериализации можно подвергнуть большинство объектов, она не всегда является допустимой или возможной. Обычно зависящие от платформы объекты, такие как дескрипторы файлов или сокет, не сериализуются.

Единственное различие между локальными и удаленными объектами, наблюдаемое в процессе RMI, состоит в том, что локальные объекты передаются по значению (включая большие объекты, такие как массивы), в то время как удаленные объекты передаются по ссылке. Другими словами, локальные объекты копируются, после чего копия используется в качестве параметра-значения. В случае удаленных объектов в качестве параметра используется ссылка на объект, без всякого копирования, как показано на рис. 2.17.

При обращении RMI в Java ссылка на удаленный объект реализуется именно так, как мы говорили в пункте 2.3.2. Эта ссылка содержит сетевой адрес и конечную точку сервера, а также локальный идентификатор необходимого объекта в адресном пространстве сервера. Как мы обсуждали, в ссылке на удаленный объект, кроме того, кодируется стек протоколов, используемых клиентом и сервером для взаимодействия. Чтобы понять, как при обращении RMI в Java кодируется стек протоколов, необходимо учитывать, что каждый объект Java представляет собой экземпляр класса. Класс же, в свою очередь, содержит реализацию одного или более интерфейсов.

В сущности, удаленный объект построен из двух различных классов. Один из классов содержит реализацию кода сервера и называется *классом сервера*. Класс сервера содержит реализацию той части удаленного объекта, которая выполняется на сервере. Другими словами, она содержит описание состояния (данных) объекта, а также реализацию методов обработки этого состояния. Из спецификации интерфейса объекта генерируется серверная заглушка, то есть скелетон.

Другой класс содержит реализацию кода клиента и называется *классом клиента*. Класс клиента содержит реализацию заместителя. Как и скелетон, этот класс также автоматически создается из спецификации интерфейса объекта. В своей простейшей форме заместитель делает только одну вещь — превращает каждый вызов метода в сообщение, пересылаемое реализации удаленного объекта, находящейся на сервере, а каждое ответное сообщение — в результат вызова метода. При каждом вызове он устанавливает связь с сервером, разрывая ее после завер-

шения вызова. Для этого, как уже говорилось, заместитель нуждается в сетевом адресе и конечной точке сервера.

Таким образом, заместитель обладает всей информацией, необходимой для обращения клиента к методу удаленного объекта. В Java заместители сериализуются. Другими словами, заместитель можно подвергнуть маршалингу и переслать в виде набора байтов другому процессу, в котором он может быть подвергнут обратной операции (демаршалингу) и использован для обращения к методам удаленного объекта. Косвенным результатом этого является тот факт, что заместитель может быть использован в качестве ссылки на удаленный объект.

Этот подход согласуется с методами интеграции локальных и распределенных приложений в Java. Напомним, что при обращении RMI локальный объект передается путем создания копии, а удаленный — через общесистемную ссылку на объект. Заместитель представляет собой просто-напросто локальный объект. Это означает, что сериализуемый заместитель можно передавать по сети как параметр RMI. В результате появляется возможность использовать заместитель как ссылку на удаленный объект.

В принципе при маршалинге заместителя вся его реализация, то есть его состояние и код, превращаются в последовательность байтов. Маршалинг подобного кода не слишком эффективен и может привести к слишком объемным ссылкам. Поэтому при маршалинге заместителя в Java на самом деле происходит генерация дескриптора реализации, точно определяющего, какие классы необходимы для создания заместителя. Возможно, некоторые из этих классов придется сперва загрузить из удаленного узла. Дескриптор реализации в качестве части ссылки на удаленный объект заменяет передаваемый при маршалинге код. В результате ссылки на удаленные объекты в Java имеют размер порядка нескольких сотен байт.

Такой подход к ссылкам на удаленные объекты отличается высокой гибкостью и представляет собой одну из отличительных особенностей RMI в Java [482]. В частности, это позволяет оптимизировать решение под конкретный объект. Так, рассмотрим удаленный объект, состояние которого изменяется только один раз. Мы можем превратить этот объект в настоящий распределенный объект путем копирования в процессе привязки всего его состояния на клиентскую машину. Каждый раз при обращении клиента к методу он работает с локальной копией. Чтобы гарантировать согласованность данных, каждое обращение проверяет, не изменилось ли состояние объекта на сервере, и при необходимости обновляет локальную копию. Таким же образом методы, изменяющие состояние объекта, передаются на сервер. Разработчик удаленного объекта должен разработать только код, необходимый для клиента, и сделать его динамически подгружаемым при присоединении клиента к объекту.

Возможность передавать заместителя в виде параметра существует только в том случае, если все процессы работают под управлением одной и той же виртуальной машины. Другими словами, каждый процесс работает в одной и той же среде исполнения. Переданный при маршалинге заместитель просто подвергается демаршалингу на приемной стороне, после чего полученный код заместителя можно выполнять. В противоположность этому в DCE, например, передача за-

глушек невозможна, поскольку разные процессы могут запускаться в разных средах исполнения, отличающихся языком программирования, операционной системой и аппаратным обеспечением. Вместо этого в DCE производится компоновка (динамическая) с локальной заглушкой, скомпилированной в расчете на среду исполнения конкретного процесса. Путем передачи ссылки на заглушку в виде параметра RPC достигается возможность выхода за границы процесса.

2.4. Связь посредством сообщений

Вызовы удаленных процедур и обращения к удаленным объектам способствуют сокрытию взаимодействия в распределенных системах, то есть повышают прозрачность доступа. К сожалению, ни один из этих механизмов не идеален. В частности, в условиях, когда нет уверенности в том, что принимающая сторона в момент выполнения запроса работает, приходится искать альтернативные пути обмена. Кроме того, синхронную по определению природу RPC и RMI, при которой на время осуществления операции необходимо блокировать клиента, временами тоже хочется заменить чем-то другим.

Это «что-то другое» — обмен сообщениями. В этом разделе мы сосредоточимся на использовании в распределенных системах взаимодействия на основе сообщений. Сначала мы кратко рассмотрим, что такое чисто синхронное поведение и каковы его области применения. Затем обсудим системы обмена сообщениями, позволяющие сторонам в процессе взаимодействия продолжать работу. И наконец, исследуем системы очередей сообщений, позволяющие процессам обмениваться информацией, даже если вторая сторона в момент начала связи не работает.

2.4.1. Сохранность и синхронность во взаимодействиях

Чтобы разобраться во множестве альтернатив в коммуникационных системах, работающих на основе сообщений, предположим, что система организована по принципу компьютерной сети, как показано на рис. 2.19. Приложения всегда выполняются на хостах, а каждый хост предоставляет интерфейс с коммуникационной системой, через который сообщения могут передаваться. Хосты соединены сетью коммуникационных серверов, которые отвечают за передачу (или маршрутизацию) сообщения между хостами. Без потери общности можно предположить, что каждый из хостов связан только с одним коммуникационным сервером. В главе 1 мы предполагали, что буферы могут быть размещены исключительно на хостах. Для более общего варианта нам следует рассмотреть варианты с размещением буферов и на коммуникационных серверах базовой сети.

В качестве примера рассмотрим разработанную в подобном стиле систему электронной почты. Хосты работают как пользовательские агенты — это пользовательские приложения, которые могут создавать, посылать, принимать и читать сообщения. Каждый хост соединяется только с одним почтовым сервером, кото-

рый является по отношению к нему коммуникационным сервером. Интерфейс пользовательского хоста позволяет пользовательскому агенту посылать сообщения по конкретным адресам. Когда пользовательский агент представляет сообщение для передачи на хост, хост обычно пересылает это сообщение на свой локальный почтовый сервер, в выходном буфере которого оно хранится до поры до времени.

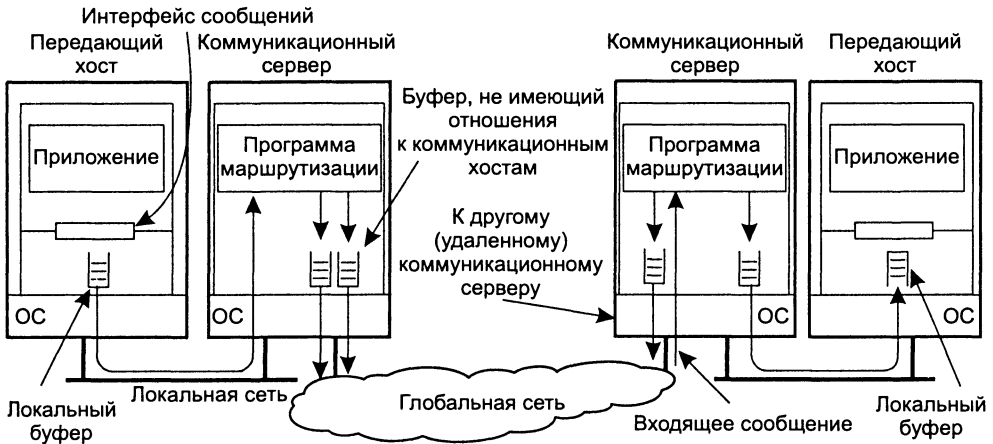


Рис. 2.19. Обобщенная организация коммуникационной системы, хосты которой соединяются через сеть

Почтовый сервер удаляет сообщение из своего выходного буфера и ищет, куда его нужно доставить. Поиски места назначения приводят к получению адреса (транспортного уровня) почтового сервера, для которого предназначено сообщение. Затем почтовый сервер устанавливает соединение и передает сообщение на другой, выбранный почтовый сервер. Последний сохраняет сообщение во входящем буфере намеченного получателя, также называемом почтовым ящиком получателя. Если искомый почтовый ящик временно недоступен, например, отключен, то хранить сообщение продолжает локальный почтовый сервер.

Интерфейс принимающего хоста предоставляет пользовательским агентам службы, при помощи которых они могут регулярно проверять наличие пришедшей почты. Пользовательский агент может работать напрямую с почтовым ящиком пользователя на локальном почтовом сервере или копировать новые сообщения в локальный буфер своего хоста. Таким образом, сообщения обычно хранятся на коммуникационных серверах, но иногда и на принимающем хосте.

Система электронной почты — это типичный пример *сохранной связи* (*persistent communication*). При сохранной связи сообщение, предназначенное для отсылки, хранится в коммуникационной системе до тех пор, пока его не удастся передать получателю. Если отталкиваться от рисунка, можно сказать, сообщение сохраняется на коммуникационном сервере до тех пор, пока его не удастся передать на следующий коммуникационный сервер. Поэтому у отправляющего сообщения приложения нет необходимости после отправки сообщения продолжать

работу. Аналогично, у приложения, принимающего сообщения, также нет необходимости находиться в рабочем состоянии во время отправки сообщения.

Система сохранной связи сравнима по принципу работы с почтовой системой Pony Express (рис. 2.20). Отправка письма начинается с доставки его в местное почтовое отделение. Почтовое отделение отвечает за сортировку почты в зависимости от того, в какое следующее почтовое отделение на пути к конечному пункту доставки ее нужно отправить. В нем также хранят соответствующие сумки с почтой, отсортированной по месту назначения, и ждут появления лошади со своим всадником. В пункте назначения письма вновь сортируются в зависимости от того, заберут ли их адресаты прямо здесь или нужно передать эти письма следующему почтальону. Отметим, что письма никогда не теряются и не пропадают. Несмотря на то что средства доставки, так же как и средства сортировки писем, за прошедшую сотню лет изменились, принципы сортировки, хранения и пересылки почты остались неизменными.

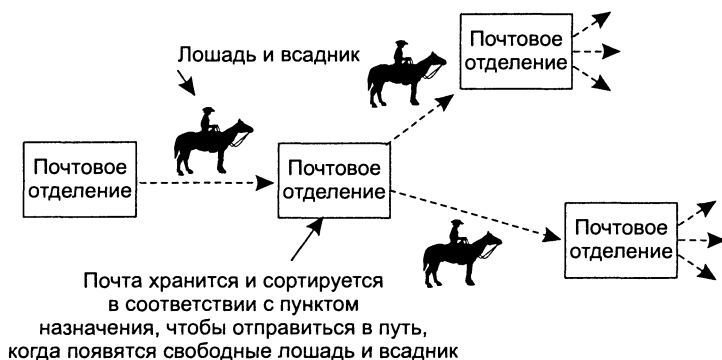


Рис. 2.20. Сохранная связь — доставка писем во времена Pony Express

В противоположность сохранной связи при *нерезидентной связи* (*transient communication*) сообщение хранится в системе только в течение времени работы приложений, которые отправляют и принимают это сообщение. Точнее говоря (если опять отталкиваться от рис. 2.20), мы имеем дело с такой ситуацией, когда коммуникационный сервер, не имея возможности передать сообщение следующему серверу или получателю, просто уничтожает его. Обычно все коммуникационные службы транспортного уровня поддерживают только нерезидентную связь. В этом случае коммуникационный сервер соответствует традиционному маршрутизатору «получил — передал». Если маршрутизатор не в состоянии переслать сообщение следующему маршрутизатору или принимающему хосту, сообщение просто теряется.

Помимо сохранной и нерезидентной связи существует деление на синхронную и асинхронную связь. Характерной чертой *асинхронной связи* (*asynchronous communication*) является немедленное после отправки письма продолжение работы отправителя. Это означает, что письмо сохраняется в локальном буфере передающего хоста или на ближайшем коммуникационном сервере. В случае *синхронной связи* (*synchronous communication*) отправитель блокируется до того момента,

пока его сообщение не будет сохранено в локальном буфере принимающего хоста или доставлено реальному получателю. Наиболее жесткая форма синхронного взаимодействия предполагает, что отправитель остается заблокированным и на время обработки его сообщения получателем.

На практике применяются различные комбинации этих типов взаимодействия. В случае сохранной асинхронной связи сообщение сохраняется в буфере либо локального хоста, либо первого коммуникационного сервера. Этот вид связи обычно используется в системах электронной почты. В случае сохранной синхронной связи сообщения хранятся только на принимающем хосте. Отправитель блокируется до момента сохранения сообщения в буфере получателя. Отметим, что приложение, принявшее сообщение, не обязано сохранять его на своем локальном хосте. «Усеченный» вариант сохранной синхронной связи состоит в том, что отправитель блокируется до момента сохранения сообщения на коммуникационном сервере, соединенном с принимающим хостом.

Нерезидентная асинхронная связь характерна для служб дейтаграмм транспортного уровня, таких как UDP. Когда приложение отправляет сообщение, оно временно сохраняется в локальном буфере передающего хоста, после чего отправитель немедленно продолжает работу. Параллельно коммуникационная система направляет сообщение в точку, из которой, как ожидается, оно сможет достигнуть места назначения, возможно, с сохранением в локальном буфере. Если получатель в момент прихода сообщения на принимающий хост этого получателя неактивен, передача обрывается. Другой пример нерезидентной асинхронной связи — асинхронный вызов RPC.

Нерезидентная синхронная связь существует в различных вариантах. В наиболее слабой форме, основанной на подтверждениях приема сообщений, отправитель блокируется до тех пор, пока сообщение не окажется в локальном буфере принимающего хоста. После получения подтверждения отправитель продолжает свою работу. В ориентированной на доставку нерезидентной синхронной связи отправитель блокируется до тех пор, пока сообщение не будет доставлено получателю для дальнейшей обработки. Мы рассматривали эту форму синхронного поведения при обсуждении асинхронных вызовов RPC. При асинхронных вызовах RPC клиент синхронизируется с сервером, ожидая, пока его запрос будет принят на дальнейшую обработку. Наиболее жесткая форма — ориентированная на ответ нерезидентная синхронная связь — предполагает блокировку отправителя до получения ответного сообщения с другой стороны, как в поведении запрос-ответ при взаимодействии клиент-сервер. Эта схема характерна также для механизмов RPC и RMI.

Все сочетания сохранности и синхронности при взаимодействиях показаны на рис. 2.21. Другая, но схожая классификация обсуждается в [445].

До недавнего времени множество распределенных систем поддерживали только ориентированную на ответ нерезидентную синхронную связь, реализованную через вызов удаленных процедур или через обращения к удаленным объектам. После того как стало ясно, что этот вид связи не всегда самый подходящий, были созданы средства для менее жестких форм нерезидентной синхронной связи, таких как асинхронные вызовы RPC (см. рис. 2.13) или отложенные синхронные операции.

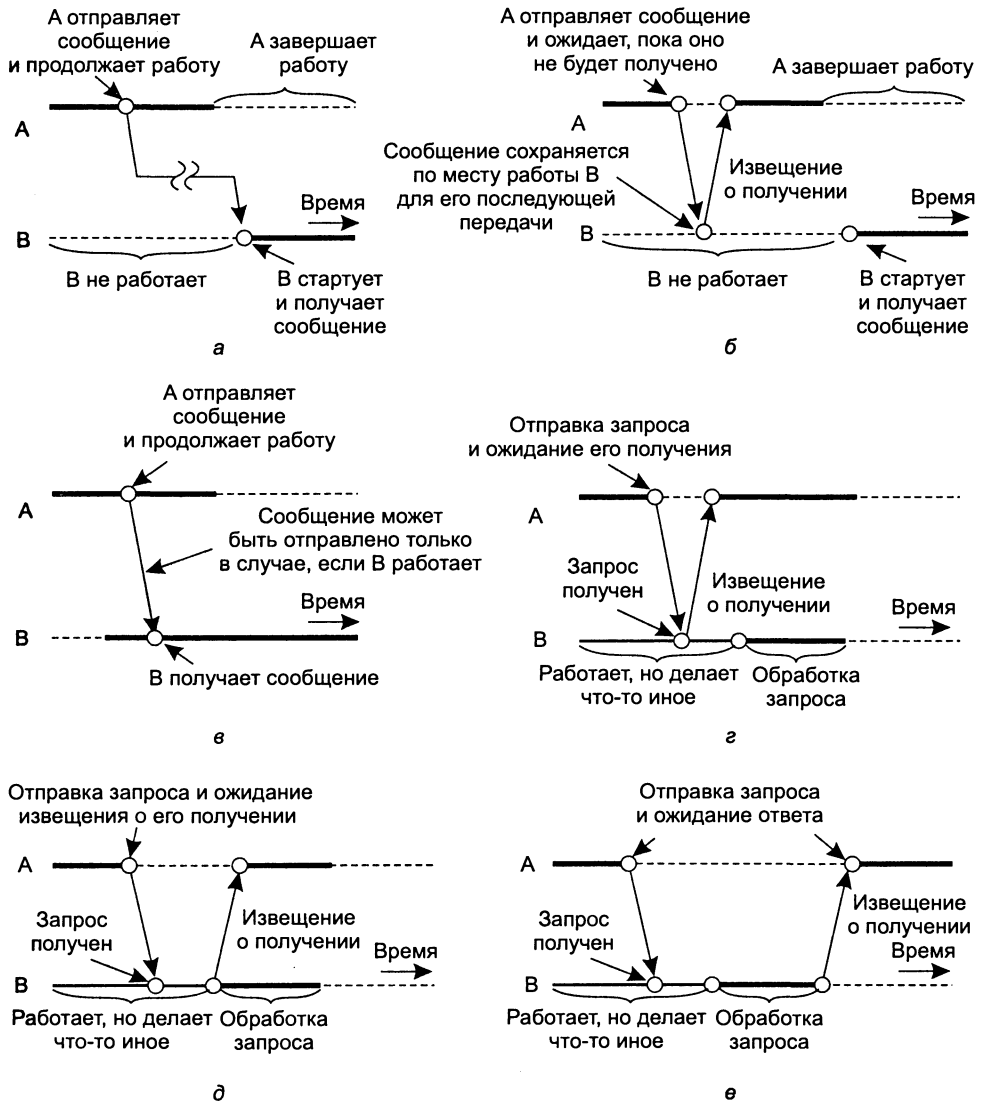


Рис. 2.21. Шесть видов связи: сохраненная асинхронная связь (а), сохраненная синхронная связь (б), нерезидентная асинхронная связь (в), нерезидентная синхронная связь с синхронизацией по приему (г), нерезидентная синхронная связь с синхронизацией по доставке (д) и нерезидентная синхронная связь с синхронизацией по ответу (е)

В корне отличный подход применен в системах передачи сообщений, использующих как отправную точку нерезидентную асинхронную связь и в качестве дополнительной возможности содержащих средства синхронной связи. Однако во всех вариантах передачи сообщений взаимодействия также предполагаются прозрачными. Другими словами, задействуются только те средства связи, кото-

рые подходят для синхронных процессов. Ограничиваться исключительно этими средствами во многих случаях нереально, особенно если принять во внимание географическую масштабируемость.

Необходимость в службах сохранной связи стала очевидной, когда разработчикам программного обеспечения промежуточного уровня потребовалось интегрировать приложения в крупные и сильно разветвленные взаимосвязанные сети. Подобные сети часто разбросаны по различным подразделениям и административным зонам, части которых не всегда могут быть доступны немедленно. Например, доступ может быть ограничен по причине сбоев в сети или процессах. Для решения подобных проблем были разработаны частные решения на базе сохранной связи, но подобные решения, как легко понять, не вполне удовлетворяли требованиям переносимости и работоспособности в разных условиях.

Другим недостатком нерезидентной связи можно считать тот факт, что в случае возникновения ошибки необходимо немедленно замаскировать ее и запустить процедуру восстановления. Невозможность отложить восстановление в данном случае означает нарушение прозрачности по отказам. В случае же сохранной связи приложение разрабатывается в расчете на длительные задержки между посылкой сообщения и получением ответа на него. Соответственно, мы можем прибегнуть к несложным, хотя возможно и медленным способам маскировки ошибок и восстановления.

Должно быть понятно, что выбор исключительно между нерезидентным и сохранным типами связи во многих случаях неприемлем. Аналогично, только синхронный и асинхронный типы связи — это еще не все. В зависимости от задач распределенной системы ей могут потребоваться все возможные типы связи. Ранее мы говорили в основном о нерезидентной синхронной связи — RPC и RMI. Другие формы взаимодействия, как правило, представлены системами, работающими на основе передачи сообщений. Эти системы мы обсудим в следующих пунктах. Мы проясним разницу между нерезидентной и сохранной связью.

2.4.2. Нерезидентная связь на основе сообщений

Множество распределенных систем и приложений непосредственно построено на базе простой модели обмена сообщениями, предоставляемой транспортным уровнем. Чтобы лучше понять и оценить системы, ориентированные на сообщения, как части решений промежуточного уровня, обсудим сперва обмен сообщениями через сокеты транспортного уровня.

Сокеты Беркли

Для стандартизации интерфейса транспортного уровня предпринимались специальные усилия. Это делалось для того, чтобы позволить программистам использовать полный комплект протоколов обмена сообщениями как простой набор примитивов. Кроме того, стандартные интерфейсы упрощают перенос приложений на другие машины.

В качестве примера мы кратко рассмотрим *интерфейс сокетов*, введенный в версии UNIX, разработанной в университете Беркли (Berkeley UNIX). Другой

важный интерфейс, *XTI*, присутствующий в транспортном интерфейсе *X/Open*, который официально именуется *интерфейсом транспортного уровня (Transport Layer Interface, TLI)*, разработан организацией *AT&T*. Сокеты и *XTI* хорошо подходят к своим моделям сетевого программирования, но имеют разный набор примитивов.

Концептуально *сокет (socket)* — это конечная точка коммуникации. В эту точку приложение может записывать данные, которые следует переслать по базовой сети, и из этой точки оно может читать приходящие данные. Сокеты образуют абстракцию, лежащую поверх реальной конечной точки сети, которая работает с локальной операционной системой по некоторому транспортному протоколу. Далее мы сосредоточимся на примитивах сокетов для *TCP*, показанных в табл. 2.1.

Таблица 2.1. Примитивы сокетов для *TCP/IP*

Примитив	Назначение
Socket	Создать новую конечную точку коммуникации
Bind	Назначить сокету локальный адрес
Listen	Обозначить готовность к установке соединения
Accept	Заблокировать вызывающую сторону до прибытия запроса о соединении
Connect	Совершить попытку установить соединение
Send	Послать через соединение некоторые данные
Receive	Принять из соединения некоторые данные
Close	Разорвать соединение

Серверы обычно выполняют первые четыре примитива, чаще всего в приведенном в таблице порядке. При вызове примитива *socket* вызывающий процесс создает новую конечную точку для некоторого транспортного протокола. Изнутри создание конечной точки коммуникации означает, что локальная операционная система резервирует ресурсы для размещения приходящих и отправляемых по некоторому протоколу сообщений.

Примитив *bind* выполняет привязку локального адреса к только что созданному сокету. Например, сервер должен связать *IP*-адрес своей машины с номером порта (возможно, общеизвестным) сокета. Привязка сообщает операционной системе, что сервер намерен получать сообщения только на указанные адрес и порт.

Примитив *listen* применяется только для коммуникаций, ориентированных на соединение. Это неблокирующий вызов, требующий от локальной операционной системы зарезервировать буфер для определенного максимального количества соединений, которое вызывающий процесс намерен поддерживать.

Вызов примитива *асерт* блокирует вызывающий процесс до прихода запроса на соединение. Когда этот запрос придет, локальная операционная система создаст новый сокет с теми же свойствами, что и у базового, и возвратит его вызывающему процессу. Такой подход позволяет серверу, например, разветвить про-

цесс, который впоследствии будет поддерживать связь через новое соединение. Сервер в это время может вернуться в состояние ожидания следующего запроса на соединение с базовым сокетом.

Рассмотрим теперь, как все это выглядит со стороны клиента. И здесь все начинается с создания сокета при помощи примитива `socket`, однако в явной привязке сокета к локальному адресу нет необходимости, поскольку операционная система может динамически выделить порт при установлении соединения. Примитив `connect` требует, чтобы вызывающий процесс указал адрес транспортного уровня, на который будет отправлен запрос на соединение. Клиент блокируется до тех пор, пока соединение не будет установлено. После установления соединения стороны начинают обмениваться информацией при помощи примитивов `write` и `read`, предназначенных для отправки и приема данных соответственно. Наконец, закрытие соединения при использовании сокетов симметрично и может быть осуществлено как клиентом, так и сервером путем вызова примитива `close`. Общая схема взаимодействия клиента и сервера с использованием сокетов для коммуникаций, ориентированных на соединение, показана на рис. 2.22. Множество подробностей, относящихся к сетевому программированию с использованием сокетов и других интерфейсов в среде UNIX, можно найти в [438].

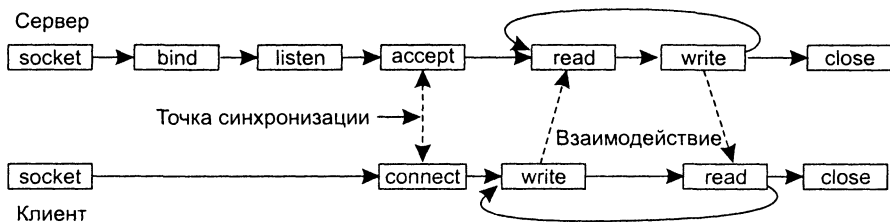


Рис. 2.22. Общая схема ориентированного на соединение взаимодействия с использованием сокетов

Интерфейс передачи сообщений

Работая на высокопроизводительных мультимедийных системах, разработчики рассматривают примитивы, ориентированные на передачу сообщений, как средство, которое облегчит им написание высокоэффективных приложений. Это означает, что такие примитивы должны поддерживать подходящий уровень абстракции (для облегчения разработки приложений), а их реализация должна вызывать минимум дополнительных накладных расходов. Для сокетов это считается неосуществимым по двум причинам. Во-первых, их уровень абстракции явно недостаточен — они поддерживают только простейшие примитивы `send` и `receive`. Во-вторых, сокеты были разработаны для связи между сетями с использованием стеков протоколов общего назначения, таких как TCP/IP. Они не подходят для специальных протоколов, разработанных для высокоскоростных взаимодействующих сетей, например таких, которые используются в системе COW или MPP (мы рассматривали их в разделе 1.3). Эти протоколы требуют интерфейса, обладающего множеством дополнительных возможностей, таких как различные варианты буферизации и синхронизации.

В результате большинство взаимодействующих сетей и мультимпьютерных систем оснащалось собственными коммуникационными библиотеками. Эти библиотеки содержали множество эффективных высокоуровневых коммуникационных примитивов. Разумеется, все библиотеки были абсолютно несовместимыми друг с другом, и теперь разработчики приложений имеют очевидные проблемы с переносимостью.

Требование независимости от аппаратного обеспечения постепенно привело к созданию стандарта пересылки сообщений, названному просто *интерфейсом передачи сообщений* (*Message-Passing Interface, MPI*). MPI разрабатывался для параллельных приложений, но затем был перенесен на нерезидентное взаимодействие. Он предполагает использование базовых сетей и не предусматривает ничего, напоминающего коммуникационные серверы (см. рис. 2.19). Кроме того, он предусматривает, что серьезные сбои в системе, такие как аварии процессов или участков сети, фатальны и не могут быть восстановлены автоматически.

MPI предполагает, что связь происходит в пределах известной группы процессов. Каждая группа получает идентификатор. Каждый процесс в группе также получает идентификатор (локальный). Пара идентификаторов (*groupID, processID*), таким образом, однозначно определяет источник или получателя сообщения и используется вместо адреса транспортного уровня. В вычислениях может участвовать несколько, возможно перекрывающихся, групп процессов, исполняемых одновременно.

В основе MPI лежат примитивы передачи сообщений, поддерживающие большинство видов нерезидентного взаимодействия, показанных на рис. 2.21 (в–е), наиболее понятные из них собраны в табл. 2.2.

Таблица 2.2. Некоторые из наиболее понятных примитивов MPI

Примитив	Назначение
MPI_bsend	Поместить исходящее сообщение в локальный буфер отсылки
MPI_send	Послать сообщение и ожидать, пока оно не будет скопировано в локальный или удаленный буфер
MPI_ssend	Послать сообщение и ожидать начала его передачи на обработку
MPI_sendrecv	Послать сообщение и ожидать ответа
MPI_isend	Передать ссылку на исходящее сообщение и продолжить работу
MPI_issend	Передать ссылку на исходящее сообщение и ожидать начала его передачи на обработку
MPI_recv	Принять сообщение, блокировать работу в случае его отсутствия
MPI_irecv	Проверить наличие входящих сообщений, не блокируя работы

Фактически без поддержки остались только синхронные взаимодействия, показанные на рис. 2.21, г. Другими словами, MPI не поддерживает синхронизацию отправителя и получателя при передаче сообщения по сети.

Нерезидентная асинхронная связь (см. рис. 2.21, в) поддерживается примитивом MPI_bsend. Отправитель отправляет сообщение на передачу. Обычно оно сперва копируется в локальный буфер исполняющей системы MPI. После того

как сообщение скопировано, отправитель продолжает работу. Локальная исполняющая система MPI удалит сообщение из буфера и примет меры по его передаче получателю сразу же, как только получатель запустит примитив, отвечающий за прием.

Также существует и операция передачи с блокировкой, которая называется MPI_send. Ее семантика зависит от реализации. Примитив MPI_send может блокировать отправителя как на время копирования сообщения в исполняющую систему MPI на стороне отправителя, так и до момента инициирования получателем операции приема. Первый случай соответствует асинхронной связи, показанной на рис. 2.21, г, второй — на рис. 2.21, д.

Синхронная связь, при которой отправитель блокируется до передачи сообщения на дальнейшую обработку, как показано на рис. 2.21, д, поддерживается при помощи примитива MPI_ssend.

И наконец, наиболее жесткая форма синхронных коммуникаций показана на рис. 2.21, е. Когда отправитель вызывает примитив MPI_sendrecv, он посылает получателю сообщение и блокируется до получения ответа. В основе работы этого примитива лежит обычный механизм RPC.

Примитивы MPI_send и MPI_ssend имеют варианты, исключающие необходимость копирования сообщения из буфера пользователя во внутренний буфер локальной исполняющей системы MPI. Эти варианты соответствуют асинхронной связи. При помощи примитива MR_isend отправитель передает указатель на сообщение, после чего исполняющая система MPI начинает взаимодействие. Отправитель немедленно продолжает свою работу. Чтобы избежать изменения сообщения до момента окончания связи, MPI предоставляет примитивы для проверки завершения передачи или, при необходимости, блокировки. Как и в случае MR_send, вопрос о том, нужно ли реально передавать сообщение или достаточно копирования в локальный буфер исполняющей системы MPI, оставлен на усмотрение разработчиков.

В случае вызова примитива MR_issend отправитель также передает исполняющей системе MPI только указатель. Когда исполняющая система показывает, что она обработала сообщение, отправитель удостоверяется, что получатель получил сообщение и в настоящее время работает с ним.

Операция MPI_recv вызывается для приема сообщения и блокирует запустивший процесс до прихода сообщения. Существует также и асинхронный вариант этой операции под именем MPI_irecv, вызовом которого получатель показывает, что он готов к приему сообщений. Получатель может проверить, имеются ли пришедшие сообщения, или заблокироваться в ожидании таковых.

Семантика коммуникационных примитивов MPI не всегда проста, и иногда замена различных примитивов никак не влияет на правильность программы. Официальная причина поддержки такого разнообразия вариантов взаимодействия состоит в том, что разработчики систем MPI должны иметь все возможности для оптимизации производительности. Циники могут сказать, что комитет, как всегда, не нашел в себе сил хорошенько подумать. Интерфейс MPI был разработан для высокопроизводительных параллельных приложений, что упрощает понимание причин подобного разнообразия коммуникационных примитивов.

Множество сведений по MPI содержится в [185]. Полное руководство, в котором детально разобрано более 100 функций MPI, можно найти в [184, 425].

2.4.3. Сохранная связь на основе сообщений

Ну вот мы и подошли к важному классу ориентированных на сообщения служб промежуточного уровня, известных как *системы очередей сообщений* (*message-queuing systems*), или *ориентированный на сообщения промежуточный уровень* (*Message-Oriented Middleware, MOM*). Системы очередей сообщений создают расширенную поддержку асинхронной сохранной связи. Смысл этих систем заключается в том, что они предоставляют возможность промежуточного хранения сообщений, не требуя активности во время передачи сообщений ни от отправителя, ни от получателя. Их существенное отличие от сокетов Беркли и интерфейса MPI состоит в том, что системы очередей сообщений обычно предназначены для поддержки обмена сообщениями, занимающего минуты, а не секунды или миллисекунды. В начале мы рассмотрим общий подход к системам очередей сообщений, а закончим этот пункт сравнением их с традиционными системами, под которыми мы понимаем системы электронной почты Интернета.

Модель очередей сообщений

Основная идея, лежащая в основе систем очередей сообщений, состоит в том, что приложения общаются между собой путем помещения сообщений в специальные очереди. Эти сообщения передаются по цепочке коммуникационных серверов и в конце концов достигают места назначения, даже в том случае, если получатель в момент отправки сообщения был неактивен. На практике большинство коммуникационных серверов напрямую соединены друг с другом. Другими словами, сообщение обычно пересылается непосредственно на сервер получателя. В принципе каждое приложение имеет собственную очередь, в которую могут посылать сообщения другие приложения. Очередь может быть прочитана только связанным с ней приложением, при этом несколько приложений могут совместно использовать одну очередь.

Важный момент в системах очередей сообщений состоит в том, что отправитель обычно в состоянии гарантировать только попадание сообщения — рано или поздно — в очередь получателя. Никакие гарантии относительно того, будет ли сообщение действительно прочитано, невозможны, это полностью определяется поведением получателя.

Подобная семантика определяет слабосвязанное взаимодействие. Именно поэтому у получателя нет необходимости быть активным в то время, когда сообщение пересылается в его очередь. Точно так же нет нужды и в активности отправителя во время обработки сообщения получателем. Отправитель и получатель могут выполняться абсолютно независимо друг от друга. На самом деле, как только сообщение поставлено в очередь, оно будет оставаться в ней до удаления, независимо от того, активен его отправитель или его получатель. Это, в зависимости от состояния отправителя и получателя, дает нам четыре комбинации, показанные на рис. 2.23.

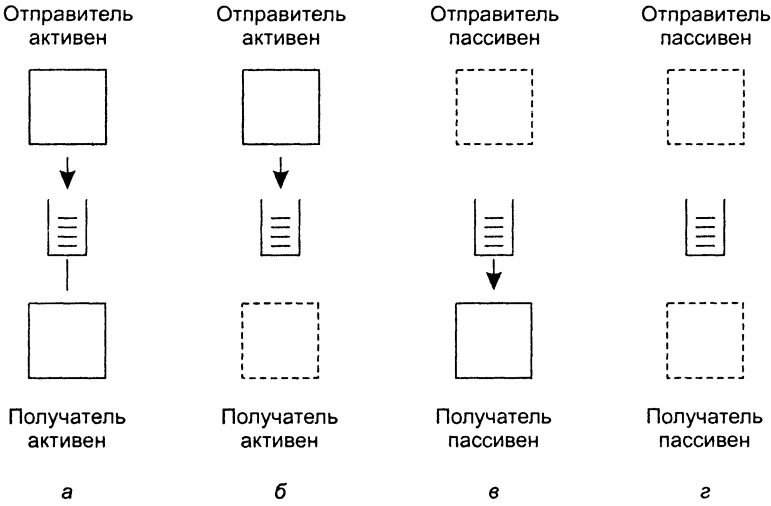


Рис. 2.23. Четыре комбинации слабосвязанных взаимодействий с использованием очередей

На рис. 2.23, *а* как отправитель, так и получатель в ходе всего процесса передачи сообщения находятся в активном состоянии. На рис. 2.23, *б* активен только отправитель, в то время как получатель отключен, то есть находится в состоянии, исключающем возможность доставки сообщения. Тем не менее отправитель все же в состоянии отправлять сообщения. Комбинация из активного получателя и пассивного отправителя приведена на рис. 2.23, *в*. В этом случае получатель может прочитать сообщения, которые были посланы ему ранее, наличия работающих отправителей этих сообщений при этом совершенно не требуется. И наконец, на рис. 2.23, *г* мы наблюдаем такую ситуацию, когда система сохраняет и, возможно, передает сообщения, даже при неработающих отправителе и получателе.

Сообщения в принципе могут содержать любые данные. Единственно важный момент — они должны быть правильно адресованы. На практике адресация осуществляется путем предоставления уникального в пределах системы имени очереди, в которую направляется письмо. В некоторых случаях размер сообщений может быть ограничен, несмотря на то, что, возможно, базовая система в состоянии разбивать большие сообщения на части и собирать их обратно в единое целое абсолютно прозрачно для приложений. Эффект подобного подхода — в том, что базовый интерфейс, предоставляемый приложениям, в результате можно сделать потрясающе простым. Это иллюстрирует табл. 2.3.

Таблица 2.3. Базовый интерфейс очереди в системе очередей сообщений

Примитив	Назначение
Put	Добавить сообщение в соответствующую очередь
Get	Приостановить работу до тех пор, пока в соответствующей очереди не появятся сообщения, затем извлечь первое сообщение

Таблица 2.3 (продолжение)

Примитив	Назначение
Poll	Проверить наличие сообщений в соответствующей очереди, затем извлечь первое из них. Ничего не блокировать
Notify	Вставить дескриптор функции, которая будет вызвана при помещении сообщения в соответствующую очередь

Примитив `put` вызывается отправителем для передачи сообщения базовой системе, где оно будет помещено в соответствующую очередь. Как мы объясняли, это неблокирующий вызов. Примитив `get` — это блокирующий вызов, посредством которого авторизованный процесс может извлечь самое старое сообщение из определенной очереди. Процесс блокируется только в том случае, если очередь пуста. Варианты этого вызова включают в себя поиск в очереди определенного сообщения, например, с использованием приоритетов или образца для сравнения. Неблокирующий вариант представлен примитивом `poll`. Если очередь пуста или искомое сообщение не найдено, вызвавший этот примитив процесс продолжает свою работу.

И, наконец, большинство систем очередей сообщений поддерживают также процесс вставки дескриптора *функции обратного вызова* (*callback function*), которая автоматически вызывается при попадании сообщения в очередь. Обратные вызовы могут быть также использованы для автоматического запуска процесса, который будет забирать сообщения из очереди, если ни один процесс в настоящее время не запущен. Подобное поведение обычно реализуется при помощи демона на стороне получателя, который постоянно проверяет очередь на наличие входящих сообщений и поступает в соответствии с результатами проверки.

Общая архитектура системы очередей сообщений

Давайте теперь рассмотрим поближе, как выглядит обобщенная система очередей сообщений. Одно из первых ограничений, которые мы сделаем, будет состоять в том, что сообщения могут быть помещены только в локальные очереди отправителя, то есть в очереди, находящиеся на той же самой машине или, во всяком случае, не дальше, чем на соседней, то есть машине, входящей в ту же локальную сеть. Подобная очередь называется *исходящей очередью* (*source queue*). Аналогично, прочитанные сообщения могут быть только из локальных очередей. Сообщение, помещенное в очередь, содержит описание *очереди назначения* (*destination queue*), в которую оно должно быть перемещено. В обязанности системы очередей сообщений входит предоставление очередей отправителям и получателям и обеспечение перемещения сообщений из исходящих очередей в очереди назначения.

Важно понимать, что набор очередей разнесен по множеству машин. Соответственно, для того чтобы система очередей сообщений могла перемещать сообщения, она должна поддерживать отображение очередей на сетевые адреса. На практике это означает, что она должна поддерживать базу данных (возможно, распределенную) *имен очередей* (*queue names*) в соответствии с их сетевым местоположением, как показано на рис. 2.24. Отметим, что это отображение полностью анало-

гично использованию системы доменных имен (DNS) для электронной почты Интернета. Так, например, для отсылки почты на логический *почтовый* адрес `steen@cs.vu.nl` почтовая система запрашивает у DNS *сетевой* адрес (то есть IP-адрес) почтового сервера получателя, чтобы затем использовать его для передачи сообщений.

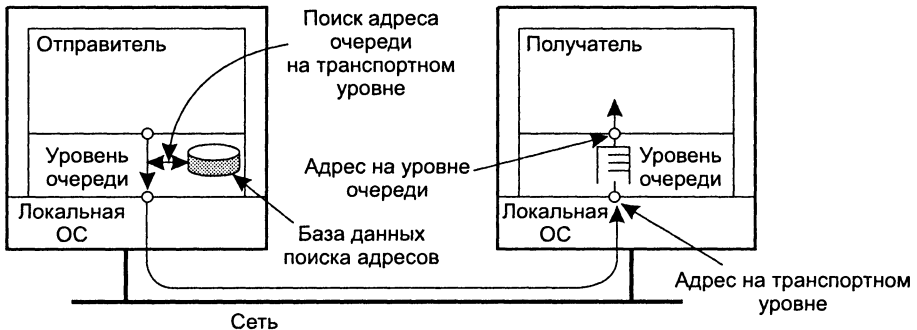


Рис. 2.24. Отношение между адресацией на уровне очередей и на сетевом уровне

Очереди управляются *менеджерами очередей* (*queue managers*). Обычно менеджер очередей взаимодействует непосредственно с отправляющими и принимающими сообщения приложениями. Существуют, однако, и специализированные менеджеры очередей, которые работают как маршрутизаторы, или ретрансляторы: они перенаправляют приходящие сообщения другим менеджерам очередей. Таким образом, система очередей сообщений может постепенно вырасти до законченной *оверлейной сети* (*overlay network*) прикладного уровня поверх существующей компьютерной сети. Этот подход схож с ранней конструкцией системы MBone поверх Интернета, в которой обычные пользовательские процессы конфигурировались для маршрутизации групповых рассылок. В наши дни многие маршрутизаторы сами поддерживают групповые рассылки и необходимость в оверлейных групповых рассылках сильно сократилась.

Ретрансляторы могут быть удобны по нескольким причинам. Так, во многих системах очередей сообщений не существует общих служб именования, которые могли бы динамически поддерживать отображение имен на адреса. Вместо этого топология сети с очередями сделана статической, а каждый менеджер очередей имеет копию отображения очередей на адреса. Нельзя не сказать, что в крупномасштабных системах очередей сообщений подобный подход легко может привести к трудностям в управлении сетью.

Одним из решений этой проблемы является использование нескольких маршрутизаторов, имеющих информацию о топологии сети. Когда отправитель *A* помещает в локальную очередь сообщение для получателя *B*, это сообщение первым делом передается на ближайший маршрутизатор *R1*, как показано на рис. 2.25. При этом маршрутизатор знает, что делать с этим сообщением, и передает его в направлении *B*. Так, *R1* может понять по имени *B*, что сообщение следует передать на маршрутизатор *R2*. Таким образом, при добавлении или удалении очере-

дей обновление информации потребуется только маршрутизаторам, в то время как остальным менеджерам очередей достаточно будет знать только местоположение ближайшего маршрутизатора.

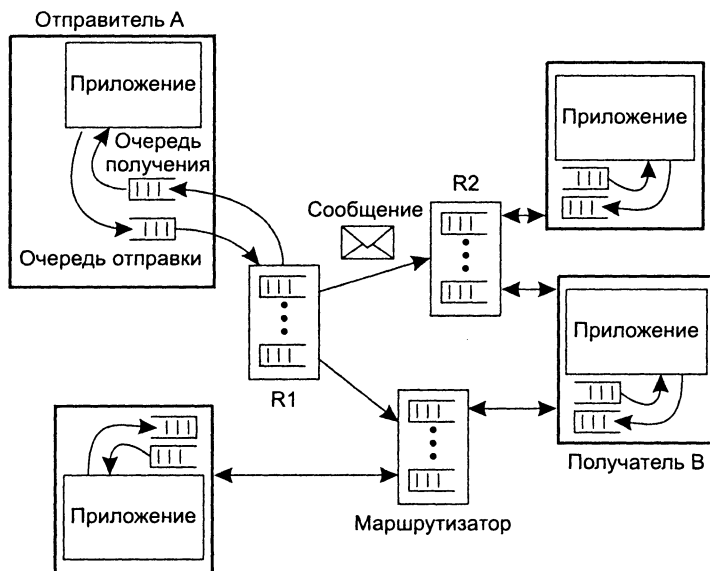


Рис. 2.25. Обобщенная организация систем очередей сообщений с маршрутизаторами

Следовательно, маршрутизаторы могут помочь в создании масштабируемых систем очередей сообщений. Однако по мере роста сети очередей сообщений становится ясно, что ручное конфигурирование этой сети скоро станет невозможным. Единственным решением будет использование динамической схемы маршрутизации, такой же как в компьютерных сетях. В этом отношении легкое удивление вызывает тот факт, что подобные решения до сих пор не встроены в какие-нибудь популярные системы очередей сообщений.

Другой причиной использования ретрансляторов является их способность производить вторичную обработку сообщений. Так, например, в целях безопасности или защиты от сбоев может быть необходимо ведение журнала сообщений. Специальный тип ретрансляторов, о котором мы поговорим в следующем пункте, в состоянии работать как шлюз, преобразуя сообщения в удобный для получателя вид.

И, наконец, ретрансляторы могут использоваться для групповой рассылки. В этом случае входящее сообщение просто помещается в каждую из исходящих очередей.

Брокеры сообщений

Важнейшей областью применения систем очередей сообщений является интеграция существующих и новых приложений в единые согласованные распределенные информационные системы. Интеграция требует, чтобы приложения понимали сообщения, которые они получают. На практике это означает, что от-

правляемые сообщения должны иметь тот формат, которого ожидает получатель.

Проблема подобного подхода в том, что каждый раз, когда в систему добавляется новое приложение, оно привносит в нее свой специфический формат сообщений и каждый потенциальный получатель должен научиться понимать этот формат, на случай, если он получит сообщение от этого нового приложения.

Альтернатива состоит в том, чтобы принять единый формат сообщений, как это было сделано с традиционными сетевыми протоколами. К сожалению, этот подход для систем очередей сообщений в общем случае неприменим. Проблема состоит в уровне абстракций, которыми оперируют эти системы. Единый формат сообщений имеет смысл, только если набор процессов, использующих этот формат, в реальности достаточно однотипен. Если набор приложений, составляющих распределенную информационную систему, в значительной степени разнороден (а так обычно и бывает), то самым лучшим единым форматом будет простая последовательность байтов.

Несмотря на то что для отдельных прикладных областей можно определить свои единые форматы сообщений, общий подход состоит в том, чтобы научиться существовать в мире разнообразных форматов, стараясь предоставить средства для максимально простого преобразования сообщений из одного формата в другой. В системах очередей сообщений преобразование производится на специальных узлах сети массового обслуживания, известных под названием *брокеров сообщений* (*message brokers*). Брокер сообщений работает как шлюз прикладного уровня в системе очередей сообщений. Его основная задача — преобразование входящих сообщений в формат, который понимается целевым приложением. Заметим, что для системы очередей сообщений брокер сообщений — это просто еще одно приложение, как показано на рис. 2.26. Другими словами, брокер сообщений обычно не считается неотъемлемой частью системы сообщений.

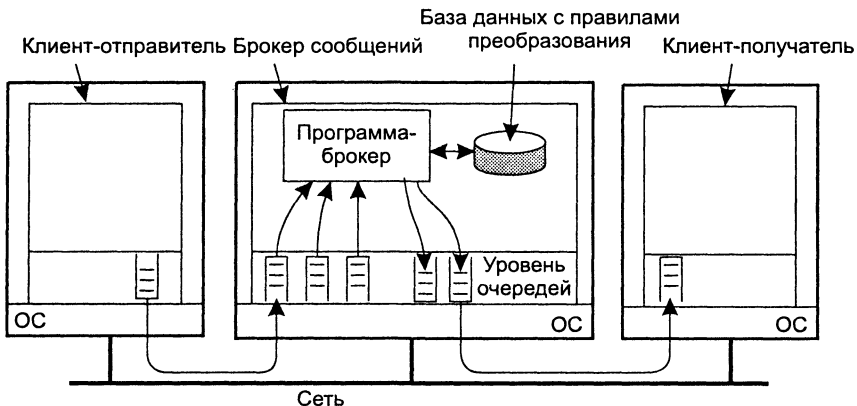


Рис. 2.26. Обобщенная организация брокера сообщений в системе очередей сообщений

Брокер сообщений может просто переформатировать сообщения. Предположим, например, что входящее сообщение содержит таблицу базы данных, записи

в которой разделены специальным разделителем, а поля в записи имеют известную фиксированную длину. Если целевое приложение рассчитано на другой разделитель записей и переменную длину полей, то для преобразования сообщений в формат, используемый приложением-получателем, можно использовать брокер сообщений.

В более совершенном варианте брокер сообщений может выполнять обязанности шлюза прикладного уровня, осуществляя, например, преобразование почтовых сообщений между сетью X.400 и Интернетом. В этих случаях часто невозможно гарантировать, что вся информация, содержащаяся в пришедшем сообщении, действительно будет преобразована в соответствующие части исходящего сообщения. Другими словами, возможно, нам придется смириться с частичной потерей информации в ходе преобразования [9, 204].

Сердцем брокера сообщений является база данных с правилами, определяющими, каким именно образом сообщение в формате *X* конвертируется в сообщение в формате *Y*. Проблема состоит в определении правил. Большинство брокеров сообщений поставляются в комплекте со сложными утилитами определения правил, но в их основе по-прежнему лежит ручной ввод правил в базу данных. Правила могут формулироваться на специальном языке преобразований, но в большинстве брокеров сообщений возможно также программное преобразование с использованием стандартных языков программирования. Установка брокера сообщений — это обычно весьма непростая задача.

Замечание по системам очередей сообщений

Обдумывая то, что мы говорили по поводу систем очередей сообщений, мы приходим к выводу, что они давно уже существуют в виде служб электронной почты. Системы электронной почты обычно реализуются посредством набора почтовых серверов, которые хранят и пересылают почту для пользователей, причем хосты пользователей соединены с сервером напрямую. Маршрутизация обычно отсутствует, поскольку системы электронной почты могут непосредственно использовать базовые транспортные службы. Так, например, в почтовом протоколе SMTP, используемом в Интернете [360], сообщения пересылаются путем установления прямого TCP-соединения с принимающим почтовым сервером.

Системы электронной почты особенно похожи на системы очередей сообщений своей направленностью на предоставление непосредственной поддержки конечным пользователям. Это объясняет, например, почему множество приложений для поддержки групповой работы основаны непосредственно на системах электронной почты [234]. Кроме того, системы электронной почты могут иметь очень специфические свойства, например способность автоматической фильтрации сообщений, поддержку специальной базы данных сообщений (например, для быстрого поиска сохраненных сообщений) и т. д.

Вообще же системы очередей сообщений предназначены не только для поддержки конечных пользователей. Важной областью их применения является организация сохранного взаимодействия между процессами, безразлично, запускают эти процессы пользовательские приложения, обслуживают доступ к базе данных, осуществляют вычисления или делают что-то еще. Такой подход приво-

дит к различиям в требованиях к системам очередей сообщений и «чистым» системам электронной почты. Так, например, системы электронной почты обычно не обеспечивают гарантированной доставки сообщений, приоритетов сообщений, средств протоколирования, эффективных групповых рассылок, выравнивания загрузки, защиты от сбоев и пр.

Поэтому системы очередей сообщений общего назначения включают широкий спектр приложений, в том числе приложений электронной почты, рабочих потоков, средств групповой работы и пакетной обработки. Однако может быть наиболее важная область применения — это интеграция наборов баз данных или приложений баз данных (возможно, сильно распределенных) в информационную систему с множеством баз данных. По этой теме можно порекомендовать работы [337, 416]. Так, запрос, затрагивающий несколько баз данных, может нуждаться в разбиении на более мелкие запросы, перенаправляемые к отдельным базам. Системы очередей сообщений помогают предоставлять основные возможности упаковки каждого из подзапросов в сообщение и их направления к соответствующим базам данных. Другие средства связи, обсуждаемые в этой главе, подходят для этого значительно хуже.

2.4.4. Пример — IBM MQSeries

Чтобы нам было легче понять, как реально работают системы очередей сообщений, рассмотрим конкретную систему, а именно MQSeries компании IBM [169]. Эта система приобрела популярность в относительно традиционной для IBM области мейнфреймов, которые используются для организации доступа и обработки больших баз данных. Важнейшая область применения MQSeries — обработка финансовых данных.

Обзор

Базовая архитектура сети массового обслуживания MQSeries весьма проста (рис. 2.27). Все очереди управляются *менеджерами очередей* (*queue managers*). Менеджер очередей отвечает за извлечение сообщений из выходных очередей и пересылку их другим менеджерам очередей. Кроме того, менеджер очередей отвечает за обработку входящих сообщений, извлекаемых им из базовой сети и сохраняемых в соответствующих входных очередях.

Менеджеры очередей попарно соединены *каналами сообщений* (*message channels*), которые представляют собой абстракцию соединений транспортного уровня. Канал сообщений — это ненаправленное надежное соединение между отправляющим и принимающим менеджерами очередей, по которому передаются находящиеся в очереди сообщения. Канал сообщений на базе Интернета, например, реализуется в виде соединения TCP. Каждый из двух концов канала сообщений управляется *агентом канала сообщений* (*Message Channel Agent, MCA*). Отправляющий агент MCA обычно только проверяет исходящие очереди на наличие в них сообщений, упаковывает их в пакеты транспортного уровня и пересылает по соединению, соответствующему принимающему агенту MCA. Основная зада-

ча принимающего агента МСА — отслеживать приход пакетов, извлекать из них содержимое и сохранять извлеченные сообщения в соответствующих очередях.

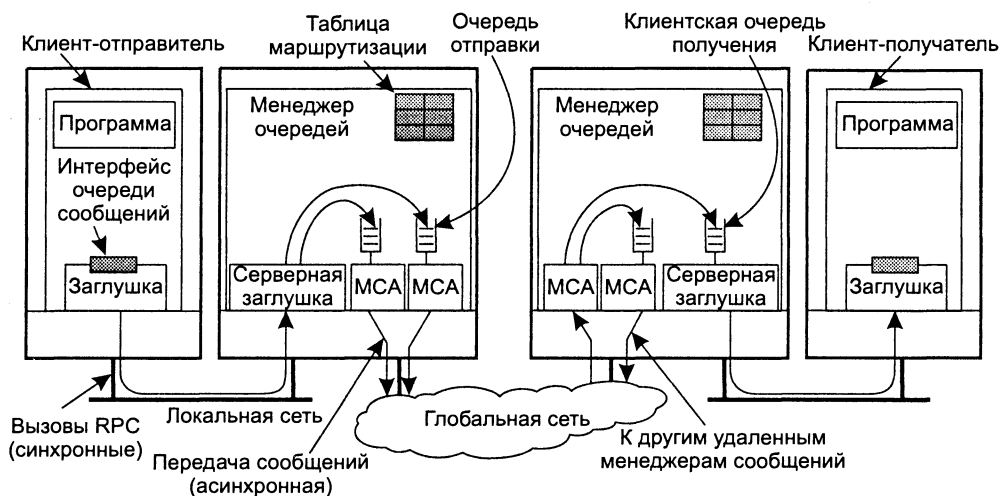


Рис. 2.27. Обобщенная организация системы очередей сообщений IBM MQSeries

Менеджеры очередей могут быть скомпонованы с каким-либо процессом в единое приложение, управляющее очередями. В этом случае очереди скрываются от приложений за стандартным интерфейсом, но непосредственная работа приложения с очередями может оказаться более эффективной. Другим вариантом организации может быть запуск и работа менеджеров очередей и приложений на разных машинах. В этом случае приложениям предоставляется такой же интерфейс, как и при размещении менеджеров очередей на одной с ними машине. Однако сам интерфейс реализуется в виде заместителя, который связан с менеджером очередей традиционной синхронной связью на базе RPC. Таким образом, система MQSeries в своей основе сохраняет модель, согласно которой доступ имеется только к локальным по отношению к приложениям очередям сообщений.

Каналы

Каналы сообщений представляют собой важный компонент системы MQSeries. Каждый канал сообщений имеет только одну связанную с ним очередь отправки. Из этой очереди выбираются сообщения, которые следует переслать на другой конец канала. Передача по каналу может происходить только в том случае, если активны как отправляющий, так и принимающий агенты МСА.

Существует несколько способов инициировать канал, альтернативных запуску обоих агентов МСА вручную, и некоторые из них мы сейчас рассмотрим.

Одной из альтернатив является прямой запуск приложением своего конца канала путем активизации принимающего или передающего агента МСА. Однако с точки зрения прозрачности это не слишком привлекательная альтернатива.

Более интересным подходом к запуску *передающего* агента МСА является конфигурирование очереди отправки канала на изменение состояния триггера при помещении в очередь первого сообщения. Этот триггер связывается с обработчиком запуска передающего агента МСА, который извлекает сообщения из очереди отправки.

Другой альтернативой является запуск МСА по сети. Так, если одна сторона канала уже активна, она может послать управляющее сообщение, требующее запуска другого агента МСА. Это управляющее сообщение посылается демону, просматривающему общедоступный адрес той машины, на которой мы хотим запустить второй агент МСА.

Каналы прекращают свое существование автоматически после того, как в течение определенного времени в очередь отправки не поступит ни одного сообщения.

Каждый из агентов МСА имеет связанный с ним набор атрибутов, которые определяют общие свойства канала. Некоторые из этих атрибутов перечислены в табл. 2.4. Значения атрибутов принимающего и передающего агентов МСА должны быть совместимыми. Обычно агенты МСА договариваются об их значениях перед запуском канала. Так, например, оба агента МСА должны, очевидно, поддерживать один и тот же транспортный протокол. Примером необсуждаемого атрибута может быть очередность передачи сообщений — та же, в которой сообщения поступают в очередь отправки, или иная. Если один из агентов МСА запрашивает отсылку по алгоритму FIFO, другой обязан выполнить это требование. Примером обсуждаемого атрибута может быть максимальная длина сообщения, величина которой просто выбирается по минимальному значению этого атрибута для обоих агентов МСА.

Таблица 2.4. Некоторые атрибуты агента канала сообщений

Атрибут	Описание
Тип транспорта	Используемый транспортный протокол
Доставка FIFO	Определяет, что доставка сообщений осуществляется в том же порядке, что и посылка
Длина сообщения	Максимальная длина одного сообщения
Установка числа повторений	Максимальное число повторений при запуске удаленного агента МСА
Попытки доставки	Максимальное число попыток МСА поместить полученное сообщение в очередь

Передача сообщения

Для передачи сообщения одним менеджером очередей другому (возможно, удаленному) необходимо, чтобы каждое сообщение несло в себе адрес назначения. Для этого используется заголовок сообщения. Адрес в MQSeries образован из двух частей. Первая часть состоит из имени менеджера очередей, которому это сообщение должно быть доставлено, а вторая часть — это имя очереди назначения, сообщающее этому менеджеру, к какой очереди добавлять сообщение.

Кроме адреса получателя необходимо также определить маршрут, которым должно следовать сообщение. Описание маршрута производится путем задания имени локальных очередей отправки, в которые должно добавляться сообщение. Нет необходимости задавать в сообщении полное описание маршрута. Напомним, что каждый канал сообщений имеет всего одну очередь отправки. Указав, в какую очередь отправки должно быть добавлено сообщение, мы однозначно определим, какому соседнему менеджеру очередей будет передано это сообщение.

В большинстве случаев маршруты явно сохраняются в таблицах маршрутизации внутри менеджеров очередей. Запись в таблице маршрутизации представляет собой пару (*destQM*, *sendQ*), где *destQM* — имя менеджера очередей, получающего сообщение, а *sendQ* — имя локальной очереди отправки, к которой следует добавлять сообщения для этого менеджера. Запись в таблице маршрутизации в MQSeries называется *псевдонимом* (*alias*).

Возможно, сообщению до прихода к адресату придется миновать несколько менеджеров очередей. Всякий раз, когда подобный промежуточный менеджер очередей получает сообщение, он просто извлекает из заголовка сообщения имя менеджера очередей, которому адресовано сообщение, и просматривает таблицу маршрутизации, отыскивая локальную очередь отправки, в которую следует добавить это сообщение.

Важно понять, что каждый менеджер очередей имеет уникальное в пределах системы имя, которое можно успешно использовать для идентификации этого менеджера очередей. Проблемы могут появиться в случае смены менеджера очередей или изменения его имени, это повлияет на все приложения, посылающие ему сообщения. Эту потенциальную проблему можно ослабить путем использования в качестве имен менеджеров очередей *локальных псевдонимов* (*local alias*). Псевдоним, определенный в менеджере очередей *M1*, — это другое имя менеджера очередей *M2*, доступное только для приложений, работающих с *M1*. Псевдоним позволяет использовать для очереди другое имя (логическое) даже при смене менеджера очередей, отвечающего за данную очередь. Изменение имени менеджера очередей означает смену его псевдонима во всех менеджерах очередей. При этом на приложениях это никак не отразится.

Принцип использования таблиц маршрутизации и псевдонимов иллюстрирует рис. 2.28. Так, например, приложение, соединенное с менеджером очередей *QMA*, может ссылаться на удаленный менеджер очередей, используя его локальный псевдоним *LA1*. Менеджер очередей первым делом ищет в таблице псевдонимов истинное место назначения, каковым оказывается менеджер очередей *QMC*. Маршрут на *QMC* ищется в таблице маршрутизации, согласно которой сообщения для *QMC* должны добавляться к очереди отправки *SQ1*, которая используется для пересылки сообщений менеджеру очередей *QMB*. Последний с помощью своей таблицы маршрутизации пересылает сообщение менеджеру *QMC*.

Руководствуясь таким подходом к маршрутизации и присвоению псевдонимов, мы приходим к относительно простому прикладному программному интерфейсу, который называется *интерфейсом очередей сообщений* (*Message Queue Interface, MQI*). Наиболее важные примитивы MQI собраны в табл. 2.5.

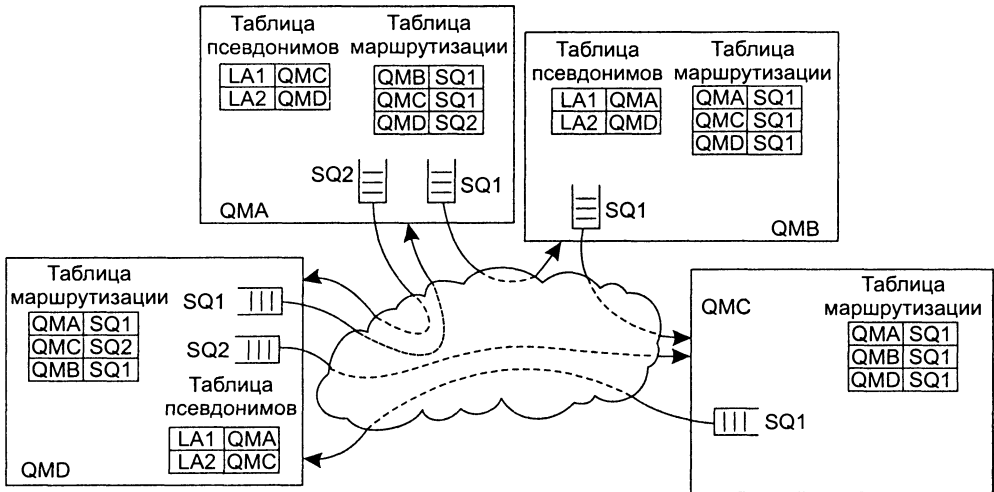


Рис. 2.28. Обобщенная организация сети очередей MQSeries с использованием таблиц маршрутизации и псевдонимов

Таблица 2.5. Примитивы MQI для системы IBM MQSeries

Примитив	Описание
MQopen	Открыть очередь (возможно, удаленную)
MQclose	Закрыть очередь
MQput	Поместить сообщение в открытую очередь
MQget	Получить сообщение из локальной очереди

Для помещения сообщения в очередь приложение вызывает примитив MQopen, определяющий очередь, для которой предназначено сообщение под управлением некоего менеджера очередей. Менеджер очередей может быть указан с использованием локально доступного псевдонима. Независимо от того, является или нет очередь получения реально удаленной, она абсолютно прозрачна для приложений. Если приложение захочет получать сообщения из своей локальной очереди, можно вызвать примитив MQopen.

Для чтения входящих сообщений можно открыть только локальную очередь. Когда приложение заканчивает свою работу с очередью, оно может закрыть ее вызовом примитива MQclose.

Сообщения могут быть записаны в очередь или считаны из нее путем использования соответственно примитивов MQput и MQget. В принципе сообщения извлекают из очереди на основании их приоритета. Сообщения с одинаковым приоритетом извлекаются из очереди по принципу первым пришел — первым ушел (FIFO), то есть дольше всех находящиеся в очереди сообщения извлекаются первыми. Кроме того, можно запросить конкретное сообщение. И наконец, MQSeries предоставляет средства для уведомления приложений о приходе сообщений. Это позволяет приложениям не опрашивать постоянно очередь на предмет прихода новых сообщений.

2.5. Связь на основе потоков данных

Модели взаимодействия, обсуждавшиеся выше, касались обмена более или менее независимыми, законченными порциями информации. Примерами таковых являются запросы и обращения к процедурам, ответы на подобные запросы и обмен сообщениями между приложениями в системах очередей сообщений. Характерной чертой подобного взаимодействия является его индифферентность к тому, в какой конкретно момент времени оно происходит. Несмотря на то, работает система очень быстро или очень медленно, это никак не сказывается на корректности ее работы.

Однако существуют формы взаимодействия, в которых временные характеристики имеют решающее значение. Рассмотрим, например, поток аудиоданных, состоящий из последовательности 16-битных выборок, каждая из которых представляет собой амплитуду звуковой волны в импульсно-кодовой модуляции. Предположим также, что поток аудиоданных имеет качество компакт-дисков, а это значит, что исходный звук был оцифрован с частотой 44 100 Гц. Для воспроизведения звука необходимо, чтобы выборки аудиопотока проигрывались в том же порядке, в котором они представлены в потоке данных, и с интервалами ровно по 1/44100 с. Воспроизведение с другой скоростью создаст неверное представление об исходном звуке.

В этом разделе мы рассмотрим вопрос о том, какие средства могут предложить нам распределенные системы для работы с информацией, критичной к временным характеристикам передачи, такой как видео- и аудиопотоки. Различные сетевые протоколы для взаимодействия на базе потоков данных рассматриваются в [192]. В [434] имеется исчерпывающее введение в вопросы мультимедиа, частью которых является взаимодействие на базе потоков данных.

2.5.1. Поддержка непрерывных сред

Поддержка обмена критичной к временным характеристикам передачи информации часто сводится к поддержке непрерывных сред. Под средой понимается то, что несет информацию. Сюда могут входить среды передачи и хранения, среда представления, например монитор, и т. д. Важнейшая характеристика среды — способ *представления* информации. Другими словами, как информация кодируется в компьютерной системе? Для различных типов информации используются различные представления. Так, текст обычно кодируется символами ASCII или Unicode. Изображения могут быть представлены в различных форматах, например GIF или JPEG. Аудиопотоки в компьютерных системах могут кодироваться, например, с помощью 16-битных выборок, использующих импульсно-кодовую модуляцию.

В *непрерывной среде представления* (*continuous representation media*) временные соотношения между различными элементами данных лежат в основе корректной интерпретации смысла данных. Мы уже давали пример получения звука при воспроизведении аудиопотока. В качестве другого примера рассмотрим представление движения. Движение может быть представлено в виде серии кар-

тинок, причем последовательно идущие картинки должны воспроизводиться в течение одинакового срока T , обычно составляющего 30–40 мс на картинку. Правильное воспроизведение требует не только показа картинок в нужной последовательности, но и поддержания постоянной частоты показа — $1/T$ картинок в секунду.

В отличие от непрерывной среды *дискретная среда представления* (*discrete representation media*), характеризуется тем, что временные соотношения между элементами данных не играют фундаментальной роли в правильной интерпретации данных. Типичными примерами дискретной среды являются представления текста и статических изображений, а также объектный код и исполняемые файлы.

Поток данных

Для обмена критичной ко времени передачи информацией распределенные системы обычно предоставляют поддержку *потоков данных* (*data streams*, или просто *streams*). Поток данных есть не что иное, как последовательность элементов данных. Потоки данных применимы как для дискретной, так и для непрерывной среды представления. Так, каналы UNIX или соединения TCP/IP представляют собой типичные примеры дискретных потоков данных (байт-ориентированных). Воспроизведение звукового файла обычно требует непрерывного потока данных между файлом и устройством воспроизведения.

Временные характеристики важны для непрерывных потоков данных. Для поддержания временных характеристик часто приходится выбирать между различными режимами передачи. В *асинхронном режиме передачи* (*asynchronous transmission mode*) элементы данных передаются в поток один за другим, но на их дальнейшую передачу никаких ограничений в части временных характеристик не вводится. Это традиционный вариант для дискретных потоков данных. Так, файл можно преобразовать в поток данных, но выяснять точный момент окончания передачи каждого элемента данных чаще всего бессмысленно.

В *синхронном режиме передачи* (*synchronous transmission mode*) для каждого элемента в потоке данных определяется максимальная задержка сквозной передачи. Если элемент данных был передан значительно быстрее максимально допустимой задержки, это не важно. Так, например, датчик может с определенной частотой измерять температуру и пересылать эти измерения по сети оператору. В этом случае временные характеристики могут вызвать наш интерес, если время сквозного прохождения сигнала по сети гарантированно ниже, чем интервал между измерениями, но никому не повредит, если измерения будут передаваться значительно быстрее, чем это действительно необходимо.

И, наконец при *изохронном режиме передачи* (*isochronous transmission mode*) необходимо, чтобы все элементы данных передавались вовремя. Это означает, что передача данных ограничена максимально, а также минимально допустимыми задержками, также известными под названием *предельного дрожания*. Isochronous режим передачи, в частности, представляет интерес для распределенных систем мультимедиа, поскольку играет значительную роль в воспроизведении аудио- и видеoinформации. В этой главе мы коснемся только непрерывных потоков данных.

Потоки данных могут быть простыми или комплексными. *Простой поток данных (simple stream)* содержит только одну последовательность данных. *Комплексный поток данных (complex stream)* состоит из нескольких связанных простых потоков, именуемых *вложенными потоками данных (substreams)*. Взаимодействие между вложенными потоками в комплексном потоке часто также зависит от времени. Так, например, стереозвук может передаваться посредством комплексного потока, содержащего два вложенных потока, каждый из которых используется для одного аудиоканала. Важно отметить, что эти два вложенных потока постоянно синхронизированы. Другими словами, элементы данных каждого из потоков должны передаваться попарно для получения стереоэффекта. Другим примером комплексного потока данных может быть поток данных для фильма. Такой поток мог бы состоять из одного видеопотока и двух аудиопотоков для передачи стереофонической звуковой дорожки к фильму. Четвертый поток мог бы содержать субтитры для глухих или синхронный перевод на другой, нежели в звуковой дорожке, язык. И вновь необходима синхронизация вложенных потоков. При нарушениях синхронизации нарушается нормальное воспроизведение фильма. Ниже мы еще вернемся к синхронизации потоков данных.

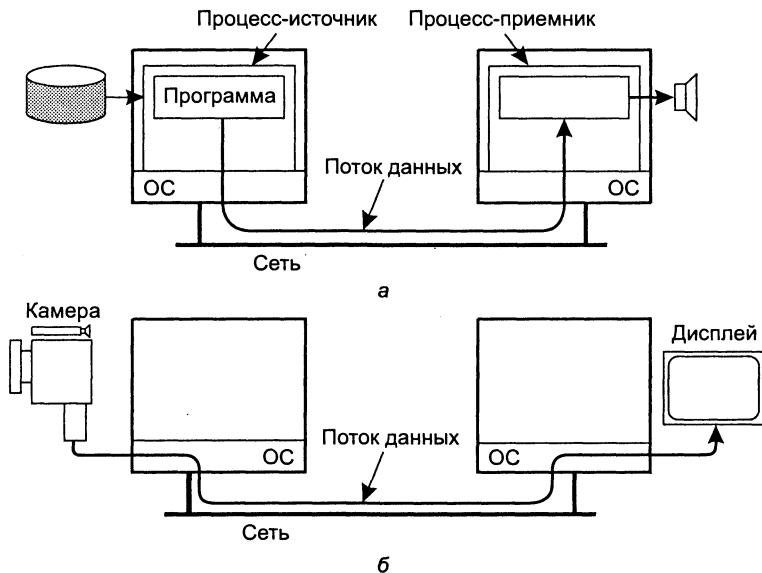


Рис. 2.29. Передача потока данных по сети между двумя процессами (а), между двумя устройствами (б)

Поток данных нередко может рассматриваться в виде виртуального соединения между источником и приемником. Источник или приемник может быть процессом или устройством. Например, при передаче данных через сеть процесс-источник может читать данные из аудиофайла и пересылать их, байт за байтом, по сети. Приемник может быть процессом, по мере поступления выбирающим байты и передающим их на локальное устройство звуковоспроизведения. Такую ситуацию иллюстрирует рис. 2.29, а. С другой стороны, в распределенных мульти-

медийных системах можно реализовать прямое соединение между источником и приемником. Так, видеопоток, создаваемый камерой, может напрямую передаваться на дисплей, как показано на рис. 2.29, б.

Другая ситуация имеет место в зависимости от того, имеется у нас всего один источник или приемник или мы можем организовать многостороннюю связь. Наиболее частая ситуация при многосторонней связи — присоединение к потоку данных нескольких приемников. Другими словами, осуществляется групповая рассылка потока данных нескольким получателям, как показано на рис. 2.30.

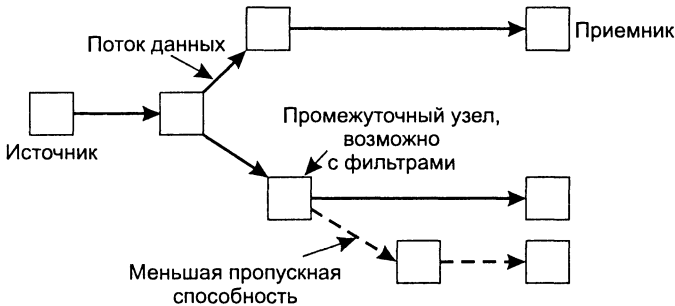


Рис. 2.30. Пример групповой рассылки потока данных нескольким получателям

Главной проблемой групповой рассылки потоков данных являются разные требования разных приемников к качеству потока. Рассмотрим, например, источник передачи высококачественного кино со стереозвуком. Он может потребовать комплексного потока данных, содержащего вложенный поток для видео, по которому с частотой 50 Гц передается картинка, и два вложенных потока для аудио с качеством на уровне компакт-дисков. Даже при использовании современных технологий сжатия комплексный поток может потребовать скорости передачи порядка 30×10^6 бит/с [434]. Не всякий приемник в состоянии обработать такой объем данных. Поэтому поток должен быть настроен на фильтрацию [500], которая приводит в соответствие качество входящего потока и отличное от него качество исходящего потока, как показано на рис. 2.30. Мы вернемся к управлению качеством потоков данных позже.

2.5.2. Потоки данных и качество обслуживания

Временные зависимости и другие нефункциональные требования обычно выражаются в виде требований к *качеству обслуживания* (*Quality of Service, QoS*). Эти требования описывают, что должны сделать базовая распределенная система и сеть для того, чтобы гарантировать, например, сохранение в потоке данных временных соотношений. Требования QoS для непрерывных потоков данных в основном характеризуются временными диаграммами, объемом и надежностью. В этом пункте мы кратко рассмотрим требования QoS и их влияние на создание потока данных.

Специфика QoS

Требования QoS могут быть выражены по-разному. Один из подходов — представить точную *спецификацию передачи (flow specification)*, содержащую требования по пропускной способности, скорости передачи, задержке и т. п. Пример такой спецификации, взятый из [343], приведен в табл. 2.6.

Таблица 2.6. Спецификация передачи

Характеристики получаемых данных	Требуемое качество обслуживания
Максимальный размер элемента данных (байт)	Чувствительность к потерям (байт)
Скорость передачи корзины элементарных пакетов (байт/с)	Чувствительность к интервалам (ммс)
Размер корзины элементарных пакетов (байт)	Чувствительность к групповым потерям (элементов данных)
Максимальная скорость передачи (байт/с)	Минимальная фиксируемая задержка (ммс)
	Максимальное отклонение задержки (ммс)
	Показатель соблюдения (единиц)

В этой модели характеристики потока формулируются в понятиях *алгоритма корзины элементарных пакетов (token bucket algorithm)*, который описывает, каким образом поток формирует сетевой трафик. Принцип работы этого алгоритма иллюстрирует рис. 2.31. Основная идея состоит в том, что элементарные пакеты генерируются с постоянной скоростью. Элементарный пакет содержит фиксированное число байтов, которые приложение намерено передать по сети. Элементарные пакеты собираются в корзину, емкость которой ограничена. После переполнения корзины элементарные пакеты просто пропадают. Каждый раз, когда приложение хочет передать в сеть элемент данных размером N , оно должно извлечь из корзины столько элементарных пакетов, чтобы их суммарный размер был не менее N байт. То есть, например, если каждый элементарный пакет имеет длину k байт, приложение должно удалить из корзины как минимум N/k элементарных пакетов.

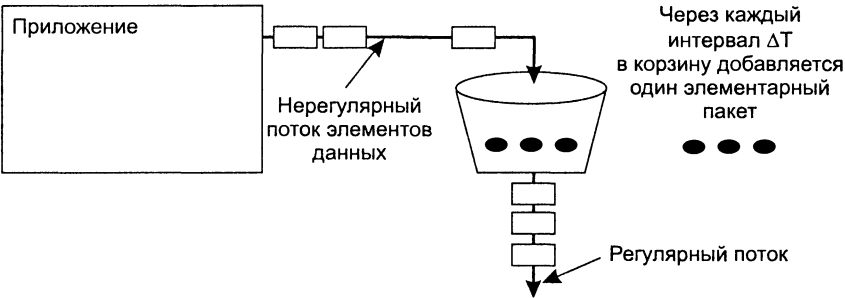


Рис. 2.31. Принцип работы алгоритма корзины элементарных пакетов

Эффект алгоритма корзины элементарных пакетов состоит в том, что данные передаются в сеть с относительно постоянной скоростью, определяемой скоростью генерации элементарных пакетов. Однако он допускает некоторые всплески, например, приложению разрешено передавать полную корзину элементарных пакетов в сеть одной операцией. Чтобы предотвратить особо сильные выплески, скорость потоков данных может быть ограничена некоторым максимумом. В спецификации передачи приложение обещает, что будет поставлять элементы данных коммуникационной системе в соответствии с алгоритмом корзины элементарных пакетов.

Вдобавок к описанию временных отношений между элементами данных, спецификация передачи также содержит требования к обслуживанию. *Чувствительность к потерям* в комбинации с *интервалом потерь* определяет максимально допустимый темп потерь (например, один байт в минуту). *Чувствительность к групповым потерям* определяет, какое количество последовательных элементов данных можно потерять.

Минимальная фиксируемая задержка определяет, на сколько сеть должна задержать доставку данных, чтобы приемник обнаружил задержку. Связанное с этой величиной *максимальное отклонение задержки* определяет максимальный предел дрожания. Величина вилки особенно важна для видео- и аудиопотоков.

И наконец, *показатель соблюдения* — число, показывающее, насколько точно должны соблюдаться требования к качеству обслуживания. Обычно небольшое число говорит, что если коммуникационная система не сможет обеспечить требуемое качество обслуживания, реально ничего страшного не случится. И наоборот, большое число указывает на то, что если невозможно получить надежные гарантии качества обслуживания, система не сможет работать с потоком данных, поскольку клиент не допускает отклонений от спецификации.

Описанная проблема со спецификацией передачи состоит в том, что приложение может попросту не знать, что ему нужно на самом деле. Так, заставляя пользователей описывать качество в понятиях параметров корзины элементарных пакетов, чувствительности к потерям и т. п., поставщик услуг вполне может вскоре растерять всех своих клиентов. Поэтому правильнее будет классифицировать потоки данных и задать разумные значения по умолчанию для параметров, включаемых в спецификации передачи. Так, например, максимум, что можно требовать от пользователя, — это выбрать требуемый поток данных (аудио или видео). При выборе аудиоданных можно предложить еще выбор между высоким, средним и низким качеством. Схожая классификация может быть предложена и для видеоданных.

Как утверждается в [344], классификация не слишком отличается от спецификаций передачи. Отличие состоит только в числе определяемых параметров и числе индивидуальных значений, которые могут принимать эти параметры.

Создание потока

После того как поток данных описан, например, в виде спецификации передачи, распределенная система должна захватить ресурсы для создания потока, удовлетворяющего требованиям QoS. В контексте управления потоками ресурсами

обычно являются пропускная способность, буферы и вычислительная мощность. Выделение пропускной способности производится для того, чтобы гарантировать соблюдение графика передачи элементов данных, например, путем задания приоритетов передачи. Выделение буферов в маршрутизаторах и операционных системах позволяет сохранять элементы данных для дальнейшей обработки. И наконец, необходимо, чтобы обработка элементов данных выполнялась за время, отводимое на это соответствующими задачам — планировщикам, кодерам и декодерам, фильтрам и др. Для этого следует правильно распределить процессорное время.

Одна из тех проблем, которые следует решить, состоит в том, что параметры, характеризующие требования QoS к потоку данных, не соотносятся напрямую с параметрами соответствующих ресурсов. Так, например, определив, что сеть должна гарантировать возможность одновременной потери не более чем k последовательных элементов данных, мы должны преобразовать это определение в размер статически выделяемых буферов во всех маршрутизаторах от источника до приемника. Этот размер может быть на самом деле подсчитан на основе других характеристик потока и привести к абсолютной или статистической гарантии обслуживания в сети.

К сожалению, в настоящее время не существует единственно лучшей модели для, во-первых, выбора параметров QoS, во-вторых, обобщенного описания ресурсов в любой коммуникационной системе и, в-третьих, преобразования параметров QoS в значения используемых ресурсов. Отсутствие подобной модели является причиной того, что описание и организация качественного обслуживания часто нелегка и что разные системы используют разные и несовместимые методы.

Чтобы дополнительно прояснить вопрос и описать, как QoS в распределенных системах зависит от служб, предоставляемых базовой сетью, взглянем на специальный протокол резервирования ресурсов для QoS в непрерывных потоках данных. *Протокол резервирования ресурсов (Resource reSerVation Protocol, RSVP)* — это управляющий протокол транспортного уровня для резервирования ресурсов сетевых маршрутизаторов [72, 503].

Передачики в RSVP предоставляют спецификацию передачи, характеризующую потоки данных в понятиях пропускной способности, задержек, дрожания и т. п., очень напоминающую спецификацию передачи, представленную в табл. 2.6. Это описание передается процессу RSVP, который работает на той же машине, что и отправитель, как показано на рис. 2.32. Процесс RSVP не занимается интерпретацией спецификацией передачи. Фактически единственное, что он делает, — это принимает спецификацию передачи от отправителя и локально сохраняет ее. RSVP — это иницилируемый получателем протокол QoS. Другими словами, получатель должен запросить у отправителя запрос на резервирование. Сохраняя спецификацию передачи, RSVP предотвращает резервирование большего, чем это необходимо, объема ресурсов.

Отправитель в RSVP определяет путь к потенциальным получателям и предоставляет спецификацию передачи для потока данных всем промежуточным узлам. Когда получатель готов принимать входящие элементы данных, он сначала

посылает запрос на резервирование по обратному маршруту своему отправителю. Формат этого запроса в сущности такой же, как у исходной спецификации передачи, но установленные значения параметров могут отражать минимальные требования QoS, которые должен определить отправитель, чтобы полностью удовлетворить запросы получателя.

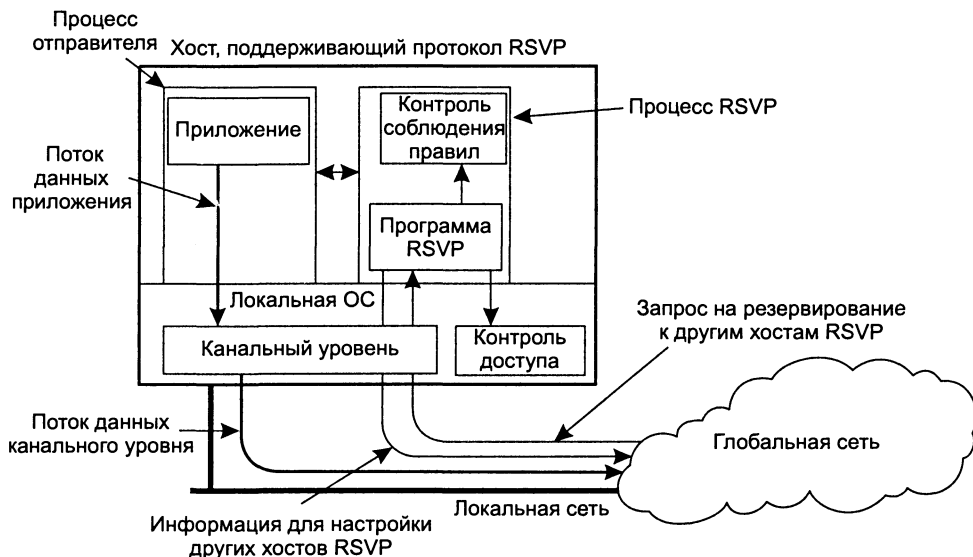


Рис. 2.32. Основы организации процесса RSVP для резервирования ресурсов в распределенных системах

Когда процесс RSVP обрабатывает запрос на резервирование, он передает запрос модулю проверки допустимости, который проверяет, имеются ли в наличии запрашиваемые ресурсы. Запрос передается также и модулю проверки правил, который проверяет, имеет ли получатель право на резервирование. Если ответ обеих проверок — «да», ресурсы могут быть зарезервированы.

Резервирование ресурсов сильно зависит от канального уровня. Фактически, чтобы протокол RSVP работал, процесс RSVP должен преобразовать параметры QoS из спецификации передачи в нечто, понятное канальному уровню. Простой пример: запрос на высокую пропускную способность транслируется в установку максимального приоритета каждого кадра, несущего данные потока. Основываясь на первоначальной спецификации передачи (которая указывает на максимальную скорость передачи данных, генерируемых отправителем) и доступной пропускной способности канального уровня, такое преобразование вполне может удовлетворить требования QoS получателя.

Другой подход применяется, если канальный уровень имеет собственный набор параметров QoS, как, например, в сетях ATM. В сетях ATM данные передаются элементами (которые называются ячейками), состоящими из 48-байтного информационного поля и заголовка длиной 5 байт. ATM позволяет процессу

RSVP задать максимальную скорость ячеек, среднюю скорость ячеек, минимально приемлемую скорость ячеек и максимально допустимое дрожание между ячейками. Существуют и другие параметры QoS. В этом случае задачей процесса RSVP будет трансляция спецификаций передачи, ориентированных на потоки данных, в значения параметров, приемлемые для сетей ATM. Соблюдение этих требований QoS будет проверяться на уровне ATM.

2.5.3. Синхронизация потоков данных

В мультимедийных системах важное значение имеет взаимная синхронизация разных потоков данных, возможно, собранных в комплексный поток. Синхронизация потоков данных подразумевает поддержание временных соотношений между ними. Существует два типа синхронизации.

Простейшая форма синхронизации имеет место между дискретными и непрерывными потоками данных. Рассмотрим, например, показ слайдов через Web, дополненный звуковым рядом. Каждый слайд передается с сервера на клиент в виде дискретного потока данных. В то же самое время клиент воспроизводит некий аудиопоток (или его часть), который соответствует текущему слайду и также поступает с сервера. В данном случае аудиопоток синхронизируется с показом слайдов.

Более сложный тип синхронизации наблюдается между непрерывными потоками данных. Дежурный пример воспроизведения фильма, при котором видеопоток должен быть синхронизирован со звуком, известен как *синхронизация артикуляции*. Другим примером может быть воспроизведение потока стереозвука, состоящего из двух вложенных потоков, по одному на каждый канал. Правильное воспроизведение требует взаимной синхронизации двух вложенных потоков, несовпадение более чем на 20 мс может исказить стереоэффект.

Синхронизация происходит на уровне элементов данных, из которых состоит поток. Другими словами, мы можем синхронизировать два потока только относительно элементов данных. Выбор элементов данных очень сильно зависит от уровня абстракции представления потока данных. Для разъяснения этого момента рассмотрим аудиопоток (одноканальный) с качеством компакт-диска. При максимальной степени детализации поток данных представляется в виде последовательности 16-битных выборок. При частоте выборки 44 100 Гц синхронизация с другим аудиопотоком может теоретически производиться каждые 23 мс. Для получения высококачественного стереоэффекта оказывается, что синхронизация на этом уровне абсолютно необходима.

Однако если мы рассмотрим синхронизацию между аудио- и видеопотоками для синхронизации артикуляции, обнаружится, что в этом случае можно использовать значительно более грубую детализацию. Как мы говорили, видеокadres показываются с частотой 25 Гц и выше. Используя широко распространенный стандарт NTSC с его частотой 30 Гц, мы можем сгруппировать аудиовыборки в логические блоки с временем воспроизведения, равным времени показа видеокadra (33 мс. При частоте выборки звука 44 100 Гц блок аудиоданных будет содержать 1470 отдельных выборок, или 11 760 байт, из расчета 16-битных выбо-

рок). На практике вполне удовлетворительное качество обеспечивается и большими блоками, по 40 или даже 80 мс [435].

Механизмы синхронизации

Ну вот, мы и подошли к вопросу о том, как на самом деле обеспечивается синхронизация. Следует рассмотреть два момента: во-первых, базовые механизмы синхронизации двух потоков данных и, во-вторых, распределение этих механизмов в сетевой среде.

Механизмы синхронизации могут рассматриваться с позиций разных уровней абстракции. С точки зрения самого нижнего, синхронизация полностью определяется работой с элементами данных простых потоков. Этот принцип иллюстрирует рис. 2.33. В сущности, здесь показан процесс, который осуществляет операции чтения и записи в нескольких простых потоках данных, гарантируя при этом обеспечение ограничений синхронизации.

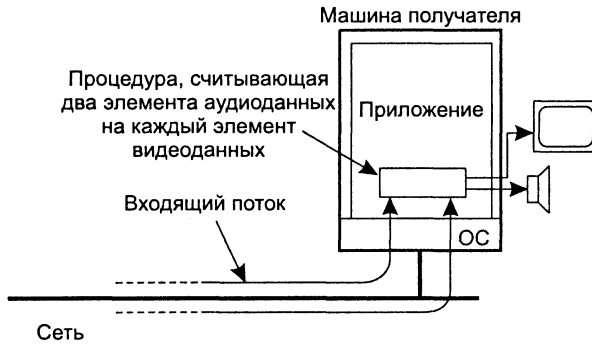


Рис. 2.33. Принцип явной синхронизации на уровне элементов данных

Рассмотрим, например, фильм, представленный двумя входными потоками данных. Видеопоток содержит несжатые изображения низкого качества с разрешением 320×240 пикселей. Каждый из пикселей при кодировании требует одного байта, что в сумме приводит к элементам видеоданных размером в 76 800 байт. Предположим, что изображения демонстрируются с частотой 30 Гц, или по 33 мс на изображение. Считаем, что аудиопоток содержит аудиовыборки, сгруппированные в элементы длиной 11 760 байт, каждый из которых, как говорилось ранее, соответствует звуку длительностью 33 мс. Если процесс ввода может обеспечить скорость 2,5 Мбайт/с, мы можем обеспечить синхронизацию, просто чередуя чтение изображений и блоков аудиовыборок каждые 33 мс.

Оборотная сторона подобного подхода состоит в том, что приложение может полностью отвечать за синхронизацию только в том случае, если оно имеет доступ к механизмам низкого уровня. Лучше будет предоставить приложению интерфейс, который упростил бы ему управление потоками и устройствами. Относительно нашего примера это будет означать, что видеодисплей получит интерфейс управления, с помощью которого сможет управлять частотой воспроизведения изображений. Кроме того, интерфейс предоставит механизм регистра-

ции пользовательского обработчика, который будет вызываться каждый раз после приема очередных k изображений. Аналогичный интерфейс будет предоставлен и устройству воспроизведения звука. Используя эти управляющие интерфейсы, разработчик приложения сможет написать простую программу мониторинга, сравнивающую два заголовка соответствующих потоков данных. Эта программа будет проверять, достаточна ли синхронизация видео- и аудиопотоков и в случае необходимости регулировать частоту передачи видео- или аудиоэлементов.

Этот последний пример (рис. 2.34) типичен для многих систем мультимедиа промежуточного уровня. В действительности системы мультимедиа промежуточного уровня содержат набор интерфейсов для управления аудио- и видеопотоками, включая интерфейсы управляющих устройств — мониторов, камер, микрофонов и т. п. Каждое устройство и поток данных имеют свои собственные интерфейсы верхнего уровня, в том числе и интерфейсы для уведомления приложений о наступлении некоторого события. Последние часто используются при написании обработчиков, предназначенных для синхронизации потоков. Примеры подобных интерфейсов можно найти в [65].

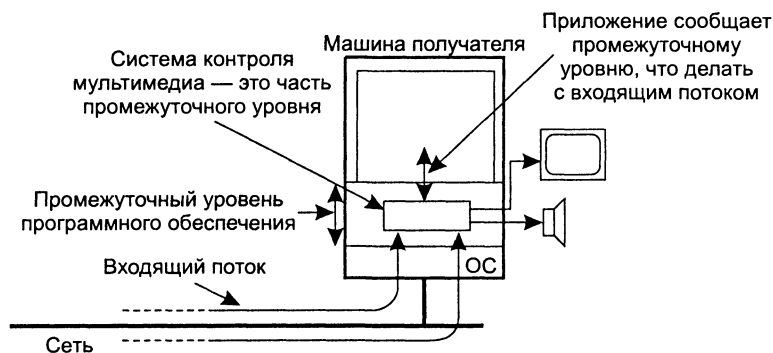


Рис. 2.34. Принцип синхронизации, поддерживаемой высокоуровневыми интерфейсами

Распределение механизмов синхронизации было вторым вопросом, который мы собирались обсудить. Прежде всего получатель комплексного потока данных, состоящего из требующих синхронизации вложенных потоков, должен точно знать, что ему делать. Другими словами, он должен иметь исчерпывающую локальную *спецификацию синхронизации*. Обычно эта информация предоставляется в неявном виде, путем мультиплексирования нескольких потоков данных в один, содержащий все элементы данных, в том числе и предназначенные для синхронизации.

Подобный подход к синхронизации используется в потоках MPEG. Стандарты *MPEG (Motion Picture Experts Group)* содержат набор алгоритмов сжатия видео- и аудиоданных. Существует несколько стандартов MPEG. MPEG-2, например, был изначально предназначен для сжатия видеозображения с качеством типичного телесигнала на скоростях от 4 до 6 Мбайт/с. Согласно этому стандарту в единый поток данных может быть собрано неограниченное количество

непрерывных и дискретных потоков. Каждый входящий поток сначала преобразуется в поток пакетов, содержащих отметку времени, базирующуюся на 90-килогерцевых системных часах. Затем эти потоки мультиплексируются в *программный поток данных* (*program stream*), состоящий из пакетов переменной длины, объединенных одинаковым базовым временем. Принимающая сторона демultipлексирует поток и использует метки времени каждого из пакетов в качестве основы механизма межпотоковой синхронизации.

Еще один важный вопрос: где проводить синхронизацию, на передающем конце или на принимающем? Если синхронизацией занимается передатчик, он может объединить потоки данных в один, содержащий различные типы элементов данных. Рассмотрим вновь поток стереозвука, содержащий два вложенных потока, по одному на канал. Мы рассматривали единственную возможность — передачу потоков независимо друг от друга с последующей синхронизацией на приемнике путем попарного объединения выборок. Ясно, что если задержки в каждом из вложенных потоков данных будут разными, синхронизация окажется крайне затруднительной. Значительно лучше объединить вложенные потоки на отправителе. Получившийся в результате поток будет содержать элементы данных, состоящие из пар выборок, по одной на канал. В этом случае приемнику достаточно просто считать элемент данных и разделить его на выборки левого и правого каналов. Задержка для обоих каналов будет абсолютно одинаковой.

2.6. Итоги

Наличие мощных и гибких механизмов взаимодействия между процессами является важным для всякой распределенной системы. В традиционных сетевых приложениях связь часто базируется на низкоуровневых примитивах передачи сообщений, предоставляемых транспортным уровнем. Важной особенностью систем промежуточного уровня является предоставляемая ими высокая степень абстракции, благодаря которой описание взаимодействия между процессами на промежуточном уровне значительно проще, чем если бы мы ограничились только интерфейсами транспортного уровня.

Одной из наиболее широко используемых абстракций является удаленный вызов процедур (*Remote Procedure Call*, *RPC*). Сущность *RPC* в том, что любая служба реализуется посредством вызова процедуры, тело которой выполняется на сервере. Клиент предоставляет только сигнатуру процедуры, то есть имя процедуры и ее параметры. Когда клиент вызывает процедуру, клиентская реализация, называемая заглушкой, упаковывает значения параметров в сообщение и пересылает его на сервер. Последний вызывает собственно процедуру и возвращает результат, снова в виде сообщения. Клиентская заглушка извлекает из этого сообщения значение результата и передает его приложению клиента, инициировавшему вызов.

Механизм *RPC* ориентирован на обеспечение прозрачности доступа. Однако он относительно слабо поддерживает передачу ссылок. В этом смысле удаленные

объекты более прозрачны. Обращение к удаленным методам (Remote Method Invocation, RMI) напоминает RPC, но отражает специфику удаленных объектов. Основная разница между ними состоит в том, что RMI позволяет использовать в качестве параметров ссылки на объекты системы.

RPC и RMI предоставляют механизмы синхронной связи, при которой клиент блокируется до получения ответа от сервера. Несмотря на вариации существующих механизмов, в которых жесткая синхронная модель смягчена, нередко более удобными оказываются универсальные высокоуровневые модели, ориентированные на передачу сообщений.

В моделях передачи сообщений неважно, является ли связь сохранной, так же как неважно и то, является ли она синхронной. Смысл сохранной связи состоит в том, что посылаемое сообщение хранится в коммуникационной системе до тех пор, пока не будет доставлено по назначению. Другими словами, ни отправитель, ни получатель при передаче сообщения не обязаны быть активными. В случае нерезидентной связи механизмы хранения не предусмотрены, а значит, получатель должен быть готов принять сообщение, когда бы оно ни было послано.

При асинхронной связи отправитель может продолжать работу сразу после установки сообщения в очередь на отправку, возможно, еще до того, как оно будет отправлено. При синхронной связи отправитель блокируется как минимум до момента получения сообщения. В других вариантах отправитель может блокироваться до момента доставки сообщения получателю или до получения от него ответа, как это сделано в RPC.

Модели обмена сообщениями промежуточного уровня обычно предоставляют сохранную асинхронную связь и используются там, где применение механизмов RPC и RMI не оправдано. В первую очередь это интеграция наборов баз данных (сильно распределенных) в крупных информационных системах. Другие области их применения включают в себя электронную почту и рабочие потоки.

Абсолютно иную связь предлагают потоки данных, проблема которых состоит в том, что любые два последовательных сообщения взаимосвязаны по времени. В непрерывных потоках данных максимальная задержка доставки своя для каждого сообщения. Кроме того, необходимо, чтобы сообщения обладали некоей минимальной задержкой доставки. Типичными примерами непрерывных потоков данных являются аудио- и видеопотоки. Часто бывает сложно описать, какими должны быть временные взаимосвязи или чего мы ожидаем от базовой подсистемы связи (в терминах качества обслуживания). При реализации мы также сталкиваемся с затруднениями. Усложняющим фактором является роль дрожания (максимальное и минимальное значения параметров). Даже если средняя производительность достижима, серьезные колебания времени доставки могут привести к неприемлемой производительности.

Вопросы и задания

1. Во многих протоколах модели OSI каждый уровень добавляет к сообщению свой заголовок. Несомненно, создавать в начале каждого сообщения единый

заголовок, содержащий всю контрольную информацию, было бы более эффективно, чем поддерживать все эти отдельные заголовки. Почему так не делается?

2. Почему коммуникационные службы транспортного уровня обычно не подходят для построения распределенных приложений?
3. Надежная массовая рассылка требует от отправителя надежной посылки сообщений группе получателей. Может ли подобная служба находиться на промежуточном уровне или должна быть частью нижнего уровня?
4. Рассмотрим процедуру `incr` с двумя целыми параметрами. Эта процедура добавляет к каждому из параметров единицу. Представим себе, что в качестве обоих параметров используется одна и та же переменная — `incr(i, i)`. Если переменная `i` изначально равна 0, какое значение будет у нее при вызове по ссылке? А при использовании механизма копирования/восстановления?
5. В языке C существует конструкция, называемая объединением (`union`), в которой поле записи — в C называемой структурой (`struct`) — может хранить любое значение из нескольких вариантов. Во время исполнения надежного способа определить, что там находится, не существует. Должна ли эта особенность языка C как-то поддерживаться при удаленном вызове процедур? Поясните свой ответ.
6. Один из способов преобразования параметров в системах RPC — требовать от каждой машины передачи параметров в их натуральном виде, с тем чтобы преобразованием занимался получатель сообщения. Базовая система представления параметров кодируется в первом байте. Однако будет ли работать подобная система, если учесть, что нахождение первого байта в первом слове — само по себе проблема?
7. Рассмотрим клиент, посылающий на сервер асинхронный вызов RPC и ожидающий возвращения сервером результата, который также будет послан при помощи асинхронного вызова RPC. Будет ли это аналогично использованию клиентом обычного вызова RPC? Что произойдет, если мы заменим асинхронные вызовы RPC синхронными?
8. Вместо регистрации некоего сервера с помощью демона, как это сделано в DCE, мы могли бы навсегда закрепить его за одной из конечных точек. Конечная точка затем может использоваться в качестве ссылки на объект в адресном пространстве сервера. Каково основное неудобство подобной схемы?
9. Приведите пример реализации ссылки на объект, которая позволила бы клиенту выполнить привязку к нерезидентному удаленному объекту.
10. Java и другие языки поддерживают обработку исключений, которые возбуждаются при возникновении ошибки. Каким образом можно применить исключения в RPC и RMI?
11. Можно ли использовать механизм исключений также и для того, чтобы отличать статические вызовы RPC от динамических?

12. Некоторые реализации систем промежуточного уровня на базе распределенных объектов целиком базируются на динамическом обращении к методам. Статические обращения при компиляции преобразуются в динамические. В чем преимущества подобного подхода?
13. Опишите, как реализуется связь без образования соединения между клиентом и сервером при использовании сокетов.
14. Объясните разницу между примитивами `MPI_bsend` и `MPI_isend` в `MPI`.
15. Предположим, что мы в состоянии использовать только примитивы нерезидентной асинхронной связи, причем исключительно примитивы асинхронного приема. Как в таком случае реализовать примитивы для нерезидентной *синхронной* связи?
16. Теперь предположим, что мы в состоянии использовать только примитивы для нерезидентной синхронной связи. Как при этом реализовать примитивы для нерезидентной *асинхронной* связи?
17. Имеет ли смысл реализовывать сохраняющую асинхронную связь средствами `RPC`?
18. В тексте мы упоминали, что для автоматического запуска процесса, извлекающего сообщения из входной очереди, часто используется демон, который следит за ее состоянием. Опишите альтернативную реализацию, не требующую использования демона.
19. Таблицы маршрутизации в системе `MQSeries` фирмы `IBM` и множестве других систем очередей сообщений настраиваются вручную. Опишите простой способ автоматизации этой работы.
20. Как встроить сохраняющую асинхронную связь в коммуникационную модель на базе удаленных объектов `RMI`?
21. В случае сохраняющей связи получатель обычно имеет свой собственный буфер, в котором в период его пассивности хранятся сообщения. Чтобы создать подобный буфер, нам может понадобиться информация о его размере. Приведите аргументы за и против указания размера.
22. Опишите, почему нерезидентная синхронная связь имеет «врожденные» проблемы с масштабируемостью, и как эти проблемы могут решаться.
23. Приведите пример, когда групповая рассылка может быть полезной для дискретных потоков данных.
24. Как можно было бы гарантировать максимальную сквозную задержку передачи при организации группы компьютеров в кольцо (физическое или логическое)?
25. Как можно было бы гарантировать минимальную сквозную задержку передачи при организации группы компьютеров в кольцо (физическое или логическое)?
26. Представьте, что у вас есть спецификация корзины элементарных пакетов с максимальным размером элемента данных — 1000 байт, скоростью работы

корзины — 10 Мбайт/с, емкостью корзины — 1 Мбайт и максимальной скоростью передачи — 50 Мбайт/с. Как долго она сможет работать на максимальной скорости?

27. В этом упражнении вам следует реализовать простую систему клиент-сервер на базе RPC. Сервер предоставляет одну процедуру `next`, которая в качестве входного параметра получает целое число и возвращает следующее за ним число. Напишите процедуру-заглушку `next` для использования на стороне клиента. Она должна будет пересылать параметры на сервер, используя протокол UDP, и ожидать ответа, причем следует рассчитывать на очень длительное время ожидания. Серверная процедура должна следить за соответствующим портом, принимать запрос, выполнять его и возвращать результат клиенту.

Глава 3

Процессы

3.1. Потоки выполнения

3.2. Клиенты

3.3. Серверы

3.4. Перенос кода

3.5. Программные агенты

3.6. Итоги

В предыдущей главе мы обсуждали взаимодействие в распределенных системах. Взаимодействие происходит между процессами, и в этой главе мы поближе рассмотрим то, какую роль играют в распределенных системах различные типы процессов. Концепция процесса зародилась в операционных системах, где этим понятием обычно обозначают выполняемую программу. С точки зрения операционных систем наиболее важные для обсуждения вопросы — это управление процессами и планирование процессов. Однако при переходе к распределенным системам другие вопросы становятся столь же или еще более важными.

Так, например, для эффективной организации систем клиент-сервер часто бывает удобно использовать многопоточные технологии. Как мы обсудим в первом разделе этой главы, основной вклад потоков выполнения в работу распределенных систем заключается в том, что они позволяют создавать клиенты и серверы так, что взаимодействие и локальная обработка выполняются параллельно, давая выигрыш в производительности.

Как мы говорили в главе 1, модель клиент-сервер крайне важна для распределенных систем. В этой главе мы детально рассмотрим основные типы организации клиентов и серверов. Мы также уделим внимание общим вопросам построения серверов. Кроме того, мы обсудим общедоступные серверы объектов, которые предоставляют базовые средства для построения распределенных объектов.

Серьезной проблемой, особенно в глобальных системах, является перенос процессов между различными машинами. Миграция процессов, или, более точно, миграция кода, может помочь повысить масштабируемость и, кроме того, помогает динамически конфигурировать клиенты и серверы. Что на самом деле под-

разумеается под миграцией кода и как она реализуется, также будет рассмотрено в этой главе.

Последняя наша тема связана с относительно новым явлением — программными агентами. В отличие от асимметричной программной модели клиент-сервер, мультиагентные системы в основном состоят из набора одинаково важных агентов, совместная работа которых приводит к достижению общей цели. Программные агенты — это еще один тип процессов, которые могут существовать в различных формах. В последнем разделе рассказывается, что такое агенты с точки зрения распределенных систем и каковы принципы их совместной работы.

3.1. Потоки выполнения

Хотя процессы являются строительными блоками распределенных систем, практика показывает, что дробления на процессы, предоставляемого операционными системами, на базе которых строятся распределенные системы, недостаточно. Вместо этого оказывается, что наличие более тонкого дробления в форме нескольких *потоков выполнения* (*threads*) на процесс значительно упрощает построение распределенных приложений и позволяет добиться лучшей производительности. В этом разделе мы подробно рассмотрим роль потоков выполнения в распределенных системах и объясним, почему они так важны. Дополнительную информацию о потоках выполнения и о том, как они используются для построения приложений, можно найти в [263, 439].

3.1.1. Знакомство с потоками выполнения

Для понимания роли потоков выполнения в распределенных системах важно понять, что такое процесс и как соотносятся между собой процессы и потоки выполнения. Для выполнения программ операционная система создает несколько виртуальных процессоров, по одному для каждой программы. Чтобы отслеживать эти виртуальные процессоры, операционная система поддерживает *таблицу процессов* (*process table*), содержащую записи для сохранения значений регистров процессора, карт памяти, открытых файлов, учетных записей пользователей, привилегиях и т. п. *Процесс* (*process*) часто определяют как выполняемую программу, то есть программу, которая в настоящее время выполняется на одном из виртуальных процессоров операционной системы. Следует отметить, что операционная система уделяет огромное внимание гарантиям того, чтобы эти независимые процессы по злому умыслу или ненамеренно не нарушили правильную работу других процессов. Другими словами, тот факт, что множество процессов совместно используют один и тот же процессор и другие аппаратные ресурсы, прозрачен. Обычно для того, чтобы подобным образом отделять процессы друг от друга, операционной системе требуется аппаратная поддержка.

Прозрачность параллельной работы обходится довольно дорого. Так, например, каждый раз при создании процесса операционная система должна создавать

абсолютно независимое адресное пространство. Выделение памяти требует инициализации сегмента памяти, например, путем обнуления сегмента данных, копирования соответствующей программы в сегмент кода и помещения в стек временных данных. Переключение процессора между двумя процессами — также довольно дорогостоящая операция. Кроме сохранения контекста процессора (в который входят значения регистров, счетчик программы, указатель на стек и т. п.) операционная система должна также изменить регистры блока управления памятью (Memory Management Unit, MMU) и объявить некорректным содержимое кэша трансляции адресов, например ассоциативного буфера страниц (Translation Lookaside Buffer, TLB). Кроме того, если операционная система поддерживает больше процессов, чем может одновременно поместиться в оперативной памяти, перед действительным переключением с одного процесса на другой может потребоваться *подкачка* (*swapping*) процессов между оперативной памятью и диском.

Поток выполнения очень похож на процесс, в том смысле, что он также может рассматриваться как программа, выполняемая на виртуальном процессоре. Однако в отличие от процесса не следует пытаться достичь высокой степени прозрачности параллельного выполнения потоков, поскольку это приводит к падению производительности. Тем не менее система потоков выполнения обычно обеспечивает лишь тот минимум информации, который позволяет совместно использовать процессор для различных потоков выполнения. В частности, *контекст потока выполнения* (*thread context*) нередко содержит просто контекст процессора и некоторую другую информацию, необходимую для управления потоком выполнения. Так, например, система потоков может отслеживать факт блокировки потока переменной мьютекса и вытекающую из этого невозможность переключения на выполнение такого потока. Информация, не являющаяся абсолютно необходимой для управления многочисленными потоками, обычно игнорируется. По этой причине задача защиты данных в потоках выполнения одного процесса от несанкционированного доступа возлагается на разработчиков приложения.

У подобного подхода имеется два аспекта. Во-первых, высокая производительность многопоточных приложений достижима с гораздо меньшими усилиями, чем в случае их однопоточного аналога. На самом деле многопоточные системы обычно дают выигрыш в производительности. Во-вторых, поскольку потоки выполнения одного процесса не защищаются автоматически друг от друга, разработка многопоточных приложений требует дополнительных интеллектуальных усилий. Обычно хорошо помогает правильное проектирование и ясность мысли. К сожалению, текущая практика показывает, что эти принципы не всегда воспринимаются должным образом.

Потоки выполнения в нераспределенных системах

До того как мы примемся обсуждать роль потоков выполнения в распределенных системах, рассмотрим сначала, как они используются в системах обычных, нераспределенных. Многопоточные процессы способны принести определенные выгоды, повышающие популярность систем с их поддержкой.

Наиболее важное преимущество вытекает из того факта, что процессы с одним потоком выполнения целиком блокируются при любом блокирующем системном вызове. В качестве иллюстрации рассмотрим какое-либо приложение, например электронные таблицы. Представим себе, что пользователь хочет постоянно изменять значения ячеек в интерактивном режиме. Важным свойством электронных таблиц является поддержка функциональных зависимостей между различными ячейками таблицы или различными таблицами. Таким образом, каждый раз при модификации ячейки все ячейки, связанные с ней, автоматически обновляются. После того как пользователь изменит значение одной из ячеек, внесенное им изменение вызовет значительный объем вычислений. Если мы будем работать только с одним потоком выполнения, во время ожидания ввода эти вычисления окажутся невозможными. Точно так же нелегко будет организовать ввод в процессе вычислений. Самым простым решением было бы создать как минимум два потока выполнения — один для поддержания работы с пользователем, другой для обновления таблиц.

Другим преимуществом многопоточности является возможность обеспечения параллелизма выполнения программ в мультипроцессорной системе. В этом случае каждый поток выполняется на своем процессоре, а совместно используемые данные размещаются в общей (разделяемой) памяти. При правильном проектировании подобный параллелизм может быть прозрачным, процесс будет работать так же хорошо и в однопроцессорной системе, разве что немного медленнее. Многопоточная структура в связи с доступностью относительно недорогих мультипроцессорных рабочих станций становится все важнее в плане поддержки параллельного выполнения. На подобных компьютерных системах обычно работают серверы в приложениях клиент-сервер.

Многопоточная структура часто используется при построении больших приложений. Подобные приложения часто разрабатываются в виде наборов совместно работающих программ, каждая из которых выполняется отдельным процессом. Этот подход типичен для среды UNIX. Кооперация между программами реализуется в виде межпроцессного взаимодействия (через механизмы IPC). Для UNIX-систем эти механизмы обычно включают в себя каналы (именованные), очереди сообщений и совместно используемые сегменты памяти [436]. Основной оборотной стороной механизмов IPC является необходимость интенсивного переключения контекстов, продемонстрированная тремя точками на рис. 3.1.

Поскольку IPC требует вмешательства в ядро, процесс обычно вынужден сначала переключиться из пользовательского режима в режим ядра (точка *S1* на рисунке). Это требует изменения карты памяти в блоке MMU, а также сброса буфера TLB. В ядре происходит переключение контекста процесса (точка *S2*), после чего второй процесс может быть активизирован очередным переключением из режима ядра в пользовательский режим (точка *S3*). Последнее переключение вновь потребует изменения карты памяти в блоке MMU и буфера сброса TLB.

Вместо использования процессов приложение может быть построено так, чтобы различные части выполнялись в отдельных потоках. Связь между этими частями обеспечивается только через разделяемую память. Переключение между

потоками выполнения может происходить исключительно в пространстве пользователя, хотя в некоторых реализациях за потоки выполнения отвечает (управляет ими) ядро. Результатом может быть значительное повышение производительности.

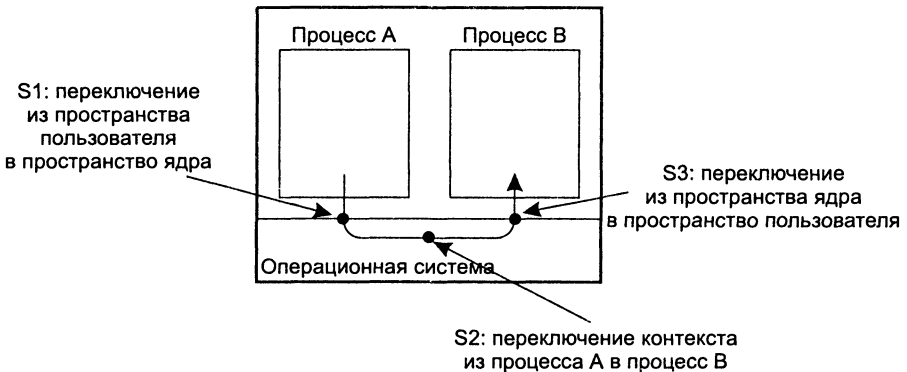


Рис. 3.1. Переключение контекстов в результате вызова IPC

И, наконец, у разработчиков имеется особая причина использовать потоки выполнения: многие приложения просто легче разрабатывать, структурировав их в виде набора взаимосвязанных потоков выполнения. Рассмотрим приложение, требующее выполнения нескольких (более или менее независимых) задач. Например, в случае текстового редактора в отдельные потоки можно выделить обработку ввода пользователя, проверку орфографии и грамматики, оформление внешнего вида документа, создание индекса и т. п.

Реализация потоков выполнения

Потоки выполнения обычно существуют в виде пакетов. Подобные пакеты содержат механизмы для создания и уничтожения потоков, а также для работы с переменными синхронизации, такими как мьютексы и условные переменные. Существует два основных подхода к реализации пакетов для потоков выполнения. Первый из них состоит в создании библиотеки для работы с потоками выполнения, выполняющейся исключительно в режиме пользователя. Второй подход предполагает, что за потоки выполнения отвечает (и управляет ими) ядро.

Библиотека для работы с потоками выполнения на пользовательском уровне имеет множество преимуществ. Во-первых, дешевле обходится создание и уничтожение потоков выполнения. Поскольку все управление потоками реализуется в адресном пространстве пользователя, стоимость создания потока выполнения определяется в первую очередь затратами на память, выделяемую для создания стека под поток. Аналогично и уничтожение потока выполнения в основном состоит в освобождении памяти, задействованной под стек, после того как необходимость в потоке отпадает. Обе операции достаточно дешевы.

Второе преимущество потоков выполнения на пользовательском уровне состоит в том, что переключение контекста требует всего нескольких инструкций.

В основном в сохранении и последующем восстановлении сохраненных значений при переключении с потока на поток нуждаются исключительно значения регистров процессора. Нет необходимости изменять карты памяти, сбрасывать буфер TLB, контролировать загрузку процессора и т. д. Переключение контекста потоков выполнения производится при необходимости в синхронизации двух потоков, например, при обращении к секции совместно используемых данных.

Основным недостатком потоков выполнения на пользовательском уровне является тот факт, что обращение к блокирующему системному вызову приводит к немедленной блокировке как процесса, к которому принадлежит поток выполнения, так и всех остальных потоков выполнения этого процесса. Как мы говорили, потоки выполнения используются, в частности, для структурирования больших приложений в виде набора одновременно исполняемых частей. В этом случае блокировка по вводу-выводу не должна мешать работе исполняемых в то же самое время частей программы. Для подобных приложений реализация потоков выполнения на пользовательском уровне неприемлема.

Эти проблемы обычно можно обойти путем реализации потоков выполнения в ядре операционной системы. К сожалению, плата за это весьма высока: любая операция с потоками выполнения (создание, уничтожение, синхронизация и т. п.), осуществляемая ядром, требует системного вызова. Переключение контекстов потоков выполнения становится столь же дорогостоящим, как и переключение контекстов процессов. В результате сходят на нет основные преимущества от замены процессов потоками выполнения.

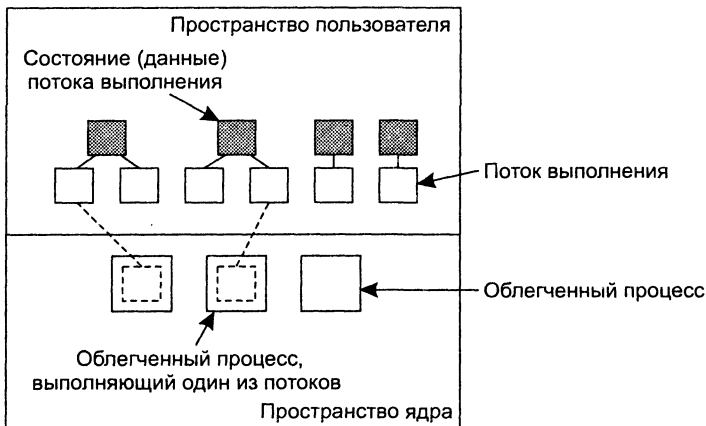


Рис. 3.2. Сочетание облегченных процессов уровня ядра и потоков выполнения уровня пользователя

Решение состоит в использовании «гибрида» из реализованных на уровне пользователя и на уровне ядра потоков выполнения (рис. 3.2). Такой гибрид носит название *облегченных процессов* (*LightWeight Processes, LWP*). Процессы LWP работают в контексте единого полновесного процесса. На один процесс может приходиться несколько LWP. Кроме LWP система предоставляет также пакет

потоков выполнения пользовательского уровня, предоставляющего приложениям операции по созданию и уничтожению потоков выполнения. Кроме того, этот пакет предоставляет средства синхронизации потоков выполнения, такие как мьютексы и условные переменные (см. также раздел 1.4). Важным моментом здесь является то, что пакет для работы с потоками выполнения реализован целиком в пространстве пользователя. Другими словами, все операции с потоками выполнения производятся без участия ядра.

Как показано на рисунке, пакет потоков выполнения может совместно использоваться несколькими облегченными процессами. Многопоточное приложение реализуется путем создания потоков выполнения с последующим назначением каждого из них своему облегченному процессу. Назначение потока выполнения облегченному процессу обычно производится неявно и скрыто от программиста.

Сочетание потоков выполнения пользовательского уровня и облегченных процессов работает так, как показано на рисунке. Подключение следующего потока выполняется специальной планирующей процедурой, входящей в пакет потоков. При создании (путем системного вызова) облегченного процесса он получает собственный стек и указание выполнить планирующую процедуру для поиска потока, который он должен исполнять. Если создается несколько облегченных процессов, планирующую процедуру выполняет каждый из них. Таблица потоков выполнения, предназначенная для отслеживания их текущего набора, используется облегченными процессами совместно. Защита этой таблицы для предотвращения одновременного доступа к ней реализуется при помощи мьютексов, создаваемых в пространстве пользователя. Иначе говоря, синхронизация между облегченными процессами не требует поддержки со стороны ядра.

Когда облегченный процесс находит готовый к выполнению поток, он переключает свой контекст на этот поток. В это время другие облегченные процессы могут искать себе другие готовые к выполнению потоки. Если поток выполнения требуется заблокировать по значению мьютекса или условной переменной, он выполняет необходимое администрирование и, в конце концов, вызывает планирующую процедуру. Если будет обнаружен другой готовый к выполнению поток, контекст может переключиться на него. Прелесть всего этого состоит в том, что облегченный процесс, выполняющий поток, не уведомляется о переключении контекста, оно происходит исключительно в пространстве пользователя и воспринимается облегченным процессом как обычный код программы.

Рассмотрим теперь, что происходит, когда поток делает блокирующий системный вызов. В этом случае выполнение переходит от пользовательского режима к режиму ядра, продолжая при этом оставаться в контексте текущего облегченного процесса. В тот момент, когда текущий облегченный процесс можно будет прервать, операционная система может принять решение переключиться на другой облегченный процесс, в результате чего контекст переключится обратно в пользовательский режим. Выбранный облегченный процесс просто продолжит свою работу с того места, где он ранее был прерван.

Использование облегченных процессов в сочетании с пакетами потоков выполнения пользовательского режима дает определенные преимущества. Во-первых, создание, уничтожение и синхронизация потоков выполнения не требует задействовать ядро и поэтому относительно недорого. Во вторых, предоставление процессу нескольких облегченных процессов приводит к тому, что блокирующий системный вызов не обеспечивает полной остановки процесса. В-третьих, приложению нет нужды знать об облегченных процессах. Все, что оно видит, — это потоки выполнения пользовательского уровня. В-четвертых, облегченные процессы легко могут быть приспособлены для работы в мультипроцессорных средах путем исполнения разных процессов на различных процессорах. Подобная параллельная обработка может происходить абсолютно незаметно для приложения. Единственным минусом облегченных процессов в сочетании с потоками выполнения пользовательского уровня можно считать необходимость создания и уничтожения самих облегченных процессов. Затраты на эти операции не отличаются от обычных затрат на работу с потоками выполнения уровня ядра. Однако в создании и уничтожении облегченных процессов мы нуждаемся лишь от случая к случаю, и нередко эти процессы полностью находятся под управлением операционной системы.

Альтернативным, но схожим с облегченными процессами подходом [14] являются *активизации планировщика* (*scheduler activations*). Основная разница между активизацией планировщика и облегченными процессами состоит в том, что когда ядро блокируется системным вызовом, оно передает вызов в пакет потоков, который осуществляет вызов процедуры планировщика. Вызванный планировщик переключает выполнение на следующий поток. Те же самые действия повторяются и после того, как процесс будет разблокирован. Преимущество подобного подхода состоит в том, что все управление облегченными процессами оказывается сосредоточенным в ядре. Однако передача вызовов на уровень пользователя выглядит менее элегантно, поскольку нарушает структуру многоуровневой системы, для которой допустима передача вызовов только на соседний с текущим нижний уровень.

3.1.2. Потоки выполнения в распределенных системах

Важным свойством потоков выполнения является удобная реализация блокирующих системных вызовов, которые происходят без блокирования всего процесса на время выполнения потока. Это свойство потоков выполнения особенно привлекательно в распределенных системах, поскольку оно значительно упрощает представление взаимодействия как одновременное поддержание значительного количества логических соединений. Мы проиллюстрируем это утверждение, рассматривая многопоточных клиентов и серверов.

Многопоточные клиенты

Чтобы добиться высокой степени прозрачности распределения, распределенные системы, работающие в глобальных сетях, могут нуждаться в маскировке боль-

ших задержек сообщений, курсирующих между процессами. Цикл задержки в глобальных сетях легко может достигать порядка сотен миллисекунд, а временами и секунд.

Традиционный способ скрыть задержки связи — инициировав взаимодействие, немедленно перейти к другой работе. Типичным примером применения этой методики являются web-браузеры. Во многих случаях web-документ, содержащийся в файле формата HTML, содержит, кроме текста, набор изображений, значков и т. п. Для получения элементов web-документа браузер открывает соединение TCP/IP, читает поступающие данные и преобразует их в компоненты визуального представления. Установка соединения, так же как и чтение данных, представляет собой блокирующие операции. При работе с медленными коммуникациями мы ощущаем неудобства, связанные с тем, что время, требующееся для завершения каждой из операций, может быть довольно длительным.

Web-браузер обычно сначала получает страницу HTML-кода, а затем показывает ее. Для того чтобы по возможности скрыть задержки связи, некоторые браузеры начинают показывать данные по мере их получения. Когда текст с механизмами прокрутки становится доступным пользователю, браузер продолжает получение остальных файлов, необходимых для правильного отображения страницы, таких как картинки. По мере поступления они отображаются на экране. Таким образом, для того чтобы увидеть страницу, пользователь не должен дожидаться получения всех ее компонентов.

В результате мы видим, что web-браузер выполняет несколько задач одновременно. Понятно, что разработка браузера в виде многопоточного клиента весьма упрощает это занятие. Как только мы получаем основной файл HTML, активизируются отдельные потоки выполнения, отвечающие за загрузку других частей страницы. Каждый из потоков выполнения создает отдельное соединение с сервером и получает от него данные. Установление соединения и чтение данных с сервера может быть запрограммировано с использованием стандартных (блокирующих) системных вызовов. Это предполагает, что блокирующие вызовы не в состоянии приостановить основной процесс. Как показано в [438], код каждого из потоков выполнения одинаков и в основном несложен. В результате пользователь, хотя и замечает задержку в показе картинок и прочих украшениях, вполне может просматривать документ.

Использование многопоточных web-браузеров, которые в состоянии открывать несколько соединений, дает также и другой выигрыш. В предыдущем примере мы создавали несколько соединений с одним и тем же сервером. Если этот сервер сильно загружен или просто медленный, нам не удастся добиться значительного повышения производительности по сравнению с последовательным (одного за другим) получением файлов.

Однако во многих случаях web-серверы могут быть реплицированы на несколько машин, при этом каждый из серверов будет содержать одинаковый набор web-документов. Реплицированные серверы находятся в одном и том же месте и имеют одно и то же имя. При поступлении запроса на web-страницу этот запрос передается одному из серверов, обычно с использованием алгоритма циклического обслуживания или другого алгоритма выравнивания нагрузки [227].

При наличии многопоточного клиента соединения могут быть установлены с различными репликами, что приведет к параллельной передаче данных, а это будет эффективно способствовать тому, что web-документ будет показан полностью значительно быстрее, чем в случае нереплицированного сервера. Этот подход срабатывает, только если клиент в состоянии обработать истинно параллельные потоки входящих данных. В этом случае потоки выполнения идеально подходят для обработки потоков данных.

Многопоточные серверы

Хотя многопоточные клиенты, как мы видели, обладают весомыми достоинствами, основные достоинства многопоточности в распределенных системах приходятся на сторону сервера. Практика показывает, что многопоточность не только существенно упрощает код сервера, но и делает гораздо проще разработку тех серверов, в которых для достижения высокой производительности требуется параллельное выполнение нескольких приложений. В число таковых входят и мультипроцессорные системы. Даже сейчас, когда мультипроцессорные компьютеры активно выпускаются в виде рабочих станций общего назначения, использование для параллельной обработки многопоточности не потеряло своей актуальности.

Чтобы ощутить достоинства потоков выполнения для написания кода серверов, рассмотрим организацию файлового сервера, который периодически оказывается блокированным ожиданием диска. Файловый сервер обычно ожидает входящего запроса на операции с файлами, после чего обрабатывает полученный запрос и возвращает ответ. Одна из возможных и особо популярных организаций показана на рис. 3.3. Здесь один из потоков выполнения, *диспетчер*, считывает приходящие запросы на файловые операции. Запросы посылаются клиентами с указанием конечной точки данного сервера. После проверки запроса сервер выбирает (то есть блокирует) находящийся в состоянии ожидания *рабочий* поток выполнения и передает запрос ему.

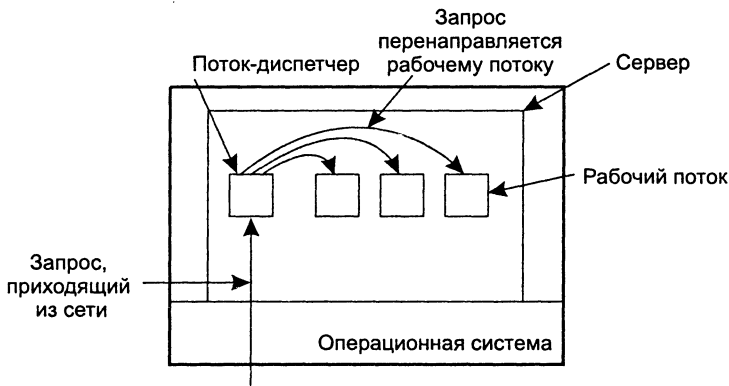


Рис. 3.3. Многопоточный сервер, организованный по схеме диспетчер—рабочий

Рабочий поток осуществляет блокирующее чтение из локальной файловой системы, что приводит к приостановке потока выполнения до считывания данных с диска. На то время, пока поток выполнения приостановлен, управление может быть передано другому потоку выполнения. Например, дополнительную работу может выполнить поток-диспетчер. Либо он может выбрать другой готовый к запуску рабочий поток.

Теперь обсудим, как файловый сервер мог бы вести запись в отсутствие потоков выполнения. Одна из возможностей — действовать так, как будто имеется единственный поток выполнения. Основной цикл файлового сервера получает запрос, проверяет его и передает на выполнение раньше, чем получит следующий. Пока сервер ожидает окончания дисковых операций, он не обрабатывает другие запросы. Таким образом, запросы других клиентов не обрабатываются. Кроме того, если файловый сервер работает на выделенной машине (а так обычно и бывает), процессор во время дисковых операций просто ничем не занят. В результате обрабатывается значительно меньшее количество запросов в секунду, чем могло бы. Таким образом, потоки выполнения обеспечивают значительное повышение производительности, и это несмотря на то, что они запускаются на выполнение поочередно.

Таким образом, мы наблюдаем две возможные архитектуры — многопоточный файловый сервер и однопоточный файловый сервер. Возможно также, что потоки выполнения отсутствуют вообще, если разработчики системы сочтут, что падение производительности из-за использования однопоточной архитектуры делает ее применение невозможным. Третья возможность заключается в использовании сервера в качестве большого конечного автомата. По приходе запроса его проверяет единственный поток выполнения. Если он может быть удовлетворен путем обращения к кэшу, отлично, в противном случае происходит обращение к дисковой системе.

Однако вместо блокировки поток выполнения записывает состояние текущего запроса в таблицу и переходит к ожиданию и получению нового сообщения. Новое сообщение может быть как запросом на совершение новой операции, так и ответом на предыдущий запрос от дисковой подсистемы. Если это новый запрос, мы начинаем новую работу. Если это ответ от диска, из таблицы извлекается соответствующая информация, ответ формируется и передается клиенту. Согласно этой схеме, сервер должен быть в состоянии осуществлять неблокирующие вызовы `send` и `receive`.

В подобной архитектуре модель «последовательных процессов», которую мы наблюдали в первых двух случаях, отсутствует. Состояние вычислений для каждого принимаемого и отправляемого сообщения должно полностью сохраняться и извлекаться из таблицы. В результате мы сложным образом моделируем потоки выполнения и их стеки. Процесс, протекающий в конечном автомате, состоит в восприятии событий и зависящей от их типа реакции на эти события.

Теперь становится ясно, зачем нужны потоки выполнения. Они делают возможным одновременное сохранение идеи последовательных процессов, осуществляющих блокирующие системные вызовы (как при вызовах `RPC` для работы с диском), и параллельной работы. Блокирующие системные вызовы упрощают

программирование, а параллельность повышает производительность. Однопоточные серверы сохраняют простоту блокирующих системных вызовов, но проигрывают в производительности. Подход, используемый в конечных автоматах, позволяет благодаря параллелизму добиться высокой производительности, но из-за неблокирующих вызовов тяжело программируется. Эти подходы суммированы в табл. 3.1.

Таблица 3.1. Три способа построения сервера

Модель	Характеристики
Потоки выполнения	Параллельность, блокирующие системные вызовы
Однопоточный процесс	Отсутствие параллелизма, блокирующие системные вызовы
Конечный автомат	Параллелизм, неблокирующие системные вызовы

3.2. Клиенты

В предыдущих главах мы обсуждали модель клиент-сервер, роли клиента и сервера и способы их взаимодействия. Теперь мы рассматриваем анатомию клиентов и серверов. Начнем с рассмотрения клиентов. Серверы будут рассматриваться в следующем разделе.

3.2.1. Пользовательские интерфейсы

Основная задача большинства клиентов — служить передаточным звеном между пользователем и удаленным сервером. Поддержка пользовательского интерфейса — основная функция большинства клиентов. Во многих случаях интерфейс между пользователем и удаленным сервером относительно прост и встроен в аппаратуру клиента. Так, например, сотовые телефоны в придачу к традиционному набору кнопок для набора номера имеют небольшой дисплей. Сколько-нибудь сложные вещи, такие как работа с электронной почтой, могут потребовать комплектации настоящей клавиатурой, электронным планшетом или устройством распознавания речи.

Важный класс интерфейсов составляют графические пользовательские интерфейсы. Далее мы сперва рассмотрим в качестве примера традиционного пользовательского интерфейса систему X-Windows, а затем обсудим современные интерфейсы, которые обеспечивают прямое взаимодействие между приложениями.

Система X-Windows

Система X-Windows, обычно называемая просто X, используется для управления растровыми терминалами, в состав которых входят монитор, клавиатура и координатное устройство, такое как мышь. На самом деле X можно рассматривать как часть операционной системы, отвечающую за терминалы. Сердце системы — это

то, что мы называем *Х-ядро* (*X kernel*). Оно содержит все необходимые для управления терминалами драйверы устройств и потому сильно зависит от аппаратной конфигурации.

Х-ядро предоставляет относительно низкоуровневый интерфейс для управления экраном и перехвата сообщений, поступающих от клавиатуры и мыши. Этот интерфейс доступен для приложений в виде библиотеки *Xlib*. Соответствующая организация показана на рис. 3.4. (В терминологии X-Windows Х-ядро называется Х-сервером, в то время как программа, использующая его функции, называется Х-клиентом. Чтобы избежать путаницы со стандартной терминологией клиент-сервер, мы воздержимся от использования терминов Х-сервер и Х-клиент.)

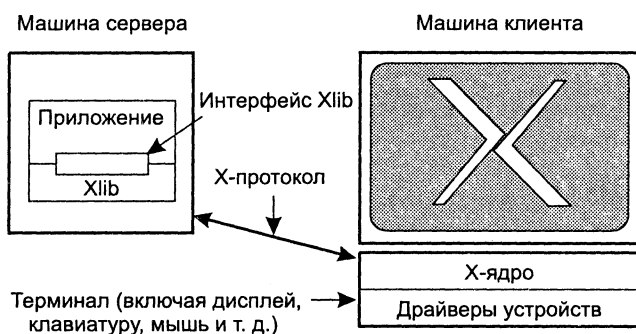


Рис. 3.4. Базовая организация системы X-Windows

Система X различает два типа программ — обычные приложения и менеджеры окон. Нормальные приложения обычно запрашивают (через *Xlib*) создание на экране окна, которое затем используется для ввода или другой работы. Кроме того, X гарантирует, что если окно приложения активно и указатель мыши находится внутри окна, значит, все сообщения от мыши и клавиатуры будут переданы в приложение.

Менеджер окон (*window manager*) — это приложение, которому дано особое право на работу со всем экраном. Обычное приложение обязано подчиняться ограничениям на работу с экраном, которые реализованы в менеджере окон. Так, менеджер окон может указать, что окна не должны перекрываться или должны всегда выводиться одним и тем же цветом. Соответственно, менеджер окон определяет «вид и цвет» самой оконной системы.

У X-системы имеется интересная особенность, заключающаяся в том, что Х-ядро и приложения, использующие X-систему, не обязательно должны располагаться на одной и той же машине. Так, X-система поддерживает X-протокол — ориентированный на использование в сети коммуникационный протокол, посредством которого экземпляры *Xlib* могут обмениваться данными и событиями с Х-ядром. Это приводит к существованию множества различных вариантов архитектуры клиент-сервер с широкой степенью варьирования уровня интеллектуальности клиента. В своей примитивнейшей форме клиент представляет собой

«голое» X-ядро, в то время как весь код приложения находится на удаленной машине. Сконфигурированный таким образом терминал часто называют просто X-терминалом (*X terminal*). В наиболее интеллектуальном виде на клиентской машине могут быть запущены многочисленные приложения, включая менеджер окон, и необходимость в сетевой связи отпадает.

Важно понимать, что системы пользовательского интерфейса, такие как X-система, только предоставляют приложениям пользовательский интерфейс. Единственная информация, которую подобные приложения могут получать из системы, — это события, определяющие основные действия пользователя с подключенными к терминалу устройствами. Примерами подобных событий могут быть нажатия клавиш, перемещение мыши и операции с кнопками мыши и т. п.

Составные документы

Как мы уже говорили в главе 1, современные пользовательские интерфейсы могут делать значительно больше, чем X-Windows. Так, они позволяют приложениям совместно использовать одно и то же графическое окно и в процессе работы пользователя обмениваться данными через это окно. Дополнительные действия, которые могут выполнять пользователи, включают в себя и то, что обычно называют соответственно операциями *перетаскивания* (*drag-and-drop*) и *редактирования по месту* (*in-place editing*).

Типичным примером функциональности перетаскивания является перемещение представляющего файл А значка в мусорную корзину, приводящее к его удалению. В данном случае пользовательский интерфейс должен быть в состоянии делать более сложную работу, чем при выравнивании значков на экране: когда значок А переносится на изображение мусорной корзины, он должен передать имя файла А приложению, ассоциированному с мусорной корзиной. Немного подумав, вы легко найдете и другие примеры.

Редактирование по месту может быть с успехом проиллюстрировано на примере документа, содержащего текст и графику. Представьте себе, что документ просматривается при помощи стандартного текстового редактора. Как только пользователь перемещает мышью на изображение, пользовательский интерфейс передает эту информацию приложению для работы с графикой, которое позволяет пользователю внести в изображение изменения. Так, пользователь может повернуть изображение, а это может повлиять на размещение рисунка в документе. Пользовательский интерфейс должен в этом случае определить новую высоту и ширину рисунка и передать эту информацию текстовому редактору. Последний, в свою очередь, сможет автоматически обновить вид страницы.

Ключевая идея, стоящая за пользовательскими интерфейсами, — это понятие о *составном документе* (*compound document*). Составной документ можно определить как набор документов, возможно различных типов (например, текст, рисунки, электронные таблицы и т. д.), которые интегрируются в одно целое на уровне пользовательского интерфейса. Пользовательский интерфейс, в котором представляется этот документ, скрывает тот факт, что с разными частями документа работают различные приложения. С точки зрения пользователя, все части соединены в одно целое. Если изменение одной из частей влечет за собой изме-

нения других, пользовательский интерфейс может производить необходимые замеры, например, для уведомления соответствующих приложений.

Аналогично ситуации, описанной для системы X-Windows, приложение, ассоциированное с составным документом, не обязательно выполняется на машине клиента. Однако должно быть понятно, что пользовательский интерфейс, поддерживающий составные документы, производит значительно больший объем работы, чем интерфейс, который этого не делает.

3.2.2. Клиентское программное обеспечение, обеспечивающее прозрачность распределения

Как мы уже говорили в разделе 1.5, в программное обеспечение клиента входит не только пользовательский интерфейс. Во многих случаях на стороне клиента выполняется также часть уровней обработки и данных приложения клиент-сервер. На основе встроенного клиентского программного обеспечения функционирует целый класс специализированных устройств, таких как автоответчики (АТМ), счетчики купюр, считыватели штрих-кодов, телеприставки и др. В подобных случаях пользовательский интерфейс является относительно небольшой частью клиентского программного обеспечения по сравнению со средствами локальной обработки и коммуникаций.

Кроме пользовательского интерфейса и другого связанного с приложением программного обеспечения, клиентское программное обеспечение содержит компоненты, обеспечивающие прозрачность распределения. В идеале клиент не должен быть осведомлен о своем взаимодействии с удаленными процессами. Для серверов же, напротив, информация о распределенной работе обычно не скрывается в целях повышения производительности и корректной работы. Так, в главе 6 мы увидим, что реплицируемые серверы должны время от времени связываться друг с другом, чтобы обеспечить определенный порядок выполнения операций в каждой из реплик.

Прозрачность доступа обычно обеспечивается путем генерации (в виде заглушки клиента) из определения интерфейса того, что должен делать сервер. Заглушка предоставляет такой же интерфейс, как и сервер, скрывая при этом разницу архитектур и реальное взаимодействие.

Существуют различные способы реализации прозрачности размещения, переноса и перемещения. Как станет ясно из материала следующей главы, важно использовать определенную систему именования. Во многих случаях важна кооперация с программным обеспечением клиентской стороны. Например, когда клиент уже привязан к серверу, он может напрямую извещаться об изменении местонахождения сервера. В этом случае промежуточный уровень клиента может скрывать истинное местоположение сервера от пользователя и при необходимости незаметно повторить привязку к этому серверу. Самое большее, что может заметить приложение клиента, — это временное падение производительности.

Аналогичным образом большинство распределенных систем реализуют прозрачность репликации на стороне клиента. Представим себе, например, распре-

деленную систему с удаленными объектами. Репликацию удаленного объекта можно осуществить путем рассылки всем репликам запроса, как показано на рис. 3.5. Заместитель клиента в состоянии прозрачно собрать все ответы и передать приложению клиента одно возвращаемое значение.

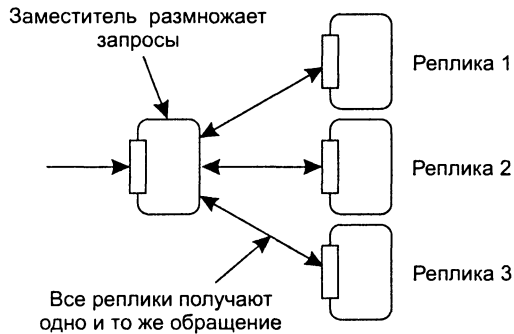


Рис. 3.5. Возможный подход к прозрачной репликации удаленных объектов с использованием клиентского программного обеспечения

И, наконец, обсудим прозрачность к сбоям. Маскирование сбоев во взаимодействии с серверами обычно выполняется при помощи клиентского программного обеспечения промежуточного уровня. Так, клиентское программное обеспечение промежуточного уровня можно сконфигурировать таким образом, чтобы оно многократно пыталось связаться с сервером или выбирало после нескольких попыток другой сервер. Возможна также ситуация, когда программное обеспечение клиента, в случае если web-браузер не в состоянии связаться с сервером, возвращало бы данные, сохраненные в кэше во время предыдущего сеанса связи.

Прозрачность параллельного исполнения может обеспечиваться специальными промежуточными серверами, так называемыми мониторами транзакций, и требует меньшей поддержки со стороны клиентского программного обеспечения. Прозрачность сохранности также реализуется серверами.

3.3. Серверы

Давайте теперь подробнее рассмотрим организацию серверов. На следующих страницах мы сосредоточимся на некоторых общих вопросах строения серверов, а затем поговорим о серверах объектов. Серверы объектов имеют большое значение — они формируют строительные блоки для реализации распределенных объектов.

3.3.1. Общие вопросы разработки

Сервер — это процесс, реализующий некоторую службу, требующуюся группе клиентов. Все серверы работают схожим образом: они ожидают входящего сооб-

щения, посылаемого клиентом, затем проверяют это сообщение на правильность, после чего ожидают следующего сообщения.

Серверы могут быть организованы различными способами. В случае *итеративного сервера* (*iterative server*) сервер сам обрабатывает запрос и при необходимости возвращает клиенту ответное сообщение. *Параллельный сервер* (*concurrent server*) не обрабатывает сообщение сам, а передает его в отдельный поток выполнения или другой процесс, после чего сразу же переходит в состояние ожидания следующего входящего сообщения. Примером параллельного сервера является многопоточный сервер. В другой реализации параллельного сервера на каждый пришедший запрос может выделяться новый процесс. Подобный подход применяется во многих UNIX-системах. Поток или процесс, обрабатывающий запрос, отвечает за отправку ответа клиенту, приславшему запрос.

Другой момент — каким образом клиент обращается к серверу. Клиент всегда посылает запросы в *конечную точку* (*endpoint*), также именуемую *портом* (*port*), той машины, на которой работает сервер. Каждый сервер просматривает указанную ему конечную точку. Как клиент узнает конечную точку конкретной службы? Одно из решений — глобальное назначение конечных точек широко распространенным службам. Так, серверы, обслуживающие запросы к FTP в Интернете, всегда работают с портом 21. HTTP-серверам World Wide Web всегда назначается TCP-порт 80. Эти конечные точки назначены организацией назначения номеров Интернета (Internet Assigned Numbers Authority, IANA) и документированы в [379]. Имея назначенные конечные точки, клиенту достаточно знать сетевой адрес той машины, на которой запущена служба. Как мы увидим в следующей главе, для этой цели используются службы именования.

Существует множество служб, которые не нуждаются в предварительном назначении конечной точки. Так, например, служба времени может использовать конечную точку, динамически выдаваемую ей локальной операционной системой. В этом случае клиент должен сначала определить конечную точку. Одно из решений, которое обсуждалось нами для среды DCE, — это создание специального демона, запускаемого на каждой машине, на которой работают серверы. Демон отслеживает текущую конечную точку каждого из серверов, реализуемых на данной машине. Демон сам просматривает общедоступные конечные точки. При первом контакте с демоном клиент запрашивает конечную точку, после чего связывается с нужным ему сервером, как показано на рис. 3.6, а.

Обычно конечная точка ассоциируется с конкретной службой. Однако на самом деле реализация каждой службы отдельным сервером была бы непозволительно щедрым расходом ресурсов. Так, например, в типичных UNIX-системах нередко имеется множество одновременно работающих серверов, при этом большая их часть пассивно ожидает запросов от клиента. Вместо ведения такого множества пассивных процессов нередко эффективнее иметь один *суперсервер* (*superserver*), просматривающий все конечные точки, относящиеся к конкретным службам, как показано на рис. 3.6, б. Подобный подход реализует, например, демон *inetd* в UNIX. Этот демон просматривает множество стандартных портов служб Интернета. В случае прихода запроса он разветвляет процесс и передает

запрос на дальнейшую обработку. После окончания обработки дочерний процесс завершается.

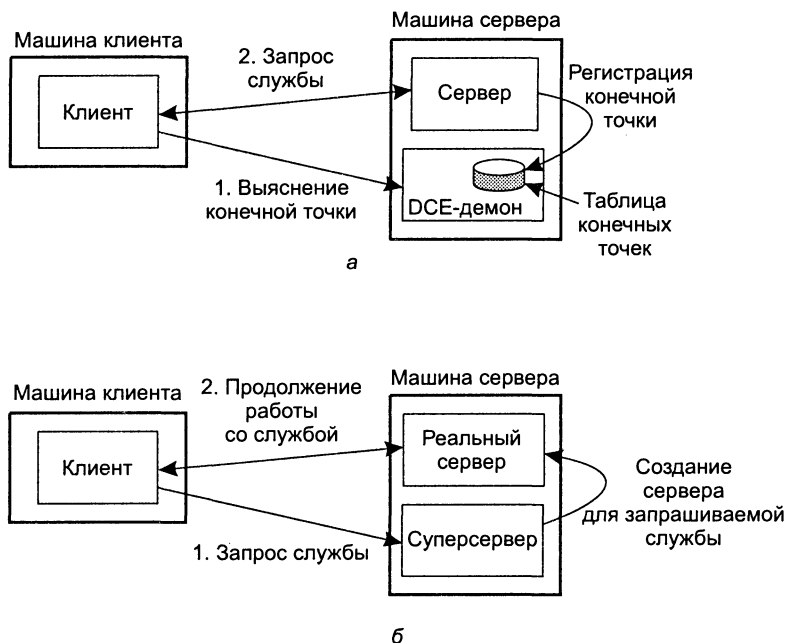


Рис. 3.6. Привязка клиента к серверу с использованием демона в среде DCE (а).
Привязка клиента к серверу с использованием суперсервера в UNIX (б)

Еще один момент, который следует принимать во внимание при создании сервера, — когда и как сервер может быть прерван. Для примера рассмотрим пользователя, который решил загрузить на FTP-сервер большой файл. Затем, начав делать это, он обнаружил, что выбрал не тот файл и хотел бы отменить дальнейшую пересылку данных. Существует несколько способов сделать это. Один из подходов, кстати единственный, надежно работающий в современном Интернете (а иногда и единственно возможный), — пользователь немедленно закрывает клиентское приложение (что автоматически вызывает разрыв соединения с сервером), тут же перезапускает его и продолжает работу. Сервер, естественно, разрывает старое соединение, полагая, что клиент прервал работу.

Более правильный способ вызвать прерывание связи — разрабатывать клиент и сервер так, чтобы они могли пересылать друг другу сигнал *конца связи* (*out-of-band*), который должен обрабатываться сервером раньше всех прочих передаваемых клиентом данных. Одно из решений — потребовать от сервера просматривать отдельную управляющую конечную точку, на которую клиент будет отправлять сигнал конца связи и одновременно (с более низким приоритетом) — конечную точку, через которую передаются нормальные данные. Другой вариант — пересылать сигнал конца связи через то же соединение, через которое клиент пересылал свой запрос. В TCP, например, можно посылать срочные данные.

Когда срочные данные достигают сервера, он прерывает свою работу (в UNIX-системах — по сигналу), после чего может просмотреть эти данные и обработать их.

И последний из важных моментов, о которых надо помнить при разработке, — должен или нет сервер хранить информацию о состоянии клиентов. *Сервер без фиксации состояния (stateless server)* не сохраняет информацию о состоянии своих клиентов и может менять свое собственное состояние, не информируя об этом своих клиентов [55]. Web-сервер, например, это сервер без фиксации состояния. Он просто отвечает на входящие HTTP-запросы, которые могут требовать загрузки файла как на сервер, так и (гораздо чаще) с сервера. После выполнения запроса web-сервер забывает о клиенте. Кроме того, набор файлов, которыми управляет web-сервер (возможно, в комбинации с файловым сервером), может быть изменен без уведомления клиентов об этом действии.

В противоположность этому, *сервер с фиксацией состояния (stateful server)* хранит и обрабатывает информацию о своих клиентах. Типичным примером такого сервера является файловый сервер, позволяющий клиенту создавать локальные копии файлов, скажем, для повышения производительности операций обновления. Подобный сервер поддерживает таблицу, содержащую записи пар (*клиент, файл*). Такая таблица позволяет серверу отслеживать, какой клиент с каким файлом работает и, таким образом, всегда определять самую «свежую» версию файла. Подобный подход повышает производительность операций чтения-записи, осуществляемых на клиенте. Рост производительности по сравнению с серверами без фиксации состояния часто является основным преимуществом и основной причиной разработки серверов с фиксацией состояния. Однако данный пример также иллюстрирует основной недостаток таких серверов. В случае сбоя сервера он вынужден восстанавливать свою таблицу с записями пар (*клиент, файл*), в противном случае не будет никакой гарантии в том, что работа происходит с последней обновленной версией файла. Как правило, серверы с фиксацией состояния нуждаются в восстановлении своего состояния в том виде, в котором оно было до сбоя. Как мы увидим в главе 7, необходимость обеспечить восстановление после сбоя сильно усложняет программу. В случае архитектуры без фиксации состояния вообще нет необходимости принимать какие-то специальные меры по восстановлению серверов после сбоя. Они просто перезапускаются и работают, ожидая запросов клиента.

При разработке сервера выбор между архитектурой с фиксацией и без фиксации состояния не отражается на службе, которую этот сервер должен предоставлять. Так, например, поскольку до выполнения чтения или записи файлы в любом случае должны открываться, сервер без фиксации состояния должен тем или иным способом воспроизвести открытие файла. Стандартное решение, которое мы более детально обсудим в главе 10, состоит в том, чтобы сервер, обрабатывая запрос на запись в файл или чтение из файла, открывал нужный файл, выполнял операцию чтения или записи и немедленно его закрывал.

В других случаях сервер может пожелать хранить записи о поведении клиентов, чтобы более эффективно обрабатывать их запросы. Так, например, web-серверы иногда предоставляют клиентам возможность немедленно отправиться на

любимые страницы. Это можно сделать только в том случае, если сервер будет хранить информацию о клиенте. Традиционное решение состоит в том, чтобы позволить клиенту отсылать дополнительную информацию о его предыдущих сеансах работы с сервером. Эта информация часто прозрачно сохраняется браузером клиента в виде файлов *cookie* — маленьких фрагментов данных, содержащих информацию о клиенте, важную для сервера. Файлы *cookie* никогда не исполняются браузером, они просто хранятся.

При первом обращении клиента к серверу последний пересылает файл *cookie* вместе с запрошенными web-страницами браузеру, а браузер сохраняет этот файл *cookie*. Каждый раз в дальнейшем при обращении клиента к серверу файлы *cookie* пересылаются браузером на сервер вместе с текстом запроса. В теории этот подход работает прекрасно, на деле же файлы *cookie* для безопасного хранения отсылаются браузером назад на сервер часто совершенно скрытно от пользователя. Такая вот «великая» секретность. В отличие от бабушкиного печенья (*cookie* — печенье), эти *cookie*, как правило, следует оставлять там, где их «испекли».

3.3.2. Серверы объектов

После того как мы рассмотрели основные моменты, относящиеся к проектированию серверов, обсудим особый тип серверов, который становится все важнее. *Сервер объектов (object server)* — это сервер, ориентированный на поддержку распределенных объектов. Важная разница между стандартным сервером объектов и другими (более традиционными) серверами состоит в том, что сам по себе сервер объектов не предоставляет конкретной службы. Конкретные службы реализуются объектами, расположенными на сервере. Сервер предоставляет только средства обращения к локальным объектам на основе запросов от удаленных клиентов. Таким образом, можно относительно легко изменить набор служб, просто добавляя или удаляя объекты.

Сервер объектов, соответственно, выступает как место для хранения объектов. Объект состоит из двух частей: данных, отражающих его состояние, и кода, образующего реализацию его методов. Будут ли эти части храниться отдельно, а также смогут ли методы совместно использоваться несколькими объектами, зависит от сервера объектов. Кроме того, существует разница и в способе обращения сервера объектов к его объектам. Например, в многопоточном сервере объектов отдельный поток выполнения может быть назначен каждому объекту или каждому запросу на обращение к объекту. Эту и другие тонкости мы сейчас и обсудим.

Альтернативы обращению к объектам

Для любого объекта, к которому происходит обращение, сервер объектов должен знать, какой код выполнять, с какими данными работать, запускать ли отдельный поток выполнения для поддержки обращения и т. д. Простейший подход — считать, что все объекты выглядят одинаково и обращение к ним может производиться единообразно. Именно так работает среда DCE. К сожалению, подобному подходу обычно не хватает гибкости, и он часто накладывает на разработчиков распределенных объектов неоправданные ограничения.

Более правильный подход со стороны сервера — поддерживать различные правила обработки объектов. Рассмотрим, например, нерезидентные объекты. Напомним, что нерезидентным называется объект, существующий только во время существования сервера, а возможно, и еще более короткий срок. Типичной реализацией нерезидентного объекта можно считать хранящуюся в памяти копию файла, предназначенную только для чтения. Калькулятор (обычно запускаемый на высокоскоростном сервере) также может быть реализован в виде нерезидентного объекта. Разумно создать нерезидентный объект при первом же запросе к нему, а уничтожить после того, как не останется связанных с этим объектом клиентов. Преимущество подобного подхода состоит в том, что нерезидентные объекты нуждаются в ресурсах сервера только до тех пор, пока в этих объектах есть необходимость. Недостаток — в том, что обращение может потребовать для выполнения определенного времени, поскольку объект еще нужно создать. Поэтому иногда применяется другая политика — все нерезидентные объекты создаются при инициализации сервера. Это дается ценой выделения ресурсов на объекты даже в том случае, если ни один клиент не станет их использовать.

Схожим образом сервер мог бы выделять каждому из своих объектов отдельный сегмент памяти. Другими словами, можно запретить совместное использование объектами кода и данных. Подобный подход мог бы применяться в тех случаях, когда для реализации каждого объекта не требуется отделять код от данных или когда объекты нужно отделить друг от друга по соображениям безопасности. В этом последнем случае сервер должен будет для гарантированной защиты границ сегментов предоставить специальные средства измерения или требовать поддержки от базовой операционной системы. При альтернативном подходе можно позволить объектам совместно использовать хотя бы код. Так, база данных, содержащая объекты, относящиеся к одному классу, может быть эффективно реализована путем однократной загрузки на сервер реализации этого класса. В случае прихода запроса на обращение к объекту серверу достаточно будет извлечь из базы данных состояние объекта и выполнить вызванный метод.

Подобным же образом существует множество разных подходов, относящихся к работе с потоками выполнения. Простейший из них — реализация сервера с единственным управляющим потоком выполнения. С другой стороны, сервер может поддерживать несколько потоков выполнения — по одному на каждый объект. В случае прихода запроса с обращением к объекту сервер просто передает запрос потоку выполнения, отвечающему за этот объект. Если поток выполнения в этот момент занят, запрос ставится в очередь. Преимущество такого подхода — в автоматической защите объектов от одновременного доступа: для единственного связанного с объектом потока выполнения все обращения выстраиваются по порядку. Разумеется, можно также выделять отдельный поток выполнения каждому запросу, но при этом необходимо предварительно защитить объекты от одновременного доступа. Независимо от того, назначается поток выполнения каждому объекту или каждому методу, нужно решить, создавать каждый поток по запросу или поддерживать на сервере пул потоков. Оптимального решения «на все случаи жизни» здесь быть не может.

Адаптер объектов

Правила обращения к объекту обычно называют *политикой активизации* (*activation policies*), чтобы подчеркнуть тот факт, что во многих случаях объект, перед тем как к нему можно будет обратиться, должен быть перемещен в адресное пространство сервера (то есть активизирован).

Нам нужен механизм группирования объектов в соответствии с политикой активизации каждого из них. Этот механизм называют *адаптером объектов* (*object adapter*), или *упаковщиком объектов* (*object wrapper*), но чаще всего его существование скрыто в наборе средств построения сервера объектов. Мы будем использовать термин «адаптер объектов». Адаптер объектов прекрасно подходит для программ, реализующих собственную политику активизации. Главное, однако, что этот адаптер объектов является общедоступным для разработчиков распределенных объектов компонентом. Его нужно только сконфигурировать под соответствующую политику.

Адаптер объектов контролирует один или несколько объектов. Поскольку сервер должен одновременно поддерживать объекты с различной политикой активизации, на одном сервере может одновременно работать несколько адаптеров объектов. При получении сервером запроса с обращением к объекту этот запрос сначала передается соответствующему адаптеру объектов, как показано на рис. 3.7.

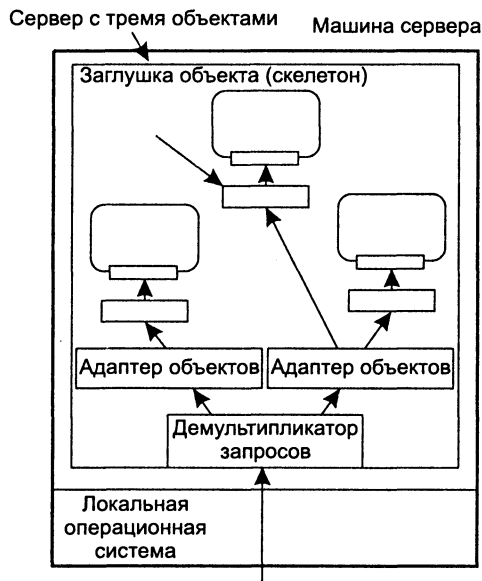


Рис. 3.7. Организация сервера объектов с разной политикой активизации

Важно отметить, что этот адаптер объектов не осведомлен о конкретных интерфейсах объектов, которые он контролирует. Другими словами, он никогда не бывает конкретен. Единственное, что для него важно, — возможность извлечь

ссылку на объект из запроса и передать запрос объекту в соответствии с его политикой активизации. Как показано на рисунке, вместо прямой передачи запроса объекту адаптер передает запрос серверной заглушке этого объекта. Заглушка, называемая еще скелетоном и обычно генерируемая из определения интерфейса объекта, выполняет демаршалинг запроса (получает параметры запроса) и обращается к соответствующему методу.

Для примера рассмотрим адаптер объектов, управляющий несколькими объектами. Адаптер реализует политику, в соответствии с которой каждым из объектов управляет отдельный поток. Для взаимодействия адаптера объектов со скелетонами (у каждого объекта скелетон собственный), выполняющими маршалинг и демаршалинг запросов, каждый скелетон должен реализовать следующую операцию:

```
invoke(unsigned in_size, char in_args[], unsigned* out_size, char* out_args[])
```

Здесь `in_args` — это массив байтов, который заглушка подвергает демаршалингу, чтобы извлечь аргументы запроса. Массив содержит идентификацию метода и значения для всех его параметров. Точный формат массива известен только заглушке, которая и отвечает за действительный вызов. Параметр `in_size` задает размер массива `in_args`. Аналогичным образом заглушка выполняет маршалинг всех выходных данных в массив `out_args`, который динамически создается заглушкой. Размер массива задается в выходном параметре `out_size` (отметим, что процедура `invoke` соответствует версии для динамических вызовов, которая обсуждалась в предыдущей главе).

В листинге 3.1 приведен заголовочный файл адаптера. Наиболее важная его часть — определение сообщений, которыми обмениваются адаптер и удаленный клиент.

Листинг 3.1. Заголовочный файл адаптера `header.h`, используемый как адаптером, так и всеми программами, которые к нему обращаются

```
/* Определения, необходимые адаптеру и тем программам, которые к нему обращаются */
#define TRUE 1
#define MAX_DATA 65536

/* Определение стандартного формата сообщения */
struct message {
    long    source:        /* определение отправителя */
    long    object_id:     /* идентификатор запрашиваемого объекта */
    long    method_id:     /* идентификатор запрашиваемого метода */
    unsigned size:        /* число байтов в списке параметров */
    char    *data:         /* параметры как последовательность байтов */
};

/* Общее определение операции, вызываемой для скелетона объекта */
typedef void (*METHOD_CALL) (unsigned, char*, unsigned*, char**);

long register_object(METHOD_CALL call); /* зарегистрировать объект */
void unregister_object(long object_id); /* отменить регистрацию объекта */
void invoke_adapter(message *request)    /* вызвать адаптер
```

Каждый клиент производитmarshaling запроса в сообщение, содержащее пять полей. Адаптер возвращает ответ в сообщении с аналогичной структурой. Поле `source` определяет отправителя сообщения. Поля `object_id` и `method_id` определяют соответственно объект и метод, к которым производится обращение. Исходные данные, передаваемые заглушке, помещаются в массив `data`, размер которого задается полем `size`. Результат обращения помещается в поле `data` нового сообщения.

Заголовочный файл содержит также определение способа вызова адаптера серверной заглушкой объекта путем определения типа `METHOD_CALL`.

И наконец, адаптер предоставляет две процедуры, которые могут быть вызваны сервером для регистрации и отмены регистрации объекта в адаптере. Регистрация производится путем передачи указателя на реализацию процедуры данного объекта `invoke` (она реализована в заглушке объекта). Функция регистрации возвращает число, которое может эффективно использоваться адаптером в качестве идентификатора объекта. Для отмены регистрации сервер должен передать это число путем вызова процедуры `unregister_object`. Реальный вызов адаптера происходит в ходе выполнения процедуры `invoke_adapter`, которая требует идентификатор объекта и запроса. Результат этой операции, как показано ниже, будет помещен в отдельный буфер.

При реализации адаптера мы предполагаем, что доступный пакет потоков выполнения предоставляет нам средства для создания (и уничтожения) потоков выполнения и для их взаимодействия друг с другом. Связь между потоками выполнения осуществляется посредством буферов. В частности, каждый поток имеет собственный буфер, из которого он может удалять сообщения путем блокирующей операции `get_msg`. Сообщения добавляются в буфер путем неблокирующей операции `put_msg`. Основная часть заголовочного файла пакета потоков выполнения представлена в листинге 3.2.

Листинг 3.2. Файл `thread.h` используется в адаптере для работы с потоками выполнения

```
typedef struct thread THREAD; /* Скрытое определение потока */

THREAD *create_thread (void (*body)(long tid), long thread_id);
/* Создать поток путем передачи указателя на функцию, которая определяет текущее
поведение потока, вместе с целым, которое используется как уникальный идентификатор
потока */

void get_msg(unsigned *size, char **data);
void put_msg(THREAD *receiver, unsigned size, char *data);
/* Вызов операции get_msg блокирует поток до тех пор, пока сообщение не попадет
в соответствующий буфер. Помещение сообщения в буфер потока – это неблокирующая
операция */
```

Мы подошли к реализации адаптера, приведенной в листинге 3.3. Каждый из объектов имеет ассоциированный с ним поток выполнения, заданный процедурой `thread_per_object`. Поток выполнения начинает работу с блокировки и остается заблокированным, пока запрос не окажется в буфере потока. Запрос немедленно

передается заглушке объекта путем вызова `invoke[object_id]` с соответствующим значением параметра. Результат обращения к объекту возвращается в переменной `results` и копируется в ответное сообщение. Это ответное сообщение собирается из полей `object_id` и `method_id`. За ними следует результат, который копируется в поле данных сообщения. В этом состоянии ответ пропускается через демультимплексор, как показано на рис. 3.7. В нашем примере демультимплексор реализуется отдельным потоком выполнения, идентифицируемом переменной `root`.

Листинг 3.3. Основная часть адаптера, реализующего политику «один объект — один поток»

```
#include <header.h>
#include <thread.h>
#define MAX_OBJECTS      100
#define NULL              0
#define ANY               -1

/* Массив указателей на заглушки */
METHOD_CALL invoke[MAX_OBJECTS];
/* Поток выполнения демультимплексатора */
THREAD *root;
/* По одному потоку выполнения на объект */
THREAD *thread[MAX_OBJECTS];

void thread_per_object(long object_id) (
    message      *req, *res;          /* сообщения — запрос/ответ */
    unsigned     size;                /* размер сообщений */
    char         *results;            /* массив со всеми результатами */

    while(TRUE) {
        get_msg(&size, (char*) &req); /* блокировка для запроса */

        /* Передача запроса соответствующей заглушке */
        /* Заглушка должна зарезервировать память для результатов */
        (invoke[object_id])(req->size, req->data, &size, &results);
        /* Создать ответное сообщение: */
        res = malloc(sizeof(message)+size);
        /* Определить объект: */
        res->object_id = object_id;
        /* Определить метод: */
        res->method_id = req->method_id;
        /* Установить размер результатов обращения: */
        res->size = size;
        /* Скопировать результаты в ответное сообщение: */
        memcpy(res->data, results, size);
        /* Добавить ответ к содержимому буфера: */
        put_msg(root, sizeof(res), res);
        /* Освободить память, выделенную под запрос: */
        free(req);
        /* Освободить память, выделенную под результаты: */
        free(results);
```

```
    }  
}  
void invoke_adapter(long oid, message *request) {  
    put_msg(thread[oid], sizeof (request), request);  
}
```

Теперь реализация процедуры `invoke_adapter` несложна. Вызываемый поток выполнения (в нашем примере — демультимплексор) добавляет свой запрос на обращение к буферу потока, ассоциированному с объектом, к которому был запрошен доступ. Позднее демультимплексор может извлечь результаты из своего собственного буфера и вернуть их клиенту, который и заказывал обращение к объекту.

Важно отметить, что реализация адаптера не зависит от объектов, обращения к которым он обрабатывает. Фактически в примере реализации нет никакого зависящего для объектов кода. Соответственно, можно создать обобщенный адаптер и поместить его на промежуточный уровень программного обеспечения. После этого разработчикам серверов объектов можно сконцентрироваться исключительно на разработке объектов, просто указывая при необходимости, какой адаптер обращения к каким объектам должен контролировать.

Последнее замечание. Хотя на рис. 3.7 показан специальный компонент, который занимается распределением поступивших запросов соответствующим адаптерам (демультимплексор), он не обязателен. Для этой цели мы вполне могли бы использовать адаптер объектов. Как мы увидим в главе 9, подобный подход реализован в технологии CORBA.

3.4. Перенос кода

До сих пор мы в основном обсуждали распределенные системы, в которых взаимодействие ограничивалось передачей данных. Однако существуют ситуации, когда передача программ, иногда даже во время их выполнения, позволяет упростить разработку распределенных систем. В этом разделе мы детально рассмотрим, как происходит перенос кода. Мы начнем с обсуждения различных подходов к переносу кода и продолжим разговором о локальных ресурсах, которые используются переносимыми программами. Отдельной сложной проблемой является перенос кода в гетерогенных системах. Об этом мы тоже поговорим. Чтобы вести предметный разговор, в конце этого раздела мы рассмотрим систему D'Agents для мобильных агентов. Отметим, что вопросы безопасности, связанные с переносом кода, вынесены в главу 8.

3.4.1. Подходы к переносу кода

Перед тем как рассматривать различные варианты переноса кода, обсудим сперва, зачем подобный перенос может понадобиться.

Причины для переноса кода

Традиционно перенос кода в распределенных системах происходит в форме *переноса процессов* (*process migration*), в случае которых процесс целиком переносится с одной машины на другую. Перенос работающего процесса на другую машину — дорогостоящая и сложная задача, и для ее выполнения должна быть веская причина. Такой причиной всегда была производительность. Основная идея состоит в том, что производительность может возрасти при переносе процессов с сильно загруженной на слабо загруженную машину. Загрузка обычно выражается в понятиях длины очереди к процессору или загрузки процессора, используются также и другие индикаторы производительности.

Алгоритм распределения загрузки, на базе которого принимаются решения, включающие распределение и перераспределение задач в соответствии с имеющимся набором процессоров, играет важную роль в системах интенсивных вычислений. Однако во многих современных распределенных системах оптимизация вычислительной мощности — менее важная задача по сравнению, например, со снижением коммуникационного трафика. Более того, учитывая гетерогенность базовых платформ и компьютерных сетей, повышение производительности путем переноса кода нередко основывается скорее на качественных рассуждениях, чем на математических моделях.

Рассмотрим, например, систему клиент-сервер, в которой сервер управляет большой базой данных. Если клиентское приложение собирается выполнять множество операций с базой данных, используя большие объемы данных, может быть лучше перенести часть клиентского приложения на сервер, а по сети передавать только результаты. В противном случае сеть может быть перегружена данными, передаваемыми с сервера на клиент. В этом случае перенос кода основан на соображении о том, что обычно имеет смысл обрабатывать данные поближе к тому месту, где они находятся.

Сходная причина может быть использована и при переносе части сервера на клиент. Например, во многих интерактивных приложениях баз данных клиент должен заполнять форму, которая затем будет преобразована в серию операций базы данных. Обработка формы на стороне клиента с пересылкой на сервер только заполненной формы нередко позволяет избежать пересылки по сети значительного количества небольших сообщений. В результате клиент покажет лучшую производительность, а сервер затратит меньше времени на обработку формы и взаимодействие.

Поддержка переноса кода может также помочь повысить производительность на основе параллелизма, но без обычных сложностей, связанных с параллельным программированием. Типичным примером может быть поиск информации в Web. Относительно несложно реализовать поисковый запрос в виде небольшой мобильной программы, переносимой с сайта на сайт. Создав несколько копий этой программы и разослав их по разным сайтам, мы можем добиться линейного возрастания скорости поиска по сравнению с единственным экземпляром программы.

Помимо повышения производительности существуют и другие причины поддержания переноса кода. Наиболее важная из них — это гибкость. Традицион-

ный подход к построению распределенных приложений состоит в разбиении приложения на части с последующим определением, где какая часть будет выполняться. Подобный подход, например, использовался в различных многозвенных приложениях клиент-сервер, обсуждавшихся в главе 1.

Однако, если код можно переносить с машины на машину, становится возможным конфигурировать распределенные системы динамически. Например, рассмотрим сервер, реализующий стандартизованный интерфейс к файловой системе. Чтобы предоставить удаленному клиенту доступ к файловой системе, сервер использует специальный протокол. В обычном варианте клиентская реализация интерфейса с файловой системой, основанная на этом протоколе, должна быть скомпонована с приложением клиента. Этот подход предполагает, что подобное программное обеспечение для клиента должно быть доступно уже тогда, когда создается клиентское приложение.

Альтернативой является предоставление сервером клиенту реализации не ранее, чем это будет действительно необходимо, то есть в момент привязки клиента к серверу. В этот момент клиент динамически загружает реализацию, производит необходимые действия по инициализации, а затем обращается к серверу. Схема такого пути показана на рис. 3.8. Подобная модель динамически переносимого из удаленного хранилища кода требует стандартизации протокола загрузки и инициализации кода. Кроме того, необходимо, чтобы загружаемый код можно было выполнять на машине клиента. Различные решения этой проблемы будут рассмотрены ниже и в последующих главах.

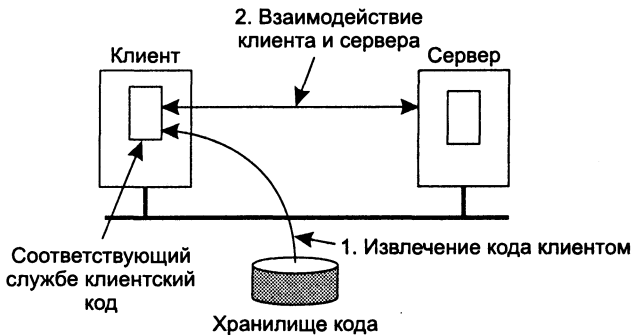


Рис. 3.8. Принцип динамического конфигурирования клиента для связи с сервером. Сначала клиент извлекает необходимое программное обеспечение, а затем обращается к серверу

Важное преимущество подобной модели динамической дозагрузки клиентского программного обеспечения состоит в том, что клиенту для общения с сервером нет необходимости иметь полный комплект предварительно устанавливаемого программного обеспечения. Вместо этого при необходимости программы могут быть перенесены на клиента и точно так же удалены, когда необходимость в них исчезнет. Другим преимуществом будет то, что поскольку интерфейсы стандартизованы, то мы можем менять протокол взаимодействия клиент-сервер и его реализацию так часто, как пожелаем. Изменения не окажут влияния на су-

ществующие клиентские приложения, связанные с сервером. Существуют, разумеется, и недостатки. Наиболее серьезный, который мы будем обсуждать в главе 8, связан с безопасностью. Слепо верить в то, что загружаемый код реализует только объявленные интерфейсы доступа к вашему незащищенному жесткому диску и не отправляет наиболее интересные фрагменты неизвестно кому, — не всегда оправдано.

Модели переноса кода

Хотя перенос кода подразумевает только перемещение кода с машины на машину, этот термин имеет более широкую область применения. Традиционно связь в распределенных системах основана на обмене данных между процессами. Перенос кода в широком смысле связан с переносом программ с машины на машину с целью исполнения этих программ в нужном месте. В некоторых случаях, таких как перенос процессов, также должны переноситься состояние программы, получаемые сигналы и прочие элементы среды.

Для лучшего понимания различных моделей переноса кода используем шаблон, описанный в [156]. Согласно этому шаблону, процесс состоит из трех сегментов. *Сегмент кода* — это часть, содержащая набор инструкций, которые выполняются в ходе исполнения программы. *Сегмент ресурсов* содержит ссылки на внешние ресурсы, необходимые процессу, такие как файлы, принтеры, устройства, другие процессы и т. п. И наконец, *сегмент исполнения* используется для хранения текущего состояния процесса, включая закрытые данные, стек и счетчик программы.

Абсолютный минимум для переноса кода предлагает модель *слабой мобильности (weak mobility)*. Согласно этой модели допускается перенос только сегмента кода, возможно вместе с некоторыми данными инициализации. Характерной чертой слабой мобильности является то, что перенесенная программа всегда запускается из своего исходного состояния. Это происходит, например, с Java-апплетами. Достоинство подобного подхода в его простоте. Слабая мобильность требуется только для того, чтобы машина, на которую переносится код, была в состоянии его исполнять. Этого вполне достаточно, чтобы сделать код переносимым. Мы вернемся к данному вопросу, когда будем обсуждать перенос программ в гетерогенных системах.

В противоположность слабой мобильности, в системах, поддерживающих *сильную мобильность (strong mobility)*, переносится также и сегмент исполнения. Характерная черта сильной мобильности — то, что работающий процесс может быть приостановлен, перенесен на другую машину и его выполнение продолжено с того места, на котором оно было приостановлено. Ясно, что сильная мобильность значительно мощнее слабой, но и значительно сложнее в реализации. Примером системы, поддерживающей сильную мобильность, является система D'Agents, которую мы рассмотрим позднее в этой главе.

Независимо от того, является мобильность слабой или сильной, следует провести разделение на системы с переносом, инициированным отправителем, и системы с переносом, инициированным получателем. При переносе, *инициированном отправителем*, перенос инициируется машиной, на которой переносимый

код постоянно размещен или выполняется. Обычно перенос, инициированный отправителем, происходит при загрузке программ на вычислительный сервер. Другой пример — передача поисковых программ через Интернет на сервер баз данных в Web для выполнения запроса на этом сервере. При переносе, *инициированном получателем*, инициатива в переносе кода принадлежит машине-получателю. Пример такого подхода — Java-апплеты.

Перенос, инициируемый получателем, обычно реализуется проще. Во многих случаях перенос кода происходит между клиентом и сервером, причем инициатива исходит от клиента. Безопасный перенос кода на сервер, как это происходит при переносе, инициированном отправителем, часто требует, чтобы клиент был сначала зарегистрирован и опознавался сервером. Другими словами, сервер должен знать всех своих клиентов, поскольку клиенты, естественно, имеют доступ к ресурсам сервера, таким как его диск. Защита этих ресурсов является необходимым делом. В противоположность этому, загрузка кода в случае инициирования этого процесса принимающей стороной может осуществляться анонимно. Более того, сервер обычно не заинтересован в ресурсах клиента. Напротив, перенос кода на клиента производится исключительно с целью увеличения производительности клиента. С этой стороны в защите нуждается лишь небольшое количество ресурсов, таких как память и сетевое соединение. Мы вернемся к защите переноса кода позже и подробно рассмотрим ее в главе 8.

В случае слабой мобильности следует также разделять варианты, когда перенесенный код выполняется в процессе-приемнике или когда он выполняется в новом, специально запущенном процессе. Так, например, Java-апплеты просто загружаются в web-браузер и выполняются в адресном пространстве браузера. Преимущество этого подхода состоит в том, что нам нет необходимости запускать новый процесс и разрывать из-за этого связь с машиной-приемником. Основной недостаток состоит в том, что процесс-приемник приходится защищать от злонамеренного или случайного выполнения кода. Простое решение — потребовать от операционной системы для перемещенного кода создать отдельный процесс. Отметим, что, как уже говорилось, это решение не решает проблем с доступом к ресурсам.

Помимо переноса работающего процесса, называемого также миграцией процесса, сильная мобильность может также осуществляться за счет удаленного клонирования. В отличие от миграции процесса клонирование создает точную копию исходного процесса, которая выполняется на удаленной машине. Клон процесса выполняется параллельно оригиналу. В UNIX-системах удаленное клонирование имеет место при ответвлениях дочернего процесса в том случае, когда этот дочерний процесс продолжает выполнение на удаленной машине. Преимущество клонирования — в схожести со стандартными процедурами, осуществляемыми в многочисленных приложениях. Единственная разница между ними состоит в том, что клонированный процесс выполняется на другой машине. С этой точки зрения миграция путем клонирования — самый простой способ повышения прозрачности распределения.

Различные варианты переноса кода иллюстрирует рис. 3.9.

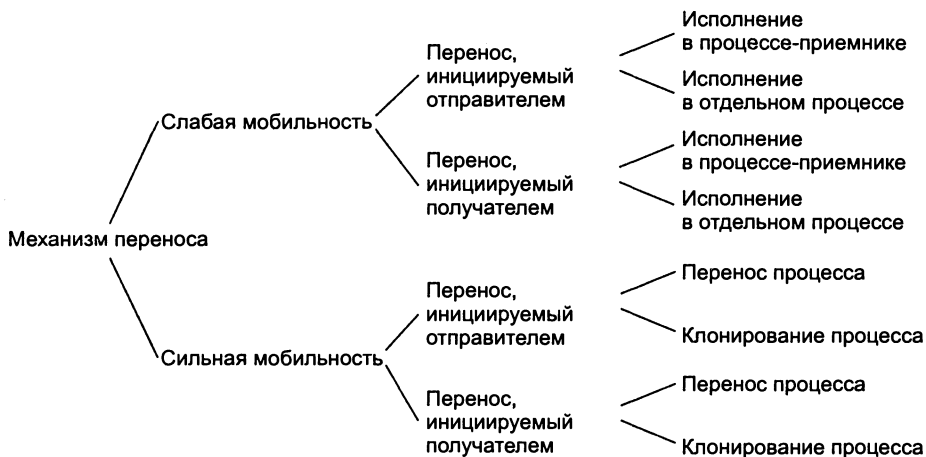


Рис. 3.9. Варианты переноса кода

3.4.2. Перенос и локальные ресурсы

Ранее мы рассматривали перенос только сегментов кода и исполнения. Сегмент ресурсов требует отдельного рассмотрения. Перенос кода нередко сильно затрудняет то, что сегмент ресурсов не всегда можно перенести с такой же легкостью без изменений, как другие сегменты. Так, например, рассмотрим процесс, содержащий ссылку на конкретный порт TCP, посредством которого он взаимодействует с другими (удаленными) процессами. Эта ссылка находится в сегменте ресурсов процесса. При переносе процесса на другую машину процесс должен освободить занятый им порт и запросить другой — на той машине, на которую он был перемещен. В других случаях перенос ссылки проблем не создает. Например, ссылка на файл с использованием абсолютного URL-адреса останется верной независимо от того, на какой машине выполняется процесс, содержащий этот URL-адрес.

Чтобы понять, какое влияние оказывает перенос кода на сегмент ресурсов, было выделено три типа связи процесса с ресурсами [156]. Наиболее сильная связь наблюдается, когда процесс ссылается на ресурс по его идентификатору. В этом случае процесс требует в точности тот ресурс, на который ссылается. Примером подобной *привязки по идентификатору (binding by identifier)* является использование процессом URL-адреса для ссылки на конкретный web-сайт или Интернет-адреса для ссылки на FTP-сервер. По этим же причинам ссылка на локальную конечную точку взаимодействия также будет считаться привязкой по идентификатору.

Более слабая связь процесса с ресурсами будет иметь место в том случае, если процессу необходимо только значение ресурса. В этом случае выполнение процесса ничуть не изменится, если такое же значение ему предоставит другой ресурс. Типичным примером *привязки по значению (binding by value)* являются обращения программ к стандартным библиотекам, как при программировании на языке C или Java. Эти библиотеки всегда доступны на локальной машине, но их

истинное местоположение в локальной файловой системе может быть различным. Для правильного выполнения процесса важны не конкретные имена файлов, а их содержимое.

И, наконец, наиболее слабая форма связи имеет место в том случае, когда процесс указывает на необходимость использования ресурса определенного типа. Подобная *привязка по типу (binding by type)* может быть проиллюстрирована ссылками на локальные устройства, такие как принтеры, мониторы и т. п.

При переносе кода мы часто нуждаемся в изменении ссылок на ресурсы, при этом изменять тип привязки ресурса к процессу запрещено. Можно ли изменять ресурсы, и если да, то как, зависит от того, могут ли они быть перенесены на машину-приемник вместе с кодом. Конкретнее, нам нужно определить связь ресурсов с машиной и рассмотреть варианты. *Неприсоединенные ресурсы (unattached resources)* могут быть с легкостью перенесены с машины на машину. Файлы (данных) в этом случае обычно связаны только с переносимой программой. В противоположность им, перенос или копирование *связанных ресурсов (fastened resources)* возможно лишь с относительно большими затратами. Типичными примерами связанных ресурсов могут быть локальные базы данных или сайты целиком. Несмотря на то что эти ресурсы теоретически не зависят от текущей машины, часто бывает невозможно перенести их в другую среду.

И, наконец, *фиксированные ресурсы (fixed resources)* изначально привязаны к конкретной машине или среде и не могут быть перенесены на другую. Фиксированными ресурсами часто бывают локальные устройства. Другой пример фиксированных ресурсов — локальные конечные точки взаимодействия.

Скомбинировав три типа привязки ресурсов к процессам и три типа привязки ресурсов к машине, получим девять комбинаций, которые следует рассмотреть, обсуждая вопрос переноса кода. Эти девять комбинаций иллюстрирует табл. 3.2. Используются следующие сокращения:

- ♦ GR — организовать глобальную ссылку;
- ♦ MV — перенести ресурс;
- ♦ CP — копировать значение ресурса;
- ♦ RB — выполнить новую привязку процесса к локальному ресурсу.

Таблица 3.2. Варианты переноса кода на другую машину

		Привязка ресурса к машине		
		Неприсоединенный ресурс	Связанный ресурс	Фиксированный ресурс
Привязка ресурса к процессу	Привязка по идентификатору	MV (или GR)	GR (или MV)	GR
	Привязка по значению	CP (или MV, GR)	GR (или CP)	GR
	Привязка по типу	RB (или MV, CP)	RB (или GR, CP)	RB (или GR)

Рассмотрим сначала возможности, возникающие при привязке процесса к ресурсу по идентификатору. Если ресурс неприсоединенный, лучше всего перене-

сти его на другую машину вместе с кодом. Однако если этот ресурс используется переносимым процессом совместно с другими, следует организовать на него глобальную ссылку — ссылку, которая в состоянии будет преодолеть границу между машинами. Примером такой ссылки может быть URL. Если ресурс связанный или фиксированный, организация глобальной ссылки также является наилучшим решением проблемы.

Важно, что реализация системы глобальных ссылок может быть сложнее простого использования URL и иногда оказывается слишком дорогостоящей. Рассмотрим, например, программу обработки высококачественных изображений на отдельной рабочей станции. Создание в реальном времени высококачественных изображений — это задача, требующая интенсивных вычислений, поэтому программа может быть перенесена на высокопроизводительный вычислительный сервер. Организация глобальных ссылок на рабочую станцию будет означать организацию связи между сервером и рабочей станцией. Кроме того, серьезная обработка, происходящая одновременно на сервере и рабочей станции, потребует соблюдения определенных требований к скорости передачи изображений. В результате может оказаться, что перенос программы на вычислительный сервер не оправдан просто потому, что цена поддержания глобальных ссылок чересчур высока.

Другим примером трудностей с поддержанием глобальных ссылок может быть перенос процесса, использующего локальную конечную точку взаимодействия. В этом случае мы имеем дело с фиксированными ресурсами, привязанными к процессу по идентификатору, поэтому имеется два основных решения. Одно из них состоит в том, чтобы позволить процессу после переноса установить соединение с исходной машиной, создав там отдельный поток выполнения, который просто будет перенаправлять все приходящие сообщения на новое «место жительства» процесса. Основным недостатком такого подхода является то, что при сбоях или повреждении исходной машины связь с перенесенным процессом будет прервана. Другое решение состоит в том, чтобы, взяв все процессы, связанные с перенесенным, поменять их глобальные ссылки и пересылать сообщения на новую конечную точку взаимодействия целевой машины.

Другая ситуация возникает в случае привязки по значению. Рассмотрим сначала фиксированные ресурсы. Комбинация фиксированных ресурсов и привязки по значению возможна, например, в случае использования процессом участка памяти совместно с другими процессами. Организация глобальных ссылок в этом случае может потребовать от нас реализации распределенной разделяемой памяти, о которой мы говорили в главе 1. Однако чаще всего подобное решение неприемлемо.

Связанные ресурсы, ссылка на которые производится по значению, — это чаще всего библиотеки времени исполнения. Обычно допускается копирование этих ресурсов на другую машину, причем это копирование может быть осуществлено до переноса кода. Организация глобальных ссылок может оказаться хорошей альтернативой копированию в том случае, если нужно скопировать большой объем данных, например словари и тезаурусы текстового редактора.

Наиболее простой случай — неприсоединенные ресурсы. Наилучшее решение при этом — скопировать (или переместить) ресурсы в новое место, исключая варианты, когда они совместно используются несколькими процессами. В последнем случае единственным выходом будет создание глобальных ссылок.

Последний вариант — привязка по типу. Независимо от способа привязки ресурса к машине решение состоит в новой привязке процесса к локальным ресурсам того же типа. Только в том случае, если ресурсы данного типа на локальной машине отсутствуют, мы можем скопировать или переместить оригинальные ресурсы на новое место или организовать глобальные ссылки на них.

3.4.3. Перенос кода в гетерогенных системах

Ранее мы предполагали, что перенесенный код может быть с легкостью выполнен на целевой машине. Это предположение относилось исключительно к гомогенным системам. Обычно, однако, распределенные системы создаются из набора гетерогенных платформ, каждая из которых имеет свою собственную машинную архитектуру и операционную систему. Перенос в подобных системах требует, чтобы поддерживались все эти платформы, то есть сегмент кода должен выполняться на всех этих платформах без перекомпиляции текста программы. Кроме того, мы должны быть уверены, что сегмент исполнения на каждой из этих платформ будет представлен правильно.

Проблемы могут быть частично устранены в том случае, если мы ограничимся слабой мобильностью. В этом случае обычно не существует такой информации времени исполнения, которую надо было бы передавать от машины к машине. Это означает, что достаточно скомпилировать исходный текст программы, создав различные варианты сегмента кода — по одному на каждую потенциальную платформу.

В случае сильной мобильности основной проблемой, которую надо будет решить, является перенос сегмента исполнения. Проблема заключается в том, что этот сегмент в значительной степени зависит от платформы, на которой выполняется задача. На самом деле перенести сегмент исполнения, не внося в него никаких изменений, можно только в том случае, если машина-приемник имеет ту же архитектуру и работает под управлением той же операционной системы.

Сегмент исполнения содержит закрытые данные процесса, его текущий стек и счетчик программы. Стек обычно содержит временные данные, такие как значения локальных переменных, но может также содержать и информацию, зависящую от платформы, например значения регистров. Важно отметить, что если бы нам удалось избавиться от данных, зависящих от платформы, то перенести сегмент на другую машину и продолжить исполнение там было бы значительно проще.

Решение, работающее в случае процедурных языков, таких как C или Java, показано на рис. 3.10. Перенос кода ограничен несколькими конкретными моментами выполнения программы. Точнее, перенос возможен только в момент вызова очередной подпрограммы. Под подпрограммой имеется в виду функция в C, метод в Java и т. п. Исполняющая система создает собственную копию про-

граммного стека, причем машинно-независимую. Мы будем называть эту копию *стеком переноса (migration stack)*. Стек переноса обновляется при вызове подпрограммы или возвращении управления из подпрограммы.

При вызове подпрограммы исполняющая система выполняет маршalling данных, которые были помещены в стек во время предыдущего вызова. Эти данные представляют собой значения локальных переменных, а также значения параметров текущего вызова процедуры. Данные после маршallingа помещаются в стек переноса вместе с идентификатором вызванной подпрограммы. Кроме того, в стек переноса помещается адрес (в форме метки перехода), с которого должно продолжаться исполнение после возвращения из подпрограммы.

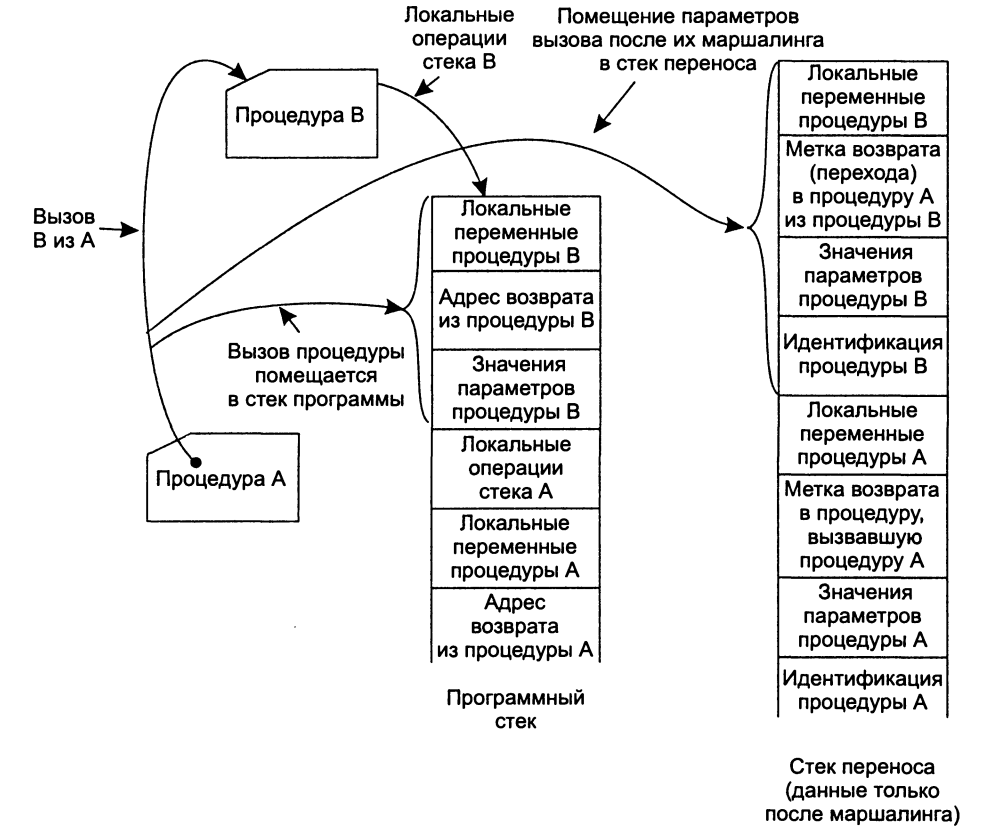


Рис. 3.10. Принцип работы стека переноса, поддерживающего перенос сегмента исполнения в гетерогенных средах

Если перенос кода происходит в точке вызова подпрограммы, исполняющая система производит сначала маршalling всех глобальных данных программы, составляющих сегмент исполнения. Данные, специфичные для данной машины, и текущий стек игнорируются. Данные после маршallingа и стек переноса передаются на ожидающую их машину. Кроме того, на машину-приемник загружает-

ся соответствующий сегмент кода, содержащий подходящие для ее архитектуры и операционной системы код. На машине-приемнике выполняется демаршалинг полученных данных сегмента исполнения, и из стека переноса формируется новый стек исполнения. После этого исполнение может быть продолжено простым входом в подпрограмму, которая была вызвана на исходном сайте.

Ясно, что подобный подход возможен только в том случае, если компилятор генерирует код для обновления стека переноса при каждом входе в подпрограмму или выходе из нее. Компилятор должен также генерировать в вызывающем коде метки, позволяющие реализовать выход из подпрограммы в виде переходов (машинно-независимых). Кроме того, мы нуждаемся в подходящей исполняющей системе. Тем не менее существует множество систем, успешно использующих подобную технологию. Так, в [127] показано, как можно обеспечить перенос программ, написанных на C/C++, в гетерогенных системах путем небольших изменений в языках, используя для добавления кода поддержки стека переноса исключительно директивы препроцессора.

Проблемы переноса кода, вызванные гетерогенностью, во многих случаях сходны с проблемами переносимости. Не будет неожиданностью то, что также сходны и методы их решения. Так, например, в конце 70-х годов было предложено простое решение, позволившее решить множество проблем с переносом языка Pascal на различные машины. Таким решением стала генерация промежуточного машинно-независимого кода для абстрактной виртуальной машины [40]. Эта машина, разумеется, требовала реализации на множестве различных платформ, благодаря которой программы на языке Pascal могли работать на них всех. Несмотря на то что эта простая идея некоторое время находила широкое применение, она никогда не считалась общим решением всех проблем переносимости для других языков, особенно C.

По прошествии приблизительно 20 лет проблему переноса кода в гетерогенных системах начали решать средствами языков сценариев, а также языков, обладающих высокой степенью переносимости, таких как Java. Все эти решения, в общем, основаны на виртуальной машине, которая интерпретирует либо непосредственно исходные тексты программ (в случае языков сценариев), либо промежуточный код, выдаваемый компилятором (для Java). Предложить свой товар в нужном месте в правильное время оказывается важно и для разработчиков языков программирования.

Единственный серьезный недостаток переносимости, реализуемой при помощи виртуальных машин, состоит в том, что приходится ограничиваться конкретным языком, да еще таким, которого частенько никто не знает. По этой причине важно, чтобы языки, предназначенные для переноса, имели интерфейс с существующими языками.

3.4.4. Пример — D'Agent

Чтобы проиллюстрировать перенос кода, рассмотрим теперь платформу промежуточного уровня, поддерживающую различные формы переноса кода. D'Agent, или полностью Agent TCL, — это система, построенная на основе концепции

агента. Агентом в системе D'Agent называется программа, которая в гетерогенной системе способна перемещаться с одной машины на другую. Сейчас мы сосредоточимся на возможностях переноса кода в системе D'Agent, а к более общему обсуждению программных агентов вернемся в следующем пункте. Мы также проигнорируем вопросы безопасности, о них мы поговорим в главе 8. Дополнительную информацию по D'Agent можно найти в [180, 242].

Обзор переноса кода в D'Agent

Агент в системе D'Agent — это программа, которая может перемещаться с одной машины на другую. В принципе программы могут быть написаны на разных языках, главное, чтобы машина, на которую переносится код, могла выполнить его. На практике это означает, что программы в D'Agent пишутся на интерпретируемых языках, а конкретнее, на командном языке утилит (Tool Command Language, Tcl), Java или Scheme [336, 375]. Использование исключительно интерпретируемых языков значительно облегчает поддержку гетерогенных систем.

Программа, или агент, исполняется в процессе, запущенном интерпретатором языка, на котором эта программа написана. Мобильность поддерживается тремя способами — посредством инициированной отправителем слабой мобильности, посредством сильной мобильности с переносом процессов и, наконец, посредством сильной мобильности с клонированием процессов.

Слабая мобильность реализуется при помощи команды `agent_submit`. В качестве параметра используется идентификатор машины, на которую переносится код. На этой же машине выполняется сценарий. Сценарий — это не что иное, как последовательность инструкций. Сценарий переносится на машину-получатель вместе со всеми описаниями процедур и копиями переменных, которые необходимы для его выполнения на этой машине. На машине-получателе процесс запускает подходящий интерпретатор, который и выполняет сценарий. В понятиях вариантов переноса кода (см. рис. 3.9), D'Agent обеспечивает инициированную отправителем слабую мобильность, когда перенесенный код выполняется в отдельном процессе.

В качестве примера слабой мобильности в D'Agent рассмотрим листинг 3.4. В нем представлена часть простого агента на языке Tcl, которая посылает сценарий на удаленную машину. В этом агенте процедура `factorial` получает единственный параметр и рекурсивно вычисляет факториал для значения параметра. Переменные `numbers` и `machine` должны быть правильно инициализированы (путем запроса значения у пользователя), после чего агент вызывает команду `agent_submit`. На целевую машину, определяемую значением переменной `machine`, вслед за определением процедуры `factorial` и исходным значением переменной `number` пересылается следующий сценарий:

```
factorial $number
```

D'Agent автоматически вычисляет результат и возвращает его отправителю. Вызов `agent_receive` приводит к блокированию агента, инициировавшего перенос, до момента возвращения ему результата.

Листинг 3.4. Простой пример агента D'Agent на языке Tcl, пересылающего сценарий на удаленную машину (адаптация сценария из [179])

```
proc factorial n {
    if {$n <= 1} {return 1;}          #fac(1) = 1
    expr $n * [factorial [expr $n - 1]] #fac(n) = n * fac(n-1)
}

set number ...      # указать, какой факториал вычислять
set machine ...     # идентифицировать целевую машину

agent_submit $machine -procs factorial -vars number -script { factorial $number }
agent_receive ...    # получить результат
```

Также поддерживается инициируемая отправителем сильная мобильность, как в форме миграции, так и в форме клонирования процессов. Для переноса работающего агента агент вызывает команду `agent_jump` с указанием целевой машины, на которую он должен быть перенесен. При вызове команды `agent_jump` исполнение агента на исходной машине приостанавливается и его сегмент ресурсов, сегмент кода и сегмент исполнения подвергаются маршалингу, укладываясь в сообщение, которое затем пересылается на целевую машину. После доставки сообщения запускается новый процесс, в котором исполняется соответствующий интерпретатор. Этот процесс выполняет демаршалинг пришедшего сообщения и продолжает выполнение с инструкции, следующей за последним вызовом `agent_jump`. Процесс, в котором агент выполнялся на исходной машине, прекращает свою работу.

Пример переноса агентов, приведенный в листинге 3.5, иллюстрирует упрощенную версию агента, который обнаруживает только что вошедшего в систему пользователя, выполняя команду `who` UNIX-системы на каждом из хостов. Поведение агента определяется процедурой `all_users`. Она поддерживает список пользователей, который изначально пуст. Набор хостов, которые будут посещаться агентом, определяется параметром `machines`. Агент переходит с хоста на хост, помещая результат выполнения команды `who` в переменную `users` и добавляя его к списку. В основной программе на текущей машине создается агент для рассылки с использованием рассмотренного ранее механизма слабой мобильности. В данном случае команда `agent_submit` будет вызвана для выполнения следующего сценария:

```
all_users $machines
```

Команде передается процедура и набор хостов в качестве дополнительных параметров.

Листинг 3.5. Пример агента D'Agent на языке Tcl, перемещающегося с машины на машину, исполняя команду `who` UNIX-системы (адаптация сценария из [73])

```
proc all_users machines {
    set list ""          # создать изначально пустой список
    foreach m $machines { # для всех хостов из заданных машин
        agent_jump $m     # переместиться на следующий хост
        set users [exec who] # выполнить команду who
    }
}
```

```
        append list $users      # добавить результаты к списку
    }
    return $list                # по окончании вернуть список
}
set machines ...                # инициализировать набор машин
set this_machine ...            # задать хост для старта агента

# Создать агент переноса для переноса сценария на ту машину,
# с которой агент начнет просматривать остальные машины.
# указанные в $machines

agent_submit $this_machine -procs all_users -vars machines \
                        -script {all_users $machines}
agent_receive ...              # получить результат
```

И, наконец, поддерживается клонирование процессов посредством команды `agent_fork`. Эта команда работает почти так же, как и `agent_jump`, за исключением того, что процесс, запустивший агента на исходной машине, просто продолжает работу с инструкции, следующей за вызовом `agent_fork`. Подобно операции `fork` в UNIX-системах, команда `agent_fork` возвращает значение, по которому вызвавший ее процесс может определить, что перед ним — клонированная версия (в UNIX именуемая «дочерней») или оригинальный процесс («родитель»).

Вопросы реализации

Чтобы рассмотреть некоторые детали внутренней реализации, рассмотрим написанные на Tcl агенты. Изнутри система D’Agent состоит из пяти уровней, как показано на рис. 3.11. Самый нижний уровень сравним с сокетами Беркли, в том смысле, что он реализует единый интерфейс механизмов взаимодействия базовой сети. В D’Agent предполагается, что базовая сеть предоставляет механизмы для работы с сообщениями TCP и электронной почты.

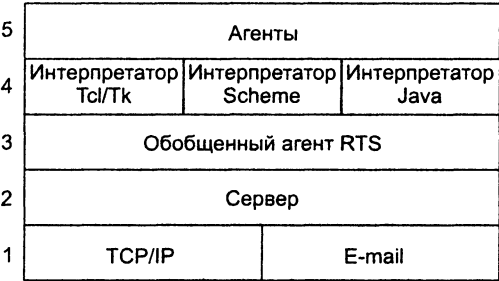


Рис. 3.11. Архитектура системы D’Agent

Следующим уровнем является сервер, работающий на каждой машине, на которой выполняется D’Agent. Сервер отвечает за управление агентами, авторизацию и управление связью между агентами. Для последнего вида деятельности сервер присваивает каждому агенту локальный уникальный идентификатор. Если использовать сетевой адрес сервера, каждый из агентов может быть обозна-

чен парой (адрес, локальный идентификатор). Подобное имя низкого уровня используется для установки связи между двумя агентами.

Третий уровень — сердце системы D'Agent. Он содержит независимое от языка ядро, которое поддерживает основные модели агентов. Так, например, этот уровень содержит реализацию запуска и окончания работы агента, реализации различных операций переноса и средства для связи между агентами. Понятно, что операции ядра недалеко ушли от операций сервера, но в отличие от сервера ядро не отвечает за управление набором агентов, размещенных на одной машине.

Четвертый уровень содержит интерпретаторы, по одному на каждый поддерживаемый в D'Agent язык. Каждый интерпретатор содержит компонент интерпретации языка, модуль безопасности, интерфейс с уровнем ядра и отдельный модуль для перехвата состояния работающего агента. Этот последний модуль необходим для поддержки сильной мобильности и будет детально рассмотрен ниже.

Самый верхний уровень содержит агенты, написанные на одном из поддерживаемых системой языков. Каждый агент D'Agent выполняется в отдельном процессе. Так, например, когда агент переносится на машину A, сервер разветвляет процесс выполнения соответствующего интерпретатора, создавая ветку для выполнения этого агента. Новый процесс подхватывает состояние мигрировавшего агента и продолжает его выполнение с той точки, на которой он был приостановлен. Сервер отслеживает локальные каналы созданного процесса, посредством которых процесс получает предназначенные для него сообщения.

Более сложная часть реализации D'Agent — это получение состояния работающего агента и передача его на другую машину. В случае Tcl состояние агента описывается частями, показанными в табл. 3.3. По существу, состояние агента описывается четырьмя таблицами, содержащими глобальные определения переменных и сценариев, и двумя стеками, отслеживающими состояние выполнения.

Таблица 3.3. Части, описывающие состояние агента в системе D'Agent

Состояние	Описание
Глобальные переменные интерпретатора	Переменные, необходимые интерпретатору агентов
Глобальные системные переменные	Коды возврата, коды ошибок, строки сообщений об ошибках и т. д.
Глобальные переменные программы	Определяемые пользователем глобальные переменные программы
Определения процедур	Определения сценариев, выполняемых агентом
Стек команд	Стек выполняемых в настоящее время команд
Стек вызовов	Стек записей об активизации, по одной на каждую выполняемую команду

Интерпретатору необходима таблица, в которой хранятся глобальные переменные. Так, обработчик событий может сообщать интерпретатору, какую процедуру следует вызывать в случае прихода сообщения от некоторого агента. Пары (событие, обработчик) хранятся в таблице интерпретатора. В другой таблице

содержатся глобальные системные переменные, в которых хранятся коды ошибок, строки с сообщениями об ошибках, коды результатов, строки сообщений, выводимые вместе с результатами и т. п. Имеется также отдельная таблица, в которой хранятся все определенные пользователем глобальные переменные программы. И наконец, в отдельной таблице хранятся определения процедур, связанных с агентами. Эти определения процедур нуждаются в переносе вместе с агентами для выбора интерпретатора на целевой машине.

Более интересная часть, напрямую связанная с переносом агентов, — два стека, в которых хранится истинное состояние выполнения агента. В основе каждого агента лежит набор команд Tcl, возможно, встроенных в конструкции, такие как циклы, инструкции множественного выбора и т. д. Кроме того, команды могут быть сгруппированы в процедуры. Как это происходит во всех интерпретируемых языках, агент выполняется команда за командой.

Сначала рассмотрим, что происходит при выполнении базовой команды Tcl, то есть команды, которая вызывается не из пользовательской процедуры. Интерпретатор анализирует команду и строит запись, помещаемую в то, что мы называем *стеком команд (command stack)*. Эта запись содержит все необходимые для выполнения команды поля, как то: значения параметров, указатель на процедуру реализации команды и т. п. Запись помещается в стек, после чего может быть использована компонентом, отвечающим за выполнение команды. Другими словами, стек команд представляет собой место хранения текущего состояния выполнения агента.

Tcl также поддерживает определяемые пользователем процедуры. Кроме стека команд среда исполнения D'Agent отслеживает стек записей активизации, также называемых фреймами вызова. Фрейм вызова в D'Agent содержит таблицу переменных, локальных для процедуры, а также имена и значения параметров, с которыми эта процедура была вызвана. Фрейм вызова создается только в результате вызова процедуры и относится к команде вызова процедуры, помещенной в стек команд. Фрейм вызова содержит ссылку на связанную с ним команду.

Рассмотрим теперь, что происходит, например, когда агент вызывает команду `agent_jump`, при помощи которой он переносится на другую машину. В этот момент полное состояние агента, описанное выше, подвергается маршалингу и превращается в последовательность байтов. Другими словами, все четыре таблицы и два стека укладываются в массив байтов и пересылаются на другую машину. Сервер D'Agent на целевой машине создает новый процесс, запуская интерпретатор Tcl. Процесс обрабатывает полученные данные, выполняет их демаршалинг и в результате получает состояние агента, в котором он находился перед вызовом команды `agent_jump`. Выполнение агента продолжается путем простого снятия с вершины стека команд очередной команды.

3.5. Программные агенты

Теперь мы рассмотрим процессы под несколько другим углом. Сначала мы сосредоточимся на одном из ключевых вопросов, управляющих потоках выполнения

внутри процессов. С позиций взаимодействия мы поближе рассмотрим обобщенную организацию клиентов и серверов. И наконец, обсудим перенос программ и процессов. Эти более или менее независимые представления процессов объединяются в то, что нередко называют программными агентами — автономные единицы, способные выполнять задания в кооперации с другими, возможно, удаленными агентами.

Агенты играют в распределенных системах все более важную роль. Однако очень близко к действительности утверждение о том, что у нас есть лишь интуитивное определение того, что такое процесс [334]. Определение программных агентов в таких условиях тоже нуждается в уточнении. В этом разделе мы поближе приглядимся к программным агентам и их роли в распределенных системах.

3.5.1. Программные агенты в распределенных системах

Существует множество мнений о том, что такое агент. Взяв за основу описание, данное в [173], мы определяем *программный агент (software agent)* как автономный процесс, способный реагировать на среду исполнения и вызывать изменения в среде исполнения, возможно, в кооперации с пользователями или с другими агентами. Свойство, которое делает агента чем-то большим, чем процесс, — это способность функционировать автономно и, в частности, проявлять при необходимости инициативу.

Наше определение программного агента получилось весьма неопределенным, и в результате многие типы процессов с легкостью могут восприниматься в качестве агентов. Вместо того чтобы делать попытки точнее определить программные агенты, будет разумнее говорить о разных типах программных агентов. То есть в литературе сделано несколько попыток разработать классификацию программных агентов, но договориться о единой классификации исследователям, по всей видимости, нелегко.

Помимо автономности важнейшее качество агентов — возможность кооперироваться с другими агентами. Сочетание автономности и кооперации приводит нас к классу кооперативных агентов [320]. *Кооперативный агент (collaborative agent)* — это агент, составляющий часть мультиагентной системы, то есть системы, в которой агенты, кооперируясь, выполняют некие общие задачи. Типичное приложение, использующее кооперативные агенты, — это электронная конференция. Каждый из докладчиков представлен агентом, имеющим доступ к вопросам, которые пользователь хочет представить на всеобщее рассмотрение. С учетом всех персональных ограничений на время, местоположение, перемещение и т. п. совместная работа отдельных агентов позволяет организовать конференцию. В перспективе, таким образом, могут производиться разработки распределенных систем, особенно предназначенных для обмена информацией.

Многие исследователи также выделяют из других типов агентов мобильные агенты. *Мобильный агент (mobile agent)* — это просто агент, у которого имеется способность перемещаться с машины на машину. В терминах, которые мы ис-

пользовали при обсуждении переноса кода в предыдущей главе, мобильные агенты часто требуют поддержки сильной мобильности, хотя это и не является абсолютно необходимым. Требование сильной мобильности вытекает из того факта, что агенты автономны и активно взаимодействуют со своей средой. Перенос агента на другую машину без учета его состояния будет сильно затруднен. Однако как было показано на примере системы D'Agent, сочетание агентов и слабой мобильности также вполне возможно. Отметим, что мобильность — это общее свойство агентов, наличие которого не приводит к выделению особого их класса. Так, например, имеет смысл говорить о существовании мобильных кооперативных агентов. Хороший пример практического использования мобильных агентов приведен в [74], где авторы описывают использование мобильных агентов для получения информации, распределенной по большой гетерогенной сети, такой как Интернет.

Способность к кооперации с другими агентами или перемещению между машинами — это системные свойства агентов. Они не говорят нам ничего о назначении агента. Для изучения функциональности агента нам требуется другая классификация.

Традиционно выделяемый класс — это класс интерфейсных агентов. *Интерфейсный агент (interface agent)* — это агент, помогающий конечному пользователю работать с одним или несколькими приложениями. Среди традиционно имеющихся у интерфейсного агента свойств можно считать способность к *обучению* [280, 320]. Чаще всего они взаимодействуют с пользователями, обеспечивая им поддержку. В контексте распределенных систем примером интересного интерфейсного агента может быть агент, отслеживающий взаимодействия между агентами и пользователями в некотором сообществе. Так, например, создаются специальные интерфейсные агенты для взаимодействия продавцов с покупателями. Правильно поняв, что хочет увидеть или что может предложить его владелец, интерфейсный агент может помочь предложить нужную группу товаров.

Очень близок к интерфейсному агенту *информационный агент (information agent)*. Основная функция подобных агентов — управление информацией из множества различных источников. Управление информацией включает в себя упорядочение, фильтрацию, сравнение и т. п. Важности информационным агентам в распределенных системах придает тот факт, что они могут работать с информацией из физически разных источников. Стационарные информационные агенты обычно работают с входящими потоками информации. Так, например, почтовый агент может применяться для фильтрации в почтовом ящике непрошенной корреспонденции, ее владельца или автоматического перенаправления входящей почты в соответствующие теме почтовые ящики. В противоположность им, мобильные информационные агенты обычно свободно путешествуют по сети, собирая по требованию их владельца необходимую ему информацию.

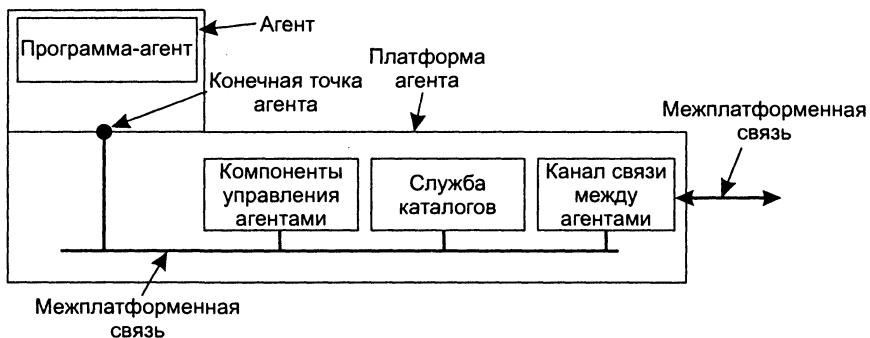
В целом агенты могут быть охарактеризованы набором свойств, приведенным в табл. 3.4 [152]. Дальнейшее разделение агентов можно провести, рассматривая, как они реально работают с точки зрения искусственного интеллекта. Краткий обзор этой стороны дела можно найти в [175, 196, 496].

Таблица 3.4. Некоторые важные свойства агентов

Свойство	Общность для агентов	Описание
Автономность	Да	Способность работать независимо от других
Реактивность	Да	Способность своевременно реагировать на изменения в своем окружении
Проактивность	Да	Способность инициировать действия, влияющие на их окружение
Коммуникативность	Да	Способность обмениваться информацией с пользователями и другими агентами
Продолжительность	Нет	Относительно долгое время жизни
Мобильность	Нет	Способность перемещаться с места на место
Адаптивность	Нет	Способность к обучению

3.5.2. Технология агентов

Представление о том, что такое агенты, бесполезно в том случае, если отсутствует поддержка в виде реально существующих систем агентов. Очень полезной была бы возможность выделить постоянно используемые компоненты агентов в распределенных системах и включить их, например, в программное обеспечение промежуточного уровня. В качестве исходной точки организация *FIPA (Foundation for Intelligent Physical Agents)* разработала обобщенную модель программных агентов. Согласно этой модели агенты регистрируются и работают под управлением платформы агентов, как показано на рис. 3.12. Платформа агентов предоставляет основные службы, необходимые любой мультиагентной системе. Сюда входят механизмы создания и уничтожения агентов, механизмы распознавания агентов и механизмы взаимодействия между агентами.

**Рис. 3.12.** Обобщенная модель платформы агента (адаптировано из [143])

Компонент управления агентами отслеживает агентов на соответствующей платформе. Он предоставляет механизмы создания и уничтожения агентов, а также для просмотра текущей конечной точки на предмет наличия конкретного

агента. В этом смысле он предоставляет службу наименования, посредством которой глобально уникальный идентификатор отображается на локальную конечную точку взаимодействия. Службы именования детально обсуждаются в следующей главе.

Кроме того, существует и отдельная локальная служба каталогов, при помощи которой агенты могут узнать, какие еще агенты имеются на этой платформе. Служба каталогов в модели FIPA основана на использовании атрибутов. Это означает, что агент предоставляет описания своих служб в понятиях имен атрибутов вместе с их значениями для данного агента. Это очень похоже на то, как устроен справочник «Желтые страницы». К службе каталогов могут иметь доступ удаленные агенты, то есть агенты, находящиеся на других платформах агентов.

Важный компонент платформы агента — *канал связи между агентами (Agent Communication Channel, ACC)*. В большинстве моделей мультиагентных систем агенты связываются друг с другом, пересылая сообщения. Модель FIPA — не исключение, она возлагает на ACC ответственность за все взаимодействие между различными платформами агентов. В частности, ACC отвечает за надежную и направленную связь точка-точка с другими платформами. Канал ACC может быть реализован просто в виде сервера, отслеживающего некоторый порт, предназначенный для входящих сообщений, которые перенаправляются другим серверам или агентам, являющимся частью платформы агентов. Для обеспечения межплатформенного взаимодействия связь между ACC соответствует Интернет-протоколу IIOP (Internet Inter-ORB Protocol), который мы будем обсуждать в главе 9. В архитектуре D'Agent в качестве ACC выступает сервер.

Языки взаимодействия агентов

Итак, у платформ агентов есть своя специфика. Отличие от других подходов к распределенным системам становится понятным при рассмотрении характера информации, которой реально обмениваются агенты. Связь между агентами происходит посредством коммуникационного протокола прикладного уровня, известного под названием *языка взаимодействия агентов (Agent Communication Language, ACL)*. В ACL присутствует жесткое разделение между *целью* сообщения и его *содержанием*. Сообщение может иметь только ограниченный набор целей. Например, целью сообщения может быть запрос на предоставление получателем определенной службы.

Также целью сообщения может быть ответ на ранее присланное сообщение с запросом. Другим примером цели сообщения может быть уведомление принимающей стороны о произошедшем событии или предложение чего-либо в ходе согласования. Некоторые из целей сообщений на языке ACL, разработанные FIPA, перечислены в табл. 3.5.

Идея ACL состоит в том, что агент-отправитель и агент-получатель как минимум одинаково понимают цель сообщения. Более того, цель сообщения обычно определяет и реакцию получателя. Так, например, при запросе предложения путем сообщения, имеющего в заголовке цель CFP, получатель на самом деле должен будет послать предложение, посредством сообщения с целью PROPOSE. В этом

смысле ACL действительно определяет высокоуровневый коммуникационный протокол для набора агентов.

Таблица 3.5. Примеры целей сообщения и описание содержания сообщений

Цель сообщения	Описание	Содержание сообщения
INFORM	Информировать, что данное предположение истинно	Предположение
QUERY-IF	Запросить, истинно ли данное предположение	Предположение
QUERY-REF	Запрос данного объекта	Выражение
CFP	Запросить предложение	Зависит от предложения
PROPOSE	Предоставить предложение	Предложение
ACCEPT-PROPOSAL	Сообщить, что данное предложение принято	Идентификатор предложения
REJECT-PROPOSAL	Сообщить, что данное предложение отвергнуто	Идентификатор предложения
REQUEST	Запросить осуществления действия	Спецификация действия
SUBSCRIBE	Подписаться на источник информации	Ссылка на источник

Как и большинство коммуникационных протоколов, сообщения ACL состоят из заголовка и реального содержания. Заголовок содержит поле цели сообщения, а также поля отправителя и получателя. Кроме того, как и во многих других коммуникационных протоколах, содержимое письма отделено и независимо от остальной его части. Другими словами, предполагается, что содержимое письма определяют вступившие в связь агенты. ACL никак не задает формат или язык содержания сообщения.

Теперь необходимо, чтобы принимающему агенту была предоставлена вся необходимая информация для правильной интерпретации содержания. Для этого заголовок сообщения ACL должен также определять язык или схему декодирования содержания. Этот подход хорошо работает до тех пор, пока отправитель и получатель одинаково интерпретируют данные, или, точнее, символы сообщения. Если единое понимание отсутствует, нередко требуются дополнительные поля, идентифицирующие стандартное отображение символов в их смысл. Такое отображение обычно называется *онтологией* (*ontology*).

Приведем простой пример. В табл. 3.6 показано сообщение на языке ACL, используемое для информирования агента о генеалогических связях в королевском семействе Нидерландов. Для определения агента, отправившего сообщение, и агента, которому оно предназначено, каждый из агентов имеет имя, состоящее из нескольких компонентов. Так, например, значение `max@http://fanclub-beatrix.royalty-sporters.nl:7239` может использоваться для ссылки на агента с именем `max`, находящегося на агентской платформе с DNS-именем `fanclub-beatrix.royalty-sporters.nl`. Для связи с агентом имя платформы должно быть сначала преобразовано из DNS в IP-адрес. Кроме того, доступ к имени, указанному для связи, должен производиться путем посылки сообщений серверу по протоколу HTTP на

его хост. Ожидает их прихода сервер на порту под номером 7239. В нашем примере агент `max` посылает информационное сообщение агенту `eike`, расположенному на платформе под именем `royalty-watcher.uk`. Сообщения должны посылаться по протоколу ИОР (мы обсудим его в главе 9) на порт под номером 5623.

Таблица 3.6. Простой пример сообщения ACL, пересылаемого одним агентом другому

Поле	Значение
Цель	INFORM
Отправитель	max@http://fanclub-beatrix.royalty-sptters.nl:7239
Получатель	eike@iiop://royalty-watcher.uk:5623
Язык	Prolog
Онтология	Genealogy
Содержимое	female(beatrix).parent(beatrix,juliana.bernhard)

Остальные поля сообщения относятся к его содержимому. Поле языка указывает на то, что содержимое сообщения представляет собой серию выражений на языке Prolog, а поле онтологии определяет, что эти конструкции Prolog интерпретируются как генеалогическая информация. В результате получающий сообщение агент должен понимать, что следующее выражение означает, что `beatrix` — это имя женщины:

```
female(beatrix)
```

С другой стороны, показанное ниже сообщение говорит о том, что мать `beatrix` зовут именем `juliana`, а отца — именем `Bernhard`:

```
parent(beatrix,juliana.bernhard)
```

3.6. Итоги

Процессы играют фундаментальную роль в распределенных системах, поскольку они формируют базис для связи между различными машинами. Важным вопросом является внутренняя организация процессов и в частности, способны ли они поддерживать несколько управляющих потоков выполнения. Потоки выполнения в распределенных системах используются, в частности, для продолжения работы с процессором во время блокирующих операций ввода-вывода. Таким образом, появляется возможность построения более эффективных серверов, в которых несколько потоков выполнения работают одновременно, причем некоторые из них могут быть заблокированы в ожидании выполнения дисковых операций ввода-вывода или операций сетевого взаимодействия.

Возможна организация распределенных приложений в понятиях клиентов и серверов. Клиентский процесс обычно реализует пользовательский интерфейс, который может варьироваться от простого вывода информации до расширенных интерфейсов, способных поддерживать составные документы. Клиентское программное обеспечение, кроме того, способно поддерживать прозрачность распре-

деления, скрывая детали, касающиеся связи с серверами, текущего местоположения серверов и репликации серверов. Кроме того, программное обеспечение клиента способно частично скрыть возникающие сбои и процессы восстановления после сбоев.

Серверы часто сложнее клиентов, но тем не менее при их построении применяется относительно немного архитектурных моделей. Так, например, серверы могут быть итеративными или параллельными, реализовывать одну или несколько служб, сохранять информацию о состоянии или не сохранять. Остальные архитектурные особенности касаются адресации служб и механизмов прерывания серверов после прихода запроса на обслуживание и возможно в ходе его выполнения.

Серверы объектов выделяются в особый класс. Коротко говоря, сервер объектов — это процесс, содержащий размещенные в своем адресном пространстве объекты и готовый принимать направленные к ним обращения. В отдельную категорию серверы объектов выделяются во многом благодаря разнообразию способов обращения к объектам. Так, например, сервер может запустить отдельный поток выполнения для каждого запроса к объекту. С другой стороны, он может выделять каждому объекту собственный поток выполнения или оставить единственный поток выполнения для всех своих объектов. Посредством адаптера объектов в различных серверах может быть реализована разная политика обращения к объектам. Коротко говоря, адаптер объектов — это компонент, реализующий только одну политику обращения. На сервере может быть несколько адаптеров объектов.

Важной для распределенных систем темой является перенос кода с машины на машину. Для того чтобы поддерживать перенос кода, имеются две веские причины — повышение производительности и мобильность. Если связь дорога, мы можем иногда уменьшить взаимодействие, перенеся вычисления с сервера на клиент и заставив клиента все что можно обрабатывать локально. Гибкость же возрастает, когда клиент имеет возможность динамически загружать программное обеспечение, необходимое для работы с конкретным сервером. Загруженное программное обеспечение может быть уже настроено на взаимодействие с этим сервером, избавляя клиента от необходимости повторно устанавливать его до начала работы.

Перенос кода приносит проблемы использования локальных ресурсов, связанные с тем, что эти ресурсы также необходимо переносить на другие машины, организовывать новые привязки кода к локальным ресурсам целевой машины или задействовать глобальные ссылки. Другая проблема заключается в том, что при переносе кода мы должны принимать во внимание гетерогенность системы. Текущая практика показывает, что, возможно, лучшим средством справиться с гетерогенностью являются виртуальные машины, которые эффективно скрывают гетерогенность с помощью интерпретируемого кода.

И, наконец, программные агенты — специальный вид процессов, работающих как автономные модули, но способных кооперироваться с другими агентами. С точки зрения распределенных систем, отличие агентов от обычных приложений в том, что агенты взаимодействуют друг с другом посредством коммуникационного протокола прикладного уровня, который называется языком взаи-

модействия агентов (ACL). В ACL имеется четкое разделение между целью сообщения и его содержимым. ACL определяет коммуникационный протокол верхнего уровня: посылка сообщения обычно предполагает конкретную реакцию получателя на основании исключительно цели сообщения.

Вопросы и задания

1. В этом задании сравнивается чтение файлов с использованием однопоточного и многопоточного файловых серверов. На то чтобы получить запрос, ответить на него и совершить все прочие действия, у вас есть 15 мс. Считается, что необходимые данные находятся в кэше основной памяти. Если потребуются дисковые операции, как это происходит в трети случаев, нужно дополнительно 75 мс, в течение которых поток выполнения будет приостановлен. Сколько запросов в секунду сможет выполнить однопоточный сервер? А многопоточный сервер?
2. Имеет ли смысл ограничивать число потоков выполнения серверного процесса?
3. В тексте мы описали многопоточный файловый сервер, показав, чем он лучше однопоточного сервера и сервера на базе конечного автомата. Существуют ли ситуации, в которых однопоточный сервер оказывается лучше? Приведите пример.
4. Статически ассоциировать с облегченным процессом единственный поток выполнения — это не самая лучшая идея. Почему?
5. Иметь в процессе только один облегченный процесс — также не лучшая идея. Почему?
6. Опишите простую схему, в которой облегченных процессов столько же, сколько работающих потоков выполнения.
7. Заместители могут поддерживать прозрачность репликации, обращаясь к каждой из реплик так, как описывалось в тексте. Может ли объект (на стороне сервера) быть предметом реплицированного обращения?
8. Создание параллельных серверов путем порождения вложенных процессов имеет свои преимущества и недостатки по сравнению с многопоточными серверами. Опишите их.
9. Набросайте архитектуру многопоточного сервера, поддерживающего посредством сокетов несколько протоколов, и его интерфейса транспортного уровня с базовой операционной системой.
10. Как мы можем предотвратить игнорирование приложениями менеджера окон и избежать полного беспорядка на экране?
11. Объясните, что такое адаптер объектов.
12. Опишите некоторые особенности адаптера объектов, используемого для поддержания сохраненных объектов.
13. Измените процедуру `thread_per_object` в примере адаптера объектов (см. листинг 3.3) так, чтобы все объекты, контролируемые этим адаптером, обрабатывались одним потоком выполнения.

14. Сервер, поддерживающий соединение TCP/IP с клиентом, — это сервер с фиксацией состояния или без фиксации состояния?
15. Представьте себе web-сервер, поддерживающий таблицу, в которой IP-адреса клиентов отображаются на наиболее часто посещаемые web-страницы. Когда клиент соединяется с сервером, сервер находит клиента в своей таблице и, если он обнаружен, возвращает записанную на него страницу. Это сервер с фиксацией состояния или без фиксации состояния?
16. В какой степени обращение RMI, написанное на языке Java, зависит от переноса кода?
17. Сильная мобильность в UNIX-системах может обеспечиваться путем разветвления процесса с образованием дочернего процесса на удаленной машине. Опишите, как это работает.
18. Из рис. 3.9 вытекает, что сильная мобильность не может сочетаться с выполнением переносимого кода в процессе-получателе. Приведите обратный пример.
19. Рассмотрим процесс P , который нуждается в доступе к файлу F , расположенному на той же машине, на которой в настоящее время запущен процесс P . При переносе процесса P на другую машину он по-прежнему требует доступа к F . Если привязка файла к машине фиксирована, как можно реализовать глобальную ссылку на F ?
20. Каждый агент в системе D'Agents реализуется в виде отдельного процесса. Агенты могут связываться друг с другом, прежде всего, через совместно используемые файлы и путем обмена сообщениями. Файлы не могут выходить за границы машин. В понятиях шаблона переноса, данных в разделе 3.4, какие части состояния агента, представленные в табл. 3.3, относятся к сегменту ресурсов?
21. Сравните архитектуру D'Agents с архитектурой платформы агентов модели FIPA.
22. Куда в модели OSI поместить языки взаимодействия агентов (ACL)?
23. Где в модели OSI размещались бы языки взаимодействия агентов (ACL), будь они реализованы поверх системы обработки электронной почты, такой как в D'Agents? Каковы были бы преимущества такого подхода?
24. Почему часто необходимо указывать онтологию сообщения ACL?

Глава 4

Именованние

4.1. Именованные сущности

4.2. Размещение мобильных сущностей

4.3. Удаление сущностей, на которые нет ссылок

4.4. Итоги

Имена играют важную роль во всех компьютерных системах. Они необходимы для совместного использования ресурсов, определения уникальных сущностей, ссылок на местоположения и т. д. Важная особенность именования состоит в том, что имя может быть разрешено, предоставляя доступ к сущности, на которую оно указывает. Разрешение имени, таким образом, представляет собой процесс доступа к именованной сущности. Для разрешения имен необходимо реализовать систему именования. Разница между именованием в распределенных и нераспределенных системах состоит в способе реализации систем именования.

В распределенных системах реализация системы именования часто сама по себе распределена по нескольким машинам. Способ этого распределения играет ключевую роль для эффективности и масштабируемости системы именования. В этой главе мы сосредоточимся на трех различных, но одинаково важных способах использования имен в распределенных системах.

Во-первых, до обсуждения некоторых общих вопросов, связанных с именованием, мы поближе рассмотрим организацию и реализацию «человеческих» имен. Типичные примеры подобных имен включают в себя имена файловой системы и World Wide Web. Построение глобальной масштабируемой системы именования имеет прямое отношение к этим типам имен.

Во-вторых, имена используются для локализации мобильных сущностей. Как оказывается, системы именования на основе «человеческих» имен не особенно подходят для поддержки большого количества мобильных сущностей, которые вдобавок могут быть разбросаны по большой сети. Необходима альтернативная организация, подобная той, что используется в мобильной телефонии, в которой имена (идентификаторы) не зависят от местоположения.

Наша третья и последняя тема будет касаться организации имен. В частности, имена, на которые больше не ссылается ни один объект и которые невоз-

можно более локализовать, чтобы получить к ним доступ, должны автоматически удаляться. Этот процесс также известен как уборка мусора, корни его следует искать в языках программирования. Однако при переходе к крупным распределенным системам автоматическая уборка объектов, на которые нет ссылок, становится особенно важной.

4.1. Именованные сущности

В этом разделе мы сначала сосредоточимся на различных типах имен и том, как они организуются в пространства имен. Далее обсудим важный вопрос о том, как разрешить имя, то есть каким образом получить доступ к сущности, на которую оно указывает. Также мы коснемся различных аспектов, связанных с распределением по нескольким машинам больших пространств имен и их реализацией. В качестве примеров больших служб именования мы рассмотрим систему доменных имен Интернета и стандарт OSI X.500.

4.1.1. Имена, идентификаторы и адреса

Имя в распределенной системе представляет собой строку битов, или символов, используемую для ссылки на сущность. Сущностью в распределенной системе является практически все. Типичными примерами являются ресурсы, включая хосты, принтеры, диски, файлы. Другие хорошо известные примеры сущностей, часто получающие имена, — это процессы, пользователи, почтовые ящики, группы новостей, web-страницы, графические окна, сообщения, сетевые соединения и т. д.

С сущностями можно работать. Например, ресурс вроде принтера предоставляет интерфейс, поддерживающий операции печати документа, запроса состояния печати и т. п. Кроме того, сущность, такая как сетевое соединение, может совершать операции по передаче и приему данных, установки параметров качества обслуживания, запроса состояния и т. д.

Чтобы работать с сущностью, необходимо иметь к ней доступ, для которого мы используем *точку доступа (access point)*. Точка доступа — это еще один специальный вид сущности в распределенных системах. Имя точки доступа называется *адресом (address)*. Адрес точки доступа сущности часто называют просто адресом сущности.

Сущность может иметь более чем одну точку доступа. Для сравнения, телефон может считаться точкой доступа к человеку, если телефонный номер рассматривать как адрес. Однако в наши дни множество людей имеет несколько телефонных номеров, каждый из которых соответствует одной из точек доступа, в которой этих людей можно застать. В распределенных системах типичным примером точки доступа является хост, на котором запущен некий сервер. Его адрес формируется сочетанием, например, IP-адреса и номера порта (то есть адресом сервера транспортного уровня).

Точка доступа сущности может с течением времени изменяться. Так, например, если мобильный компьютер перенести в другое место, ему, скорее всего,

будет присвоен не тот IP-адрес, который был у него раньше. Точно так же если человек переезжает в другой город или другую страну, ему обычно приходится менять телефон. Аналогично, смена работы или Интернет-провайдера влечет за собой изменение адреса электронной почты.

Адрес — это специальный тип имени, указывающий на точку доступа к сущности. Поскольку точка доступа тесно связана с сущностью, кажется удобным использовать адрес в качестве постоянного имени соответствующей сущности. Однако делать это можно не всегда.

В трактовке адресов в качестве имен особого типа имеется множество достоинств. Так, например, нет ничего необыкновенного в регулярных реорганизациях распределенных систем, в результате которых какой-либо сервер, например тот, который обрабатывает запросы FTP, оказывается на другом хосте. Та машина, на которой этот сервер работал раньше, может быть перенацелена на совершенно другую задачу, например резервное копирование локальных файловых систем. Другими словами, сущность может с легкостью поменять точку доступа, а точка доступа может быть перенацелена на другую сущность.

Если для ссылки на сущность использовать адрес, то после изменения точки входа или назначения ее другой сущности мы получим неверную ссылку. Для примера представьте себе, что FTP-служба организации известна только по адресу хоста, на котором запущен FTP-сервер. Как только сервер будет переведен на другой хост, FTP-служба окажется абсолютно недоступной до тех пор, пока новый адрес не станет известен всем пользователям этой службы. В этом случае было бы значительно лучше сделать так, чтобы FTP-служба имела собственное имя, никак не связанное с адресом соответствующего FTP-сервера.

Точно так же если сущность имеет более одной точки входа, непонятно, какой адрес использовать для ссылки. Например, как мы говорили в главе 1, множество организаций разносят свои web-службы по нескольким серверам. Если мы будем использовать адреса этих серверов для ссылок на web-службы, неясно, какой из адресов лучше выбрать. Значительно лучше использовать для web-службы одно имя, не связанное с адресами web-серверов.

Эти примеры иллюстрируют тот факт, что имя сущности, не связанное с ее адресами, часто значительно проще и удобнее. Такие имена мы называем *локально независимыми* (*local independent*).

Кроме адресов существуют и другие типы имен, требующие особого рассмотрения, например имена, используемые для однозначной идентификации сущности. *Правильный идентификатор* (*true identifier*) — это имя со следующими свойствами [491]:

- ◆ идентификатор ссылается не более чем на одну сущность;
- ◆ на каждую сущность ссылается не более одного идентификатора;
- ◆ идентификатор всегда ссылается на одну и ту же сущность (то есть не может быть использован повторно).

Использование идентификаторов значительно упрощает создание однозначных ссылок на сущность. Для примера рассмотрим два процесса, каждый из которых ссылается на некие сущности посредством идентификаторов. Для того

чтобы понять, что оба процесса ссылаются на одну и ту же сущность, достаточно сравнить на эквивалентность два идентификатора. При использовании обычных (не «идентификаторных») имен такого теста мало. Так, в качестве уникальной ссылки на конкретного человека указывать только его имя и фамилию, например «Джон Смит», явно недостаточно.

Точно так же если один и тот же адрес может быть присвоен другой сущности, мы не можем задействовать его в качестве идентификатора. Обсудим использование телефонных номеров, которые относительно постоянны в том смысле, что номер обычно относится к одному и тому же человеку или организации. Однако задействовать телефонный номер в качестве идентификатора нельзя, потому что он с течением времени может поменяться. Соответственно, новая булочная Боба может получить старый телефонный номер склада аппаратуры Алисы. В этом случае лучше использовать правильный идентификатор Алисы, а не ее телефонный номер.

Адреса и идентификаторы — два очень важных типа имен, каждый из которых предназначен для своих целей. Во многих компьютерных системах адреса и идентификаторы хранятся только в форме, удобной для использования машиной, то есть в форме строк битов. Так, адреса Ethernet — это, по сути, случайные 48-битные строки. Адреса памяти обычно представлены 32- или 64-битными строками.

Еще один важный тип имен — *имена, приспособленные для восприятия человеком (human-friendly names)*, или просто *удобные для восприятия*. В противоположность адресам и идентификаторам удобное для восприятия имя обычно представляется в виде строки символов. Эти имена принимают множество различных форм. Например, файлы в UNIX-системах имеют имена, состоящие из строк, длина которых может достигать до 255 символов и определяется только желанием пользователя. Аналогично, имена DNS представляются относительно простой строкой, символы в которой не зависят от регистра.

Пространства имен

Имена в распределенных системах организуются в некоторую сущность, которая носит название *пространства имен (name space)*. Пространство имен может быть представлено как направленный граф с двумя типами узлов. *Листовой узел (leaf node)*, или *лист*, представлен именованной сущностью и не имеет исходящих из него ребер. Листовой узел обычно содержит информацию о представляемой сущности — например, ее адрес, — к которой имеют доступ клиенты. С другой стороны, мы можем сохранить в листовом узле состояние этой сущности — как в случае файловой системы, когда листовой узел содержит файл, который он представляет. Позже мы еще вернемся к вопросу о содержимом узлов.

В противоположность листовому узлу, *направляющий узел (directory node)* имеет несколько исходящих из него ребер, каждое из которых именовано, как показано на рис. 4.1. Каждый узел в графе именования рассматривается как еще одна сущность распределенной системы и, следовательно, имеет отдельный идентификатор. Направляющий узел хранит таблицу, в которой все исходящие ребра представлены в виде пар (метка ребра, идентификатор узла). Эта таблица называется *направляющей таблицей (directory table)*.

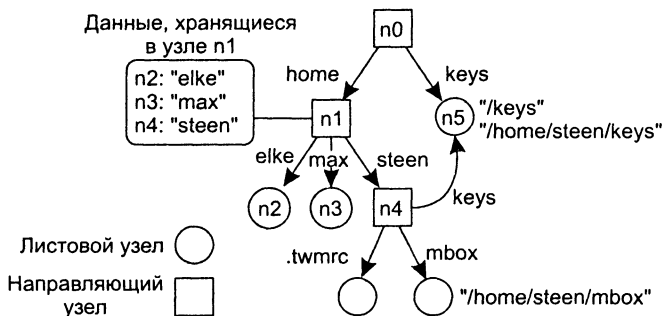


Рис. 4.1. Обобщенный граф именования с одним корневым узлом

Граф именования, показанный на рисунке, имеет один узел, а именно n0, у которого имеются только исходящие ребра. Входящих ребер у него нет. Такой узел называется *корневым узлом* (*root node*), или *корнем*, графа именования. Несмотря на то что граф именования может иметь несколько корневых узлов, для простоты многие системы именования имеют лишь один корень. Любой путь в графе именования должен быть представлен последовательностью меток, соответствующих ребрам графа, например:

N:<label-1, label-2, ..., label-n>

Здесь N соответствует первому из узлов пути. Эта последовательность называется *именем пути* (*path name*). Если первый узел пути — корень графа именования, это имя называется *абсолютным именем пути* (*absolute path name*). В противном случае оно называется *относительным именем пути* (*relative path name*).

Важно понимать, что имена всегда организуются в пространство имен. Вследствие этого имя всегда определяется только относительно направляющего узла. В этом смысле термин «абсолютное имя» — это, в некотором смысле, обман. Точно так же и разница между глобальными и локальными именами порой может ввести в заблуждение. *Глобальным именем* (*global name*) называется имя, которое обозначает одну и ту же сущность, вне зависимости от того, где в системе оно используется. Другими словами, глобальное имя всегда соответствует одному и тому же направляющему узлу. В противоположность ему, *локальное имя* (*local name*) — это имя, интерпретация которого зависит от того, где это имя используется. Иначе говоря, локальное имя — это лишь относительное имя, и должен быть известен (подразумеваться) направляющий узел, к которому оно относится. Мы вернемся к этим вопросам, когда будем обсуждать разрешение имен.

Это описание графа именования похоже на реализацию многих файловых систем. Однако вместо представления имени пути последовательностью именованных ребер, имена путей в файловых системах обычно представляются в виде единой строки, в которой метки разделяются специальным разделяющим символом, например косой чертой (/). Этот символ также используется и для того, чтобы показать, абсолютное это имя или нет. Например, вместо задания имени пути в виде n0:<home, steen, mbox> (см. рис. 4.1) обычно используется его строковое представление /home/steen/mbox. Отметим также, что если в один и тот же узел

приходят несколько путей, этот узел может соответствовать различным именам путей. Так, например, на узел `p5` можно сослаться в виде `/home/steen/keys` или просто в виде `/keys`. Строковое представление имени пути успешно применяется к графу именования не только в файловых системах. В Plan 9 все ресурсы, как то: процессы, хосты, устройства ввода-вывода и сетевые интерфейсы, именуются таким же образом, как файлы [353]. Такой подход соответствует реализации единого графа именования для всех ресурсов распределенной системы.

Существует множество различных способов организации пространства имен. Как мы говорили, большая часть пространств имен имеют лишь один корневой узел. Во многих случаях пространства имен имеют вдобавок жесткую иерархию, то есть граф именования организован в виде дерева. Это означает, что каждый узел, за исключением корневого, имеет лишь одно входящее в него ребро. Корневой узел входящих ребер не имеет. Соответственно, каждый узел имеет лишь одно соответствующее ему абсолютное имя.

Граф именования, изображенный на рис. 4.1, является примером *направленного ациклического графа* (*directed acyclic graph*). При подобной организации узел может иметь более одного входящего ребра, но граф не может иметь циклов. Существуют также пространства имен, не имеющие подобного ограничения.

Чтобы приблизить наши рассуждения к практике, рассмотрим способ именования файлов в стандартной файловой системе UNIX. В графе именования UNIX направляющий узел соответствует файловому каталогу, а листовой узел — файлу. Существует единственный корневой каталог, соответствующий в графе именования корневому узлу. Реализация графа именования является составной частью файловой системы. Эта реализация состоит из непрерывного набора блоков логического диска, обычно поделенных на загрузочный блок, суперблок, наборы индексных узлов и блоки файловых данных [117, 319, 450]. Эта структура приведена на рис. 4.2.

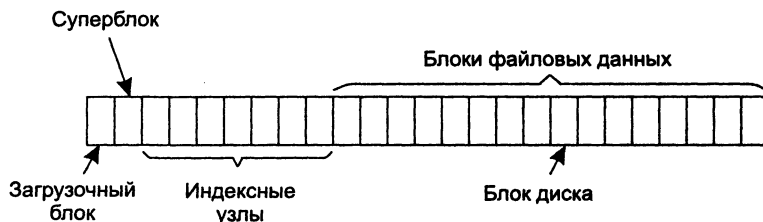


Рис. 4.2. Обобщенная организация реализации файловой системы UNIX на логическом диске в виде непрерывного набора дисковых блоков

Загрузочный блок — это специальный блок данных и инструкций, которые автоматически загружаются в оперативную память при загрузке системы. Загрузочный блок используется при загрузке оперативной системы в основную память.

Суперблок содержит информацию обо всей файловой системе — это размер незанятых дисковых блоков, неиспользованные индексные узлы и т. д. Индексные узлы нумеруются, начиная с нуля. Ноль зарезервирован для индексного узла, соответствующего корневному каталогу.

Каждый индексный узел содержит информацию о данных расположенного на диске соответствующего файла. Кроме того, индексный узел содержит информацию о своем владельце, времени создания и последней модификации, защите и подобных же вещах. Соответственно, имея номер индексного узла, можно получить доступ к соответствующему файлу. Каждый из каталогов также реализован в виде файла. Это относится и к корневому каталогу, который обеспечивает отображение между именами файлов и индексами индексных узлов. То есть индекс индексного узла соответствует идентификатору узла в графе именования.

4.1.2. Разрешение имен

Пространства имен предоставляют удобный способ сохранения и извлечения информации о сущностях по их именам. В общем виде, зная имя пути, можно извлечь всю информацию, которая хранится в узле, соответствующем этому имени. Процесс поиска информации называется *разрешением имени* (*name resolution*).

Чтобы понять, как происходит разрешение имени, рассмотрим путь $N: \langle \text{label-1}, \text{label-2}, \dots, \text{label-n} \rangle$. Разрешение этого имени начинается с узла N графа именования, при этом в направляющей таблице ищется имя *label-1* и возвращается идентификатор узла, на который указывает это имя. Разрешение продолжается поиском в направляющей таблице указанного узла имени *label-2* и т. д. Предполагая, что названный путь действительно существует, разрешение завершается при обнаружении последнего узла, соответствующего имени *label-n*, возвращением содержимого этого узла.

При поиске имен возвращается идентификатор узла, с которого продолжается процесс разрешения. В частности, необходимо, получить доступ к направляющей таблице найденного узла. Рассмотрим снова файловую систему UNIX. Как уже говорилось, идентификатором узла является индекс (номер) индексного узла. Доступ к направляющей таблице означает возможность сначала прочитать индексный узел, чтобы определить, где на диске находятся данные, а затем прочитать блоки данных.

Механизм свертывания

Разрешение имен может происходить только в том случае, если мы знаем, откуда и как начать. В нашем примере начальный узел задан, причем предполагается, что мы знаем, как получить доступ к его направляющей таблице. Знание того, откуда и как начинать разрешение имен, называется *механизмом свертывания* (*closure mechanism*). По существу, механизм свертывания относится к выбору начального узла пространства имен, с которого должно начинаться разрешение имени [367]. О функционировании механизмов свертывания иногда довольно трудно судить. Это происходит из-за их частично неявной реализации и значительного отличия друг от друга.

Так, например, разрешение имен в графе именования файловой системы UNIX предполагает знание того факта, что индексный узел корневого каталога является первым индексным узлом логического иска, на котором находится рассматриваемая файловая система. Его истинное смещение в байтах вычисляется из зна-

чений других полей суперблока, а также внутренней информации операционной системы об организации суперблока.

Чтобы прояснить этот момент, рассмотрим строковое представление имени файла `/home/steen/mbox`. Чтобы разрешить это имя, необходимо иметь доступ к направляющей таблице корневого узла соответствующего графа именования. Будучи корневым узлом, этот узел не может быть разрешен, если только он не реализован в виде другого узла иного графа именования, скажем `G`. Но в этом случае необходимо иметь прямой доступ к корневому узлу `G`. Из этого следует, что разрешение имени файла требует предварительной реализации некоторых механизмов, которые начнут процесс разрешения.

Абсолютно другой пример — использование строки `"0031204430784"`. Большинство людей не поймут, что делать с этими цифрами. Исключением будут лишь те, кому скажут, что эта цепочка цифр — номер телефона. Подобная информация необходима для того, чтобы начать разрешение имени, в частности, для набора номера. После этого телефонная система сделает то, чего от нее требуют.

В качестве последнего примера рассмотрим использование в распределенных системах глобальных и локальных имен. Типичный пример локального имени — переменная среды. Так, в UNIX-системах переменная с именем `HOME` используется для ссылки на домашний каталог пользователя. Каждый пользователь имеет свою копию этой переменной, инициализируемую глобальным общесистемным именем, соответствующим домашнему каталогу пользователя. Механизм свертывания, ассоциированный с переменными среды, гарантирует, что имя этой переменной будет правильно разрешено путем поиска в направляющей таблице пользователя.

Организация ссылок и монтирование

С разрешением имен связано использование *псевдонимов (aliases)*. Псевдоним — это другое имя той же сущности. Переменные среды — пример псевдонимов. В понятиях графа именования существует два основных способа реализации псевдонимов. Первый способ — просто предоставление нескольких абсолютных путей к каждому узлу графа именования. Подобный подход иллюстрирует рис. 4.1, на котором на узел `n5` можно сослаться с использованием двух различных путей. В терминологии UNIX оба пути, `/keys` и `/home/steen/keys`, называются *жесткими ссылками (hard links)* к узлу `n5`.

Другой подход состоит в том, чтобы представить сущность в виде листового узла, скажем `N`, но вместо сохранения в нем адреса или состояния этой сущности, сохранить в нем ее абсолютный путь. Когда первое разрешение абсолютного пути приведет к узлу `N`, разрешение имени вернет путь, сохраненный в `N`, после чего мы продолжим разрешение нового пути. Этот принцип соответствует использованию *символических ссылок (symbolic links)* в файловых системах UNIX и его иллюстрирует рис. 4.3. В этом случае имя пути `/home/steen/keys`, ссылающееся на узел, содержащий абсолютное имя пути `/keys`, является символической ссылкой на узел `n5`.

Описанное разрешение имени полностью относится к одиночным пространствам имен. Однако разрешение имен может также использоваться и при про-

зрачном слиянии нескольких пространств имен. Давайте сначала рассмотрим монтируемые файловые системы. В понятиях нашей модели именования монтируемая файловая система — это система, в которой направляющий узел хранит идентификатор направляющего узла *другого* пространства имен, называемого внешним пространством имен. Направляющий узел, содержащий такой идентификатор узла, называется *монтажной точкой* (*mount point*). Соответственно, направляющий узел во внешнем пространстве имен называется *точкой монтирования* (*mounting point*). Обычно точка монтирования является корнем пространства имен. В ходе разрешения имен отыскивается точка монтирования, и процесс разрешения продолжается поиском в ее направляющей таблице.

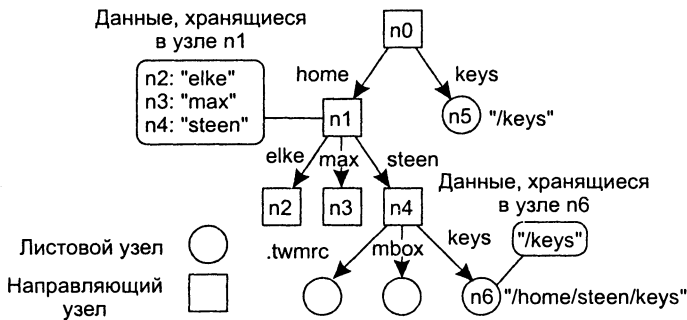


Рис. 4.3. Концепция символической ссылки в графе именования

Принципы монтирования могут быть обобщены и на другие пространства имен. В частности, все, что необходимо для использования направляющего узла в качестве монтажной точки, — это чтобы в ней хранилась вся информация, необходимая для идентификации монтирующей точки чужого пространства имен и доступа к ней. Этот подход соблюдается в системе именования Jade [371], ему также следуют многие распределенные файловые системы.

Рассмотрим набор пространств имен, распределенных по нескольким машинам. В частности, пусть каждое из пространств имен реализовано на своем сервере, которые, возможно, работают на разных машинах. Соответственно, если мы хотим смонтировать внешнее пространство имен *NS2* с пространством имен *NS1*, нужно (если серверы *NS2* и *NS1* работают на разных машинах) организовать связь по сети между этими серверами. Чтобы смонтировать внешнее пространство имен в распределенной системе, необходима как минимум следующая информация:

- ◆ имя протокола доступа;
- ◆ имя сервера;
- ◆ имя монтирующей точки во внешнем пространстве имен.

Отметим, что каждое из этих имен должно разрешаться. Имя протокола доступа должно разрешаться реализацией этого протокола, при помощи которой организуется связь с сервером внешнего пространства имен. Имя сервера должно разрешаться посредством адреса сервера. И наконец, имя монтирующей точки должно разрешаться посредством идентификатора узла во внешнем пространстве имен.

В нераспределенных системах любая из этих трех точек может оказаться невостребованной. Так, в системе UNIX нет ни протоколов доступа, ни серверов. Имя монтирующей точки также не является необходимым, поскольку это просто корневой каталог внешнего пространства имен.

Имя монтирующей точки может быть разрешено сервером внешнего пространства имен. Однако мы также нуждаемся в пространствах имен и реализациях для протокола доступа и имени сервера. Одна из возможностей предоставить все эти три имени — это URL.

Чтобы сделать разговор более конкретным, рассмотрим ситуацию, когда пользователь с портативным компьютером хочет получить доступ к удаленному файловому серверу. И машина клиента, и файловый сервер сконфигурированы под *сетевую файловую систему (Network File System, NFS)* фирмы Sun, которую мы рассмотрим в подробностях в главе 10. NFS — это распределенная файловая система, в которую включен протокол, детально описывающий, как клиент может получить доступ к файлу, хранящемуся на (удаленном) файловом сервере NFS. В частности, чтобы система NFS работала через Интернет, клиент должен указать, к какому файлу он хочет получить доступ посредством URL, например, `nfs://flits.cs.vu.nl//home/steen`. Этот URL-адрес именуется файл (который может быть и каталогом) с именем `/home/steen` на файловом сервере NFS `flits.cs.vu.nl`, доступ к которому возможен с использованием протокола NFS [80].

Имя `nfs` широко известно, в том смысле, что существует глобальное соглашение о его интерпретации. Другими словами, при разборе того, что мы использовали в качестве URL, имя `nfs` разрешается реализацией протокола NFS. Имя сервера разрешается его адресом DNS с помощью системы DNS, которую мы обсудим в следующем пункте. Как мы уже говорили, имя `/home/steen` разрешается сервером внешнего пространства имен.

Организация файловой системы на клиентской машине частично приведена на рис. 4.4. Корневой каталог имеет несколько определенных пользователем сущностей, включая вложенный каталог под названием `/remote`. Этот вложенный каталог предназначен для того, чтобы хранить монтажную точку внешних пространств имен, таких как домашний каталог пользователя университета Vrije. С этой стороны узел каталога с именем `/remote/vu` используется для хранения URL `nfs://flits.cs.vu.nl//home/steen`.

Теперь рассмотрим имя `/remote/vu/mbox`. Это имя разрешается, начиная с корневого каталога клиентской машины, и продолжает разрешаться там, пока мы не достигнем узла `/remote/vu`. Затем разрешение имени продолжится возвращением URL `nfs://flits.cs.vu.nl//home/steen`, что, в свою очередь, заставляет клиентскую машину установить связь с файловым сервером `flits.cs.vu.nl` по протоколу NFS и получить доступ в каталог `/home/steen`. Затем разрешение имени продолжается чтением файла `mbox` в этом каталоге.

Распределенные системы позволяют монтировать удаленные файловые системы так, как это было продемонстрировано, разрешая клиентской машине, к примеру, выполнить следующие команды:

```
cd /remote/vu
ls -l
```

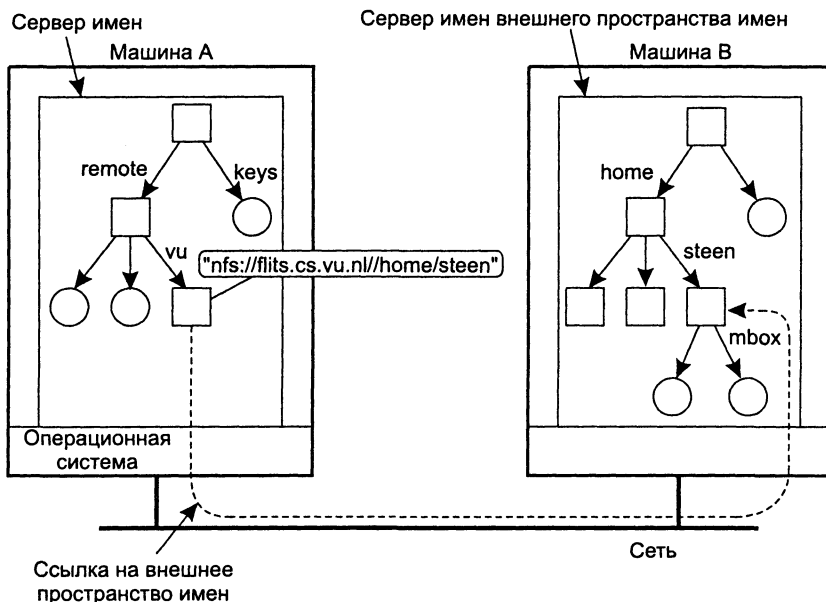


Рис. 4.4. Монтирование удаленного пространства имен посредством особого протокола доступа

Эти команды создают список файлов в каталоге `/home/steen` удаленного файлового сервера. Прелесть всего этого в том, что пользователь освобождается от деталей доступа к удаленному серверу. В идеале он заметит разве что некоторую потерю производительности по сравнению с доступом к локальным файлам. Фактически пользователь будет считать, что пространства имен, расположенные на локальной машине и в каталоге `/home/steen` на удаленной машине, образуют единое пространство имен.

Монтирование — это лишь один из способов объединения различных пространств имен. Другой подход, применяемый в *глобальной службе имен (Global Name Service, GNS)* компании DEC, состоит в том, чтобы создать новый корневой узел, а существующие корневые узлы сделать дочерними от него [253]. Этот принцип, который иллюстрирует рис. 4.5, рассмотрен ниже.

Проблема подобного подхода состоит в том, что созданные имена иногда придется изменять. Например, абсолютный путь `/home/steen` в пространстве имен *NS1* изменен на относительный путь, который разрешается, начиная с узла `n0`, и соответствует абсолютному пути `/vu/home/steen`. Чтобы решить эту проблему и позволить в будущем добавлять и другие пространства имен, имена в GNS всегда включают (неявно) идентификатор узла, с которого должно начинаться разрешение. Так, например, в пространстве имен *NS1* на рисунке имя `/home/steen/keys` всегда расширяется для включения идентификатора узла `n0`, превращаясь в имя `n0:/home/steen/keys`. Расширение обычно скрыто от пользователей. Оно означает, что идентификатор узла абсолютно уникален. Соответственно, узлы из различных пространств имен будут всегда иметь разные идентификаторы.

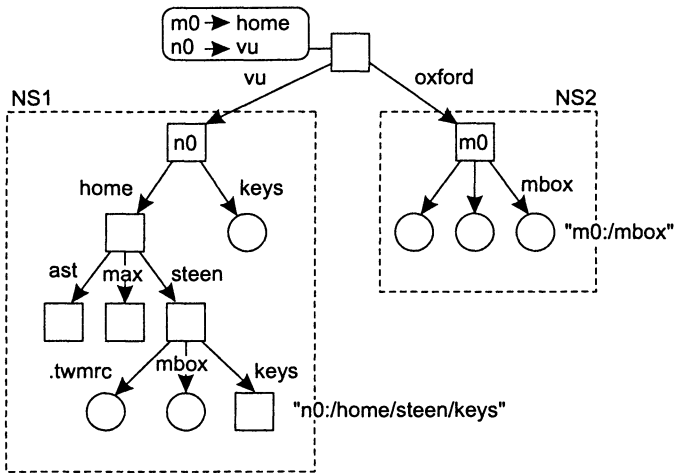


Рис. 4.5. Организация глобальной службы имен компании DEC

Объединение двух пространств имен *NS1* и *NS2* без изменения существующих имен происходит так, как показано на рисунке. При добавлении нового корневого узла этот узел получает на хранение таблицу, отображающую идентификатор корневого узла *NS1* на имя, под которым этот корень будет известен в новом пространстве адресов. То же самое происходит и с корневым узлом *NS2*. Разрешение имен всегда начинается с корня нового адресного пространства, имя `n0:/home/steen` после обнаружения идентификатора узла `n0` в таблице корневого узла сначала преобразуется в имя `/vu/home/steen`.

Потенциальная проблема GNS состоит в том, что корневой узел объединенного пространства имен требует поддерживать отображение идентификаторов старых корневых узлов в их новые имена. Если объединить тысячи пространств имен, этот подход может вызвать проблемы производительности.

4.1.3. Реализация пространств имен

Пространство имен формирует сердце службы именования, службы, которая позволяет пользователям и процессам добавлять, удалять и находить имена. Служба именования реализуется при помощи серверов имен. Если распределенная система сокращена до размеров локальной сети, она вполне в состоянии реализовать службу именования при помощи всего лишь одного сервера имен. Однако в крупных распределенных системах с множеством сущностей, возможно, разнесенных по множеству географических зон, необходимо разнести реализацию пространства имен по нескольким серверам имен.

Распределение пространства имен

Пространства имен крупномасштабных, возможно, глобальных распределенных систем обычно организованы иерархически. Как и ранее, мы предполагаем, что

пространство имен имеет только один корневой узел. Для эффективной реализации этого пространства имен удобно разбить его на логические уровни. В [98] выделяют три уровня:

- ✦ глобальный уровень;
- ✦ административный уровень;
- ✦ управленческий уровень.

Глобальный уровень (global layer) формируется узлами верхнего уровня, то есть корневыми узлами и другими направляющими узлами, которые логически связаны с корневыми, то есть их дочерними узлами. Узлы глобального уровня обычно характеризуются своей стабильностью, в том смысле, что их направляющие таблицы изменяются редко. Эти узлы могут представлять организации или группы организаций, имена которых хранятся в пространстве имен.

Административный уровень (administrational layer) формируется из направляющих узлов, которые вместе представляют одну организацию. Характерной чертой направляющих узлов административного уровня является то, что они представляют группы сущностей, относящихся к одной и той же организации или административной единице. Например, это может быть направляющий узел всех отделений организации или направляющий узел всех хостов. Другой направляющий узел может использоваться в качестве исходной точки для именования всех пользователей и т. д. Узлы административного уровня относительно стабильны, хотя изменения в них вносятся в основном чаще, чем в узлы глобального уровня.

И, наконец, *управленческий уровень (managerial layer)* состоит из узлов, которые обычно регулярно изменяются. Например, в этот уровень входят узлы, представляющие хосты локальной сети. По той же причине этот уровень включает узлы, предоставляющие совместно используемые файлы, такие как библиотеки или объектный код. Другим важным классом узлов управленческого уровня являются те, которые представляют каталоги и файлы пользователей. В противоположность глобальному и административному уровням, узлы управленческого уровня обслуживаются не только системными администраторами, но и отдельными конечными пользователями распределенных систем.

Чтобы конкретизировать предмет обсуждения, рассмотрим пример разбиения части пространства имен DNS, включающего имена файлов организации, к которым можно получить доступ через Интернет (например, web-страниц и файлов, предназначенных для загрузки). Этот пример иллюстрирует рис. 4.6. Пространство имен разделено на неперекрывающиеся части, которые [301] в DNS называются *зонами (zones)*. Зона — это часть пространства имен, реализованная отдельным сервером имен (некоторые из зон показаны на рисунке).

Что касается доступности и производительности, серверы имен каждого уровня имеют к ним различные требования. Высокая доступность особенно важна для серверов имен глобального уровня. Если происходит сбой сервера имен, большая часть пространства имен оказывается недоступной, поскольку разрешение имен не может «перескочить через голову» иерархии имен на оконечные серверы.

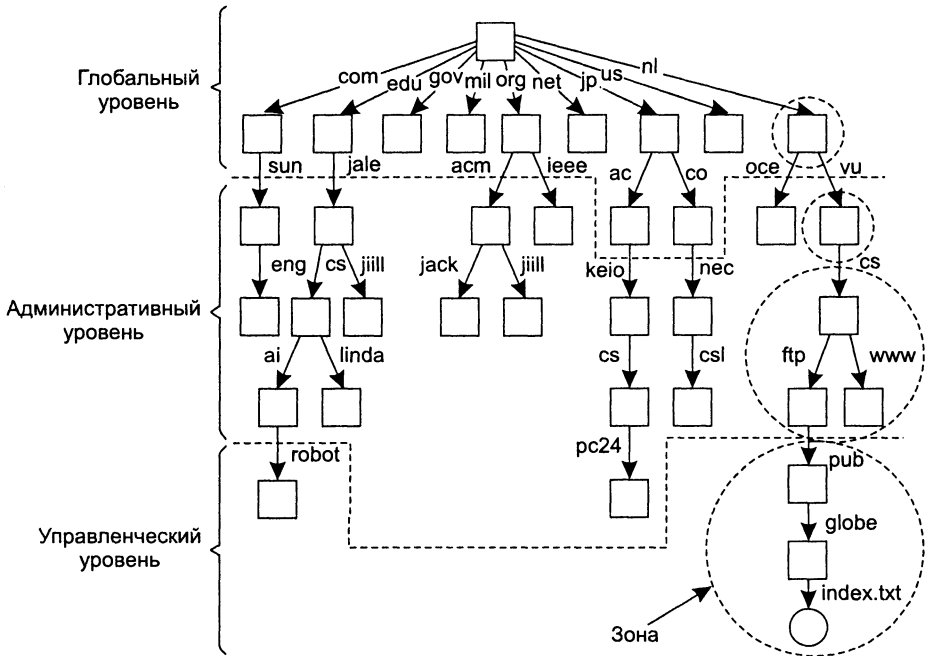


Рис. 4.6. Пример разбиения пространства имен DNS на три уровня

Требования к производительности не столь жесткие. Поскольку узлы глобального уровня изменяются медленно, результаты операции поиска будут верны в течение длительного времени. Соответственно, эти результаты могут быть эффективно подвергнуты кэшированию (то есть локально сохранены) на клиенте. При осуществлении той же операции поиска в следующий раз результат может быть извлечен из кэша клиента без обращения к серверам имен. В результате серверы имен глобального уровня не должны часто отвечать на одинаковые поисковые запросы. С другой стороны, производительность может быть важным фактором, особенно в крупных системах с миллионами пользователей.

Требования по доступности и производительности серверов имен глобального уровня должны удовлетворяться репликацией серверов в комбинации с кэшированием на стороне клиента. Как мы выясним в главе 6, изменения, вносимые на этом уровне, не обязаны вступать в силу немедленно, что значительно облегчает поддержание непротиворечивости реплик.

Доступность серверов имен административного уровня особенно важна для клиентов, относящихся к той же организации, которую обслуживает данный сервер имен. Если с сервером имен происходит сбой, множество ресурсов организации оказываются недоступными, поскольку их просто невозможно найти. С другой стороны, для пользователей вне этой организации временная недоступность ее ресурсов не столь важна.

Что касается производительности, серверы имен административного уровня имеют параметры, сходные с серверами глобального уровня. Поскольку измене-

ния узлов происходят не слишком часто, кэширование результатов поиска может быть весьма эффективным, а это делает производительность менее критичной. Однако в отличие от глобального уровня административный уровень должен обеспечить возвращение результатов поиска в течение нескольких миллисекунд, независимо от их источника (непосредственно с сервера или из локального кэша клиента). Изменения также должны вноситься быстрее, чем на глобальном уровне. Так, например, недопустимо, чтобы от создания до активизации нового пользователя проходило несколько часов.

Эти требования обычно подразумевают использование в качестве серверов имен высокопроизводительных машин. Кроме того, может применяться и кэширование на стороне клиента в сочетании с репликацией серверов для повышения общей доступности.

Требования к доступности серверов имен управленческого уровня обычно менее серьезны. В частности, для работы сервера имен нередко достаточно одной машины (выделенной), даже с риском временной недоступности. Однако критична производительность. Пользователи требуют, чтобы операции осуществлялись немедленно. Поскольку изменения вносятся постоянно, кэширование на клиенте обычно малоэффективно, если только не применять специальных мер, которые будут обсуждаться в главе 6.

Сравнение серверов имен различных уровней приведено в табл. 4.1. В распределенных системах серверы имен глобального и административного уровней реализовывать нелегко. Трудности связаны с репликацией и кэшированием, которые необходимы для достижения нужной производительности и доступности, но создают проблемы с непротиворечивостью. Некоторые из этих проблем усугубляются тем фактом, что кэши и реплики разнесены по глобальной сети, которая вносит длительные задержки связи и значительно затрудняет синхронизацию. Репликация и кэширование детально обсуждаются в главе 6.

Таблица 4.1. Сравнение серверов имен при реализации узлов в большом пространстве имен

Элемент	Глобальный уровень	Административный уровень	Управленческий уровень
Географический масштаб сети	Всемирная	Организация	Отдел
Общее число узлов	Несколько	Много	Огромное число
Время поиска	Секунды	Миллисекунды	Немедленно
Распространение изменений	Медленное	Немедленное	Немедленное
Число реплик	Множество	Мало или нет	Нет
Кэширование на клиенте	Да	Да	Иногда

Реализация разрешения имен

Распределенность пространства имен по множеству серверов имен затрудняет реализацию разрешения имен. Чтобы пояснить реализацию разрешения имен в крупных службах имен, представим себе, что серверы не реплицируются и кэширование на стороне клиента не используется. Каждый клиент имеет доступ к ло-

кальной *процедуре разрешения имен (name resolver)*, которая и отвечает за этот процесс. В соответствии с рис. 4.6 предположим, что разрешается (абсолютный) путь:

root:<nl, vu, cs, ftp, pub, globe, index.txt>

Если использовать форму записи URL, этот путь соответствует имени ftp://ftp.cs.vu.nl/pub/globe/index.txt. Теперь у нас есть два способа реализации разрешения имен.

При *итеративном разрешении имени (iterative name resolution)* процедура разрешения имен передает полное имя корневому серверу имен. Предполагается, что адрес корневого сервера, с которым контактирует процедура разрешения имен, общеизвестен. Корневой сервер разрешит ту часть пути, которую сможет, и вернет результат клиенту. В нашем примере корневой сервер может разрешить только метку nl, для которой он и вернет адрес ассоциированного с ней сервера имен.

После этого клиент передаст этому серверу имен оставшуюся часть пути (то есть имя nl:<vu, cs, ftp, pub, globe, index.txt>). Сервер сможет разрешить только метку vu и вернет адрес ассоциированного с этой меткой сервера имен вместе с оставшейся частью пути, vu:<cs, ftp, pub, globe, index.txt>.

Процедура разрешения имен клиента свяжется со следующим сервером имен, который сможет разрешить метку cs, а также ftp и вернет адрес FTP-сервера вместе с путем ftp:<pub, globe, index.txt>. После этого клиент свяжется с FTP-сервером и потребует от него разрешить остаток исходного пути. FTP-сервер разрешит последовательно метки pub, globe и index.txt и передаст запрошенный файл (в данном случае по протоколу FTP). Подобный процесс итеративного разрешения иллюстрирует рис. 4.7 (запись #<cs> используется для указания адреса сервера, отвечающего за обработку метки <cs>).

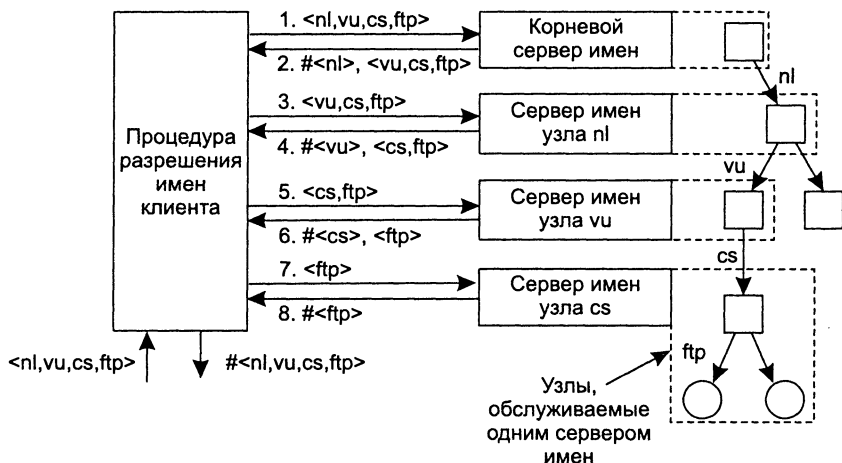


Рис. 4.7. Принцип итеративного разрешения имен

На практике последний шаг, а именно связь с FTP-сервером и запрос на передачу файла с путем ftp:<pub, globe, index.txt>, выполняется клиентским процес-

сом отдельно от всего остального. Другими словами, клиент обычным образом обрабатывает только путь `root:<nl, vu, cs, ftp>`, из которого он извлекает адрес, по которому находится FTP-сервер, как и показано на рисунке.

Альтернативой итеративному разрешению имен является использование в ходе разрешения рекурсии. Вместо того чтобы возвращать процедуре разрешения имен клиента промежуточные результаты, при *рекурсивном разрешении имени* (*recursive name resolution*) сервер имен передает эти результаты следующему обнаруженному серверу имен. Рассмотрим, например, что происходит, когда корневой сервер обнаруживает адрес сервера имен, реализованного на узле с именем `nl`. Он требует от сервера имен разрешить путь `nl:<vu, cs, ftp, pub, globe, index.txt>`. Используя и далее рекурсивное разрешение имен, этот следующий сервер разрешит путь целиком и вернет файл `index.txt` корневому серверу, который, в свою очередь, передаст его процедуре разрешения имен клиента.

Рекурсивное разрешение имен иллюстрирует рис. 4.8. Как и при итеративном разрешении имен, последний шаг разрешения, а именно контакт с FTP-сервером и запрос на передачу файла, происходят на клиенте в виде отдельного процесса.

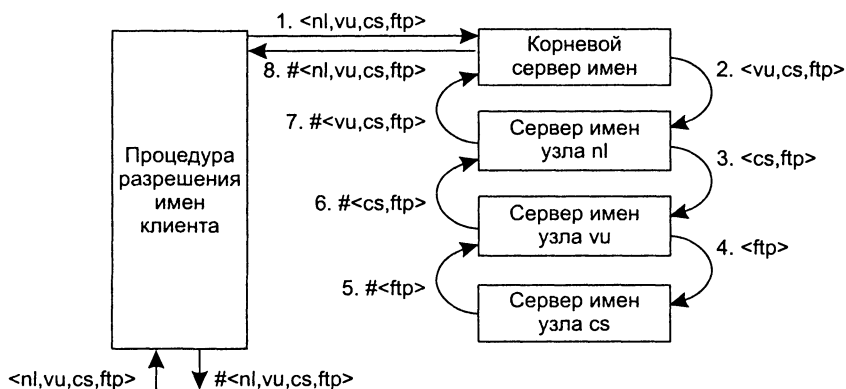


Рис. 4.8. Принцип рекурсивного разрешения имен

Основной недостаток рекурсивного разрешения имен состоит в том, что к производительности каждого из серверов имен предъявляются повышенные требования. Сервер имен должен быть в состоянии осуществить полное разрешение пути, хотя он мог бы делать это и в кооперации с другими серверами имен. Эта дополнительная нагрузка обычно столь высока, что серверы имен глобального уровня поддерживают только итеративное разрешение имен.

У рекурсивного разрешения имен имеется два важных достоинства. Первое из них состоит в том, что кэширование результатов по сравнению с итеративным разрешением имен более эффективно. Второе — это снижение затрат на взаимодействие. Эти преимущества объясняются тем, что процедура разрешения имен клиента получает только пути к узлам глобального или административного уровня. Для разрешения частей пути, относящихся к управленческому уровню, клиент отдельно связывается с сервером имен, адрес которого был возвращен ему процедурой разрешения имен. Мы говорили об этом выше.

Рекурсивное разрешение имен позволяет каждому серверу имен последовательно получать адрес каждого из серверов, ответственных за узлы нижнего уровня. В результате можно с успехом применить кэширование для повышения производительности. Так, если корневой сервер требует разрешить путь `root:<nl, vu, cs, ftp>`, он может в результате получить адрес сервера имен, реализующего узел, соответствующий этому пути. Чтобы прийти к этому, сервер имен узла `nl` должен найти адрес сервера имен узла `vu`, а последний, в свою очередь, должен найти адрес сервера имен, отвечающего за узел `cs`.

Поскольку изменения в узлах глобального и административного уровней происходят нечасто, корневой сервер имен может успешно кэшировать полученный адрес. Более того, поскольку в результате рекурсии также будут возвращены адреса серверов имен, отвечающих за реализацию узлов `vu` и `nl`, они также могут быть с успехом кэшированы на этих серверах.

Таким образом, результаты промежуточного поиска имен также могут быть кэшированы и извлечены из кэша. Например, сервер узла `nl` может найти адрес сервера узла `vu`. Этот адрес может быть возвращен корневому серверу в ходе возвращения сервером `nl` результатов поиска по исходному имени. Полный обзор процесса разрешения результатов, которые могут быть кэшированы на каждой из стадий этого процесса, приведен в табл. 4.2.

Таблица 4.2. Рекурсивное разрешение имени <nl, vu, cs, ftp>

Сервер узла	Объект разрешения	Поиск	Передача дочернему узлу	Получение и кэширование	Возвращение в ответ на запрос
cs	<ftp>	#<ftp>	—	—	#<ftp>
vu	<cs,ftp>	#<cs>	<ftp>	#<ftp>	#<cs> #<cs, ftp>
nl	<vu,cs,ftp>	#<vu>	<cs,ftp>	#<cs> #<cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>
root	<nl,vu,cs,ftp>	#<nl>	<vu,cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>	#<nl> #<nl,vu> #<nl,vu,cs> #<nl,vu,cs,ftp>

Достоинство подобного подхода в том, что операции поиска со временем могут стать потрясающе эффективными. Так, представим себе, что позже разрешить путь `root:<nl, vu, cs, flits>` потребует другой клиент. Это имя поступит в корневой узел, а оттуда немедленно будет передано серверу имен узла `cs`, к которому и поступит запрос на разрешение остатка пути `cs:<flits>`.

При итеративном разрешении имен кэширование по необходимости ограничивается процедурой разрешения имен клиента. Соответственно, если клиент *A* запрашивает разрешение некоторого имени, а затем клиент *B* требует разрешить то же самое имя, разрешение имен потребует повторного прохода через те же самые серверы, через которые мы уже проходили для клиента *A*. В качестве компромиса многие организации задействуют локальные промежуточные серверы

имен, совместно используемые всеми клиентами. Эти локальные серверы имен обрабатывают все запросы на имена и кэшируют их результаты. Такие промежуточные серверы удобны и с точки зрения управления. Например, только этот сервер должен знать, где находится корневой сервер имен, другие машины в этой информации не нуждаются.

Второе достоинство рекурсивного разрешения имен состоит в том, что оно часто экономней с точки зрения взаимодействия. Рассмотрим вновь разрешение пути `root:<nl, vu, cs, ftp>` в предположении, что клиент находится в Сан-Франциско. Предполагая, что клиент знает адрес сервера узла `nl`, при рекурсивном разрешении имен образуется связь между хостом клиента в Сан-Франциско и сервером в Нидерландах, помеченная на рис. 4.9 как *R1*. После этого вам потребуется связь между сервером `nl` и сервером имен университета *Vrije* в университетском городке в Амстердаме, Нидерланды. Эта связь обозначена как *R2*. И наконец, необходима связь между сервером `vu` и сервером имен факультета компьютерных дисциплин (Computer Science Department), обозначенная как *R3*. Ответ пройдет по тому же пути, но в обратном направлении. Понятно, что затраты на взаимодействие будут определяться обменом сообщениями между хостом клиента и сервером `nl`.

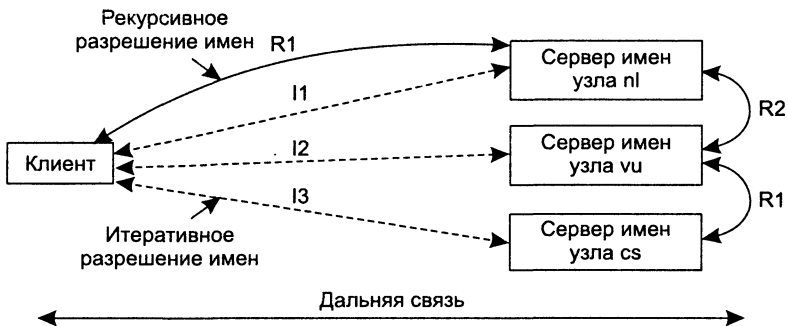


Рис. 4.9. Сравнение между рекурсивным и итеративным разрешением имен с точки зрения затрат на взаимодействие

В противоположность этому, при итеративном разрешении имен хост клиента связывается по отдельности с серверами `nl`, `vu` и `cs`, в результате чего общие затраты будут почти в три раза выше, чем при рекурсивном разрешении. Стрелки на рисунке с метками *I1*, *I2* и *I3* обозначают взаимодействие при итеративном разрешении имен.

4.1.4. Пример — система доменных имен

Одной из самых больших на сегодня распределенных служб именованья является система доменных имен (Domain Name System, DNS) Интернета. DNS используется в первую очередь для поиска адресов хостов и почтовых серверов. На следующих страницах мы сосредоточимся на организации пространства имен системы DNS и информации, хранящейся на ее узлах. Кроме того, мы поближе

ознакомимся с существующей реализацией DNS. Дополнительную информацию по этой теме можно найти в [8, 301].

Пространство имен DNS

Пространство имен DNS иерархически организовано в виде дерева с корнем. Метка представляет собой не зависящую от регистра строку алфавитно-цифровых символов. Максимальная длина метки — 63 символа, полная длина пути ограничена 255 символами. Строковое представление пути состоит из списка меток, начиная справа, разделенных точками. Корень также представлен точкой. Так, например, путь `root:<nl, vu, cs, flits>` представляется строкой `flits.cs.vu.nl.`, включая точку справа, которая означает корень. Обычно для удобства чтения мы будем опускать эту точку.

Поскольку каждый из узлов в пространстве имен DNS имеет ровно одно входящее ребро (за исключением корня, который входящих ребер не имеет), метка входящего в узел ребра используется также и в качестве имени этого узла. Поддерево называется *доменом* (*domain*), а путь к корневому узлу домена — *доменным именем* (*domain name*). Отметим, что, как и путь, доменное имя может быть абсолютным или относительным.

Содержимое узла комплектуется из набора *записей о ресурсах* (*resource records*). Существуют различные типы записей о ресурсах, наиболее важные из них перечислены в табл. 4.3.

Таблица 4.3. Наиболее важные типы записей о ресурсах

Тип записи	Сущность записи	Описание
SOA	Зона	Информация о соответствующей зоне
A	Хост	IP-адрес хоста, на котором установлен узел
MX	Домен	Почтовый сервер, обрабатывающий почтовые адреса узла
SRV	Домен	Сервер, предоставляющий некоторую службу
NS	Зона	Сервер имен, отвечающий за определенную зону
CNAME	Узел	Символическая ссылка на первичное имя соответствующего узла
PTR	Хост	Каноническое имя хоста
HINFO	Хост	Информация о хосте, на котором установлен узел
TXT	Любая сущность	Дополнительная информация, зависящая от сущности

Узел в пространстве имен DNS часто представляет одновременно несколько сущностей. Так, например, доменное имя `vu.nl` используется для представления домена и зоны. В данном случае домен реализован в нескольких зонах.

Запись о ресурсе SOA (*start of authority* — начало полномочий) содержит такую информацию, как, например, почтовый адрес системного администратора, отвечающего за указанную зону, имя хоста, на котором находятся данные о зоне и т. д.

Запись A (address — адрес) представляет отдельный хост в Интернете. Эта запись содержит IP-адрес хоста, используемый при взаимодействии. Если хост имеет несколько IP-адресов, например, в случае машин с множественной адресацией, узел будет содержать запись A для каждого адреса.

Важный тип записи о ресурсах — MX (mail exchange — почтовый обмен). Эта запись является символической ссылкой на узел, представляющий почтовый сервер. Например, пусть узел, представляющий домен `cs.vu.nl`, имеет запись MX, содержащую имя `zephyr.cs.vu.nl`, которое относится к почтовому серверу. Этот сервер будет обрабатывать все адреса для входящих почтовых сообщений пользователей домена `cs.vu.nl`. Узел может иметь несколько записей MX.

Записи SRV похожи на записи MX. Такая запись содержит имя сервера конкретной службы. Записи SRV определены в [477]. Конкретная служба определяется по ее имени вместе с именем протокола. Так, web-сервер домена `cs.vu.nl` может быть поименован при помощи такой записи SRV, как `http.tcp.cs.vu.nl`. Эта запись может ссылаться затем и на реальное имя сервера (`soling.cs.vu.nl`).

Узлы, представляющие зону, содержат одну или более записей NS (name server — сервер имен). Как и запись MX, запись NS содержит имя сервера имен, реализующего зону, представленную узлом. В принципе каждый узел пространства имен может иметь запись NS, ссылающуюся на сервер имен, который его реализует. Однако, как мы ранее говорили, реализация пространства имен DNS такова, что необходимость содержать записи NS существует только для узлов, представляющих зону.

DNS выделяет псевдонимы для того, что называется *каноническими именами* (*canonical names*). Предполагается, что каждый хост имеет каноническое, или первичное, имя. Псевдоним реализуется посредством узла, хранящего запись CNAME (canonical name — каноническое имя), которая содержит каноническое имя хоста. Имя узла, на котором хранится эта запись, совпадает с именем символической ссылки, как показано на рис. 4.3.

DNS поддерживает обратное отображение IP-адресов на имена хостов посредством записей PTR (pointer — указатель). Для того чтобы выполнить поиск среди имен хостов, когда известен только IP-адрес, DNS поддерживает домен `in-addr.arpa`, который содержит узлы, представляющие хосты Интернета и перечисленные в соответствии с IP-адресами этих хостов. Так, например, для хоста `www.cs.vu.nl`, имеющего IP-адрес `130.37.24.11`, DNS создает узел с именем `11.24.37.130.in-addr.arpa`, который используется для хранения в записи PTR канонического имени этого хоста (а именно, `soling.cs.vu.nl`).

Последние два типа записей — HINFO и TXT. Запись HINFO (host info — информация о хосте) используется для хранения дополнительной информации о хосте, например о типе машины и операционной системе. Также и записи TXT (text — текст) используются для хранения любой дополнительной информации, которую пожелает сохранить пользователь о представляемых узлом сущностях.

Реализация DNS

Реализация DNS очень похожа на то описание, которое мы привели в предыдущем пункте. В частности, пространство имен DNS можно разделить на глобаль-

ный и административный уровни именно так, как это было показано на рис. 4.6. Управленческий уровень, обычно складывающийся из локальных файловых систем, формально не является частью системы DNS и не контролируется ею.

Каждая зона реализована в виде сервера имен. Для лучшей доступности эти серверы постоянно подвергаются виртуальной репликации. Изменения в зонах, которые обычно обрабатываются первичным сервером имен, вносятся путем модификации баз данных DNS локально. Вторичные серверы имен не имеют прямого доступа к базам данных, они посылают запросы первичному серверу, который в ответ передает им свое содержимое. Этот процесс в терминологии DNS называется *переносом зон (zone transfer)*.

База данных DNS реализуется в виде набора (небольшого) файлов, среди которых наиболее важный содержит все записи о ресурсах со всех узлов соответствующей зоны. Такой подход позволяет легко идентифицировать узлы по их доменным именам, в результате понятие идентификатора узла сводится (неявно) к индексу файла.

Чтобы лучше понять эту особенность реализации, в табл. 4.4 приведена часть файла, в котором находится основная информация о домене cs.vu.nl (отметим, что для лучшего понимания материала файл был изменен). Файл демонстрирует содержимое восьми различных узлов, представляющих собой часть домена cs.vu.nl, при этом каждый узел идентифицируется собственным доменным именем.

Таблица 4.4. Выдержка из базы данных DNS для зоны cs.vu.nl

Имя	Тип записи	Значение записи
cs.vu.nl	SOA	star(1999121502,7200,3600,2419200,86400)
cs.vu.nl	NS	star.cs.vu.nl
cs.vu.nl	NS	top.cs.vu.nl
cs.vu.nl	NS	solo.cs.vu.nl
cs.vu.nl	TXT	"Vrije Universiteit - Math. & Comp. Sc."
cs.vu.nl	MX	1 zephyr.cs.vu.nl
cs.vu.nl	MX	2 tornado.cs.vu.nl
cs.vu.nl	MX	3 star.cs.vu.nl
star.cs.vu.nl	HINFO	Sun Unix
star.cs.vu.nl	MX	1 star.cs.vu.nl
star.cs.vu.nl	MX	10 zephyr.cs.vu.nl
star.cs.vu.nl	A	130.37.24.6
star.cs.vu.nl	A	192.31.231.42
zephyr.cs.vu.nl	HINFO	Sun Unix
zephyr.cs.vu.nl	MX	1 zephyr.cs.vu.nl
zephyr.cs.vu.nl	MX	2 tornado.cs.vu.nl
zephyr.cs.vu.nl	A	192.31.231.66
www.cs.vu.nl	CNAME	solving.cs.vu.nl

Таблица 4.4 (продолжение)

Имя	Тип записи	Значение записи
ftp.cs.vu.nl	CNAME	soling.cs.vu.nl
soling.cs.vu.nl	HINFO	Sun Unix
soling.cs.vu.nl	MX	1 soling.cs.vu.nl
soling.cs.vu.nl	MX	10zephyr.cs.vu.nl
soling.cs.vu.nl	A	130.37.24.11
laser.cs.vu.nl	HINFO	PC MS-DOS
laser.cs.vu.nl	A	130.37.30.32
vucs-das.cs.vu.nl	PTR	0.26.37.130.in-addr.arpa
vucs-das.cs.vu.nl	A	130.37.26.0

Узел `cs.vu.nl` представляет как домен, так и зону. Его запись о ресурсах SOA содержит специальную информацию о подлинности этого файла, которую мы рассматривать не будем. В этой зоне имеется три сервера имен, канонические имена их хостов содержатся в записях NS. Запись TXT используется для хранения дополнительной информации и не может быть обработана серверами имен автоматически. Кроме того, мы видим три почтовых сервера, обрабатывающих приходящую почту, адресованную пользователям этого домена. Число перед именем почтового сервера указывает на приоритет выбора. Отправляющий почту сервер сначала пытается установить контакт с принимающим сервером с наименьшим номером, в данном примере это `zephyr.cs.vu.nl`.

Хост `star.cs.vu.nl` выполняет обязанности сервера имен этой зоны. Серверы имен важны для любой службы имен. Что мы можем сказать об этом сервере имен? Ему присуща дополнительная устойчивость, создаваемая двумя независимыми сетевыми интерфейсами, каждый из которых представлен отдельной записью о ресурсах A. Таким образом, обрыв сетевой связи до некоторой степени пройдет без последствий.

Следующие четыре строки предоставляют необходимую информацию по почтовому серверу. Заметим, что этот почтовый сервер кроме всего прочего архивируется на другой почтовый сервер, путь к которому — `tornado.cs.vu.nl`.

Следующие шесть строк демонстрируют типовую конфигурацию web-сервера факультета, в котором на одной из машин реализован ftp-сервер с именем `soling.cs.vu.nl`. При исполнении обоих серверов на одной машине (и использовании этой машины только для служб Интернета) поддержка системы значительно облегчается. Так, например, оба сервера могут работать с одним представлением файловой системы, а для эффективности часть файловой системы можно реализовать на сервере `soling.cs.vu.nl`. Подобный подход часто применяется для служб WWW и FTP.

Следующие две строки содержат информацию об одном из лазерных принтеров, соединенных с локальной сетью. Последние две строки демонстрируют обратное отображение адресов в канонические имена. В данном случае имя суперкомпьютера отдела может быть найдено по его адресу в домене `in-addr.arpa`.

Поскольку домен `cs.vu.nl` реализован в одной зоне, табл. 4.4 не содержит ссылок на другие зоны. Способ создания ссылок на поддомены, реализованные

в другой зоне, иллюстрирует табл. 4.5. Все, что необходимо, — это указать сервер имен для поддомена, просто задав его доменное имя и IP-адрес. При разрешении имени узла, который не входит в домен cs.vu.nl, разрешение имени продолжится в месте, адрес которого будет считан из базы данных DNS для сервера имен домена cs.vu.nl.

Таблица 4.5. Часть описания домена vu.nl, содержащая домен cs.vu.nl

Имя	Тип записи	Значение записи
cs.vu.nl	NS	solo.cs.vu.nl
solo.cs.vu.nl	A	130.37.24.1

4.1.5. Пример — X.500

Система DNS — пример традиционной службы именования. Когда заданы имена (возможно, иерархически построенные), DNS разрешает имя как узел в графе именования и возвращает содержимое этого узла в виде записи о ресурсах. В этом смысле разрешение имени в DNS сравнимо с поиском номера телефона в телефонной книге.

В том, что мы называем службами каталогов, применяется другой подход. *Служба каталогов (directory service)* — это особый тип службы именования, в которой клиент может вести поиск сущности на основании описания ее свойств, а не полного имени. Этот подход очень похож на способ, которым люди используют справочник «Желтые страницы», когда им нужно, к примеру, починить разбитое окно. В этом случае пользователь ищет заголовок «Ремонт окон», а под ним находит список названий фирм, которые вставляют в окна стекла.

В этом пункте мы кратко рассмотрим службу каталогов OSI X.500. Несмотря на то что службы каталогов доступны уже более десятилетия, они стали особенно популярными лишь недавно в виде упрощенных версий, реализованных как службы Интернета. Детальную информацию по X.500 можно найти в [88, 368]. Практические сведения по различным службам каталогов, включая и X.500, содержатся в [414].

Пространство имен X.500

Концептуально служба каталогов X.500 состоит из множества записей, которые обычно называются элементами каталога. Элемент каталога в X.500 похож на запись о ресурсах системы DNS. Каждая запись состоит из набора пар (атрибут, значение), причем каждый атрибут имеет ассоциированный с ним тип. Различаются атрибуты с одним значением (однозначные) и атрибуты с несколькими значениями (многозначные). Последние обычно представляют собой массивы или списки. Так, например, простой элемент каталога, определяющий сетевые адреса нескольких основных серверов из табл. 4.4, приведен в табл. 4.6.

В этом примере мы использовали соглашение об именованиях, описанное в стандартах на X.500, которое применили к первым пяти атрибутам. Атрибуты Organization и OrganizationUnit описывают соответственно организацию и отдел,

ассоциирующиеся с данными, хранящимися в записи. Атрибуты `Locality` и `Country` предоставляют дополнительную информацию о том, где хранится элемент. Атрибут `CommonName` часто используется в качестве имени (не слишком точно-го) для идентификации элемента в ограниченной части каталога. Так, например, имени «Main server» может быть достаточно для того, чтобы найти элемент из нашего примера, имеющий соответствующие значения остальных четырех атри-бутов — `Country`, `Locality`, `Organization` и `OrganizationalUnit`. В нашем примере толь-ко атрибут `Mail_Servers` имеет несколько значений. Все остальные атрибуты од-нозначны.

Таблица 4.6. Простой пример элемента каталога X.500

Атрибут	Аббревиатура	Значение
Country	C	NL
Locality	L	Amsterdam
Organization	O	Vrije Universiteit
OrganizationalUnit	OU	Math. & Comp. Sc.
CommonName	CN	Main server
Mail_Servers	–	130.37.24.6, 192.31.231.42, 192.31.231.66
FTP_Server	–	130.37.24.11
WWW_Server	–	130.37.24.11

Набор всех элементов каталога службы каталогов X.500 называется *информа-ционной базой каталога (Directory Information Base, DIB)*. Важный момент — ка-ждая запись в DIB имеет уникальное имя, чтобы ее можно было найти. Глобаль-но уникальное имя получается из последовательности атрибутов именования каждой записи. Каждый атрибут именования называется *относительно различи-мым именем (Relative Distinguished Name, RDN)*. В нашем примере из табл. 4.6 первые пять атрибутов — это и есть все атрибуты именования. Если использо-вать стандартные сокращения для представления атрибутов именования X.500, приведенные в табл. 4.6, атрибуты `Country`, `Organization` и `OrganizationalUnit` можно использовать для формирования глобально уникального имени:

/C=NL/O=Vrije Universiteit/OU=Math. & Comp.Sc.

Это имя аналогично имени `nl.vu.cs` в системе DNS. Как и в DNS, использование глобально уникальных имен, образуемых после-довательным перечислением имен RDN, приведет нас к иерархии наборов эле-ментов каталога, которую мы будем называть *информационным деревом катало-гов (Directory Information Tree, DIT)*. DIT, в сущности, образует граф именования службы каталогов X.500, в котором каждый узел представляет собой элемент ка-талога. Кроме того, узел может также работать каталогом в традиционном смыс-ле, у него может быть несколько дочерних узлов, для которых он будет роди-телем. Для пояснения рассмотрим граф именования, частично показанный на рис. 4.10.

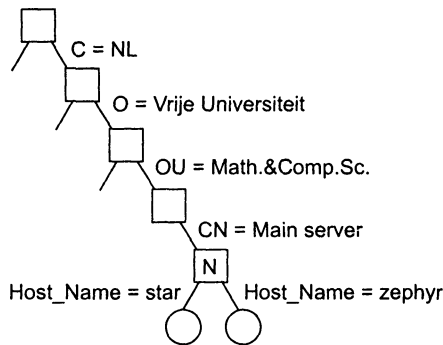


Рис. 4.10. Часть информационного дерева каталогов

Узел *N* соответствует элементу каталога, приведенному в табл. 4.6. В то же время этот узел выступает в качестве родителя нескольких других элементов каталога, которые имеют дополнительный атрибут именования *Host_Name*, используемый в качестве RDN. Эти сущности могут, например, задействоваться для представления хостов, как показано в табл. 4.7.

Таблица 4.7. Два элемента каталога, использующие в качестве RDN имя *Host_Name*

Первый хост		Второй хост	
Атрибут	Значение	Атрибут	Значение
Country	NL	Country	NL
Locality	Amsterdam	Locality	Amsterdam
Organization	Vrije Universiteit	Organization	Vrije Universiteit
OrganizationalUnit	Math. & Comp. Sc.	OrganizationalUnit	Math. & Comp. Sc.
CommonName	Main server	CommonName	Main server
Host_Name	star	Host_Name	zephyr
Host_Address	192.31.231.42	Host_Address	192.31.231.66

Узел в графе именования X.500 может, таким образом, быть представлен и в виде каталога в традиционном смысле, как мы обсуждали выше, и в виде записи X.500. Это разница поддерживается двумя различными операциями поиска. Операция *read*, предназначенная для чтения одиночной записи, дает ее путь в дереве DIT. С другой стороны, операция *list* используется для построения списка имен всех ребер, исходящих из данного узла дерева DIT. Каждое имя соответствует узлу, дочернему для данного. Отметим, что операция *list* не возвращает записей, она возвращает только имена. Рассмотрим, например, вызов операции *read* со следующим именем в качестве исходных данных:

/C=NL/O=Vrije Universiteit/OU=Math. & Comp. Sc./CN=Main server

Этот вызов вернет запись, приведенную в табл. 4.6, в то время как вызов операции `read` с теми же исходными данными вернет имена `star` и `zephyr` из сущностей, представленных в табл. 4.7, а также имена других хостов, зарегистрированных подобным образом.

Реализация X.500

Реализация службы каталогов X.500 выполняется в основном таким же образом, как и реализация службы имен, такой как DNS, за исключением того, что X.500 поддерживает больше операций поиска. Мы это кратко обсудили. При операциях с большим каталогом дерево DIT обычно разбивается и разносится по нескольким серверам, которые в терминологии X.500 называются *агентами службы каталогов* (*Directory Service Agents, DSA*). Каждая часть разбитого дерева DIT соответствует зоне в DNS. Точно так же каждый агент DSA ведет себя очень похоже на обычный сервер имен, за исключением того, что он реализует несколько стандартных для службы каталогов служб, таких как расширенные операции поиска.

Клиенты представлены тем, что называется *агентами пользователей каталога* (*Directory User Agents, DUA*). Агент DUA подобен процедуре разрешения имен из традиционной службы именованя. DUA обмениваются информацией с DSA в соответствии со стандартным протоколом доступа.

Что делает реализацию X.500 отличной от реализации DNS — так это механизмы поиска в базе DIB. В частности, имеются механизмы для поиска элемента каталога по заданному набору критериев, в который могут входить атрибуты искомым элементов. Например, предположим, что мы хотим получить список всех главных серверов университета Vrije. Если использовать запись, определенную в [206], этот список можно вернуть, используя следующую операцию поиска:

```
answer = search("&(C=NL)(O=Vrije Universiteit)(OU=*)(CN=Main server)")
```

В этом примере мы определили, что местом, где мы будем искать главные серверы, является организация под названием Vrije Universiteit в стране NL, при этом нас не интересует конкретный отдел этой организации, однако каждый возвращаемый результат должен иметь значение атрибута CN, равное Main server.

Важно заметить, что подобный поиск в службе каталогов — обычно достаточно дорогостоящая операция. Так, например, чтобы найти все главные серверы университета Vrije, необходимо найти все сущности в каждом отделе и собрать результаты поиска в единый ответ. Другими словами, для получения ответа мы обычно должны перебрать множество листовых узлов дерева DIT. На практике это еще означает, что следует перебрать также и множество агентов DSA. В противоположность этому, службы имен часто могут быть реализованы при помощи операции поиска, нуждающейся в доступе только к одному листовому узлу.

Система X.500 находится в одном ряду с множеством других протоколов OSI. Доступ к каталогу X.500 в соответствии с официальными правилами — дело не простое. Чтобы приспособить службу каталогов X.500 к Интернету, был создан

более простой протокол, известный как *упрощенный протокол доступа к каталогам* (*Lightweight Directory Access Protocol, LDAP*).

LDAP — это протокол прикладного уровня, реализованный непосредственно поверх TCP [481, 501], что уже способствует его простоте по сравнению с официальным протоколом доступа OSI. Кроме того, параметры операций поиска и обновления могут быть переданы просто в виде строк. Раздельного кодирования, необходимого по протоколу OSI, не нужно. Протокол LDAP постепенно становится стандартом де-факто для служб каталогов в Интернете. Он включается во многие распределенные системы, включая, например, Windows 2000, о которой мы поговорим в главе 9. Практическую информацию по LDAP можно найти в [217].

4.2. Размещение мобильных сущностей

Службы имен, которые мы обсуждали, используются в первую очередь для именованных сущностей, имеющих постоянное местоположение. По своей природе традиционные системы именования плохо подходят для поддержания отображения имени в адрес, если тот регулярно изменяется, как это происходит в случае мобильных сущностей. Эту проблему, в том числе и решения для размещения мобильных сущностей, мы и рассмотрим в этом разделе.

4.2.1. Именованное и локализация сущностей

Как мы узнали в предыдущем разделе, сущности именуются для того, чтобы иметь возможность их найти и получить к ним доступ. Мы выделили три типа имен: имена, удобные для восприятия, идентификаторы и адреса. Поскольку распределенные системы строятся для людей и для доступа к сущности, необходимо знать ее адрес, фактически все системы именования поддерживают отображение имен, удобных для восприятия, в адреса.

Как мы говорили, для эффективной реализации полномасштабного пространства имен, такого как в DNS, удобно разбить пространство имен на три уровня. Глобальный и административный уровни характеризуются тем, что имена изменяются нечасто. Точнее, содержимое узлов этих частей пространства имен относительно постоянно. Вследствие этого репликация и кэширование способны повысить эффективность реализации.

Содержимое узлов управленческого уровня часто изменяется. Поэтому производительность операций поиска и обновления на этом уровне становится особенно важной. На практике требования по производительности можно удовлетворить путем реализации узлов на локальных высокопроизводительных серверах имен.

Рассмотрим теперь поближе, какое допущение следует сделать на самом деле и как подобный подход можно использовать для реализации крупномасштабной системы именования. Прежде всего рассмотрим поиск адреса удаленного хоста `ftp.cs.vu.nl`. Если считать содержимое узлов глобального и административного уровней стабильным, клиент, вероятно, сможет найти адрес сервера имен домена `cs.vu.nl`

в локальном кэше. Соответственно, ему понадобится всего один запрос к серверу имен, чтобы найти адрес `ftp.cs.vu.nl`.

Рассмотрим далее изменение адреса `ftp.cs.vu.nl`, например, из-за переноса FTP-сервера на другую машину. До тех пор пока сервер будет оставаться на машине, входящей в домен `cs.vu.nl`, обновление можно выполнить быстро. В этом случае изменению подвергнется только база данных DNS сервера имен `cs.vu.nl`. Поиск будет столь же эффективен, как и ранее.

Соответственно, в том случае, если узлы глобального и административного уровней изменяются редко, а изменения обычно ограничиваются одним сервером имен, системы именования, такие как DNS, весьма эффективны.

Посмотрим теперь, что произойдет, если сервер `ftp.cs.vu.nl` придется перенести на машину с именем `ftp.cs.umsa.edu.au`, находящуюся в абсолютно другом домене. Первое, на что следует обратить внимание, — имя `ftp.cs.vu.nl` предпочтительно было бы не менять, поскольку многие приложения и пользователи могли бы иметь на него символическую ссылку. Другими словами, это имя предположительно используется в качестве идентификатора. Его изменение сделает все ссылки на него неверными.

Существует два основных решения. Первое решение — записать адрес новой машины в базе данных DNS для `cs.vu.nl`. Второе решение — записать имя новой машины, а не ее адрес, включив `ftp.cs.vu.nl` в символическую ссылку. Оба решения имеют серьезные недостатки.

Сначала рассмотрим запись адреса новой машины. Ясно, что на операцию поиска подобный подход не повлияет. Однако если сервер `ftp.cs.vu.nl` придется еще раз переносить на новую машину, придется обновлять и его элемент базы данных DNS в `cs.vu.nl`. Важно отметить, что само по себе обновление выполняется не дольше локальной операции, однако в реальности может потребовать сотен миллисекунд. Другими словами, подобный подход нарушает предположение о том, что операции в узлах управленческого уровня эффективны.

Основной недостаток использования символических ссылок состоит в том, что теряется эффективность операций поиска. Каждая операция поиска выполняется за два шага.

1. Поиск имени новой машины.
2. Поиск адреса, соответствующего этому имени.

Однако, если сервер `ftp.cs.vu.nl` снова переместится, скажем, на адрес `ftp.cs.berkeley.edu`, мы сможем осуществить операцию локального обновления, поместив имя `ftp.cs.unisa.edu.au` в символическую ссылку на `ftp.cs.berkeley.edu` и сохранив для `cs.vu.nl` элемент в базе данных DNS таким, как он есть. Недостаток — добавление еще одного шага к операции поиска.

Для сущностей с высокой мобильностью ситуация может быть только хуже. Каждый раз при переносе сущности либо приходится выполнять нелокальную операцию внесения изменений, либо добавляется еще один шаг к операции поиска.

Другая серьезная проблема обсуждаемых подходов состоит в том, что имя `ftp.cs.vu.nl` не должно изменяться. Таким образом, становится чрезвычайно важ-

ным выбирать имена, которые (предположительно) не придется менять в течение всего времени существования той сущности, которую они идентифицируют. Более того, такое имя нельзя использовать для каких-либо других сущностей. На практике подбор подобных имен, особенно для долгоживущих сущностей, затруднен, что демонстрирует нам система именования в World Wide Web. Так, многие сайты известны под разными именами и все эти имена остаются правильными, то есть всегда ссылаются на одну и ту же сущность, даже в случае мобильности.

По этим причинам традиционные службы имен, такие как DNS, не справляются с мобильными сущностями. Тут необходимо другое решение. По сути, проблемы связаны с тем, что традиционные службы именования поддерживают прямое отображение имен, удобных для восприятия человеком, в адреса сущностей. Каждый раз, когда имя или адрес изменяются, приходится изменять и отображение, как показано на рис. 4.11, а.

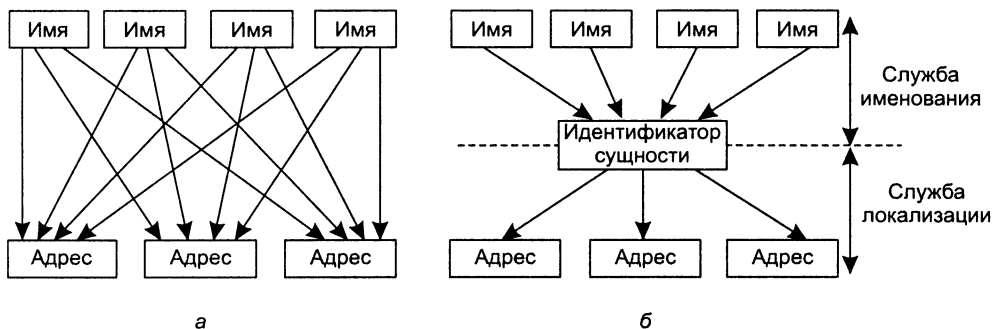


Рис. 4.11. Прямое одноуровневое отображение имен в адреса (а). Двухуровневое отображение с использованием идентификаторов (б)

Наилучшее решение состоит в том, чтобы отделить именование сущностей от их размещения, введя идентификаторы, как показано на рис. 4.11, б. Напомним, что идентификаторы никогда не изменяются, каждая сущность имеет только один идентификатор, и идентификатор не может быть назначен другой сущности [212]. В общем случае идентификатор не предназначен для восприятия человеком. Иначе говоря, он оптимизирован исключительно для машинной обработки.

При поиске сущности средствами службы именования она возвращает идентификатор. Идентификатор может быть сохранен на локальной машине на любой необходимый срок, поскольку он не может ни начать указывать на другую сущность, ни измениться. Под каким именем он сохранен, не важно. Соответственно, когда этот идентификатор потребуется снова, его можно будет просто взять с локальной машины, не обращаясь к поиску средствами службы именования.

Размещение сущности определяется посредством отдельной *службы локализации* (location service). Служба локализации, собственно, использует в качестве исходных данных идентификатор и возвращает текущий адрес соответствующей

ему сущности. Если имеется несколько копий сущности, будет возвращено несколько адресов. В этом пункте мы сосредоточимся исключительно на проблеме реализации эффективной службы локализации.

4.2.2. Простые решения

Обсудим сначала два простых решения по локализации сущностей. Оба эти решения применимы только в локальных сетях. Тем не менее в этой среде они обычно успешно работают, что делает их простоту особенно привлекательной.

Широковещательная и групповая рассылки

Рассмотрим распределенную систему, построенную на основе компьютерной сети, предоставляющей эффективные механизмы широковещательной рассылки. Обычно подобные механизмы предоставляются в локальной сети, в которой все машины присоединены к одному кабелю. Беспроводные локальные сети также попадают в эту категорию.

Локализация сущности в такой среде проста: сообщение, содержащее идентификатор сущности, широковещательной рассылкой доносится до каждой машины и каждая из машин откликается на этот запрос проверкой, не на ней ли размещена эта сущность. Те машины, которые могут предоставить точку входа к искомой сущности, посылают ответное сообщение, содержащее адрес точки входа.

Этот принцип использован в *протоколе разрешения адресов (Address Resolution Protocol, ARP)* Интернета для поиска канального адреса машины, для которой известен только IP-адрес [358]. В сущности, машина, посылающая широковещательный пакет в локальную сеть, спрашивает, кто владеет указанным IP-адресом. Когда машина принимает подобное сообщение, получатель проверяет, не отслеживает ли он указанный IP-адрес. Если это так, он посылает ответный пакет, содержащий, например, его адрес в сети Ethernet.

Широковещательная рассылка с ростом сети теряет эффективность. Потери пропускной способности сети на пересылку сообщений — не единственная проблема, более важно то, что множество хостов вынуждены прерывать свою работу из-за запроса, на который они не будут отвечать. Одно из возможных решений этой проблемы — переход к групповой рассылке, при которой запрос получает лишь ограниченная группа хостов. Так, например, сеть Ethernet аппаратно поддерживает групповую рассылку канального уровня.

Групповая рассылка может также использоваться для локализации сущностей в некоммутируемых сетях. Так, например, в Интернете поддерживается групповая рассылка сетевого уровня, при которой хостам разрешено присоединяться к конкретной группе рассылки. Эта группа определяется адресом групповой рассылки. Когда хост посылает сообщение на адрес групповой рассылки, сетевой уровень предоставляет удобную службу по доставке этого сообщения каждому из членов группы. Эффективная реализация групповой рассылки в Интернете обсуждается в [121, 122].

Адрес групповой рассылки может использоваться в качестве обобщенной службы локализации для множества сущностей. Рассмотрим, например, органи-

зацию, в которой каждый сотрудник имеет собственный мобильный компьютер. Когда этот компьютер присоединяется к локальной сети, ему динамически присваивается IP-адрес. Кроме того, он включается в группу рассылки. Когда процесс желает локализовать компьютер *A*, он посылает запрос «Где *A*?» в группу рассылки. Если компьютер *A* в этот момент подключен к сети, он отвечает, сообщая свой текущий IP-адрес.

Другой способ использования адреса групповой рассылки — ассоциировать его с реплицируемой сущностью и путем групповой рассылки выполнить поиск *ближайшей* реплики. При посылке запроса на адрес групповой рассылки каждая реплика сообщает свой текущий (обычный) IP-адрес. Примитивный способ отобрать ближайшую реплику — взять ту, ответ от которой пришел первым. Более разумные способы описаны в [189]. Как оказывается, выбрать ближайшую реплику в общем не так-то просто.

Передача указателей

Другой популярный подход к локализации мобильных сущностей основан на передаче указателей [149]. Принцип прост. Когда сущность перемещается из *A* в *B*, она сохраняет ссылку на свое новое местоположение в *A*. Преимущество этого подхода — в его простоте: как только сущность локализуется при помощи, например, традиционной службы именования, клиент может найти ее текущий адрес, пройдя по цепочке переданных указателей.

Имеется в наличии и комплект недостатков. Во-первых, если не предпринимать специальных мер, цепочка может стать настолько длинной, что локализация сущности станет слишком дорогим удовольствием. Во-вторых, все промежуточные местоположения в цепочке должны поддерживать свою часть цепочки так долго, как это будет необходимо. Третий и самый серьезный недостаток — уязвимость к потере ссылок. Как только пересылаемый указатель по каким-то причинам будет утрачен, сущность окажется невозможным локализовать. Таким образом, важно сохранить цепочку короткой и гарантировать сохранность пересылаемых указателей.

Чтобы лучше понять, как работает передача указателей, рассмотрим ее в контексте распределенных объектов. В соответствии с подходом *цепочек SSP* (*SSP chains*), каждый из пересылаемых указателей реализуется в виде пары (заместитель, скелетон), как показано на рис. 4.12 [408].

ПРИМЕЧАНИЕ

В SSP заместитель (*proxy*) называется заглушкой (*stub*), а скелетон (*skeleton*) — побером (*scion*), что приводит нас к *паре* (*stub, scion*), от которой и происходит аббревиатура SSP.

Скелетон (то есть серверная заглушка) содержит либо локальную ссылку на реальный объект, либо локальную ссылку на заместитель (то есть клиентскую заглушку) этого объекта. Чтобы подчеркнуть, что скелетоны выступают для удаленных ссылок в роли *входных элементов*, а заместители — в роли *выходных элементов*, они обозначены на рисунке особыми образом.

Всякий раз при передаче объекта из адресного пространства *A* в адресное пространство *B* он оставляет вместо себя в адресном пространстве *A* заместите-

ля, а для связи с этим заместителем в адресное пространство *B* устанавливает скелетон. Интересно, что эти перемещения абсолютно незаметны клиенту. Все, что он видит, — это заместитель. Как и куда заместитель передает вызовы, от клиента скрыто. Заметим также, что передача указателей не имеет ничего общего с поиском адреса. Вместо того чтобы отыскивать адрес, мы просто передаем запрос клиента по цепочке реальному объекту.

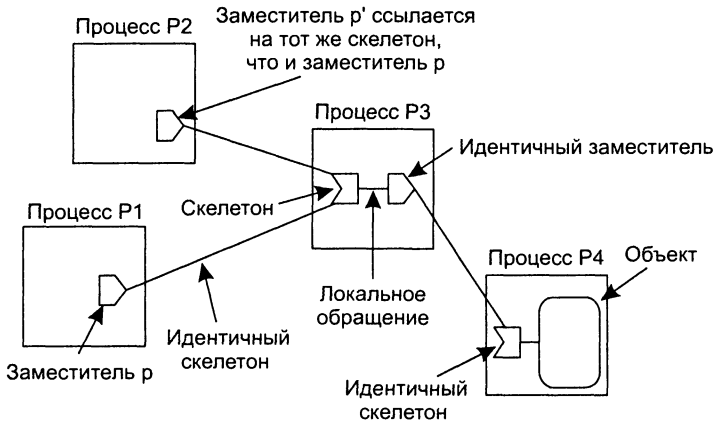


Рис. 4.12. Принцип передачи указателей с использованием пар (заместитель, скелетон)

Для сокращения длины цепочки пар (заместитель, скелетон) вместе с вызовом передается идентификатор того заместителя, который инициировал этот вызов. В идентификатор заместителя входит адрес клиента транспортного уровня, совмещенный с числом, идентифицирующим конкретный заместитель. Когда вызов добирается до объекта в том месте, где он в настоящее время расположен, ответ возвращается непосредственно тому заместителю, который инициировал вызов. Текущее местоположение объекта вкладывается в ответное сообщение, и заместитель меняет тот скелетон, с которым он работал, на скелетон из текущего местоположения объекта. Этот принцип иллюстрирует рис. 4.13.

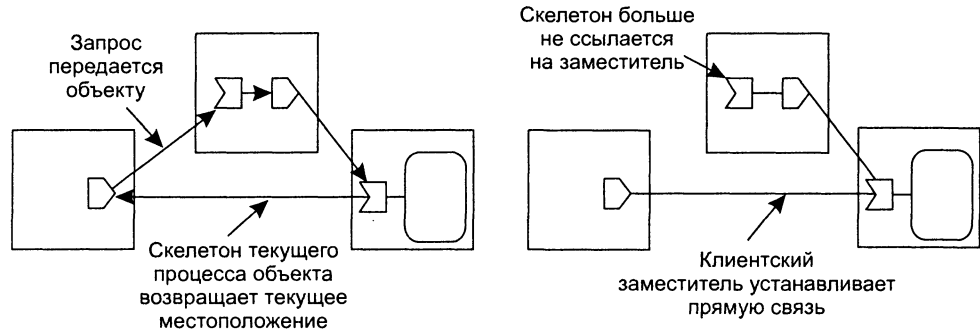


Рис. 4.13. Перенаправление передаваемого указателя путем настройки заместителя на прямую связь

Существует большая разница между посылкой ответа непосредственно начавшему обмен заместителю и посылкой ответа обратно по цепочке передающих указателей. В первом случае связь быстрее, поскольку во взаимодействие вовлечены немногие процессы. С другой стороны, подстроить можно только заместитель, начавший обмен, в то время как посылка ответа обратно по тому же пути дает возможность подстроить все промежуточные заместители.

После того как скелетон перестанет ссылаться на какой бы то ни было заместитель, его можно удалить. Как сделать это автоматически, рассматривается в разделе 4.3.

Как мы говорили в главе 2, ссылки на объекты в распределенных системах можно реализовать как заместители, передаваемые в виде параметров при обращении к методу. Эта схема подходит также и для пересылки указателей. Предположим, что процесс $P1$ (см. рис. 4.12) передает свою ссылку на объект O процессу $P2$. Передача ссылки производится путем установки копии заместителя p в адресное пространство процесса $P2$ (при этом копия превращается в заместитель p'). Заместитель p' ссылается на тот же самый скелетон, что и p , так что механизм пересылки обращения работает так же, как раньше.

Проблемы возникают, когда процесс в цепочке пар (заместитель, скелетон) зависает. Это делает остальные процессы недоступными. Возможно несколько решений. Одна из возможностей, использованная в Emerald [220] и системе LII [64], — позволить машине, на которой был создан объект, — ее называют *базовой точкой* (*home location*) объекта — всегда хранить ссылку на его текущее местонахождение. Эта ссылка хранится и обрабатывается так, чтобы исключить сбои. При обрыве цепочки текущее местонахождение объекта запрашивается у его базовой точки. Для изменения базовой точки объекта можно использовать традиционную службу имен, в которой записывается текущая базовая точка. Подобные подходы мы обсудим далее.

4.2.3. Подходы на основе базовой точки

Использование широковещательной рассылки и передачи указателей создает проблемы масштабируемости. Широковещательные и групповые рассылки трудно эффективно реализовать в крупномасштабных сетях, а длинные цепочки пересылаемых указателей создают проблемы с производительностью и чувствительны к обрывам связей.

Популярный подход к поддержке мобильных сущностей в полномасштабных сетях состоит во введении понятия *базовой точки* (*home location*), из которой отслеживается текущее местоположение объекта. Для защиты базовой точки от сбоев в сети или процессах можно применять специальные методики. На практике базовой точкой обычно выбирается то место, где была создана сущность.

Как говорилось ранее, подход на основе базовой точки используется в качестве аварийного метода служб локализации, основанных на передаче указателей. Другим примером подобного подхода является схема мобильного IP-адреса [348]. Каждый мобильный хост имеет фиксированный IP-адрес. Всякая связь с этим IP-адресом изначально перенаправляется *агенту базы* (*home agent*) мобильного хоста. Этот агент находится в локальной сети, соответствующей сетевому адресу, со-

держателю IP-адрес мобильного хоста. Каждый раз, когда мобильный хост перемещается в другую сеть, он запрашивает временный адрес для связи. Этот адрес, который называется *контрольным адресом* (*care-of address*), регистрируется агентом базы.

Когда агент базы получает от мобильного хоста пакет, он проверяет местонахождение хоста. Если хост находится в текущей локальной сети, пакет просто пересылается ему. В противном случае он передается туда, где в настоящее время находится хост, то есть помещается в виде данных в IP-пакет и пересылается на контрольный адрес. Одновременно отправитель пакета уведомляется о текущем местоположении хоста. Этот принцип иллюстрирует рис. 4.14. Отметим, что IP-адрес успешно используется в качестве идентификатора мобильного хоста.

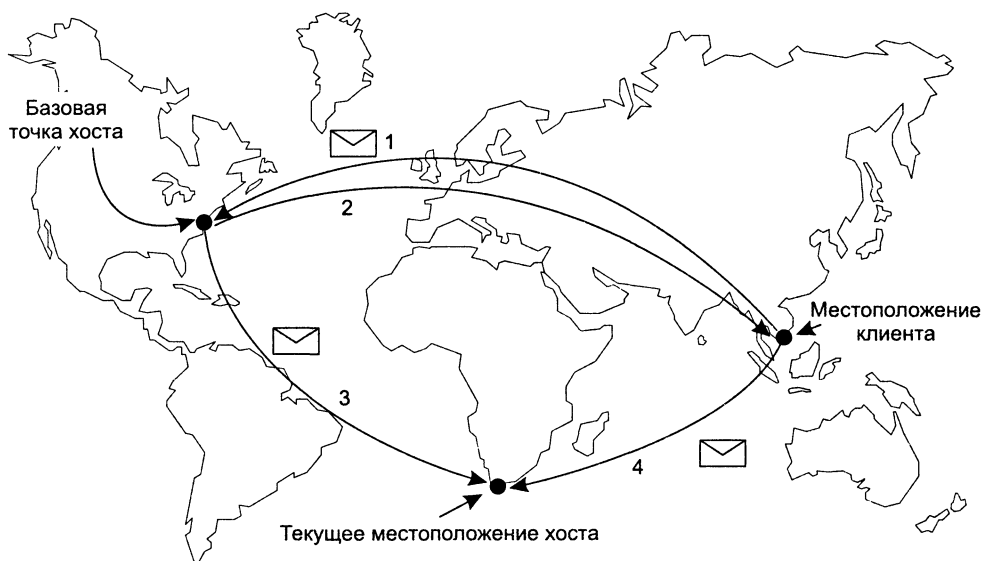


Рис. 4.14. Принцип работы схемы мобильного IP-адреса

В соответствии с рисунком схема мобильного IP-адреса работает следующим образом.

1. Отправка пакета хосту на его базу.
2. Возвращение текущего адреса.
3. Передача сигнального пакета по текущему адресу.
4. Передача последующих пакетов по текущему адресу.

Рисунок также иллюстрирует недостатки подходов, основанных на базовой точке, для крупномасштабных сетей. Для связи с мобильной сущностью клиент должен сперва связаться с базой, которая может находиться совершенно не там, где сама сущность. В результате задержки связи возрастают.

Решение, которое применяется в мобильной телефонии, — это двухуровневая схема [302]. При установке связи с мобильной сущностью клиент сначала прове-

ряет локальный реестр, чтобы определить, нельзя ли связаться с клиентом локально. Если нельзя, осуществляется контакт с базой сущности для определения ее текущего местоположения. Позднее мы обсудим расширенный вариант этой схемы с множеством иерархических уровней.

Другой важный недостаток подходов, основанных на базовой точке, состоит в ее обязательной фиксированности. То есть мы всегда должны гарантировать существование базовой точки. В противном случае связаться с сущностью не удастся. Проблемы усугубляются, если сущность имеет длительный срок существования и постоянно перемещается в удаленные от базовой точки части сети. В этом случае было бы лучше позволить базе перемещаться вместе с хостом.

Решение этой проблемы состоит в том, чтобы регистрировать базовую точку в традиционной службе именования и требовать от клиента искать сперва базовую точку. Поскольку местонахождение базовой точки можно считать относительно стабильным, после нахождения его можно успешно кэшировать.

4.2.4. Иерархические подходы

Двухэтапный подход к локализации сущностей, основанный на базовой точке, можно обобщить на случай множества уровней. В этом пункте мы рассмотрим сначала общий подход к иерархической схеме локализации, а затем представим способы его оптимизации. Подход, о котором мы будем говорить, основан на службе локализации Globe, описанной в [470]. Это служба локализации общего назначения, представленная множеством иерархических служб локализации [355, 487], реализованных в *персональных системах связи (personal communication systems)*.

Основной механизм

В иерархической схеме сеть делится на домены. Это сильно напоминает иерархическую организацию DNS. Домен верхнего уровня охватывает сеть целиком. Каждый домен делится на множество поддоменов. Домен самого нижнего уровня, называемый *листовым доменом (leaf domain)*, обычно соответствует локальной сети в компьютерных сетях или соту в мобильной телефонии.

Также аналогично DNS и другим иерархическим системам именования. Каждый домен D имеет ассоциированный с ним направляющий узел $dir(D)$, который отслеживает сущности домена. Таким образом, мы получаем дерево направляющих узлов. Направляющий узел домена верхнего уровня, именуемый *корневым направляющим узлом (root directory node)*, содержит сведения обо всех сущностях. Подобная обобщенная организация сети из доменов и направляющих узлов показана на рис. 4.15.

Чтобы отслеживать местонахождение сущностей, каждая сущность, находящаяся в домене D , представлена *локализирующей записью (location record)* в направляющем узле $dir(D)$. Локализирующая запись для сущности E в направляющем узле N листового домена D содержит текущий адрес сущности в этом домене. С другой стороны, направляющий узел N' в домене следующего уровня D' , в который

входит D , содержит в локализирующей записи для E только указатель на N . Точно так же и родительский для N' узел содержит в локализирующей записи для E только указатель на N' . Соответственно, на корневом узле находятся локализирующие записи для всех сущностей, и каждая локализирующая запись содержит указатель на направляющий узел в домене нижнего уровня, в котором находится сущность, идентифицируемая этой записью.

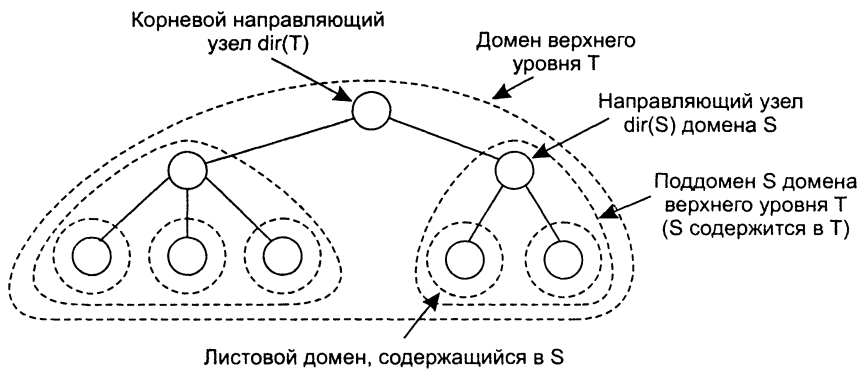


Рис. 4.15. Иерархическая организация службы локализации в виде доменов, каждый из которых имеет ассоциированный с ним направляющий узел

Сущность может иметь несколько адресов, например, когда она реплицирована. Если сущность имеет адреса в листовых доменах $D1$ и $D2$ соответственно, то направляющий узел наименьшего из доменов, в который входят $D1$ и $D2$, будет содержать два указателя — по адресу на каждый поддомен. Это приводит нас к обобщенной организации сети (рис. 4.16).

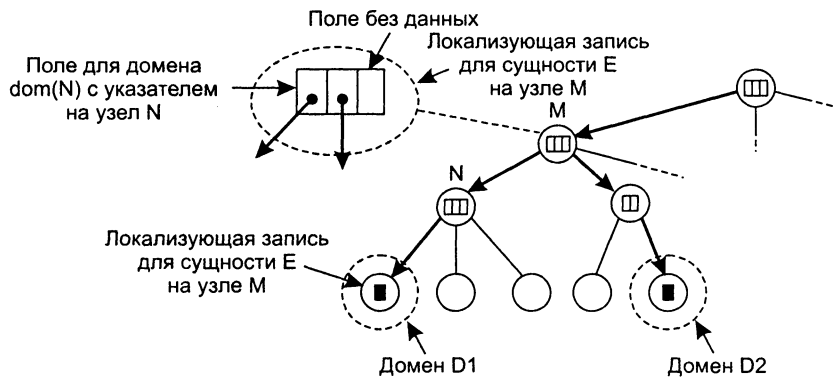


Рис. 4.16. Пример хранения информации о сущности, имеющей два адреса в различных листовых доменах

Рассмотрим теперь, как в подобной иерархической службе осуществляется операция поиска. Как показано на рис. 4.17, клиент, желающий найти сущность E , посылает запрос на поиск направляющему узлу листового домена D , к которому

этот клиент прикреплен. Если направляющий узел не содержит локализующей записи для этой сущности, значит, в настоящее время она находится за пределами домена D . Соответственно, узел пересылает запрос своему родителю. Отметим, что родительский узел представляет больший домен, чем дочерний. Если и родительский домен не содержит локализующей записи для E , запрос на поиск передается уровнем выше и т. д.

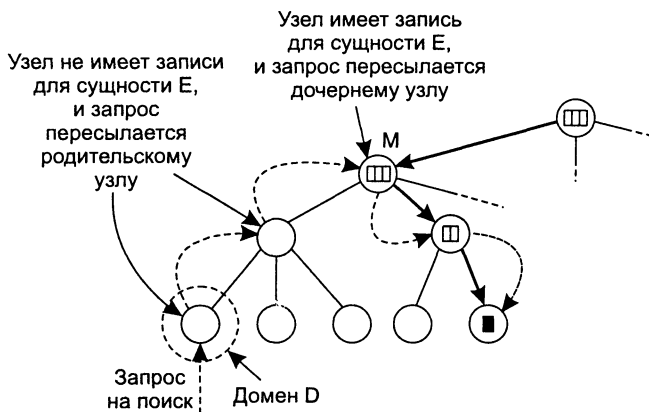


Рис. 4.17. Поиск сущности в иерархической службе локализации

После того как запрос достигнет направляющего узла M , где хранится локализующая запись для сущности E , мы узнаем, что E находится где-то в домене $dom(M)$, содержимое которого отражено на узле M . На рисунке показано, что на M хранится локализующая запись, содержащая указатель на один из поддоменов. Запрос на поиск будет переслан направляющему узлу этого поддомена, который в свою очередь переправит его дальше, вниз по дереву, пока он не достигнет листового узла. Локализующая запись, хранящаяся на листовом узле, содержит адрес сущности E в этом листовом домене. Этот адрес и будет возвращен клиенту, инициировавшему поиск.

По поводу иерархических служб локализации следует сделать важное замечание: операция поиска разворачивается наступательно. Сущность ищется в постепенно увеличивающемся кольце, центром которого является отправивший запрос клиент. Область поиска расширяется каждый раз, когда запрос на поиск пересылается направляющему узлу верхнего уровня. В наиболее неудачном случае поиск продолжается до тех пор, пока запрос не достигнет корневого узла. Поскольку корневой узел содержит локализующие записи всех сущностей, запрос после этого будет просто пересылаться от указателя к указателю, пока не доберется до одного из листовых узлов.

Операции обновления разворачиваются аналогичным образом, как продемонстрировано на рис. 4.18. Рассмотрим сущность E , для которой в листовом домене D была создана реплика. Необходимо вставить в направляющий узел адрес этой реплики. Вставка начинается с листового узла $dir(D)$ домена D , который не-

медленно пересылает запрос на вставку своему родителю. Родитель также пересылает запрос на вставку, и так до тех пор, пока он не достигнет направляющего узла M , в котором уже имеется локализирующая запись для E .

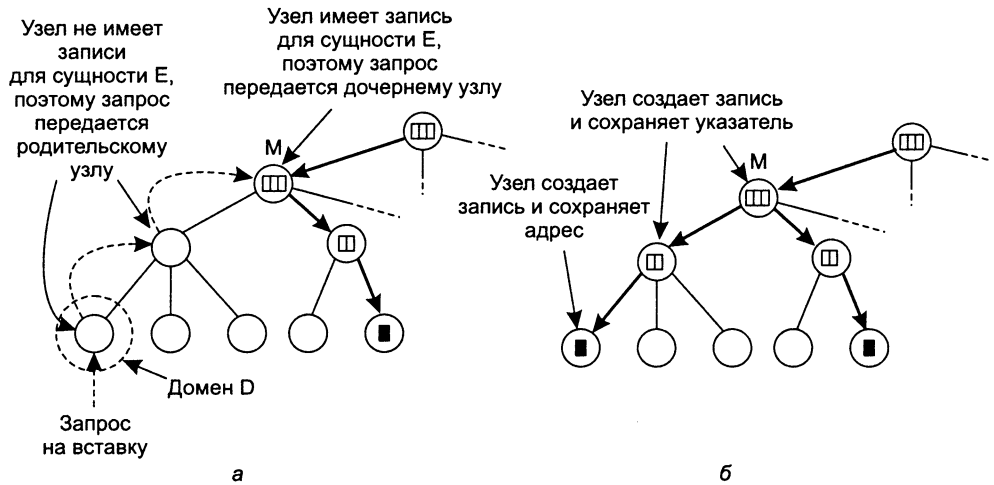


Рис. 4.18. Запрос на вставку пересылается первому узлу, имеющему информацию о сущности E (а). Создана цепочка передачи указателей до листового узла (б)

Узел M сохранит указатель в локализирующей записи для сущности E , ссылающийся на дочерний узел, от которого пришел запрос на вставку. После этого дочерний узел также создает локализирующую запись для сущности E , содержащую указатель на узел еще более низкого уровня, от которого запрос пришел к нему. Этот процесс продолжается, пока мы не доберемся до листового узла, который инициировал вставку. Листовой узел, наконец, создает запись с адресом сущности в ассоциированном с ним листовом домене.

Таким образом, вставка адреса, как было описано, ведет к образованию цепочки указателей, ведущей сверху вниз и начинающейся с направляющего узла самого нижнего из имеющих локализирующую запись для сущности E уровня. Альтернативой является создание локализирующей записи до передачи запроса родительскому узлу. Другими словами, цепочка указателей строится снизу вверх. Преимущество этого варианта состоит в том, что адрес для поисковых запросов оказывается доступным немедленно. Соответственно, если родительский узел временно недоступен, адрес по-прежнему можно будет найти в домене, связанном с текущим узлом.

Операция удаления аналогична операции вставки. При необходимости удаления адреса сущности E из листового домена D направляющему узлу $dir(D)$ поступает запрос на удаление этого адреса из локализирующей записи для E . Если эта локализирующая запись оказывается пустой, то есть других адресов для сущности E в домене D нет, запись можно удалять. В этом случае родительский узел для узла $dir(D)$ должен удалить свой указатель на $dir(D)$. Если локализирующая запись для E на родительском узле стала пустой, эта запись также может быть удалена, при этом следует уведомить следующий уровень в иерархии. Таким образом,

этот процесс продолжается до тех пор, пока, удалив очередной указатель из очередной локализирующей записи, мы не обнаружим там еще один или пока не доберемся до корневого узла.

Кэширование указателей

Иерархическая служба локализации предназначена для поддержания мобильных сущностей, то есть таких сущностей, местоположение которых регулярно меняется. В традиционных службах именования отображение имени на адрес предполагается неизменным, как минимум для узлов глобального и административного уровней. Это означает, что сохранение результатов поиска этих узлов в локальном кэше может сильно повысить эффективность работы.

Локальное кэширование адресов для служб локализации обычно не слишком эффективно. Зачем хранить адрес найденной мобильной сущности, если она в следующий раз вполне может оказаться в другом месте? Соответственно, потребуется провести полную процедуру поиска, которую только что описывали. Это неизбежно делает иерархические службы локализации значительно более дорогими, чем большинство служб именования.

Кэширование эффективно только в том случае, если кэшируемые данные изменяются редко. Рассмотрим мобильную сущность E , которая регулярно перемещается в пределах домена D . Перемещение в пределах домена означает, что E регулярно меняет текущий адрес. Однако цепочка указателей сущности E от корневого узла до направляющего узла $dir(D)$ не изменяется. Другими словами, место, где хранится наиболее свежая информация о местонахождении E , остается тем же самым, в данном случае это направляющий узел $dir(D)$. Таким образом, кэширование ссылок на направляющий узел вполне оправдано.

Обычно если D — наименьший домен, в пределах которого мобильная сущность регулярно перемещается, имеет смысл начинать поиск текущего местонахождения E с направляющего узла $dir(D)$, а не какого-либо другого узла. Этот подход используется в службе, описанной в [212], а также службе локализации Globe [23, 470] и называется *кэшированием указателей (pointer caching)*. Ссылка на $dir(D)$ может в принципе кэшироваться каждым узлом на пути к листовому узлу, с которого начинался поиск, как показано на рис. 4.19.

Дальнейшие усовершенствования можно произвести, не позволив узлу $dir(D)$ сохранить указатель на поддомен, в котором в настоящее время находится E , а заменив его текущим адресом E . В сочетании с кэшированием указателей операция поиска может быть реализована всего за два шага. Первый шаг требует просмотра локального кэша указателей, выводя нас прямо на соответствующий направляющий узел. Второй шаг включает в себя запрос к узлу с возвращением текущего адреса E .

Несмотря на то что принцип кэширования указателей в иерархических службах локализации работает, существует множество нюансов, требующих особого внимания. Один из них состоит в выборе направляющего узла, наиболее подходящего для хранения текущего адреса мобильной сущности. Представьте себе пользователя с мобильным компьютером, регулярно перемещающегося между двумя различными городами, скажем, Сан-Франциско и Лос-Анджелесом.

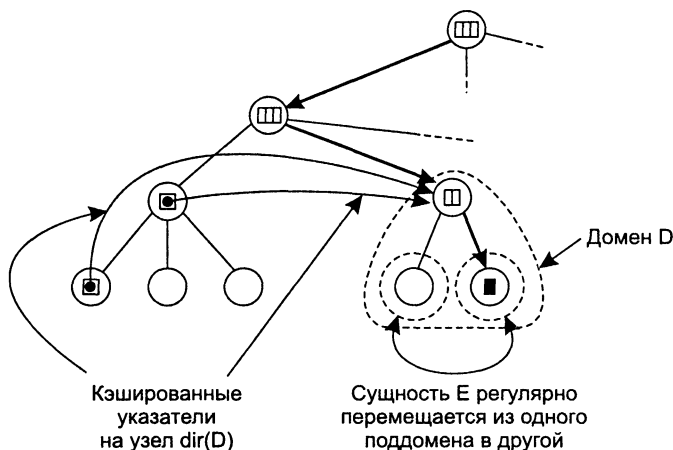


Рис. 4.19. Кэширование указателей на направляющий узел домена самого нижнего уровня из тех, на которых сущность находится подавляющую часть времени

Когда пользователь находится в Сан-Франциско, можно предположить, что он будет регулярно менять свое местоположение внутри города. Соответственно, имеет смысл сохранять его текущее местоположение в направляющем узле, соответствующем домену Сан-Франциско. Такая же модель поведения будет характерна для нашего путешественника и в Лос-Анджелесе.

Однако может статься, что пользователь все свое время летает из Сан-Франциско в Лос-Анджелес. В этом случае, может быть, разумнее сохранять его текущее местоположение в направляющем узле более высокого уровня, например, соответствующем штату Калифорния, независимо от того, где находится пользователь — в Сан-Франциско или Лос-Анджелесе.

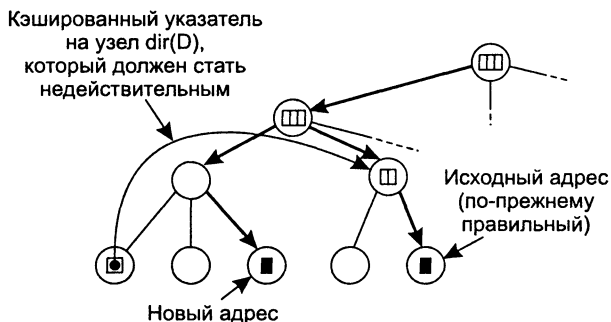


Рис. 4.20. Запись в кэше становится неправильной, поскольку возвращает нелокальный адрес, в то время как существует локальный

Другой оставшийся открытым вопрос — когда содержимое кэша перестает соответствовать действительности. Предположим, что пользователь из нашего примера получил столько заявок из Нью-Йорка, что решил открыть офис на Ман-

хэттене и поручил одному из своих приятелей отправлять все приходящие на его имя запросы в район Нью-Йорка. В терминах службы локализации у него появился постоянный адрес в домене Манхеттен. На все запросы на поиск из Нью-Йорка должен возвращаться этот новый адрес, а не кэшированный указатель на направляющий узел Калифорнии, как показано на рис. 4.20.

Проблемы масштабирования

Одна из основных проблем иерархических служб локализации состоит в том, что корневой узел должен хранить локализирующие записи и обрабатывать запросы ко всем сущностям. Объем хранимой информации сам по себе не является главной проблемой. Локализирующая запись невелика, она состоит только из идентификатора сущности и одного или более указателей на направляющие узлы нижнего уровня. Если размер каждой локализирующей записи приблизительно составляет 1 Кбайт, то для хранения, скажем, миллиарда сущностей необходим всего один терабайт. Этот объем обеспечат десять дисков по 100 Гбайт.

Настоящая проблема состоит в том, что без принятия специальных мер корневой узел должен будет обрабатывать такое количество запросов на поиск и обновление, что здесь неизбежно возникнет узкое место. Решение этой проблемы состоит в разбиении корневого узла и других направляющих узлов верхнего уровня на вложенные узлы. Каждый вложенный узел будет отвечать за обработку запросов к определенному поднабору сущностей, поддерживаемых службой локализации.

К сожалению, просто разделить узлы верхнего уровня на части недостаточно. Чтобы продемонстрировать проблему масштабируемости, рассмотрим разбиение корневого узла, скажем, на 100 вложенных узлов. Вопрос состоит в том, где будет находиться каждый из вложенных узлов в сети, охваченной службой локализации, физически.

Одна из возможностей — расположить вложенные узлы в соответствии с централизованным подходом, недалеко друг от друга, например собрать их в кластер. Действительно, корневой узел в этом случае реализуется посредством параллельного компьютера типа COW или MPP (с которыми мы познакомились в главе 1). Однако несмотря на то, что вычислительной мощности в этом случае будет достаточно, пропускной способности входящих и исходящих сетевых соединений для выполнения всех запросов может не хватить.

Следовательно, более удачной альтернативой является равномерное распределение вложенных узлов по сети. Однако, если сделать это неправильно, подобный способ может привести к появлению проблемы масштабирования. Рассмотрим снова мобильного пользователя, перемещающегося в основном между Сан-Франциско и Лос-Анджелесом. Корневой узел разбит на вложенные узлы, и вопрос, который предстоит решить, — какой вложенный узел сделать ответственным за этого пользователя.

Предположим, что для постоянного хранения локализирующих записей данного пользователя выбран вложенный узел, находящийся в Финляндии. Опуская кэширование указателей, это будет означать, что запрос, например, из Бразилии, перед тем как он будет передан на направляющий узел в Калифорнии, должен

пройти через корневой узел в Финляндии. Однако, как показано на рис. 4.21, разумней было бы пропустить запрос через вложенный узел, расположенный, например, в Калифорнии. Вопрос о том, какому вложенному узлу за какую сущность отвечать, для крупномасштабных служб локализации остается открытым.



Рис. 4.21. Иллюстрация проблем масштабирования, связанных с задачей равномерного распределения вложенных узлов разделенного корневого узла

Возможным решением было бы использовать то местоположение, в котором была создана сущность *E*. В частности, вложенный узел корневого узла, расположенный неподалеку от места, где была создана сущность *E*, может быть назначен ответственным за обработку относящихся к *E* запросов корневого уровня. Это решение срабатывает для сущностей, которые имеют тенденцию находиться неподалеку от места, где они были созданы. Однако если сущность перемещается в отдаленное место, проблема остается. Детальное описание этого подхода можно найти в [31].

4.3. Удаление сущностей, на которые нет ссылок

Службы именования и локализации предоставляют глобальные службы ссылок на сущности. До тех пор пока в службе имеются ссылки на сущность, эта сущность доступна пользователям и может использоваться. После того как доступ к сущности прекращается, ее следует удалить.

Во многих системах удаление сущностей реализуется явным образом. Например, если процесс *P* знает, что он — последний из процессов, использующих не-

который файл, причем в будущем этот файл ни одному процессу не понадобится, *P* может при своем завершении удалить этот файл. К сожалению, управление удалением сущностей в распределенных системах обычно сложнее. Так, в частности, нередко неизвестно, где в системе были созданы ссылки на удаляемую сущность с намерением воспользоваться ими позднее для доступа к ней. Если впоследствии к удаленной сущности действительно произойдет попытка доступа с помощью одной из таких ссылок, это приведет к ошибке.

С другой стороны, недопустимо также вообще не удалять сущность просто из-за отсутствия гарантий, что на нее не существует ссылок. Если ссылок все же не существует, мы окажемся в ситуации, когда сущность, которая продолжает потреблять ресурсы, никогда не будет использоваться. Понятно, что эта сущность — просто мусор и ее следует удалить.

Для снижения остроты проблем, связанных с удалением сущностей, на которые нет ссылок, распределенные системы могут предоставить механизмы их автоматического удаления. Эти механизмы известны под общим названием *распределенных сборщиков мусора* (*distributed garbage collectors*). В этом разделе мы поближе познакомимся с взаимоотношениями сущностей именования и локализации, а также с автоматической уборкой «ничейных» сущностей.

4.3.1. Проблема объектов, на которые нет ссылок

Чтобы понять, как производится уборка мусора, сосредоточимся на уборке распределенных объектов, в частности удаленных объектов. Напомним, что удаленный объект реализуется так, что его состояние целиком находится на сервере объектов, а клиент работает исключительно с заместителем. Как мы узнали в разделе 2.3, ссылка на удаленный объект обычно реализуется парой (заместитель, скелетон). Заместитель клиента содержит всю информацию, необходимую для работы с объектом через связанный с ним скелетон, реализованный на сервере. В наших примерах скелетон принимает участие в администрировании, необходимом для сборки мусора, вместе с заместителем. Другими словами, все что необходимо сделать для сборки мусора, скрыто как от пользователя, так и от самого объекта. Отметим, что и сам объект может содержать удаленные ссылки на другие объекты, например локальные ссылки на заместители этих объектов. Точно так же удаленные ссылки могут передаваться другим процессам путем копирования заместителя, связанного с процессом, в другой процесс.

Из этого следует, что к объекту можно получить доступ, только если у нас есть удаленная ссылка на него. Объект, на который отсутствуют удаленные ссылки, можно удалять из системы. С другой стороны, наличие удаленных ссылок не означает, что объект будет доступным всегда. По различным причинам возможно возникновение двух объектов, ссылающихся друг на друга и не имеющих других ссылок. Эта ситуация может быть легко обобщена на цепочку объектов, каждый из которых ссылается на соседа. Подобные объекты также следует выявлять и удалять.

В общем виде модель ссылок на объекты может быть представлена в виде графа, каждый узел которого — это объект. Ребро, идущее от узла *M* к узлу *N*, отра-

жает тот факт, что объект M содержит ссылку на объект N . Подмножество объектов, в которых нет ссылок друг на друга, известно под названием *корневого набора* (*root set*). Объекты корневого набора обычно представляют системные службы, пользователей и т. д.

На рис. 4.22 показан граф ссылок. Все белые узлы представляют объекты, на которые у объектов корневого набора нет прямых или косвенных ссылок. Такие объекты могут быть удалены.

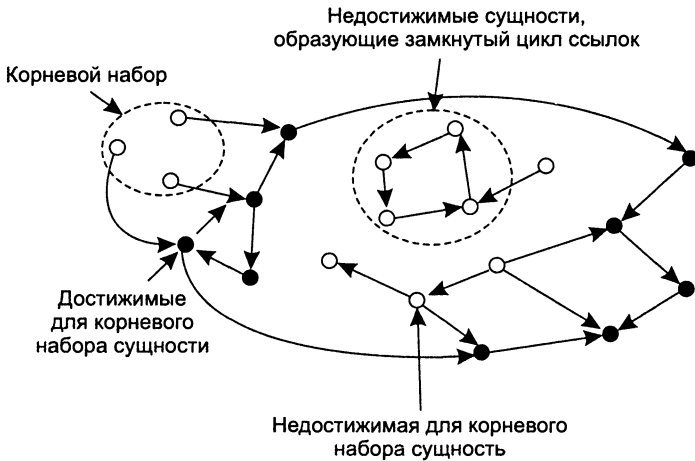


Рис. 4.22. Пример графа объектов, содержащих ссылки друг на друга

В однопроцессорных системах обнаружение и удаление объектов, на которые нет ссылок, относительно несложно по сравнению с тем, что происходит в распределенных системах (описание сборки мусора в однопроцессорных системах можно найти в [226]). Поскольку объекты в нашем случае разбросаны по множеству машин, распределенная сборка мусора нуждается в сетевой связи. Как выясняется, эта связь оказывает решающее влияние на производительность и масштабируемость решений. Кроме того, связь (как и машины, как и процессы) подвержена сбоям, которые способны еще более усложнить задачу.

В этом пункте мы рассмотрим несколько общеизвестных решений проблемы распределенной сборки мусора. В большинстве случаев эти решения лишь частично решают проблему. Наш подход подобен подходу, примененному в [356], где описывается целый обзор способов распределенной сборки мусора. Дополнительную информацию можно почерпнуть также из [2].

4.3.2. Подсчет ссылок

Метод проверки (который весьма популярен в однопроцессорных системах) возможности удаления того или иного объекта состоит в том, чтобы просто вести подсчет ссылок на этот объект. Каждый раз при создании ссылки на объект счетчик ссылок на этот объект увеличивается на единицу. Когда ссылка на объект

уничтожается, значение счетчика уменьшается. Как только значение счетчика станет равным нулю, объект можно удалять.

Простой подсчет ссылок

Простой подсчет ссылок в распределенных системах приводит к некоторым проблемам, возникающим отчасти из-за ненадежности связи. Без потери общности мы можем предположить, что объект хранит свой счетчик ссылок в соответствующем скелетоне, который управляется сервером объекта, отвечающим за этот объект. Этот случай показан на рис. 4.23.

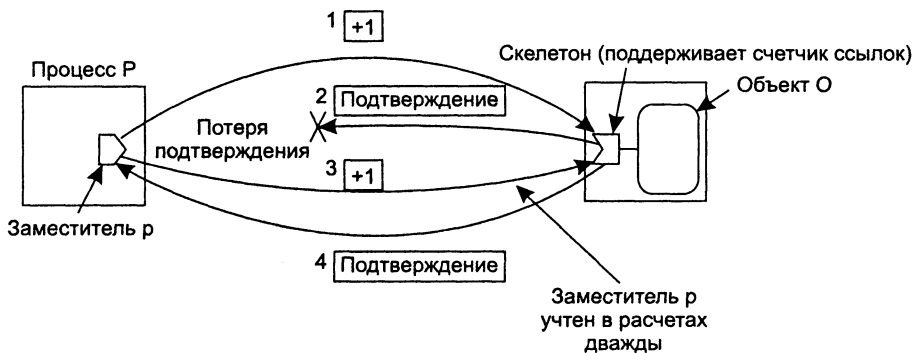


Рис. 4.23. Проблема сохранения правильности счетчика ссылок в условиях ненадежной связи

Когда процесс P создает ссылку на удаленный объект O , он, как показано на рисунке, устанавливает заместителя этого объекта в свое адресное пространство. Для увеличения счетчика ссылок заместитель посылает сообщение (1) скелону объекта и ожидает, что последний вернет подтверждение. Однако если подтверждение затеряется (2), заместитель произведет повторную посылку сообщения (3). Если не предпринимать никаких мер для обнаружения повторных сообщений, скелон может ошибочно увеличить счетчик еще раз. На практике относительно несложно организовать обнаружение повторных сообщений.

Подобные же проблемы могут возникнуть и при уничтожении удаленной ссылки. В этом случае заместитель посылает сообщение, уменьшающее счетчик ссылок. Если подтверждение вновь потеряется, повторная посылка сообщения приведет к повторному, на этот раз ошибочному уменьшению счетчика. Таким образом, при распределенном подсчете ссылок важно обнаруживать повторные сообщения и в случае обнаружения игнорировать их.

Другая проблема, которую необходимо решить, возникает при копировании удаленной ссылки в другой процесс. Если процесс $P1$ передает ссылку процессу $P2$, объект O , или точнее, его скелон, может остаться в неведении относительно создания новой ссылки. Таким образом, если процесс $P1$ решит уничтожить свою ссылку, содержимое счетчика ссылок может стать равным нулю и объект O может быть удален до того, как $P2$ сможет с ним связаться. Эту проблему иллюстрирует рис. 4.24, а.

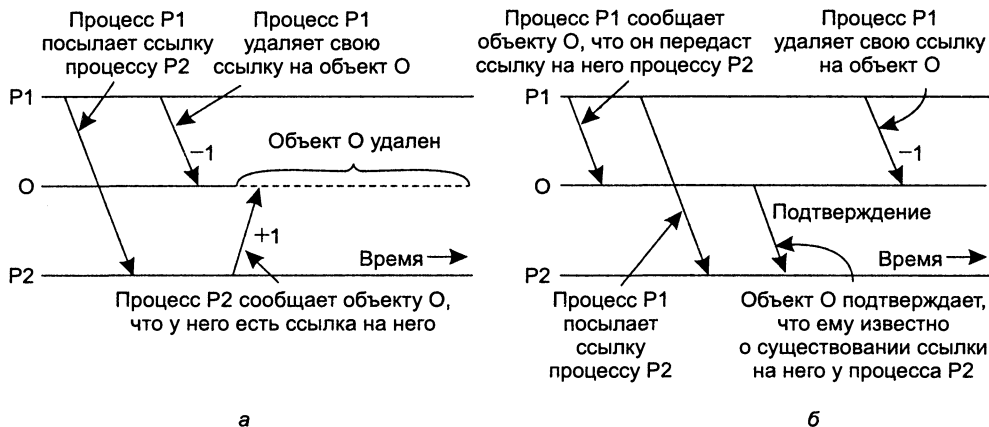


Рис. 4.24. Копирование ссылки в другой процесс и запоздавшее увеличение счетчика (а). Решение (б)

Решение состоит в том, чтобы потребовать от *P1* уведомлять скелетон объекта о том, что он собирается передать ссылку процессу *P2*. Кроме того, процесс не должен удалять свою ссылку до тех пор, пока скелетон не подтвердит ему, что знает о создании ссылки. Это решение продемонстрировано на рис. 4.24, б. Подтверждение, посылаемое от объекта *O* процессу *P2*, уведомляет *P2*, что объект *O* зарегистрировал ссылку, что позволяет *P1* позднее удалить свою ссылку. До тех пор пока процесс *P2* не уверен, что *O* знает о существовании у него ссылки, *P1* не может требовать у *O* уменьшения счетчика ссылок.

Заметим, что вдобавок к надежной связи передача ссылки требует теперь трех сообщений. Понятно, что в крупномасштабных распределенных системах это быстро приведет нас к проблемам с производительностью.

Улучшенные механизмы подсчета ссылок

Как было показано, простой распределенный подсчет ссылок требует соблюдения определенных условий в период между увеличением и уменьшением счетчика ссылок. Эти условия можно не соблюдать, если иметь дело только с уменьшением счетчика. Такое решение реализовано во *взвешенном подсчете ссылок (weighted reference counting)*, при котором каждый объект имеет фиксированный общий вес. При создании объекта этот вес сохраняется в ассоциированном с ним скелетоне вместе с его частичным весом, который инициализируется общим весом, как показано на рис. 4.25, а.

При создании новой удаленной ссылки половина частичного веса, хранящегося в скелетоне объекта, присваивается новому заместителю, как показано на рис. 4.25, б. Оставшаяся половина хранится в скелетоне. Когда удаленная ссылка дублируется, например, при ее передаче из процесса *P1* в процесс *P2*, половина частичного веса заместителя из *P1* передается заместителю процесса *P2*, в который производится копирование, а вторая половина остается в заместителе процесса *P1*, как показано на рис. 4.25, в.

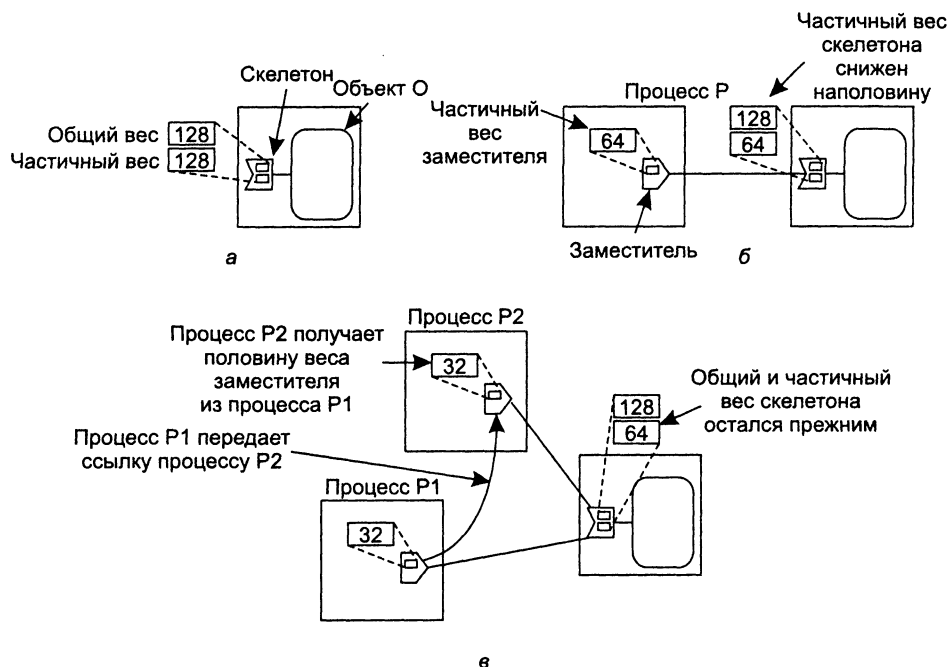


Рис. 4.25. Исходное соотношение весов при взвешенном подсчете ссылок (а). Присвоение веса при создании новой ссылки (б). Присвоение веса при копировании ссылки (в)

При уничтожении ссылки скелетону объекта посылается сообщение, уменьшающее счетчик. Затем из общего веса вычитается частичный вес уничтоженной ссылки. Как только общий вес станет равным нулю, объект можно удалять. Отметим, что в этом случае также предполагается, что сообщения не теряются и не дублируются.

Основная проблема взвешенного подсчета ссылок состоит в том, что количество создаваемых ссылок ограничено. Как только частичный вес скелетона и удаленных ссылок достигнет нуля, создавать или копировать новые ссылки будет невозможно. Решить проблему можно с помощью косвенных ссылок. Предположим, что процесс *P1* хочет передать ссылку процессу *P2*, но частичный вес его заместителя равен 1, как показано на рис. 4.26. В этом случае *P1* создает в своем адресном пространстве скелетон *s'* с подходящим общим весом и частичным весом, равным общему. Это абсолютно аналогично созданию в адресном пространстве объекта скелетона *s*. Затем процессу *P2* пересылается заместитель, которому передается половина частичного веса скелетона *s'*. Другая половина веса остается у *s'* для раздачи другим заместителям.

Заметим, что если общий вес скелетона *s'* установить в 1, этот подход превратится в подобие передачи указателя из процесса *P1* в процесс *P2*. Если, в свою очередь, *P2* захочет передать свою ссылку, он может создать еще один передаваемый указатель. Наиболее серьезная проблема передачи указателей состоит в том,

что длинные цепочки существенно снижают производительность. Кроме того, цепочки весьма уязвимы при сбоях.

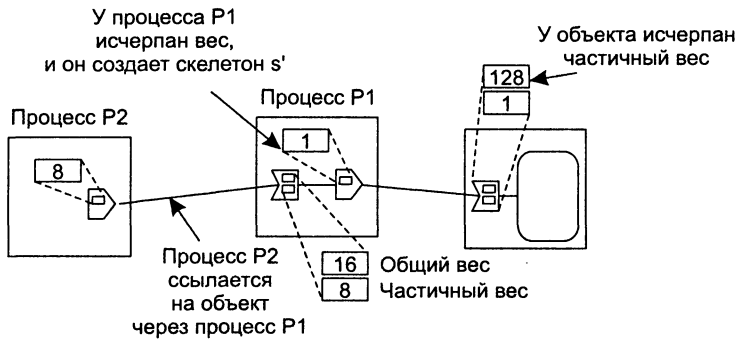


Рис. 4.26. Создание косвенной ссылки в случае исчерпания частичного веса

Альтернативой косвенным ссылкам может быть подсчет поколений ссылок (*generation reference counting*). Вновь предположим, что каждая удаленная ссылка представляет из себя пару (заместитель, скелетон), причем скелетон принадлежит тому же адресному пространству, что и объект. Каждый заместитель хранит счетчик копий и номер поколения. При создании новой ссылки вида (заместитель, скелетон) номер поколения соответствующего заместителя устанавливается в ноль. Поскольку пока еще не было сделано копий заместителя, его счетчик копий также равен нулю.

Копирование удаленной ссылки (заместитель, скелетон) в другой процесс производится обычным образом — пересылкой ему копии заместителя. В этом случае счетчик копий в исходном заместителе увеличивается на единицу, а счетчик копий в скопированном заместителе выставляется в ноль. Однако поскольку этот заместитель был скопирован, он принадлежит к следующему поколению, и по этой причине его номер поколения на единицу больше, чем у исходного заместителя, что и демонстрирует рис. 4.27.

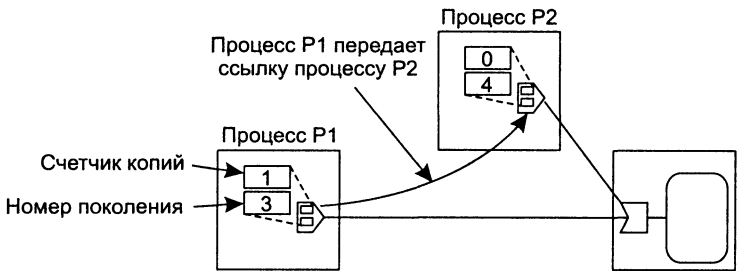


Рис. 4.27. Создание и копирование удаленных ссылок при подсчете поколений ссылок

Скелетон поддерживает таблицу G , в которой $G[i]$ обозначает число созданных копий для поколения i . Если заместитель p удаляется, сообщение об удале-

нии посылается скелетону, содержащему номер поколения заместителя, скажем k , и число копий, порожденных от заместителя p , скажем n . Скелетон приводит G в соответствие с действительностью, сначала уменьшая на единицу $G[k]$ и показывая, таким образом, что удалена ссылка k -го поколения. Затем он увеличивает $G[k+1]$ на n , чтобы показать, что удаленная ссылка создала n потомков или, другими словами, что она была скопирована в n ссылок следующего поколения (отметим, что скелетон должен сначала создать элемент $G[k+1]$, если ранее он ничего не знал о поколении $k+1$). В том случае, если все элементы $G[i]$ равны нулю, на объект больше нет ссылок и его можно удалить.

Подсчет поколений ссылок также требует надежных коммуникаций, но способен обеспечивать копирование ссылок, не обращаясь при создании копий к скелетону.

4.3.3. Организация списка ссылок

Другой подход к управлению ссылками состоит в том, чтобы заставить скелетон отслеживать заместители, которые содержат ссылки на него. Другими словами, вместо подсчета ссылок скелетон должен поддерживать полный список всех заместителей, которые на него указывают. Подобный *список ссылок* (*reference list*) имеет следующие важные свойства. Добавление заместителя в список ссылок не должно давать никакого результата, если этот заместитель в списке уже присутствует. Точно так же попытка удаления заместителя, отсутствующего в списке, не должна ни к чему вести. Добавление и удаление заместителей являются таким образом *идемпотентными* (*idempotent*) операциями.

Идемпотентные операции характеризуются тем, что их многократное повторение не влияет на конечный результат. В частности, при создании новой ссылки на объект создавший ссылку процесс может многократно отправлять сообщения скелетону объекта, требуя добавить ссылку в список ссылок. Он прекращает посылать сообщения после получения подтверждения о доставке. Соответственно, об удалении ссылки можно уведомлять посылкой (возможно, многократной) скелетону сообщения с требованием исключить ссылку из списка ссылок. Понятно, что операции увеличения и уменьшения счетчика не являются идемпотентными операциями.

Таким образом, список ссылок не требует от связи надежности, а также не нуждается в механизмах обнаружения и отбрасывания повторных сообщений (однако необходимо, чтобы добавление ссылок в список и удаление их из списка подтверждались). Это серьезное преимущество по сравнению с механизмами подсчета ссылок.

Списки ссылок, используемые в системе RMI языка Java, основаны на методе, описанном в [60]. Согласно этому методу, когда процесс P создает удаленную ссылку на объект, он посылает свой идентификатор скелетону объекта, который добавляет P в список ссылок. Когда приходит подтверждение, процесс создает заместителя объекта в своем адресном пространстве.

Передача ссылки другому процессу, то есть посылка копии заместителя, обрабатывается сходным образом. Когда процесс $P1$ посылает копию своего замес-

тителя объекта O процессу $P2$, процесс $P2$ сначала требует у скелетона объекта добавить $P2$ в список ссылок. Когда это сделано, процесс $P2$ встраивает заместитель в свое адресное пространство.

Проблемы могут возникнуть в том случае, если процесс $P1$ удалит свой заместитель до того, как $P2$ подаст запрос на вставку в список ссылок. В этом случае если запрос на удаление, который $P1$ посылает скелетону объекта, будет обработан раньше запроса на вставку от $P2$, список ссылок может оказаться пустым, то есть скелетон может прийти к неправильному выводу о возможности удаления объекта. Эта ситуация полностью аналогична ситуации из варианта с подсчетом ссылок, показанного на рис. 4.24, а, и может быть решена таким же способом.

Другим важным достоинством списка ссылок является простота сохранения его непротиворечивости при сбоях процессов. Скелетон объекта регулярно проверяет, все ли имеющиеся в списке процессы в порядке. Это делается посредством послышки сообщения `ring` с запросом, живы ли они еще и сохраняют ли ссылку на объект. Процесс должен немедленно ответить на это сообщение. Если ответ не получен, возможно, даже после нескольких попыток, скелетон удаляет процесс из списка.

Основной недостаток списка ссылок состоит в том, что он плохо масштабируется, особенно если скелетон должен отслеживать много ссылок. Один из способов сохранить управляемость состоит в том, чтобы указать скелетону на необходимость регистрировать ссылки на ограниченный срок. Если заместитель до истечения этого срока не подтверждает скелетону свою регистрацию, ссылка просто удаляется из списка. Этот подход называется также *арендой* (*lease*). Мы вернемся к аренде в главе 6.

4.3.4. Идентификация сущностей, на которые нет ссылок

Как уже было продемонстрировано с помощью рис. 4.22, набор сущностей в распределенных системах может содержать сущности со ссылками только друг на друга, то есть недоступные из корневого набора. Такие сущности должны удаляться. К сожалению, методы сборки мусора, описанные ранее, не в состоянии выявить подобные сущности.

Нам необходим метод, который позволит отследить все сущности в распределенной системе. В основном он состоит в проверке доступности сущностей из корневого набора с последующим удалением всех недоступных. Подобные методы носят общее название *трассировочной сборки мусора* (*tracing-based garbage collection*). В противоположность распределенным ссылкам, которые мы рассматривали ранее, трассировочной сборке мусора присущи проблемы масштабируемости, поскольку необходимо отслеживать все сущности распределенной системы.

Примитивная трассировка в распределенных системах

Чтобы разобраться в распределенной трассировочной сборке мусора, полезно обсудить трассировку в однопроцессорных системах. Наиболее простой подход

к однопроцессорной трассировке представлен сборщиками типа «помечай и подметай». Эти сборщики работают в два приема.

В ходе фазы пометки сущности отслеживаются по существующим цепочкам ссылок, исходящим из сущностей корневого набора. Каждая сущность, которая может быть достигнута оттуда, помечается, например, путем записи сущности в отдельную таблицу. Фаза подметания состоит из тщательной проверки памяти для локализации «ничейных» сущностей. Эти сущности считаются мусором, который должен быть удален.

Другой вариант работы сборщиков «помечай и подметай» — трехцветная пометка сущностей. Изначально каждая сущность, которую следует проверить, окрашена в белый цвет. К концу фазы пометки все сущности, доступные из корня, помечаются черным, а недоступные остаются белыми. Серый цвет используется для индикации хода фазы пометки. Сущность помечается серым, если она доступна, но ссылки, которые содержит эта сущность, еще нуждаются в проверке. Когда все сущности, на которые ссылается данная сущность, окрашиваются серым, сама она становится черной.

Распределенный вариант схемы «помечай и подметай» был реализован в системе Emerald, описанной в [220]. В Emerald каждый процесс запускает локальный сборщик мусора, причем все сборщики работают параллельно. Сборщики окрашивают заместители, скелетоны и сами объекты. Изначально все они помечаются белым цветом. Когда объект, расположенный в адресном пространстве процесса P , доступен из корневого набора, объект помечается серым. При этом все заместители, содержащиеся в этом объекте, также помечаются серым. Пометка заместителя серым цветом означает, что его локальный сборщик мусора указывает на необходимость проверки ссылок данного удаленного объекта ассоциированным с ним локальным сборщиком мусора.

Когда заместитель помечается серым, ассоциированному с заместителем скелетону посылается сообщение, маркирующее серым и его. Объект, к которому относится скелетон, помечается серым вслед за своим скелетоном. Далее, рекурсивно, все заместители в этом объекте помечаются серым. В этот момент скелетон и ассоциированный с ним объект помечаются черным, и соответствующее сообщение посылается всем ассоциированным с ним заместителям. Отметим, что хотя в этом подходе скелетону известны связанные с ним заместители, это не означает, что они достижимы из скелетона. Логически рассуждая, пара (заместитель, скелетон) — это исключительно однонаправленная связь от заместителя к скелетону.

Когда заместитель получает сообщение о том, что ассоциированный с ним скелетон стал черным, он тоже помечается черным. Другими словами, локальный сборщик мусора теперь знает, что удаленный объект, на который идет ссылка через заместителя, признан доступным.

Когда все локальные сборщики мусора заканчивают фазу пометки, каждый из них по отдельности собирает все белые объекты, считая их мусором. Фаза пометки завершается, когда все объекты, скелетоны и заместители приобретут либо белый, либо черный цвет. Удаление белых объектов означает также удаление ассоциированных с ними скелетонов, а также заместителей этих объектов.

Основной недостаток алгоритма «помечай и подметай» состоит в том, что он требует, чтобы граф доступности в течение обеих фаз оставался неизменным. Иначе говоря, выполнение программы, для которой изначально был создан процесс, должно быть временно приостановлено и система должна переключиться на сборку мусора. Для распределенных систем это означает, что все процессы должны быть синхронизированы. Каждый из них должен переключиться на сборку мусора, после чего все они смогут продолжить свою работу.

Такой сценарий, называемый также синхронизацией «все замри!», обычно недоступен для распределенных сборщиков мусора. Усовершенствование может заключаться в использовании инкрементных сборщиков мусора, которые позволяют программе работать, перемежая ее выполнение со сборкой мусора. К сожалению, подобные сборщики мусора в распределенных системах плохо масштабируются. Поскольку они работают параллельно с программой, которая изменяет граф доступности, объекты часто должны помечаться серым, а это приведет к распространению серых пометок по удаленным процессам. Результатом будет большой трафик сообщений, способный снизить общую производительность системы.

Трассировка в группах

Рассматривая проблемы масштабируемости, присущие многим трассировочным системам сборки мусора, разработчики придумали способ, в ходе которого процессы (содержащие объекты) в больших распределенных системах собираются в группы [255]. Сборка мусора производится внутри групп путем сочетания методов «помечай и подметай» и подсчета ссылок. Сосредоточимся на базовом алгоритме сборки мусора в группе процессов.

Группа — это просто набор процессов. Единственная причина работы с группами — масштабируемость. Базовая идея состоит в том, чтобы сначала собрать весь мусор в группе, включая цепочки ссылок, полностью находящиеся внутри группы. На следующем шаге выполняется переход к группе большего размера, содержащей несколько подгрупп, каждая из которых была очищена сборщиком мусора.

Чтобы понять трассировку в группе, предположим, что удаленные ссылки вновь реализованы в виде пар (заместитель, скелетон). Для каждого объекта существует лишь один скелетон, находящийся в адресном пространстве объекта, и множество заместителей, способных связываться со скелетоном. Скелетон поддерживает счетчик ссылок, описанный в пункте 4.3.2, который подсчитывает число ассоциированных с этим скелетоном заместителей. Процесс может иметь максимум один заместитель на каждый распределенный объект.

Как только будет сформирована группа процессов, вступает в работу базовый алгоритм сборки мусора, состоящий из пяти шагов.

1. Первичная маркировка, помечаются только скелетоны.
2. Распространение маркировки внутри процессов со скелетонов на заместителей.
3. Распространение маркировки между процессами с заместителей на скелетоны.
4. Стабилизация путем многократного повторения двух предыдущих шагов.
5. Удаление мусора.

До начала каждого из этих шагов важно понять, что означает маркировка сущности. Алгоритм относится в основном к маркировке заместителей и скелетонов. Важно отметить, что ни заместители, ни скелетоны не относятся к корневому набору.

Скелетоны могут помечаться как *нетвердые* или *твердые*, а заместители — как *отсутствующие*, *нетвердые* или *твердые*. Если скелетон помечен как *твердый*, значит, он достижим либо из заместителя процесса, не входящего в группу, либо из корневого объекта, входящего в группу, то есть объекта, содержащегося в корневом наборе и входящего в группу процесса. Скелетон, помеченный как *нетвердый*, доступен только из заместителя внутри группы. Скелетон может быть помечен только как *нетвердый* или *твердый*.

Заместитель, помеченный как *твердый*, достижим из объектов корневого набора. Если заместитель помечен как *нетвердый*, он достижим из скелетона, также помеченного как *нетвердый*. Такие скелетоны потенциально могут расположиться цепочкой, к которой не будет доступа у объектов корневого набора. Заместитель, помеченный как *отсутствующий*, недоступен ни из скелетонов, ни из объектов корневого набора. Только заместители, помеченные как *отсутствующие*, могут изменить маркировку на *твердые*. Как мы увидим, заместители, помеченные как *нетвердые*, сохраняют эту маркировку и далее.

Первый шаг состоит в пометке одних скелетонов. Скелетон помечается как *твердый* или *нетвердый* в зависимости от того, доступен ли он для заместителей, не входящих в группу. Эта доступность может быть легко проверена по счетчику ссылок скелетона. Значение этого счетчика показывает, сколько заместителей других процессов на него ссылаются. Некоторые из этих процессов входят в группу, другие не входят. Если среди них имеются заместители процессов, не входящих в группу, скелетон помечается как *твердый*. Просто пересчитав, сколько заместителей, ассоциированных со скелетоном, входят в группу, и вычтя это число из значения счетчика ссылок, мы можем решить, имеются ли у скелетона ассоциированные с ним заместители вне группы. Это приводит нас к следующему алгоритму.

1. Для каждого заместителя внутри группы уменьшается счетчик ссылок соответствующего скелетона на единицу, в том случае, если скелетон также находится внутри группы.
2. Скелетон внутри группы помечается как *нетвердый*, если его счетчик ссылок сброшен в ноль. В противном случае он доступен из-за границ группы и помечается как *твердый*.

Первый шаг иллюстрирует рис. 4.28, *а*, на котором все скелетоны, кроме одного, являются *нетвердыми*. Только один скелетон, который является *твердым*, доступен заместителю из-за границ группы.

Второй шаг включает в себя переключение каждого из процессов на свой сборщик мусора. Эти локальные сборщики работают независимо от глобального сборщика мусора. Единственное, что от них требуется в ходе работы, — это распространить маркировку со скелетонов на заместителей. Конкретнее, мы полагаем, что внутри одного процесса заместители достижимы из скелетона (отметим, что заместитель и скелетон принадлежат разным объектам). В результате ло-

кального распространения пометок заместители будут помечены как минимум так же, как и скелетон. Более того, если заместитель будет доступен объекту корневого набора, он будет помечен как *твердый*.

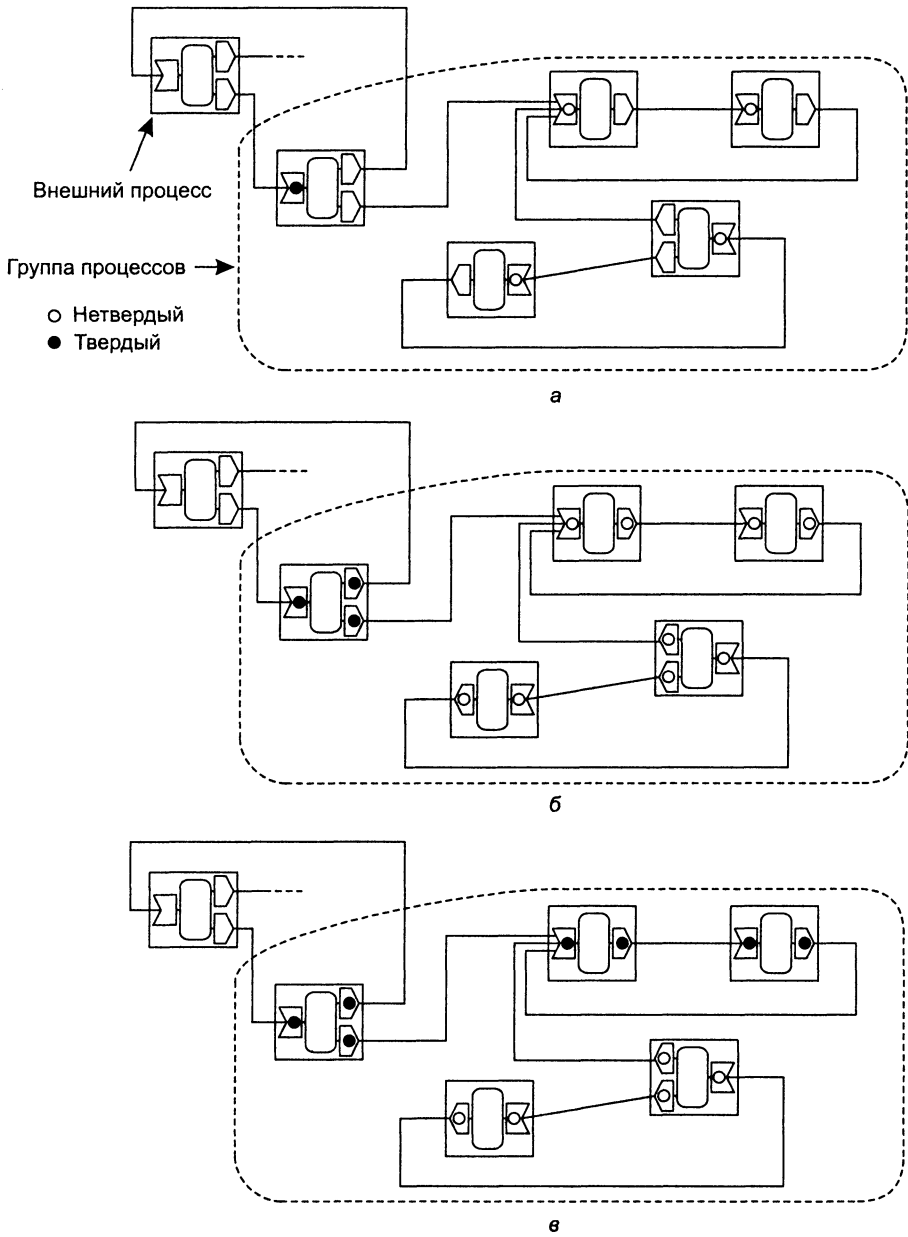


Рис. 4.28. Исходная маркировка скелетона (а). Ситуация после локального распространения маркировки в каждом процессе (б). Итоговая маркировка (в)

Локальное распространение внутри процесса *P* может происходить следующим образом. Изначально все заместители помечаются как *отсутствующие*. Локальный сборщик мусора начинает трассировку с набора, в который входят скелетоны, которые ранее были помечены как *твердые*, а также объекты корневого набора. *Твердые* пометки распространяются на все объекты (то есть на локальные объекты и заместители), доступные из этого набора. Второй проход осуществляется, начиная со скелетонов, помеченных как *нетвердые*. Если заместители, доступные из этого набора, помечены как *отсутствующие*, их маркировка меняется на *нетвердые*. Если достижимые заместители уже помечены как *твердые*, за ними сохраняется эта маркировка. Таким образом, после локального распространения каждый заместитель процесса получает одну из трех маркировок — *отсутствующий*, *нетвердый* или *твердый*. Рисунок 4.28, б отражает момент после локального распространения маркировок в группе, изображенной на рис. 4.28, а.

Третий шаг состоит из распространения пометок от заместителей к ассоциированным с ними скелетонам. Другими словами, пометки распространяются от процесса к процессу. Так, если заместитель был помечен как *твердый*, ассоциированному с ним скелетону посылается сообщение о том, что он также получает пометку *твердый*, если этот скелетон раньше был маркирован иначе. Сообщение посылается только тем скелетонам, которые входят в группу. Пометки *нетвердый* распространять не нужно: исходная разметка уже привела к тому, что все скелетоны группы имеют пометку *твердый* или *нетвердый*.

Четвертый шаг приводит к глобальному распространению пометок *твердый* предыдущего шага. Скелетон, скажем, процесса *P* может при этом поменять свою пометку с *нетвердый* на *твердый*. Эти изменения будут обусловлены тем фактом, что к скелетону получают доступ объекты, содержащиеся в корневом наборе удаленного процесса. Соответственно, пометка *твердый* сначала распространится локально, на заместителей процесса *P*, а затем глобально, на смежные процессы. Другими словами, второй и третий шаги будут повторяться до тех пор, пока пометки не перестанут распространяться глобально или локально. Когда ситуация стабилизируется, то есть прекратятся изменения пометок в процессах, входящих в группу, алгоритм перейдет к следующему шагу. В нашем примере результат повторений шагов 2 и 3 приведет к итоговой разметке, показанной на рис. 4.28, в.

Пятый и последний шаг состоит из удаления недоступных объектов, включая недоступные заместители, а также те заместители и скелетоны, которые были помечены как *нетвердые*. Важно отметить, что последние недоступны из-за пределов группы или из объектов корневого набора. Другими словами, *нетвердые* заместители и скелетоны ссылаются только друг на друга и могут быть удалены.

Сборка мусора может в действительности выполняться в виде вторичного эффекта локального распространения. Вместо явного удаления сущностей на последнем шаге скелетоны, помеченные как *нетвердые*, могут заменяться ссылкой на нуль. Таким образом, они могут быть обработаны позже, при повторном запуске локальных сборщиков мусора. Кроме того, если объект, ассоциированный с подобным скелетоном, становится недоступным, он также может быть обработан при повторном проходе. Если заместители, локально ссылающиеся на подоб-

ный объект, также становятся недоступными, они могут быть помечены как *отсутствующие* и обработаны точно так же. Поэтому для безопасного использования локальных сборщиков мусора следует обрабатывать заместители, помеченные как *отсутствующие*, после отправки сообщения об уменьшении счетчика скелетону удаленного процесса, ассоциированному с заместителем.

В иерархически организованных процессах в группах необходимо использовать более масштабируемое решение для распределенной сборки мусора. Основная идея состоит в том, что низкоуровневые группы собирают свой мусор, а анализ ссылок между группами оставляют на группы более высокого уровня. Поскольку количество объектов в низкоуровневых группах, которые необходимо отслеживать, уменьшается, группы верхнего уровня в основном работают со схожим количеством объектов, которые просто распределены по большей сети. Мы опустим детали, которые при необходимости можно найти в [255].

4.4. Итоги

Имена используются для ссылок на сущности. По сути, существует три основных типа имен. Адрес — имя точки доступа, ассоциированной с сущностью, часто называемый просто адресом сущности. Другой тип имени — идентификатор. Он имеет три свойства: каждая сущность имеет только один идентификатор, идентификатор указывает на единственную сущность и не может быть переназначен другой. И наконец, имена, удобные для восприятия, предназначены для использования людьми и представляют собой строку символов.

Имена организованы в пространства имен. Пространство имен может быть представлено в виде графа именования, в узлах которого расположены именуемые сущности, а метки на ребрах представляет собой имена, под которыми эти сущности известны. Узлы, из которых выходит несколько ребер, представляют собой наборы сущностей и известны также под названием направляющих узлов. Крупномасштабные графы именования часто организуются в виде корневых ациклических направленных графов.

Графы именования подходят для структурирования имен, удобных для восприятия. Доступ к сущности может осуществляться посредством пути. Разрешение имен — это процесс прохода по графу именования в поисках компонентов, входящих в путь, по одному за раз. Крупномасштабный граф именования реализуется путем распределения узлов по нескольким серверам имен. При разрешении пути путем обхода графа именования, если разыскиваемый узел находится на следующем сервере имен, то разрешение имен продолжается на нем.

Системы именования для имен, удобных для восприятия, невозможно использовать для высокоомобильных сущностей. Локализация мобильных сущностей более успешно может быть произведена с помощью не зависящих от местоположения идентификаторов. Существует четыре основных подхода к локализации мобильных сущностей.

Первый подход состоит в использовании широковебчательных или групповых рассылок. Идентификатор сущности посылается широковебчательной посылкой

каждому процессу в распределенной системе. Процесс предлагает точку доступа в ответ на предоставление ему адреса этой точки доступа. Очевидно, что этот подход имеет ограниченную масштабируемость.

Второй подход состоит в пересылке указателей. Каждый раз при перемещении сущности в другое место она оставляет за собой указатель, информирующий о том, куда она переместилась. Локализация сущности требует обхода цепочки пересылаемых указателей. Чтобы устранить длинную цепочку указателей, важно уменьшить длину цепи после перемещения.

Третий подход состоит в создании базы сущности. Каждый раз при перемещении сущности в другое место она уведомляет об этом свою базу. При локализации сущности первым делом о текущей локализации запрашивается ее база.

Четвертый подход состоит в построении иерархического дерева поиска. Сеть разбивается на неперекрывающиеся области (домены). Домены могут группироваться в домены верхнего уровня (неперекрывающиеся) и т. д. Существует один домен верхнего уровня, охватывающий всю сеть. Всякий домен на любом уровне имеет ассоциированный с ним направляющий узел. Если сущность находится в домене *D*, направляющий узел в домене следующего уровня содержит указатель на *D*. Направляющие узлы самого нижнего уровня содержат адреса сущностей. Направляющий узел самого верхнего уровня содержит сведения обо всех сущностях.

Сущности, локализация которых больше не нужна, могут быть удалены. Важная цель использования имен в распределенных системах — создание ссылок на сущности так, чтобы сущности, на которые нет ссылок, удалялись автоматически. Такая сборка мусора требует подсчета ссылок или трассировки.

В случае подсчета ссылок сущность просто считает количество созданных на нее ссылок. Когда счетчик достигнет нуля, сущность можно удалять. В отличие от подсчета ссылок можно также поддерживать список ссылок на процессы, ссылающиеся на сущность. Список ссылок более устойчив, чем счетчик ссылок, но имеет проблемы с масштабированием.

В методах трассировки все сущности прямо или косвенно ссылаются на заданный набор корневых сущностей, которые помечаются как доступные. Недоступные сущности удаляются. Распределенная трассировка трудна, поскольку требуется проверить все сущности в системе. Решения разнообразны, но в основном они основаны на традиционных сборщиках мусора, характерных для однопроцессорных систем.

Вопросы и задания

1. Приведите пример ситуации, в которой для реального доступа к сущности *E* ее адрес разрешается в другой адрес.
2. Зависит или нет от локализации URL-адрес <http://www.acme.org/index.html>? Как насчет адреса <http://www.acme.nl/index.html>?
3. Приведите несколько примеров правильных идентификаторов.
4. Как найти точку монтирования в большинстве UNIX-систем?

5. Jade — это распределенная файловая система, использующая отдельные пространства имен для каждого пользователя [371]. Другими словами, каждый пользователь имеет собственное закрытое пространство имен. Могут ли имена из подобных пространств имен при разделении ресурсов совместно использоваться несколькими пользователями?
6. Рассмотрим DNS. Для ссылки на узел N в поддомене, реализованном в другой нежели текущий домен зоне, должен быть определен сервер имен для этой зоны. Всегда ли необходимо включать запись о ресурсах для адреса этого сервера или иногда достаточно указать только его доменное имя?
7. Может ли идентификатор содержать информацию о сущности, которую он идентифицирует?
8. Опишите эффективную реализацию глобально уникальных идентификаторов.
9. Приведите пример того, как должен работать механизм свертывания в URL.
10. Объясните разницу между жесткой и мягкой ссылкой в UNIX-системах.
11. Высокоуровневые серверы имен в DNS (то есть серверы имен, реализующие узлы пространства имен DNS, близкие к корню) обычно не поддерживают рекурсивного разрешения имен. Можно ли достичь значительного повышения производительности, если они будут поддерживать рекурсивное разрешение имен?
12. Опишите, как можно использовать DNS для реализации ориентированного на наличие базы способа локализации мобильных хостов.
13. Существует особая форма локализации сущности, называемая индивидуальной рассылкой (anycasting), в которой служба определяется по IP-адресу [345]. Посылка запроса на адрес индивидуальной рассылки приводит к получению ответа от сервера, который реализует службу, определяемую по адресу индивидуальной рассылки. Опишите реализацию службы индивидуальной рассылки, основанную на иерархической службе локализации, описанной в пункте 4.2.4.
14. Где корень, если считать, что двухзвенный подход, основанный на наличии базы, является специализацией иерархической службы локализации?
15. Предположим, известно, что конкретная мобильная сущность практически никогда не выйдет за пределы домена D , а если это и случится, то быстро вернется обратно. Как можно использовать эту информацию для ускорения операции поиска в иерархической службе локализации?
16. Какое максимальное число локализуемых записей необходимо обновить в иерархической службе локализации глубины k при изменении местонахождения сущности?
17. Рассмотрим сущность, перемещающуюся из места A в место B с посещением нескольких промежуточных мест, в которых эта сущность находится относительно недолго. По прибытии в B она на время затихает. Изменение адреса в иерархической службе локализации может потребовать относительно долгого времени, поэтому в ходе посещения промежуточных мест такого изменения следует избегать. Как можно обнаружить сущность в промежуточном месте?

18. Как при передаче удаленной ссылки из процесса $P1$ в $P2$ при распределенном подсчете ссылок увеличить счетчик $P1$ независимо от $P2$?
19. Поясните, почему взвешенный подсчет ссылок эффективнее простого подсчета ссылок. Считайте связь надежной.
20. Возможна ли при подсчете поколений ссылок такая ситуация, когда объект будет сочтен мусором и уничтожен, поскольку невозможно определить, к какому поколению относится этот объект, хотя на него еще имеются ссылки?
21. Возможно ли при подсчете поколений ссылок, чтобы значение $G[i]$ оказалось меньше нуля?
22. При работе со списком ссылок, если после посылки команды `ping` процессу P не был получен ответ, процесс удаляется из списка ссылок на объект. Будет ли правильным удалить этот процесс?
23. Опишите очень простой способ определения момента достижения шага стабилизации при сборке мусора путем трассировки по [255].

Глава 5

Синхронизация

- 5.1. Синхронизация часов
- 5.2. Логические часы
- 5.3. Глобальное состояние
- 5.4. Алгоритмы голосования
- 5.5. Взаимное исключение
- 5.6. Распределенные транзакции
- 5.7. Итоги

В предыдущих главах мы рассматривали процессы и связь между процессами. Хотя связь и очень важна, это еще не все. Не менее важны вопросы взаимодействия и синхронизации процессов друг с другом. Взаимодействие частично обеспечивается именованием, которое позволяет процессам как минимум совместно использовать ресурсы и вообще сущности.

В этой главе мы в основном сосредоточимся на том, как процессы синхронизируются между собой. Так, например, важно, что несколько процессов не способны одновременно получать доступ к совместно используемым ресурсам, таким как принтеры, а должны взаимодействовать, позволяя друг другу временно получать эксклюзивный доступ к этому ресурсу. Другой пример. Несколько процессов могут время от времени нуждаться в соглашениях о порядке прохождения сообщений, о том, например, что сообщение $m1$ от процесса P будет отправлено раньше, чем сообщение $m2$ от процесса Q .

Как оказывается, синхронизация в распределенных системах часто значительно сложнее, чем синхронизация в однопроцессорных или мультипроцессорных системах. Проблемы и решения, которые обсуждаются в этой главе, являются по своей природе глобальными и случаются во множестве различных ситуаций в распределенных системах.

Мы начнем с обсуждения синхронизации, основанной на реальном времени, продолжим обсуждение синхронизацией, у которой всего один относительный параметр упорядочивания, не считая упорядочивания по абсолютному времени. Мы обсудим также понятие распределенного глобального состояния и то, как это состояние записывается в процессе синхронизации.

Во многих случаях важно, чтобы группа процессов назначила один из процессов координатором. Это обычно происходит в соответствии с алгоритмом голосования. Мы рассмотрим различные алгоритмы голосования в отдельном разделе.

Две отдельные темы, касающиеся синхронизации, — это взаимные исключения в распределенных системах и распределенные транзакции. Распределенные взаимные исключения позволяют совместно использовать ресурсы, предотвратив возможность одновременного доступа нескольких процессов. Распределенные транзакции делают нечто похожее, но посредством совершенствования механизма контроля за параллельным выполнением. Взаимные исключения и транзакции будут обсуждаться в отдельных разделах.

Существуют распределенные алгоритмы на любой «вкус и цвет» для распределенных систем самых разных типов. Множество примеров (и дополнительной информации) можно найти в [17, 421, 497]. Более формальный подход к изобилию существующих алгоритмов проводится в [277].

5.1. Синхронизация часов

В централизованных системах время определяется однозначно. Если процессу необходимо время, он организует системный вызов, и ядро выдает ему ответ. Если процесс *A* запрашивает время, а немного позже то же самое делает процесс *B*, значение, которое получит *B*, будет больше (или, возможно, равно) значению, полученному *A*. Оно не может быть меньше. В распределенных системах достижение договоренности о времени не столь тривиально.

Для того чтобы понять, к чему могут привести проблемы определения глобального времени, достаточно одного примера с программой `make` операционной системы UNIX. Обычно в UNIX большие программы разбиваются на несколько файлов исходного текста, и внесение изменений в один из этих файлов требует повторной компиляции не всех, а только одного из этих файлов. Такой подход значительно повышает производительность работы программистов. Например, если программа содержит 100 файлов, а исправлен был только один, достаточно перекомпилировать только этот файл.

Обычная методика применения программы `make` проста. Когда программист заканчивает свою работу, изменив все файлы с текстом программы, он запускает программу `make`, которая проверяет время последней модификации всех файлов с исходным текстом и всех объектных файлов программы. Если файл с исходным текстом `input.c` имеет время последней модификации 2151, а соответствующий ему объектный файл `input.o` — 2150, `make` считает, что `input.c` со времени создания `input.o` был изменен, и перекомпилирует его. С другой стороны, если `output.c` имеет время последней модификации 2144, а `output.o` — 2145, компиляция не требуется. Таким образом, программа `make` проверяет все файлы с исходным текстом в поисках тех из них, которые нуждаются в повторной компиляции, вызывая при необходимости компилятор.

Представим теперь, что произойдет, если в распределенной системе отсутствует глобальное соглашение о времени. Предположим, что `output.o`, как и ранее,

имеет отметку времени изменения 2144, а `output.c` после создания был модифицирован, но получил отметку времени 2143, потому что часы на машине, где он находится, немного запаздывают, как это показано на рис. 5.1. Тогда программа `make` не станет вызывать компилятор. В результате исполняемый файл программы будет содержать смесь объектных файлов из старых и новых исходных файлов. При исполнении это легко может привести к ошибкам, и программист сойдет с ума, пытаясь понять, что в его коде не так.

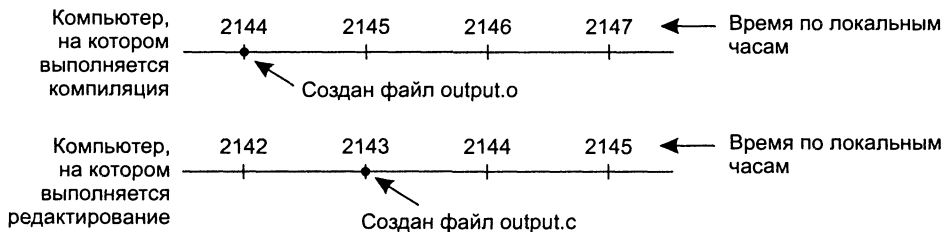


Рис. 5.1. Когда каждая из машин имеет собственные часы, событие, происшедшее позже другого, может быть отнесено к более раннему времени

Поскольку время лежит в основе человеческого мышления, а эффект от отсутствия синхронизации может быть, как мы сейчас показали, столь печален, изучение синхронизации можно начать с простого вопроса: Можно ли синхронизировать все часы в распределенной системе?

5.1.1. Физические часы

Почти все компьютеры имеют встроенные периодические процессы для подсчета времени. Несмотря на постоянное использование для этих устройств названия «часы», они не являются часами в обычном смысле этого слова. Пожалуй, более правильным словом было бы *таймер* (*timer*). Таймер компьютера — это обычно особым образом обработанный кристалл кварца. Находясь под напряжением, этот кристалл колеблется с постоянной частотой, которая зависит от свойств кристалла, таких как способ разрезания и величина напряжения. С каждым кристаллом ассоциированы два регистра — *счетчик* (*counter*) и *регистр хранения* (*holding register*). Каждое колебание кристалла уменьшает счетчик на единицу. Когда значение счетчика достигает нуля, генерируется прерывание и счетчик перезагружается из регистра хранения. Таким образом, можно запрограммировать таймер генерировать прерывания 60 раз в секунду или с любой другой желаемой частотой. Каждое прерывание вызывается одним *тиком таймера* (*clock tick*).

Когда система загружается в первый раз, она обычно просит пользователя ввести дату и время, которые пересчитываются в число тиков, начиная с определенной стартовой даты, и сохраняются в памяти. Многие компьютеры имеют специальную микросхему CMOS RAM с питанием от аккумулятора, в результате в них нет необходимости вводить время при каждой загрузке. После каждого тика таймера процедура обработки прерывания добавляет единицу к хранящемуся в памяти времени. Таким образом, часы (программные) сохраняют верное время.

Для единственного компьютера и единственных часов маленькая неточность этих часов не вызовет проблемы. Поскольку все процессы в машине используют одни и те же часы, они будут внутренне непротиворечивы. Так, например, если файл `input.c` имеет отметку времени 2151, а файл `input.o` — 2150, `make` перекомпилирует этот файл, даже если часы отстают на два деления и истинное время составляет соответственно 2153 и 2152. Все это, разумеется, относится к относительному времени.

В том случае, если мы перейдем к рассмотрению нескольких компьютеров, каждый из которых имеет собственные часы, картина изменится. Несмотря на то что частота каждого из кристаллов обычно весьма стабильна, невозможно гарантировать, что кристаллы на различных компьютерах будут иметь абсолютно одинаковую частоту. На практике, когда в систему входит n компьютеров, все n кристаллов работают с несколько отличающейся частотой, что ведет к постепенной потере синхронизации и выдаче часами при обращении к ним различных значений. Эта разница в показаниях часов называется *рассинхронизацией* часов (*clock skew*). В результате рассинхронизации часов программы, которые ожидают, что время, ассоциирующееся с файлом, объектом или сообщением, корректно и не зависит от машины, на которой оно определялось (то есть часы которой использовались), могут работать неправильно. Мы продемонстрировали это чуть выше на примере программы `make`.

В некоторых системах (это так называемые системы реального времени) точность часов необходима. Для таких систем нужно использовать внешние физические часы, причем для эффективности и защищенности желательно иметь несколько физических часов. Это вызывает две проблемы. Во-первых, как синхронизировать их с часами реального мира? Во-вторых, как синхронизировать эти часы друг с другом?

Перед тем как ответить на эти вопросы, давайте немного отступим от нашей темы, чтобы понять, как на самом деле измеряют время. Это вовсе не так просто, как можно было бы подумать. С тех пор как в XVII веке были изобретены механические часы, время стало определяться по астрономическим наблюдениям. Каждый день солнце встает на востоке, поднимается на максимальную высоту и заходит на западе. Момент подъема солнца на максимально возможную высоту называется *солнечным переходом* (*transit of sun*). Это событие происходит около полудня. Интервал между двумя последовательными солнечными переходами называется *солнечным днем* (*solar day*). День делится на 24 часа, каждый из которых содержит 3600 с. *Солнечная секунда* (*solar second*) определяется как $1/86\,400$ солнечного дня. Геометрические построения, необходимые для расчета солнечного дня, приведены на рис. 5.2.

В сороковых годах двадцатого века было установлено, что период обращения земли непостоянен. Земля замедляется из-за приливного трения и вязкости атмосферы. На основе изучения колец роста древних кораллов геологи полагают, что 300 миллионов лет назад в году было около 400 дней. Продолжительность года (время одного оборота вокруг солнца) при этом не изменилась, сутки просто стали длиннее. Вдобавок к этому медленному изменению существуют краткие вариации продолжительности суток, связанные, вероятно, с завихрениями

глубоко в земном ядре, состоящем из расплавленного железа. Эти открытия потребовали от астрономов расчета ситуации для множества дней с усреднением результатов перед делением их на 86 400. Получившееся значение было названо *средней солнечной секундой* (*mean solar second*).

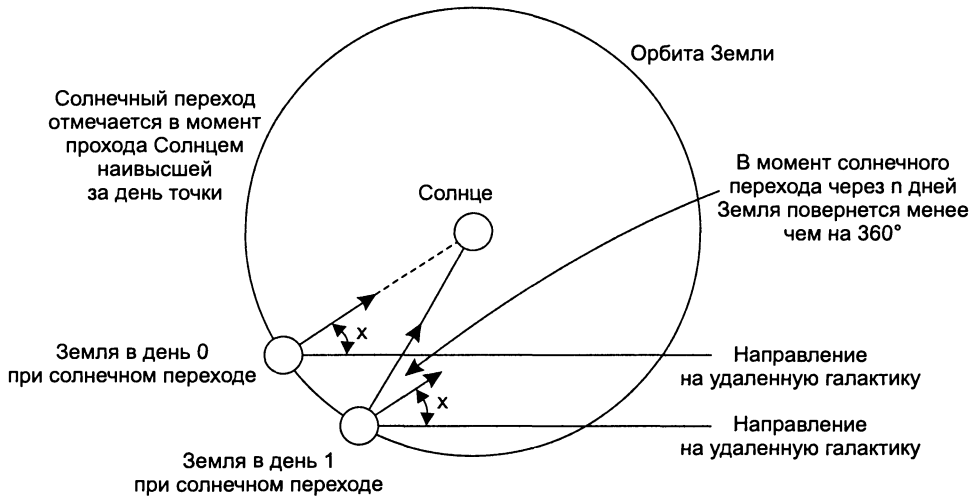


Рис. 5.2. Расчет среднего солнечного дня

После изобретения в 1948 году атомных часов появилась возможность еще более точного измерения времени, независимо от вращения и колебаний земли, путем подсчета переходов атома цезия-133. Физики переняли работу по хранению времени у астрономов и определили секунду как время, за которое атом цезия-133 совершит ровно 9 192 631 770 переходов. Выбор числа 9 192 631 770 сделал атомную секунду равной средней солнечной секунде в год ее расчета. В настоящее время около 50 лабораторий по всему миру имеют часы на цезии-133. Периодически каждая лаборатория сообщает в международное бюро мер и весов в Париже, сколько времени на их часах. Международное бюро усредняет их результаты и выдает *глобальное время по атомным часам* (*International Atomic Time, TAI*). TAI — это среднее время тиков часов на цезии-133, прошедшее с полуночи 1 января 1958 года (начала времен) и деленное на 9 192 631 770.

Хотя время TAI весьма стабильно и доступно каждому, кто в состоянии преодолеть проблемы, связанные с покупкой цезиевых часов, имеется серьезная проблема: 86 400 с TAI в настоящее время приблизительно на 3 мс меньше среднего солнечного дня (потому что средний солнечный день все время удлиняется). Используя TAI для хранения времени, надо понимать, что постепенно полдень будет наступать все раньше и раньше, пока, наконец, от утра почти ничего не останется. Это станет заметно населению, и мы столкнемся с ситуацией вроде той, что была в 1582 году, когда Папа Григорий XIII объявил, что 10 дней календаря пропускаются. Это событие привело к уличным бунтам, поскольку домовладельцы требовали у работников плату, а банкиры — процент за полный месяц, в то

время как работодатели отказывались платить работникам за те 10 дней, что они не работали. Но это были лишь небольшие столкновения. Протестантские страны принципиально отвергали любые папские декреты, не переходя на григорианский календарь еще 170 лет.

Международное бюро решило эту проблему, используя *потерянные секунды* (*leap seconds*) всякий раз, когда разница между временем TAI и солнечным временем возрастает до 800 мс (рис. 5.3). Эта коррекция позволила перейти к системе, основанной на постоянных секундах TAI, в которой, однако, соблюдается соответствие с периодичностью очевидно видимого движения солнца. Она называется *универсальным согласованным временем* (*Universal Coordinated Time, UTC*). UTC — это основа всей системы хранения времени в наши дни. Оно, по существу, заменило старый стандарт — *среднее время по Гринвичу* (*Greenwich mean time*), которое основывалось на астрономических наблюдениях и расчетах.



Рис. 5.3. Секунды TAI в отличие от солнечных секунд имеют постоянную длительность, поэтому для синхронизации времени UTC и времени TAI используют потерянные секунды

Большинство электрических компаний положили в основу измерения времени для своих 60-герцевых или 50-герцевых часов систему UTC, а когда международное бюро объявляет потерянную секунду, компании, производящие электроэнергию, меняют частоту на 61 или 51 Гц для 60 или 50 с, чтобы чуть подвести часы в той зоне, где они работают. Поскольку для компьютера 1 с — это заметный интервал, операционная система, которой нужно знать точное время за несколько лет, может обрабатывать объявленные потерянные секунды с помощью специального программного обеспечения (если только она не использует для определения времени линию питания, что обычно слишком неточно). Общее число потерянных секунд, введенных во время UTC, к настоящему времени составляет примерно 30.

Для предоставления времени UTC тем, кому необходимо точное время, национальный институт стандартного времени (National Institute of Standard Time, NIST) имеет коротковолновую радиостанцию с позывными WWV из форта Коллинз (Fort Collins), штат Колорадо. Радиостанция WWV широкоэвещательно рассылает короткий импульс в начале каждой секунды UTC. Точность самой радиостанции WWV составляет около ± 1 мс, но из-за различных атмосферных флуктуаций длина сигнала может меняться, так что на практике точность составляет не более ± 10 мс. В Англии станция MSF, работающая из Рерби (Rugby), район Варвикшир (Warwickshire), предоставляет похожую службу. Существуют также станции и в некоторых других странах.

Некоторые спутники Земли также предоставляют службы UTC. Рабочий спутник геостационарного окружения (Geostationary Environment Operational Satellite) может предоставлять время UTC с точностью до 0,5 мс, а некоторые другие — и с более высокой точностью.

Использование как коротковолновых радиосигналов, так и спутниковых служб требует точных сведений об относительном положении отправителя и получателя с целью компенсации задержки распространения сигнала. Радиоприемники для WWV, GEOS и других источников UTC имеются в продаже.

5.1.2. Алгоритмы синхронизации часов

Если одна машина имеет приемник WWV, нашей задачей является поддержание синхронизации с ней всех остальных машин. Если приемников WWV нет ни на одной из машин, то каждая из них отсчитывает свое собственное время, и нашей задачей будет по возможности синхронизировать их между собой. Для проведения синхронизации было предложено множество алгоритмов [113, 129, 240]. Их обзор имеется в [370].

Все алгоритмы имеют одну базовую модель системы, которую мы сейчас рассмотрим. Считается, что каждая машина имеет таймер, который инициирует прерывание H раз в секунду. Когда этот таймер срабатывает, обработчик прерываний добавляет единицу к программным часам, которые сохраняют число тиков (прерываний), начиная с какого-либо момента в прошлом, о котором была предварительная договоренность. Будем считать, что мы можем вызвать значение этих часов C . Конкретнее, когда время UTC равно t , значение часов машины p — $C_p(t)$. В идеальном мире мы можем считать, что $C_p(t) = t$ для всех p и всех t . Другими словами, dC/dt — точно единица.

Реальные таймеры не генерируют прерывания точно H раз в секунду. Теоретически таймер с $H = 60$ должен генерировать 216 000 тиков в час. На практике относительная ошибка, допустимая в современных микросхемах таймеров, составляет порядка 10^{-5} . Это означает, что конкретная машина может выдать значение в диапазоне от 215 998 до 216 002 тиков в час. Уточним. Пусть имеется константа ρ :

$$1 - \rho \leq dC/dt \leq 1 + \rho.$$

В этих пределах таймер может считаться работоспособным. Константа ρ определяется производителем и известна под названием *максимальной скорости дрейфа* (*maximum drift rate*). Отстающие, правильные и спешащие часы иллюстрирует рис. 5.4.

Если двое часов уходят от UTC в разные стороны за время Δt после синхронизации, разница между их показаниями может быть не более чем $2\rho \cdot \Delta t$. Если разработчики операционной системы хотят гарантировать, что никакая пара часов не сможет разойтись более чем на δ , синхронизация часов должна производиться не реже, чем каждые $\delta/2\rho$ с. Различные алгоритмы отличаются точностью определения момента проведения повторной синхронизации.

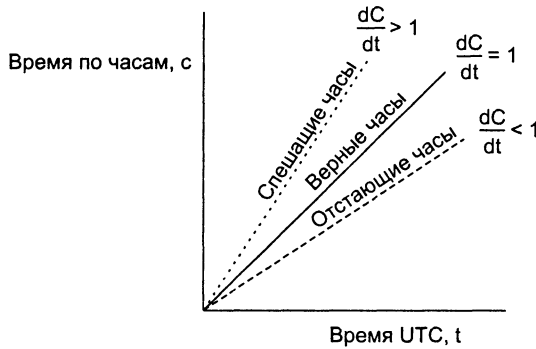


Рис. 5.4. Соотношение между временем по часам и временем UTC при различной частоте тиков

Алгоритм Кристиана

Давайте начнем с алгоритма, хорошо подходящего для систем, в которых одна из машин имеет приемник WWV, а наша задача состоит в синхронизации всех остальных машин по ней. Назовем машину с приемником WWV *сервером времени* (*time server*). Наш алгоритм [113] основан на работах Кристиана (Cristian) и более ранних. Периодически, гарантировано не реже, чем каждые $\delta/2$ с, каждая машина посылает серверу времени сообщение, запрашивая текущее время. Эта машина так быстро, как это возможно, отвечает сообщением, содержащим ее текущее время, C_{UTC} , как показано на рис. 5.5.

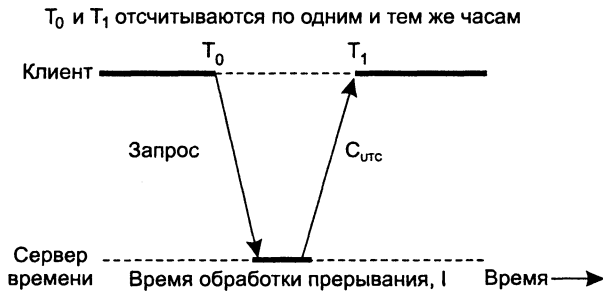


Рис. 5.5. Получение текущего времени с сервера времени

В качестве первого приближения, когда отправитель получает ответ, он может просто выставить свои часы в значение C_{UTC} . Однако такой алгоритм имеет две проблемы: главную и второстепенную. Главная проблема состоит в том, что время никогда не течет назад. Если часы отправителя спешат, C_{UTC} может оказаться меньше текущего значения C у отправителя. Простая подстановка C_{UTC} способна вызвать серьезные проблемы, связанные с тем, что объектные файлы, скомпилированные после того, как было изменено время, помечены временем более ранним, чем модифицированные исходные тексты, которые поправлялись до изменения времени.

Изменения могут вноситься постепенно. Один из способов таков. Предположим, что таймер настроен так, что он генерирует 100 прерываний в секунду. В нормальном состоянии каждое прерывание будет добавлять ко времени по 10 мс. При запаздывании часов процедура прерывания будет добавлять каждый раз всего по 9 мс. Соответственно, часы должны быть исправлены так, чтобы добавлять при каждом прерывании 11 мс вместо того, чтобы переключать все время одновременно.

Менее серьезная проблема состоит в том, что перенос ответного сообщения с сервера времени отправителю требует ненулевого времени. Хуже всего, что эта задержка может быть весьма велика и зависеть от загрузки сети. По Кристиану, метод решения проблемы состоит в измерении этой величины. Отправителю достаточно просто аккуратно записать интервал между посылкой запроса и приходом ответа. То и другое время (начальное T_0 и конечное T_1), измеряется по одним и тем же часам, а значит, интервал будет относительно точно измерен даже в том случае, если часы отправителя имеют некоторое расхождение с UTC.

В отсутствии какой-либо дополнительной информации наилучшим приближением времени прохождения сообщения будет $(T_1 - T_0)/2$. После получения ответа, чтобы получить приблизительное текущее время сервера, значение, содержащееся в сообщении, следует увеличить на это число. Если теоретическое минимальное время прохождения известно, следует рассчитать другие параметры времени.

Эта оценка может быть улучшена, если приблизительно известно, сколько времени сервер времени обрабатывает прерывание и работает с пришедшим сообщением. Обозначим время обработки прерывания I . Тогда величина интервала времени между T_0 и T_1 , затраченного на прохождение сообщения, будет равна $T_1 - T_0 - I$, и лучшей оценкой времени на прохождение сообщения в одну сторону будет половина этой величины. Существуют системы, в которых сообщение из A в B постоянно движется по одному маршруту, а из B в A — по другому. Они будут иметь другое время прохождения, но мы здесь не будем рассматривать эти случаи.

Для повышения точности Кристиан предложил производить не одно измерение, а серию. Все измерения, в которых разность $T_1 - T_0$ превосходит некоторое пороговое значение, отбрасываются как ставшие жертвами перегруженной сети, а потому недостоверные. Оценка делается по оставшимся замерам, которые могут быть усреднены для получения наилучшего значения. С другой стороны, сообщение, пришедшее быстрее всех, можно рассматривать как самое точное, поскольку оно предположительно попало в момент наименьшего трафика и потому наиболее точно отражает чистое время прохождения.

Алгоритм Беркли

В алгоритме Кристиана сервер времени пассивен. Прочие машины периодически запрашивают у него время. Все, что он делает, — это отвечает на запросы. В операционной системе UNIX разработкой университета Беркли (Berkeley) принят прямо противоположный подход [188]. Здесь сервер времени (а на самом деле демон времени) активен, он время от времени опрашивает каждую из машин, какое

время на ее часах. На основании ответов он вычисляет среднее время и предлагает всем машинам установить их часы на новое время или замедлить часы, пока не будет достигнуто необходимое уменьшение значения времени на сильно ушедших вперед часах. Этот метод применим для систем, не имеющих машин с приемником WWV. Время демона может периодически выставляться вручную оператором. Это продемонстрировано на рис. 5.6.

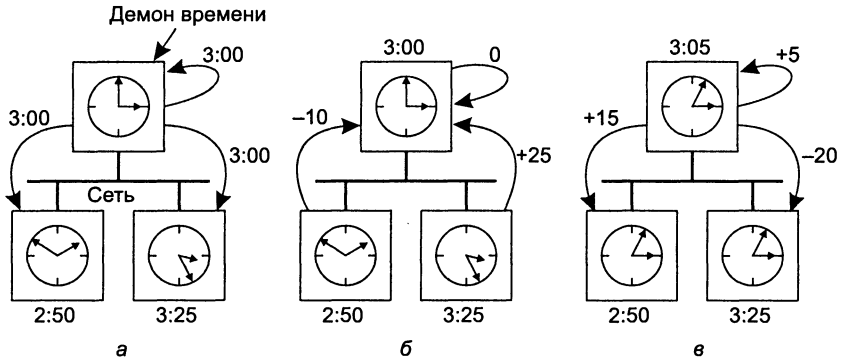


Рис. 5.6. Демон времени запрашивает у всех остальных машин значения их часов (а). Ответы машин (б). Демон времени сообщает всем, как следует подвести их часы (в)

В 3:00 демон времени сообщает всем машинам свое время и запрашивает эти машины об их времени (рис. 5.6, а). Затем демон времени получает ответ о том, насколько спешат или отстают часы машин от часов демона (рис. 5.6, б). Зная эти числа, демон времени вычисляет среднее и сообщает каждой из машин, как ей следует подвести свои часы (рис. 5.6, в).

Усредняющие алгоритмы

Оба ранее описанных алгоритма сильно централизованы с неизбежно вытекающими отсюда недостатками. Известны также и децентрализованные алгоритмы. Один из классов алгоритмов децентрализованной синхронизации часов работает на основе деления времени на синхронизационные интервалы фиксированной продолжительности. В этом случае i -й интервал начинается в момент времени $T_0 + iR$ и продолжается до $T_0 + (i + 1)R$, где T_0 — заранее согласованный прошедший момент, а R — параметр системы. В начале каждого интервала каждая машина производит широковещательную рассылку значения текущего времени на своих часах. Поскольку часы на различных машинах идут с немного различной скоростью, эти широковещательные рассылки будут сделаны не одновременно.

После рассылки машиной своего времени она запускает локальный таймер и начинает собирать все остальные широковещательные пакеты в течение некоторого интервала S . Когда будут собраны все широковещательные пакеты, запускается алгоритм вычисления по ним нового времени. Простейший алгоритм состоит в усреднении значений всех остальных машин. Незначительные его вариации заключаются, во-первых, в отбрасывании m самых больших и m самых маленьких значений и усреднении оставшихся. Отбрасывание крайних значений можно рассматривать как самозащиту от m неправильных часов, посылающих ерунду.

Другая вариация состоит в том, чтобы попытаться скорректировать каждое из сообщений, добавляя к ним оценку времени прохождения от источника. Эта оценка может быть сделана на основе знания топологии сети или измерением времени прохождения эха.

Дополнительные алгоритмы синхронизации часов рассматриваются в [274, 369, 429]. Один из наиболее часто используемых алгоритмов в Интернете — это *протокол сетевого времени (Network Time Protocol, NTP)*, описанный в [293]. NTP известен тем, что позволяет обеспечить точность (глобальную) 1–50 мс. Такая точность достигается путем использования усовершенствованных алгоритмов синхронизации часов, дальнейшие их усовершенствования описаны в [294].

Множественные внешние источники точного времени

Для систем, которым необходима особо точная синхронизация по UTC, можно предложить использование нескольких приемников WWV, GEOS или других источников UTC. Однако из-за врожденной неточности самих источников времени и флуктуаций на пути сигнала лучшее, что могут сделать операционные системы, — это установить интервал, в который попадает UTC. В основном различные источники точного времени будут порождать различные диапазоны, и машины, к которым они присоединены, должны прийти к какому-то общему соглашению.

Чтобы достичь этого соглашения, каждый процессор с источником UTC может периодически делать широковещательную рассылку своих данных, например, точно в начале каждой минуты по UTC. Ни один процессор не получит пакеты значений времени мгновенно. Что еще более печально, задержка между посылкой и приемом будет зависеть от длины кабеля и числа маршрутизаторов, через которые должен будет пройти пакет. Эти значения различны для каждой пары (источник UTC, процессор). Будут также играть свою роль и другие факторы, такие как задержка по причине коллизий, когда несколько машин попытаются послать что-то по Ethernet в один и тот же момент. Более того, если процессор занят обработкой предыдущего пакета, он, возможно, не сумеет просмотреть пакет значений времени за оговоренное число миллисекунд, что внесет дополнительную неясность.

5.1.3. Использование синхронизированных часов

За прошедшие несколько лет стали легкодоступными необходимая аппаратура и программное обеспечение, предназначенные для синхронизации часов в глобальном масштабе (то есть во всем Интернете). Благодаря этим новым технологиям можно синхронизировать миллионы часов с точностью до нескольких миллисекунд по UTC. Стали появляться новые алгоритмы, использующие эти синхронизированные часы. Один из примеров касается того, как добиться доставки серверу не более одного сообщения, даже в случае сбоя [269]. Традиционный подход состоит в том, что каждому сообщению приписывается уникальный номер сообщения, а каждый сервер сохраняет все номера сообщений, которые он принял, чтобы можно было отличить новое сообщение от повторной посылки. Проблема этого алгоритма состоит в том, что при сбое и перезагрузке сервера он теряет эту

таблицу номеров сообщений. Неясно также, сколько времени хранить эти номера сообщений.

С использованием времени традиционный алгоритм модифицируется следующим образом. Итак, каждое сообщение имеет идентификатор связи (выбранный отправителем) и метку времени. Для каждого соединения сервер заносит в таблицу последнюю полученную метку времени. Если какое-либо входящее сообщение имеет отметку времени меньшую, чем отметка, сохраненная в таблице для этого соединения, сообщение считается дубликатом и не рассматривается.

Чтобы можно было удалить старую метку времени, каждый сервер постоянно поддерживает глобальную переменную G , определяемую следующим образом:

$$G = \text{CurrentTime} - \text{MaxLifetime} - \text{MaxClockSkew}.$$

Здесь MaxLifetime — максимальное время жизни сообщения, а MaxClockSkew указывает на то, как далеко от времени UTC могут уйти часы. Любая метка времени старше G может быть легко удалена из таблицы, поскольку все сообщения старше G уже прошли. Если входящее сообщение имеет неизвестный идентификатор связи, оно будет принято, если его метка времени более ранняя, чем G , и отвергнуто, если она более поздняя, чем G , поскольку всякое более позднее сообщение — это, несомненно, дубликат. В действительности G — это сумма номеров всех старых сообщений. Каждые ΔT текущее время сбрасывается на диск.

В случае сбоя и последующей перезагрузки сервера он загружает значение G из файла, сохраненного на диске, и увеличивает его в ходе периода обновления, ΔT . Любое входящее сообщение с отметкой времени старше G — это дубликат. В результате каждое сообщение, которое могло быть принято до сбоя, теперь отвергается. Некоторые новые сообщения могут быть отвергнуты ошибочно, но при соблюдении всех условий алгоритм поддерживает семантику «не более одного сообщения».

Вдобавок к этому алгоритму в [269] также описано, как можно использовать синхронизированные часы для достижения непротиворечивости кэша, как использовать послышки тайм-аута для аутентификации распределенных систем и как обработать обязательства по атомарным транзакциям. Мы обсудим некоторые из этих алгоритмов в следующих пунктах. По мере улучшения синхронизации таймеров для них, несомненно, будут найдены и другие приложения.

5.2. Логические часы

Во многих случаях необходимо, чтобы все машины договорились об использовании одного и того же времени. Не столь уж и важно, чтобы это время совпадало с истинным временем, которое каждый час объявляют по радио. Для работы программы `make`, например, достаточно, чтобы все машины считали, что сейчас 10:00, даже если на самом деле сейчас 10:02. Так, для некоторого класса алгоритмов подобная внутренняя непротиворечивость имеет гораздо большее значение, чем то, насколько их время близко к реальному. Для таких алгоритмов принято говорить о *логических часах* (*logical clocks*).

В своей классической статье (см. [249]) Лампорт (Lamport) показал, что хотя синхронизация часов возможна, она не обязательно должна быть абсолютной. Если два процесса не взаимодействуют, нет необходимости в том, чтобы их часы были синхронизированы, поскольку отсутствие синхронизации останется незамеченным и не создаст проблем. Кроме того, он указал, что обычно имеет значение не точное время выполнения процессов, а его порядок. В примере с программой `make`, который мы приводили в предыдущем разделе, нас интересовало, чтобы файл `input.c` был более старым или более новым, чем `input.o`, а не абсолютное время их создания.

В этом пункте мы обсудим алгоритм Лампорта, предназначенный для синхронизации логических часов. Кроме того, мы рассмотрим расширение метода Лампорта, векторные отметки времени. Лампорт сделал дополнения к своей работе в [251].

5.2.1. Отметки времени Лампорта

Для синхронизации логических часов Лампорт определил отношение под названием «происходит раньше». Выражение $a \rightarrow b$ читается как « a происходит раньше b » и означает, что все процессы согласны с тем, что первым происходит событие a , а позже — событие b . Отношение «происходит раньше» непосредственно исполняется в двух случаях.

- ◆ Если a и b — события, происходящие в одном и том же процессе, и a происходит раньше, чем b , то отношение $a \rightarrow b$ истинно.
- ◆ Если a — это событие отсылки сообщения одним процессом, а b — событие получения того же сообщения другим процессом, то отношение $a \rightarrow b$ также истинно. Сообщение не может быть получено до отсылки или даже в тот же самый момент, когда оно было послано, поскольку для пересылки необходимо конечное ненулевое время.

Отношение «происходит раньше» — это транзитивное отношение, то есть в случае, если $a \rightarrow b$ и $b \rightarrow c$, выполняется условие $a \rightarrow c$. Если два события, x и y , происходят в разных процессах, которые не обмениваются сообщениями (ни непосредственно, ни через третий процесс), то отношение $x \rightarrow y$ не истинно, впрочем, как и $y \rightarrow x$. Такие события называются *параллельными* (*concurrent*). В данном случае это означает только то, что никто не может (или не хочет) знать о том, где и какое из этих событий произошло.

Нам нужен способ измерения времени каждого события, который позволил бы поставить в соответствие каждому событию a отметку времени $C(a)$, которая подошла бы всем процессам. Эти отметки времени должны быть такими, чтобы при $a \rightarrow b$ соблюдалось соотношение $C(a) < C(b)$. Перефразируя условие, которое мы ранее установили, если a и b — два события одного процесса и a происходит раньше, чем b , то $C(a) < C(b)$. Например, если a — это посылка сообщения одним процессом, а b — прием этого сообщения другим процессом, то $C(a)$ и $C(b)$ должны быть назначены так, чтобы значения $C(a)$ и $C(b)$ соответствовали отношению $C(a) < C(b)$. Кроме того, время по часам, C , всегда идет вперед (увеличивается),

а назад — никогда (не уменьшается). Коррекция времени должна производиться только путем добавления к нему положительного значения, а не его вычитанием.

Давайте теперь рассмотрим алгоритм Лампорта, применяемый для присвоения времен событиям. Обсудим три процесса, описанные на рис. 5.7, а. Процессы запущены на различных машинах, каждая из которых имеет собственные часы и скорость работы. Как можно увидеть по рисунку, когда часы процесса 0 поставят отметку времени 6, в процессе 1 они покажут 8, а в процессе 2 — 10. Каждые часы идут с постоянной скоростью, но эти скорости различны из-за разницы в кристаллах.

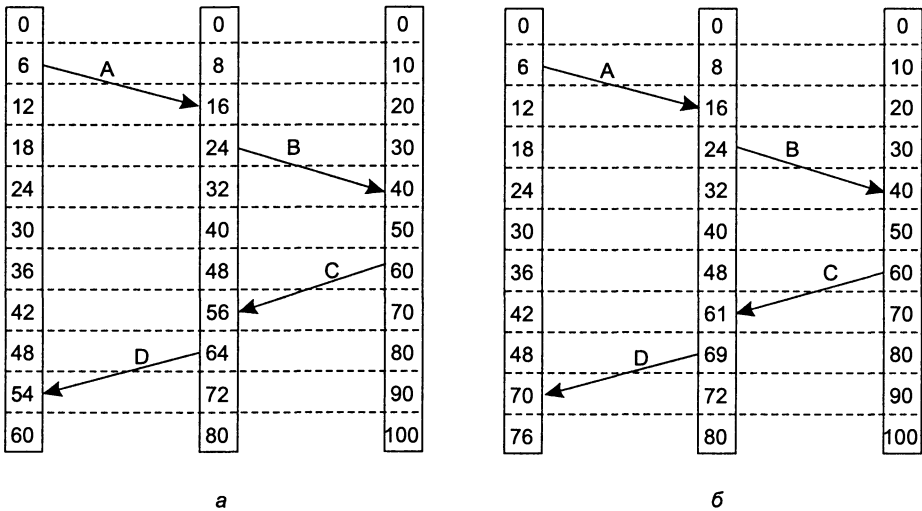


Рис. 5.7. Три процесса, каждый с собственными часами, которые ходят с разной скоростью (а). Подстройка часов по алгоритму Лампорта (б)

В момент 6 процесс 0 посылает сообщение А процессу 1. Сколько времени это сообщение проведет в пути, зависит от того, каким часам мы верим. В любом случае, когда оно придет в процесс 1, его часы будут показывать 16. Если сообщение будет содержать время отправки, 6, процесс 1 сочтет, что на пересылку ушло 10 тиков. Это значение вполне разумно. В соответствии с этими рассуждениями сообщение В от процесса 1 процессу 2 будет доставлено за 16 тиков, это вновь разумное значение.

Перейдем к самой забавной части наших рассуждений. Сообщение С от процесса 2 процессу 1 будет послано на отметке 60 и достигнет цели на отметке 56. Сообщение D от процесса 1 процессу 0 будет послано на отметке 64 и достигнет цели на отметке 54. Эти цифры абсолютно невозможны. Это та ситуация, которую мы должны предотвратить.

Решение, найденное Лампортом, прямо вытекает из отношения «происходит раньше». Поскольку С посылается на отметке 60, оно должно достичь цели на отметке 61 или позднее. Поэтому каждое сообщение содержит время отправки по часам отправителя. Когда сообщение достигает цели, но часы получателя по-

казывают время, более раннее чем время отправки, получатель быстро подводит свои часы так, чтобы они показывали время на единицу большее времени отправки. На рис. 5.7, б мы видим, что теперь сообщение C приходит на отметке 61. Таким же образом сообщение D приходит теперь на отметке 70.

С одним небольшим дополнением этот алгоритм удовлетворяет нашим требованиям по глобальному времени. Это добавление состоит в том, что между любыми двумя событиями часы должны «протикать» как минимум один раз. Если процесс посылает или принимает в коротком сеансе два сообщения, он должен сделать так, чтобы его часы успели протикать один раз (как минимум) в промежутке между сообщениями.

В некоторых случаях желательно удовлетворять дополнительному требованию: никакие два сообщения не должны происходить одновременно. Чтобы выполнить это требование, мы можем добавить номер процесса, в котором происходят события, справа от метки времени и отделить его от целой части десятичной точкой. Таким образом, если в процессах 1 и 2 в момент времени 40 происходят события, то первый из них будет происходить в 40.1, а второй в 40.2.

Используя этот способ, мы получаем возможность указания времени для всех событий в распределенной системе, подпадающих под перечисленные ниже условия.

- ♦ Если a происходит раньше b в одном и том же процессе, то $C(a) < C(b)$.
- ♦ Если a и b представляют собой отправку и получение сообщения, соответственно $C(a) < C(b)$.
- ♦ Для всех различимых событий a и b выполняется соотношение $C(a) \neq C(b)$.

Этот алгоритм дает нам возможность полностью упорядочить все события в системе. В отличие от этого множество других распределенных алгоритмов для устранения неясностей требуют упорядоченности, так что этот алгоритм широко цитируется в литературе.

Пример — полностью упорядоченная групповая рассылка

В качестве области приложения алгоритма Лампорта рассмотрим ситуацию, когда необходима репликация базы данных на нескольких сайтах. Например, для повышения производительности запросов банк может разместить копию базы данных по счетам в двух разных местах, скажем в Нью-Йорке и Сан-Франциско. Запросы всегда пересылаются к ближайшей копии. Ценой быстрого ответа на запрос отчасти являются высокие затраты на обновление, поскольку каждая операция обновления должна быть передана каждой из реплик.

На самом деле в отношении изменений существуют более точные требования. Предположим, клиент в Сан-Франциско хочет добавить \$100 на свой счет, на котором в настоящее время лежит \$1000. В то же самое время сотрудник банка в Нью-Йорке начинает обновлять счета пользователей, увеличивая сумму на них на процент по вкладу — 1 %. Оба изменения должны быть внесены в обе копии базы данных. Однако из-за задержек в базовой сети передачи данных обновления могут дойти в ином порядке (рис. 5.8).

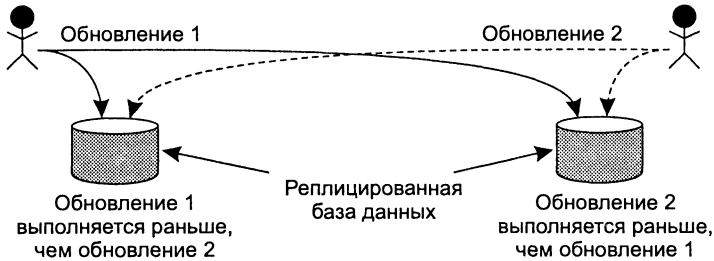


Рис. 5.8. Обновление реплицированной базы данных, после которого она приходит в противоречивое состояние

Пользовательская операция обновления будет выполнена в Сан-Франциско раньше обновления для начисления процентов. В Нью-Йорке же наоборот, копия счета клиента будет обновлена после получения одного процента прибыли, а уж после этого по ней будет проведена операция с депозитом в \$100. Соответственно, база данных в Сан-Франциско будет иметь общую сумму счета \$1,111, а база данных в Нью-Йорке — \$1,110.

Проблема, с которой мы столкнулись, состоит в том, что две операции обновления должны быть выполнены в одинаковом порядке в каждой из копий. Хотя очередность обработки разная, с точки зрения непротиворечивости она абсолютно не важна. Важно, чтобы обе копии были абсолютно одинаковыми. Обычно такого рода ситуации требуют *полностью упорядоченной групповой рассылки* (*totally-ordered multicasting*), то есть такой операции групповой рассылки, при которой все сообщения доставляются всем получателям с одинаковой очередностью. Для реализации такой рассылки абсолютно распределенным методом могут использоваться отметки времени Лампорта.

Рассмотрим группу процессов, посылающих друг другу сообщения путем групповой рассылки. Каждое из сообщений всегда имеет отметку времени с текущим временем (логическим) отправителя. Если сообщение передается путем групповой рассылки, оно по идее приходит также и отправителю. Кроме того, мы предполагаем, что сообщения каждого из отправителей принимаются в том порядке, в котором они посылались, и что ни одно сообщение не теряется.

Когда процесс принимает сообщение, оно попадает в локальную очередь в порядке, определяемом отметкой времени. Получатель рассылает групповое подтверждение остальным процессам. Заметим, что если для корректировки локальных часов мы используем алгоритм Лампорта, отметка времени на полученном сообщении должна быть меньше, чем на подтверждении.

Интересный аспект такого подхода состоит в том, что все процессы будут иметь одну и ту же копию локальной очереди. Каждое сообщение рассылается всем процессам, *включая подтверждения*, а предполагает ответ от всех процессов. Напомним также, что мы решили, что сообщения принимаются в порядке их отправки. Каждый процесс помещает полученное сообщение в свою локальную очередь в соответствии с отметкой времени на этом сообщении. Часы Лампорта гарантируют, что никакие два сообщения не могут иметь одинаковую отметку времени, а также то, что отметки времени отражают общий порядок сообщений.

Процесс может доставить поставленное в очередь сообщение приложению, которое работает, только если это сообщение находится на вершине очереди и его получение подтверждено всеми остальными процессами. В этот момент сообщение удаляется из очереди и обрабатывается приложением, а соответствующие ему подтверждения просто удаляются. Поскольку каждый процесс имеет такую же копию очереди, все сообщения доставляются повсюду в одинаковом порядке. Другими словами, мы создали полностью упорядоченную групповую рассылку.

5.2.2. Векторные отметки времени

Отметки времени Лампорта приводят к ситуации, когда все события в распределенной системе имеют единую последовательность, такую, что если событие a происходит раньше события b , то a оказывается перед b , то есть $C(a) < C(b)$.

Однако при помощи отметок времени Лампорта ничего нельзя сказать о взаимосвязи между двумя сообщениями a и b , просто сравнивая их временные отметки, соответственно $C(a)$ и $C(b)$. Другими словами, если $C(a) < C(b)$, это не обязательно означает, что события a действительно произошло раньше, чем b . Для этого необходимо что-то еще.

Чтобы понять это, рассмотрим систему рассылки сообщений. Один из наиболее популярных примеров подобной системы сообщений — это служба электронных досок сообщений в Интернете, *сетевые новости* (например, описанные в [110]). Пользователи, а следовательно, процессы, выбирают определенные дискуссионные группы. Послания в такую группу, будь то письма или реакция на другие письма, рассылаются групповой рассылкой всем членам группы. Чтобы гарантировать, что ответы присылаются после писем, на которые в них отвечали, мы можем попытаться использовать описанную выше схему полностью упорядоченной групповой рассылки. Однако эта схема не означает, что если сообщение B пришло после сообщения A , значит, B представляет собой отклик на то, что было послано в сообщении A . На самом деле они могут быть абсолютно независимыми друг от друга. Полностью упорядоченная групповая рассылка слишком строга для этого случая.

Проблема состоит в том, что отметки времени Лампорта не в состоянии уловить *причинно-следственную связь*. В нашем примере получение письма из соображений причинно-следственной связи всегда предваряет отсылку ответа на него. Соответственно, если в группе процессов поддерживаются отношения причинно-следственной связи, то получение ответа на письмо всегда должно следовать за получением этого письма. Не более, но и не менее. Если два письма или ответы на них независимы друг от друга, порядок их получения обычно не важен.

Причинно-следственная связь может быть соблюдена посредством *векторных отметок времени* (*vector timestamps*). Векторная отметка времени $VT(a)$ присвоенная событию a , имеет следующее свойство: если $VT(a) < VT(b)$ для события b , значит, событие a является причинно предшествующим событию b . Векторные отметки времени создаются путем приписыванию каждому процессу P_i вектора V_i , с перечисленными ниже свойствами.

- ♦ $V_i[i]$ — это число событий, которые произошли с процессом P_i к настоящему времени.
- ♦ Если $V_i[j] = k$, то процесс P_i знает, что с процессом P_j произошло k событий.

Первое свойство поддерживается путем увеличения $V_i[i]$ на единицу при каждом новом событии, происходящем в процессе P_j . Второе свойство поддерживается вложением векторов в посылаемые сообщения. Так, когда процесс P_i посылает сообщение m , он пересылает вместе с ним и его текущий вектор как отметку времени vt .

Таким образом, получатель оказывается информированным о номере сообщения, которое вызвало активность процесса P_i . Более важно, однако, что получатель уведомляется о том, сколько сообщений от *других* процессов должны прийти к нему до того момента, как процесс P_j посылает ему сообщение m . Другими словами, отметка времени vt в сообщении m говорит получателю, сколько событий в других процессах должны предвять m и с каким из них у сообщения m может быть причинно-следственная связь. Когда процесс P_j получает m , это побуждает его установить каждый элемент $V_j[k]$ в собственном векторе в $\max\{V_j[k], vt[k]\}$. Вектор теперь отражает число сообщений, которые должен получить процесс P_j и которые, как он видел, предшествовали посылке m . Затем элемент $V_j[i]$ увеличивается на единицу, отражая факт получение сообщения m , как следующего сообщения от процесса P_i [374].

Векторная отметка времени может использоваться для доставки сообщений только без нарушения причинно-следственной связи. Рассмотрим далее пример с электронной доской объявлений. Когда процесс P_i отправляет письмо, он осуществляет множественную рассылку этого письма в виде сообщения a с пометкой времени $vt(a)$, равной V_i . Когда другой процесс, P_j , получает a , он исправляет свой собственный вектор, устанавливая $V_j[i] > vt(a)[i]$.

Допустим теперь, что P_j посылает ответ на это письмо. Он делает это посредством множественной рассылки сообщения r с отметкой времени $vt(r)$, равной V_j . Отметим, что $vt(r)[i] > vt(a)[i]$. Если считать связь надежной, оба эти сообщения — a , содержащее письмо, и r , содержащее ответ, — в конце концов достигнут другого процесса, P_k . Поскольку мы не делали никаких предположений относительно очередности доставки сообщений, сообщение r может прийти процессу P_k раньше, чем сообщение a . Получив r , P_k проверяет отметку времени $vt(r)$ и решает отложить доставку до того момента, пока не будут приняты все сообщения, которые по соображениям причинности идут перед r . Если подробнее, сообщение r будет доставлено только в случае выполнения двух условий:

- ♦ $vt(r)[j] = V_k[j] + 1$.
- ♦ $vt(r)[i] < V_k[i]$ для всех $i \neq j$.

Первое условие означает, что r — следующее сообщение, получаемое P_k от процесса P_j . Второе условие означает, что P_k видит все те же сообщения, которые видел процесс P_j , отправляя сообщение r . В частности, это означает, что процесс P_k уже получил сообщение a .

Замечание по упорядоченной доставке сообщений

Некоторые системы промежуточного уровня, например ISIS и ее преемница Nogus [54], предоставляют поддержку полностью упорядоченной и упорядоченной в плане причинности (надежной) групповой рассылки. В свое время шли споры о том, должна ли эта поддержка быть реализована как часть уровня обмена сообщениями или упорядочиванием должны заниматься приложения (см., к примеру, [59, 99]).

В том чтобы поручить выяснение очередности сообщений уровню обмена сообщениями, имеется две основные проблемы. Во-первых, поскольку уровень обмена сообщениями не знает, что на самом деле содержится в сообщениях, возможен учет лишь *потенциальной* причинности. Так, например, два сообщения от одного отправителя, абсолютно независимые, на уровне обмена сообщениями всегда будут считаться имеющими причинно-следственную связь. Подобный подход слишком ограничен и может привести к проблемам с эффективностью.

Вторая проблема состоит в том, что не всякую причинно-следственную связь удастся выявить. Рассмотрим еще раз службу новостей. Представим себе, что Алиса написала письмо. Если затем она позвонит Бобу и обсудит с ним написанное, Боб может написать ей в ответ другое письмо, не читая послания Алисы в группе новостей. Другими словами, между письмом Боба и письмом Алисы имеется причинно-следственная связь, обусловленная внешним взаимодействием.

Такая причинно-следственная связь не выявляется системой сетевых новостей.

В сущности, вопросы упорядочивания, как и множество других вопросов связи, относящихся к приложению, могут в достаточной мере быть решены путем рассмотрения приложений, с которыми осуществляется взаимодействие. Это явление [393] известно в проектировании систем под названием *сквозного аргумента* (*end-to-end argument*). Обратная сторона опоры исключительно на решения прикладного уровня — в том, что разработчик вынужден сосредоточиваться на вопросах, которые не относятся непосредственно к базовой функциональности приложения. Так, например, упорядочение может не быть важнейшей проблемой при разработке системы сообщений, например, сетевых новостей. В этом случае наличие базового уровня передачи данных, обслуживающего упорядоченность, может оказаться весьма кстати. Мы еще не раз вернемся к сквозным аргументам, в частности, при рассмотрении вопросов безопасности в распределенных системах.

5.3. Глобальное состояние

Во многих случаях полезно знать глобальное состояние, в соответствии с которым распределенная система функционирует в данное время. *Глобальное состояние* (*global state*) распределенной системы включает в себя локальные состояния каждого процесса вместе с находящимися в пути сообщениями (то есть посланными, но еще не доставленными). Что именно считать локальным состоянием процесса, зависит от того, что нас интересует [197]. В случае распределенной базы данных в него могут входить только те записи, из которых складывается часть

базы данных, а временные записи, используемые для вычислений, можно исключить. В нашем примере сборки мусора путем трассировки, который рассматривался в предыдущей главе, локальное состояние могло бы состоять из переменных, представляющих метки для тех заместителей, скелетонов и объектов, которые содержатся в адресном пространстве процесса.

Знание глобального состояния распределенной системы может быть полезно по многим причинам. Так, например, если известно, что локальные вычисления прекратились и ни одного сообщения в пути нет, ясно, что система перешла в состояние прекращения всяческой деятельности. Анализируя глобальное состояние, можно заключить, что мы либо зашли в тупик [71], либо распределенные вычисления были корректно завершены. Пример такого анализа рассматривается ниже.

В простом способе непосредственной записи глобального состояния распределенной системы, предложенном в [89], вводится понятие *распределенного снимка состояния* (*distributed snapshot*), отражающего состояние, в котором находилась распределенная система. Важным его свойством является то, что он отражает непротиворечивое глобальное состояние. В частности, это означает следующее: если в снимке записано, что процесс P получил сообщение от процесса Q , то там также должно быть записано, что процесс Q послал это сообщение. В противном случае снимок состояния содержал бы записи о сообщениях, которые были приняты, но никогда никем не посылались. Тем не менее допустима обратная ситуация, когда процесс Q послал сообщение, а процесс P его еще не получил.

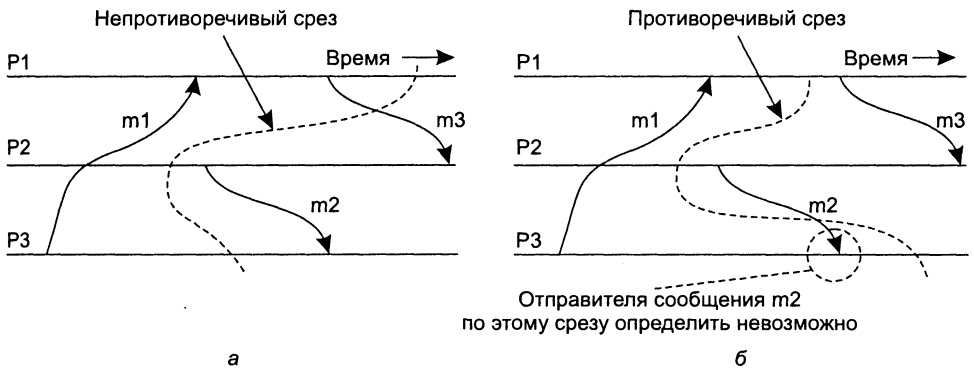


Рис. 5.9. Непротиворечивый срез (а). Противоречивый срез (б)

Понятие глобального состояния может быть представлено графически при помощи так называемого *среза* (*cut*). На рис. 5.9, а непротиворечивый срез, показанный пунктирной линией, пересекает оси времени трех процессов — $P1$, $P2$ и $P3$. Срез отражает последнее записанное событие для каждого из процессов. В этом случае мы можем легко убедиться, что для всех записей о пришедших сообщениях зафиксированы соответствующие события их отправки. В противоположность этому на рис. 5.9, б показан противоречивый срез. Приход сообщения $m2$ в процесс $P3$ был записан, но в снимке состояния не зафиксировано соответствующего события его отправки.

Для упрощения объяснения алгоритма создания распределенного снимка состояния мы предположим, что распределенную систему можно представить в виде набора процессов, соединенных друг с другом посредством однонаправленных прямых коммуникационных каналов. Так, например, процессы могут на первом этапе взаимодействия устанавливать соединения ТСП.

Инициировать алгоритм может любой процесс. Инициировавший получение распределенного снимка состояния процесс, скажем P , начинает с записи собственного локального состояния. Затем он посылает маркер по каждому из своих исходящих каналов, давая понять, что получатель этого сообщения должен принять участие в записи глобального состояния.

Когда процесс Q получает из входящего канала C маркер, все зависит от того, записал он уже свое локальное состояние или нет. Если нет, он сначала записывает свое локальное состояние, а затем также рассылает маркеры по всем исходящим каналам. Если же Q уже записал свое состояние, маркер, пришедший из канала C , показывает, что Q должен записать состояние канала. Это состояние складывается из последовательности сообщений, которые были приняты процессом Q с момента записи им своего локального состояния и до прихода маркера. Запись этого состояния иллюстрирует рис. 5.10.

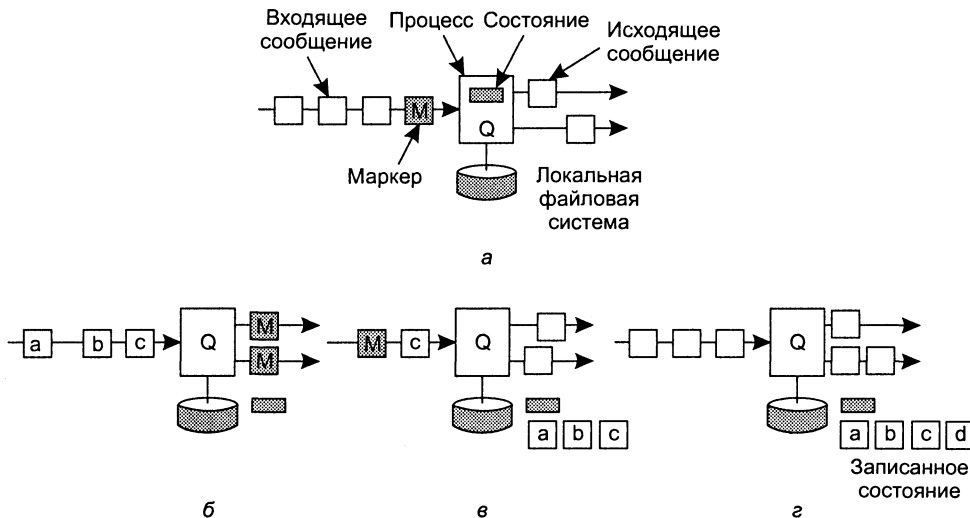


Рис. 5.10. Организация процесса и каналов в распределенном снимке состояния (а).

Процесс Q получает маркер впервые и записывает свое локальное состояние (б).

Процесс Q записывает все входящие сообщения (в).

Процесс Q получает маркер из входящего канала и заканчивает запись состояния входящего канала (г)

Процесс сообщает о том, что он закончил свою часть алгоритма, после получения и обработки маркера от каждого из своих входящих каналов. В этот момент записанное локальное состояние, а также состояние, записанное для каждого из входящих каналов, собираются вместе и посылаются, например, процессу, инициировавшему создание снимка состояния. Последний может затем проана-

лизовать текущее состояние. Отметим, что тем временем распределенная система в целом продолжает работать обычным образом.

Следует отметить, что поскольку алгоритм может быть инициирован любым процессом, одновременно может создаваться несколько снимков состояния. Поэтому маркер должен иметь идентификатор (а возможно, и номер версии) процесса, который инициировал алгоритм. Только после того, как процесс получит этот маркер по всем входным каналам, он может закончить построение своей части ассоциированного с маркером снимка состояния.

Пример — определение момента завершения

Выяснив, как приложение получает снимок состояния, рассмотрим, как определить момент завершения распределенных вычислений. Если процесс Q получает маркер, запрашивающий снимок состояния, в первый раз, он рассматривает процесс, приславший этот маркер, как предшественника. Когда Q завершает создание своей части снимка состояния, он посылает своему предшественнику сообщение *ГОТОВО*. Рекурсивным образом, когда инициатор создания распределенного снимка состояния получает сообщения *ГОТОВО* от всех своих преемников, он знает, что снимок состояния полностью готов.

Однако снимок состояния может показать глобальное состояние, в котором сообщения еще находятся в пути. В частности, представим себе записи процесса о том, что он получил сообщение по одному из своих входящих каналов в период между записью локального состояния и получением маркера из этого канала. Ясно, что мы не можем сделать заключения об окончании распределенных вычислений, поскольку это сообщение может породить новые сообщения, которые уже не будут частью снимка состояния.

Нам нужно получить снимок состояния со всеми пустыми каналами. Соответствующая несложная модификация алгоритма описывается ниже. Когда процесс Q заканчивает создание своей части снимка состояния, он посылает предшественнику либо сообщение *ГОТОВО*, либо сообщение *ПРОДОЛЖИТЬ*. Сообщение *ГОТОВО* возвращается только в том случае, если выполнены условия:

- ♦ все предшественники Q вернули сообщение *ГОТОВО*;
- ♦ между записью своего состояния и получением маркера процесс Q не принимал сообщений ни по одному из своих входящих каналов.

Во всех прочих случаях процесс Q посылает предшественнику сообщение *ПРОДОЛЖИТЬ*.

Существенно, что инициатор создания снимка состояния, скажем, процесс P , может получать от каждого из своих преемников либо сообщение *ПРОДОЛЖИТЬ*, либо сообщение *ГОТОВО*. Если были получены только сообщения *ГОТОВО*, то понятно, что обычный обмен сообщениями отсутствует, а значит, вычисления завершены. В противном случае процесс P иницирует создание нового снимка состояния и продолжает делать это, пока не получит полного комплекта сообщений *ГОТОВО*.

Были разработаны и другие решения задачи определения момента завершения вычислений, рассмотренной в этом пункте. Дополнительные ссылки и при-

меры можно почерпнуть из [17, 421]. Обзор и сравнение различных решений можно найти также в [285, 372].

5.4. Алгоритмы голосования

Многие распределенные алгоритмы требуют, чтобы один из процессов был координатором, инициатором или выполнял другую специальную роль. Обычно не важно, какой именно процесс выполняет эти специальные действия, главное, чтобы он вообще существовал. В этом разделе мы рассмотрим алгоритмы, предназначенные для выбора координатора (это слово мы будем использовать в качестве обобщенного имени специального процесса).

Если все процессы абсолютно одинаковы и не имеют отличительных характеристик, способа выбрать один из них не существует. Соответственно, мы должны считать, что каждый процесс имеет уникальный номер, например сетевой адрес (для простоты будем считать, что на каждую машину приходится по процессу). В общем, алгоритмы голосования пытаются найти процесс с максимальным номером и назначить его координатором. Алгоритмы различаются способами поиска.

Мы также считаем, что каждый процесс знает номера всех остальных процессов. Чего процессы не знают, так это то, какие из них в настоящее время работают, а какие нет. Алгоритм голосования должен гарантировать, что если голосование началось, то оно, рассмотрев все процессы, решит, кто станет новым координатором. Известны различные алгоритмы, например, [153, 159, 419].

5.4.1. Алгоритм забияки

В качестве первого примера рассмотрим *алгоритм забияки* (*bully algorithm*), предложенный в [159]. Когда один из процессов замечает, что координатор больше не отвечает на запросы, он инициирует голосование. Процесс, например P , проводит голосование следующим образом.

1. P посылает всем процессам с большими, чем у него, номерами сообщение **ГОЛОСОВАНИЕ**.
2. Далее возможно два варианта развития событий:
 - ♦ если никто не отвечает, P выигрывает голосование и становится координатором;
 - ♦ если один из процессов с большими номерами отвечает, он становится координатором, а работа P на этом заканчивается.

В любой момент процесс может получить сообщение **ГОЛОСОВАНИЕ** от одного из своих коллег с меньшим номером. По получении этого сообщения получатель посылает отправителю сообщение **ОК**, показывая, что он работает и готов стать координатором. Затем получатель сам организует голосование. В конце концов, все процессы, кроме одного, отпадут, этот последний и будет новым ко-

ординатором. Он уведомит о своей победе посылкой всем процессам сообщения, гласящего, что он новый координатор и приступает к работе.

Если процесс, который находился в нерабочем состоянии, начинает работать, он организует голосование. Если он оказывается процессом с самым большим из работающих процессов номером, он выигрывает голосование и берет на себя функции координатора. Итак, побеждает всегда самый большой парень в городишке, отсюда и название «алгоритм забияки».

На рис. 5.11 приведен пример работы алгоритма забияки. Группа состоит из восьми процессов, пронумерованных от 0 до 7. Ранее координатором был процесс 7, но он завис. Процесс 4 первым замечает это и посылает сообщение ГОЛОСОВАНИЕ всем процессам с номерами больше, чем у него, то есть процессам 5, 6 и 7, как показано на рис. 5.11, а. Процессы 5 и 6 отвечают ОК, как показано на рис. 5.11, б. После получения первого из этих ответов процесс 4 понимает, что «ему не светит» и координатором будет одна из этих более важных личностей. Он садится на место и ожидает, кто станет победителем (хотя уже сейчас он может сделать довольно точное предположение).

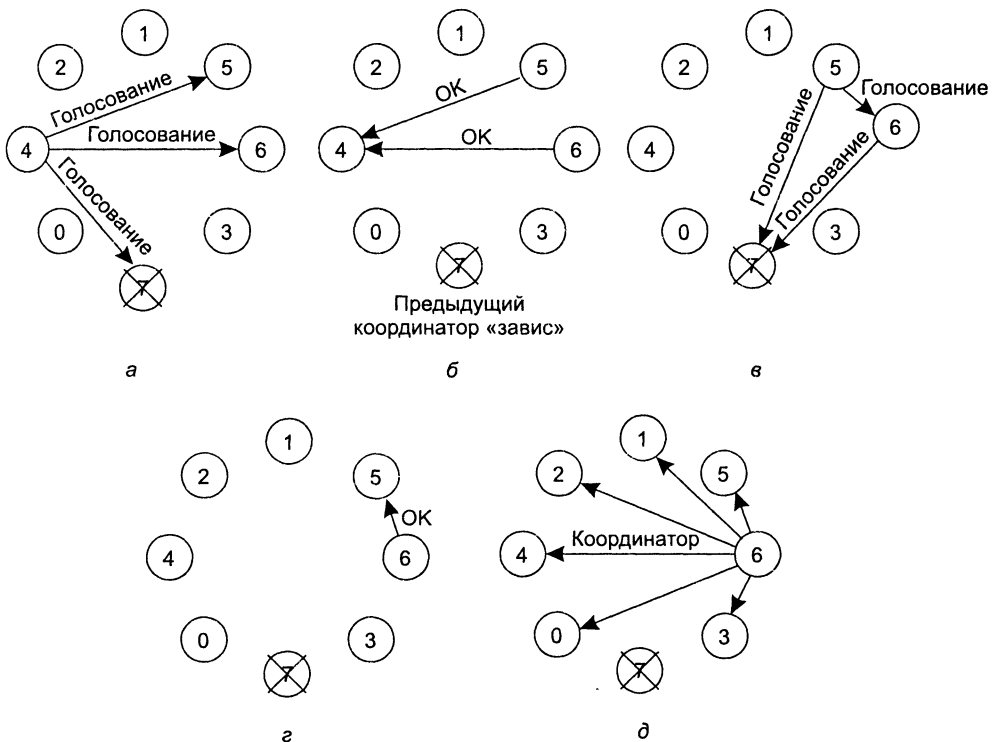


Рис. 5.11. Голосование по алгоритму забияки

На рис. 5.11, в показано, как оба оставшихся процесса, 5 и 6, продолжают голосование. Каждый посылает сообщения только тем процессам, номера у которых больше их собственных. На рис. 5.11, г процесс 6 сообщает процессу 5, что

голосование будет вести он. В это время 6 понимает, что процесс 7 мертв, а значит, победитель — он сам. Если информация о состоянии сохраняется на диске или где-то еще, откуда ее можно достать, когда с прежним координатором что-нибудь случается, процесс 6 должен записать все что нужно на диск. Готовый занять свою должность процесс 6 заявляет об этом путем рассылки сообщения *КОординАТОР* всем работающим процессам. Когда 4 получит это сообщение, он продолжит работу с той операции, которую пытался выполнить, когда обнаружил, что процесс 7 мертв, используя теперь в качестве координатора процесс 6. Таким образом, мы обошли сбой в процессе 7, и работа продолжается.

Если процесс 7 запустится снова, ему будет достаточно послать всем остальным сообщение *КОординАТОР* и вынудить их подчиниться.

5.4.2. Кольцевой алгоритм

Другой алгоритм голосования основан на использовании кольца. В отличие от некоторых других кольцевых алгоритмов в этом не требуется маркер. Мы предполагаем, что процессы физически или логически упорядочены, так что каждый из процессов знает, кто его преемник. Когда один из процессов обнаруживает, что координатор не функционирует, он строит сообщение *ГОЛОСОВАНИЕ*, содержащее его номер процесса, и посылает его своему преемнику. Если преемник не работает, отправитель пропускает его и переходит к следующему элементу кольца или к следующему, пока не найдет работающий процесс. На каждом шаге отправитель добавляет свой номер процесса к списку в сообщении, активно продвигая себя в качестве кандидата в координаторы.

В конце концов, сообщение вернется к процессу, который начал голосование. Процесс распознает это событие по приходу сообщения, содержащего его номер процесса. В этот момент тип сообщения изменяется на *КОординАТОР* и вновь отправляется по кругу, на этот раз с целью сообщить всем процессам, кто стал координатором (элемент списка с максимальным номером) и какие процессы входят в новое кольцо. После того как это сообщение один раз обойдет все кольцо, оно удаляется и процессы кольца возвращаются к обычной работе.

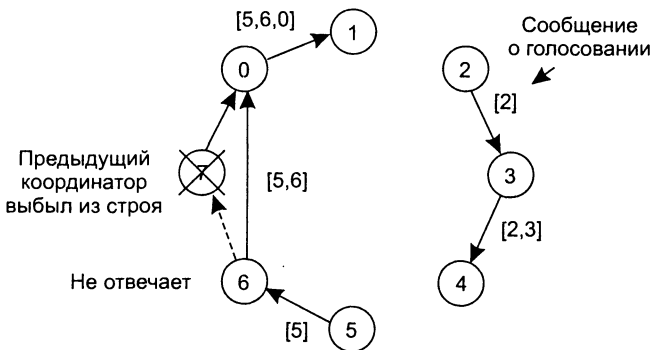


Рис. 5.12. Кольцевой алгоритм голосования

На рис. 5.12 мы видим, что происходит, когда два процесса, 2 и 5, одновременно обнаруживают, что предыдущий координатор, процесс 7, перестал работать. Каждый из них строит сообщение *ГОЛОСОВАНИЕ* и запускает это сообщение в путь по кольцу независимо от другого. В конце концов, оба сообщения проходят все кольцо, и процессы 2 и 5 превращают их в сообщения *КООДИНАТОР*, которые пересылаются тем же элементам кольца и в том же порядке. Когда эти сообщения, в свою очередь, совершают полный круг, они удаляются. Больше не происходит ничего такого, что потребовало бы дополнительного обмена сообщениями. Даже в наихудшем случае все сделанное столь незначительно загружает сеть, что это нельзя назвать расточительством.

5.5. Взаимное исключение

Системы, состоящие из множества процессов, обычно проще всего программировать, используя критические области. Когда процесс нуждается в том, чтобы считать или обновить совместно используемые структуры данных, он сначала входит в критическую область, чтобы путем взаимного исключения убедиться, что ни один из процессов не использует одновременно с ним общие структуры данных. В однопроцессорных системах критические области защищаются семафорами, мониторами и другими конструкциями подобного рода. Теперь мы рассмотрим несколько примеров того, как именно критические области и взаимные исключения реализуются в распределенных системах. Описание этих методов и библиографию по ним можно найти в [373, 420].

5.5.1. Централизованный алгоритм

Наиболее простой способ организации взаимных исключений в распределенных системах состоит в том, чтобы использовать методы их реализации, принятые в однопроцессорных системах. Один из процессов выбирается координатором (например, процесс, запущенный на машине с самым большим сетевым адресом). Каждый раз, когда этот процесс собирается войти в критическую область, он посылает координатору сообщение с запросом, в котором уведомляет, в какую критическую область он собирается войти, и запрашивает разрешение на это. Если ни один из процессов в данный момент не находится в этой критической области, координатор посылает ответ с разрешением на доступ, как показано на рис. 5.13, а. После получения ответа процесс, запросивший доступ, входит в критическую область.

Предположим теперь, что другой процесс, 2, запрашивает разрешение на вход в ту же самую область (рис. 5.13, б). Координатор знает, что в этой критической области уже находится другой процесс, и не дает разрешения на вход. Конкретный способ запрещения доступа зависит от особенностей системы. На рис. 5.13, б координатор просто не отвечает, блокируя этим процесс 2, который ожидает ответа. Он также может послать ответ, гласящий «доступ запрещен». В любом случае он ставит запрос от процесса 2 в очередь и ожидает дальнейших сообщений.

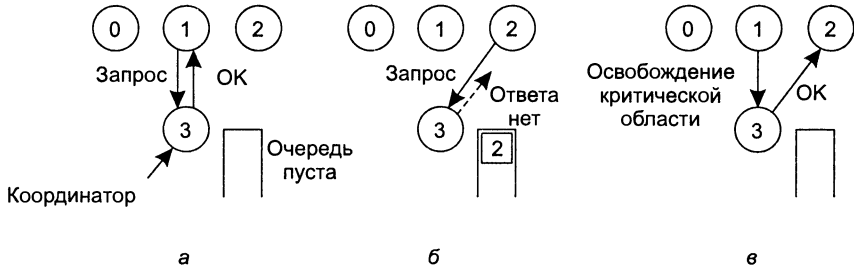


Рис. 5.13. Процесс 1 запрашивает у координатора разрешение на вход в критическую область, и разрешение дано (а). Процесс 2 запрашивает разрешение на вход в ту же критическую область, и координатор не отвечает (б). Когда процесс 1 выходит из критической области, он сообщает об этом координатору, который разрешает доступ процессу 2 (в)

Когда процесс 1 покидает критическую область, он посылает координатору сообщение, отказываясь от эксклюзивного доступа на область, как показано на рис. 5.13, в. Координатор выбирает первый элемент из очереди отложенных запросов и посылает процессу разрешающее сообщение. Если процесс был блокирован (то есть для него это первое сообщение от координатора), он разблокируется и входит в критическую область. Если процессу было отослано сообщение с запретом на доступ, он будет опрашивать приходящие сообщения или блоки. В любом случае, увидев разрешение, он войдет в критическую область.

Легко заметить, что алгоритм гарантирует взаимное исключение: координатор позволяет войти в каждую критическую область только одному процессу за раз. Это, помимо всего прочего, честно, поскольку разрешения выдаются в том же порядке, в каком они запрашивались. Никакой процесс никогда не ждет вечно (зависания отсутствуют). Схема также проста в реализации и использует для работы с критической областью всего три сообщения (запросить, разрешить и высвободить). Подобный алгоритм может использоваться и для другого выделения ресурсов, а не только для работы с критическими областями.

5.5.2. Распределенный алгоритм

Наличие даже одного неработающего места в централизованных алгоритмах часто недопустимо, поэтому исследователи обратились к распределенным алгоритмам взаимного исключения [380].

Рассматриваемый алгоритм требует наличия полной упорядоченности событий в системе. То есть в любой паре событий, например отправки сообщений, должно быть однозначно известно, какое из них произошло первым. Алгоритм Лампорта, представленный в пункте 5.2.1, является одним из способов введения подобной упорядоченности и может быть использован для расстановки отметок времени распределенных взаимных исключений.

Алгоритм работает следующим образом. Когда процесс собирается войти в критическую область, он создает сообщение, содержащее имя критической области, свой номер и текущее время. Затем он отправляет это сообщение всем процес-

сам, концептуально включая самого себя. Посылка сообщения, как предполагается, надежная, то есть на каждое письмо приходит подтверждение в получении. Вместо отдельных сообщений может быть использована доступная надежная групповая связь.

Когда процесс получает сообщение с запросом от другого процесса, действие, которое оно производит, зависит от его связи с той критической областью, имя которой указано в сообщении. Можно выделить три варианта.

- ♦ Если получатель не находится в критической области и не собирается туда входить, он отправляет отправителю сообщение *ОК*.
- ♦ Если получатель находится в критической области, он не отвечает, а помещает запрос в очередь.
- ♦ Если получатель собирается войти в критическую область, но еще не сделал этого, он сравнивает метку времени пришедшего сообщения с меткой времени сообщения, которое он отослал. Выигрывает минимальное. Если пришедшее сообщение имеет меньший номер, получатель отвечает посылкой сообщения *ОК*. Если его собственное сообщение имеет меньшую отметку времени, получатель ставит приходящие сообщения в очередь, ничего не посылая при этом.

После посылки сообщения-запроса на доступ в критическую область процесс приостанавливается и ожидает, что кто-нибудь даст ему разрешение на доступ. После того как все разрешения получены, он может войти в критическую область. Когда он покидает критическую область, то отправляет сообщения *ОК* всем процессам в их очереди и удаляет все сообщения подобного рода из своей очереди.

Попытаемся теперь понять, как работает алгоритм. Если конфликты отсутствуют, все идет хорошо. Однако представим себе, что два процесса пытаются одновременно войти в одну и ту же критическую область, как показано на рис. 5.14, а.

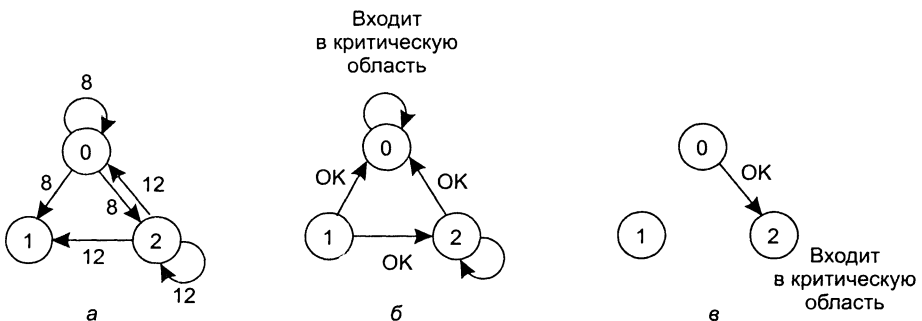


Рис. 5.14. Два процесса одновременно хотят получить доступ в одну и ту же критическую область (а). Процесс 0 имеет меньшую отметку времени и потому выигрывает (б). Когда процесс 0 завершает работу с критической областью, он отправляет сообщение *ОК*, и теперь процесс 2 может войти в критическую область (в)

Процесс 0 рассылает всем запрос с отметкой времени 8, и одновременно с ним процесс 2 рассылает всем запрос с отметкой времени 12. Процесс 1 не ин-

тересуется входом в критическую область и в ответ посылает сообщение *ОК* им обоим. Процессы 0 и 2 замечают конфликт и сравнивают отметки времени. Процесс 2 видит, что проиграл, и разрешает доступ процессу 0, посылая ему сообщение *ОК*. Процесс 0 ставит ответ от процесса 2 в очередь для дальнейшей обработки и входит в критическую секцию, как показано на рис. 5.14, б. Когда процесс 0 заканчивает работу в критической области, он удаляет ответ 2 из очереди сообщений и отправляет процессу 2 сообщение *ОК*, разрешая ему войти в критическую область, что и показано на рис. 5.14, в. Алгоритм работает, поскольку в случае конфликтов наименьшая отметка времени выигрывает, а с очередностью отметок времени способен разобраться любой процесс.

Отметим, что показанная на рисунке ситуация могла бы в корне измениться, если бы процесс 2 послал свое сообщение раньше, чем это сделал процесс 0, и получил разрешение на доступ до создания своего ответа на сообщение от процесса 0. В этом случае процесс 2, зная о том, что в момент отсылки ответа он находится в критической области, просто поместил бы запрос от процесса 0 в очередь, не посылая никакого ответа.

Как и централизованный алгоритм, который мы обсуждали ранее, распределенное взаимное исключение гарантировано от тупиков и зависимостей. Число сообщений, приходящихся на один процесс, — $2(n-1)$, где n — общее число процессов в системе. Больше нет единой точки, сбоя в которой мог бы погубить всю систему.

Однако, к сожалению, одна точка сбоя сменилась на n точек сбоев. Если какой-либо из процессов «рухнет», он не сможет ответить на запрос. Это молчание будет воспринято (неправильно) как отказ в доступе и блокирует все последующие попытки всех процессов войти в какую-либо из критических областей. Поскольку вероятность того, что рухнет один из n процессов как минимум в n раз больше, чем вероятность сбоя единственного координатора, мы пришли к тому, что заменили слабый алгоритм в n раз худшим и требующим более интенсивного сетевого трафика.

Этот алгоритм может быть исправлен тем же приемом, который мы использовали ранее. Когда приходит запрос, его получатель посылает ответ всегда, разрешая или запрещая доступ. Всякий раз, когда запрос или ответ утеряны, отправитель выжидает положенное время и либо получает ответ, либо считает, что получатель находится в нерабочем состоянии. После получения запрещения отправитель ожидает последующего сообщения *ОК*.

Другая проблема этого алгоритма состоит в том, что либо должны использоваться примитивы групповой связи, либо каждый процесс должен поддерживать список группы самостоятельно, обеспечивая внесение процессов в группу, удаление процессов из группы и отслеживание сбоев. Метод наилучшим образом работает, когда группа процессов мала, а членство в группе постоянно и никогда не меняется.

И, наконец, вспомним, что одной из проблем централизованного алгоритма было то, что обработка всех запросов в одном месте могло стать его узким местом. В распределенном алгоритме все процессы вынуждены участвовать во всех решениях, касающихся входа в критические области. Если один из процессов

оказывается неспособным справиться с такой нагрузкой, маловероятно, что возьмется успех попытка их всех сделать то же самое параллельно.

В этот алгоритм можно внести разнообразные дополнительные усовершенствования. Например, получение каждым из процессов разрешения на вход в критическую область — это, по правде говоря, чересчур. Ведь нам необходимо только предотвратить одновременный вход двух процессов в критическую область. Алгоритм можно модифицировать так, чтобы разрешить процессу вход в критическую область после того, как он соберет разрешения простого большинства, а не всех остальных процессов. Разумеется, в этом случае, после того как процесс даст разрешение на вход в критическую область одному из процессов, он не сможет дать то же самое разрешение другому процессу до тех пор, пока первый не покинет критическую область. Возможны также и другие улучшения, в частности предложенные в [279], но они легко могут запутать алгоритм.

Несмотря на все возможные улучшения, этот алгоритм остается более медленным, более сложным, более затратным и менее устойчивым, чем исходный централизованный алгоритм. Зачем тогда его вообще изучать? Для единственной цели. Мы показали, что распределенные алгоритмы как минимум возможны. Когда мы начинали рассмотрение, это не было очевидно. Кроме того, только обратив внимание на недостатки, мы можем подвигнуть будущих теоретиков на разработку алгоритмов, которые действительно можно будет использовать. И наконец, подобно питанию шпинатом и изучению латыни в средней школе, некоторые вещи выглядят очень хорошо, но только до тех пор, пока остаются теорией.

5.5.3. Алгоритм маркерного кольца

Абсолютно иной подход к реализации взаимных исключений в распределенных системах предлагает алгоритм *маркерного кольца* (*Token Ring*). Пусть мы имеем магистральную сеть (например, Ethernet), но без внутреннего упорядочения процессов (рис. 5.5, а). Программно создается логическое кольцо, в котором каждому процессу назначается положение в кольце, как показано на рис. 5.15, б. Положение в кольце может быть назначено по порядку следования сетевых адресов или как-то иначе. Неважно, как именно задана упорядоченность. Главное — как дать процессу знать, кто в кольце является следующим после него.

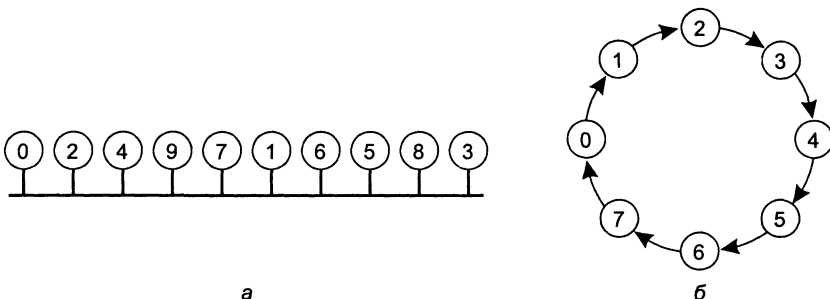


Рис. 5.15. Беспорядочная группа процессов в сети (а).
Логическое кольцо, созданное программно (б)

При инициализации кольца процесс 0 получает *маркер*, или *token (token)*. Маркер циркулирует по кольцу. Он передается от процесса k процессу $k + 1$ (это модуль размера кольца) сквозными сообщениями. Когда процесс получает маркер от своего соседа, он проверяет, не нужно ли ему войти в критическую область. Если это так, он входит в критическую область, выполняет там всю необходимую работу и покидает область. После выхода он передает маркер дальше. Входить в другую критическую область, используя тот же самый маркер, запрещено.

Если процесс, получив от соседа маркер, не заинтересован во входе в критическую область, он просто передает этот маркер дальше. Соответственно, если ни один из процессов не находится в критических областях, маркер просто циркулирует по кольцу с максимально возможной скоростью.

Легко видеть, что этот алгоритм корректен. Только один процесс в любой момент времени обладает маркером, а значит, только один процесс может находиться в критической области. Поскольку маркер перемещается от процесса к процессу в общеизвестном порядке, зависания не происходит. Когда процесс решает войти в критическую область, в худшем случае ему придется ждать, пока все остальные процессы последовательно не войдут в критическую область и не выйдут из нее.

Обычно этот алгоритм также имеет проблемы. Если маркер однажды потеряется, он должен быть восстановлен. На самом деле, понять, что он пропал, довольно сложно, поскольку срок между последовательными появлениями маркера в сети не ограничен. Тот факт, что маркера не было видно в течение часа, вовсе не означает, что он потерян, кто-нибудь просто может его использовать.

Алгоритм также сталкивается с проблемами при сбоях процессов, однако справиться с этим проще, чем в других случаях. Если мы потребуем от процесса, получающего маркер, подтверждать получение, неработающий процесс будет обнаружен при первой же попытке соседа передать ему маркер. В этот момент неработающий процесс можно удалить из группы, и хранитель маркера сможет перебросить маркер через его «голову» следующему члену кольца или, при необходимости следующему за ним. Разумеется, это требует поддержания текущей конфигурации кольца.

5.5.4. Сравнение трех алгоритмов

Мы сделаем краткое сравнение трех алгоритмов взаимных исключений, поскольку оно поучительно. В табл. 5.1 мы перечислили алгоритмы и три их ключевых свойства: число сообщений, необходимое процессу для того, чтобы войти в критическую область или выйти из нее, возможная задержка перед входом (предполагается, что сообщения передаются по сети последовательно) и некоторые проблемы, связанные с описываемым алгоритмом.

Централизованный алгоритм наиболее прост и наиболее эффективен. На то, чтобы войти в критическую область, ему достаточно всего трех сообщений — запрос, разрешение на вход и сообщение о выходе. Распределенному алгоритму (мы предполагаем, что используются только сквозные коммуникации) требуется

для запроса $n - 1$ сообщений, по одному на каждый процесс, и дополнительно $n - 1$ сообщений на разрешение, всего $2(n - 1)$. В алгоритме маркерного кольца число сообщений различно. Если каждый из процессов будет постоянно требовать входа в критическую область, каждая передача маркера станет результатом выхода из критической области одного из процессов и в среднем на вход в критическую область понадобится одно сообщение. В другом крайнем случае маркер может циркулировать по кольцу часами и им никто и не заинтересуется. В этом случае число сообщений на вход в критическую область одного процесса не ограничено.

Таблица 5.1. Сравнение трех алгоритмов взаимных исключений

Алгоритм	Число сообщений на вход-выход	Задержка перед входом в число сообщений	Возможные проблемы
Централизованный	3	2	Крах координатора
Распределенный	$2(n - 1)$	$2(n - 1)$	Сбой в одном из процессов
Маркерного кольца	От 1 до ∞	От 0 до $n - 1$	Потеря маркера, сбой в одном из процессов

Задержка с момента, когда процессу понадобилось войти в критическую область, до момента входа для этих трех алгоритмов также различна. Если критические области малы и используются редко, основным фактором задержки является механизм входа в критическую область. Если критические области и используются постоянно, основным фактором задержки является ожидание своего хода. В таблице иллюстрируется первый случай. В случае централизованного алгоритма, для того чтобы войти в критическую область, требуется всего два сообщения. В случае распределенного алгоритма требуется $2(n - 1)$ сообщений, с учетом того, что они посылаются одно за другим. Для маркерного кольца время варьируется от 0 (маркер у процесса, входящего в критическую область) до $n - 1$ (маркер был только что передан дальше по кольцу).

И, наконец, все три алгоритма тяжело реагируют на сбои. Чтобы сбой не привел к полному краху системы, приходится предпринимать специальные меры и дополнительно усложнять алгоритм. Забавно, но распределенные алгоритмы более чувствительны к сбоям, чем централизованный. В защищенных от сбоя системах неприменим ни один из них, но если сбои нечасты, они будут работать.

5.6. Распределенные транзакции

Концепция транзакций тесно связана с концепцией взаимных исключений. Алгоритмы взаимного исключения обеспечивают одновременный доступ не более чем одного процесса к совместно используемым ресурсам, таким как файл, принтер и т. п. Транзакции, в общем, также защищают общие ресурсы от одновременного доступа нескольких параллельных процессов. Однако транзакции могут и многое другое. В частности, они превращают процессы доступа и модификации множе-

ства элементов данных в одну атомарную операцию. Если процесс во время транзакции решает остановиться на полпути и повернуть назад, все данные восстанавливаются с теми значениями и в том состоянии, в котором они были до начала транзакции. В этом разделе мы поближе ознакомимся с понятием транзакции, и в частности сосредоточимся на возможностях транзакций по синхронизации нескольких процессов при защите совместно используемых данных.

5.6.1. Модель транзакций

Базовая модель транзакций пришла к нам из делового мира. Допустим, некой международной корпорации потребовалась партия каких-то шуток. Они обращаются к потенциальному поставщику, ООО «Штуковины», хорошо известному во всем мире качеством своих шуток, с заказом на 100 000 10-сантиметровых пурпурных шуток с поставкой в июне. ООО «Штуковины» предлагает 100 000 4-дюймовых розовых шуток с поставкой в декабре. Некая международная корпорация согласна с ценой, но не любит розовый цвет, хочет получить заказ в июле и настаивает на 10-сантиметровых шутовинах для своих зарубежных заказчиков. ООО «Штуковины» отвечают предложением 3 15/16-дюймовых лавандовых шуток в октябре. После дальнейших переговоров они сходятся, наконец, на 3 959/1024-дюймовых фиолетовых шутовинах, которые будут поставлены к 15 августа.

До этого самого момента обе стороны имели полную возможность прекратить обсуждение, после чего мир вернулся бы к тому же состоянию, в котором он пребывал до начала переговоров. Однако после того, как обе компании подписали контракт, они связали себя обязательством выполнить сделку. Таким образом, пока обе стороны шли к этому Рубикону, любая из них могла дать отбой, и ничего бы не произошло, но в момент появления подписей они перешли ту черту, из-за которой нет возврата. После этого транзакция должна быть выполнена.

Компьютерная модель выглядит очень похоже. Один процесс объявляет, что хочет начать транзакцию с одним или несколькими другими процессами. После этого они могут согласовывать различные условия, создавать и удалять сущности, выполнять операции. Затем инициатор объявляет, что он предлагает всем остальным подтвердить, что работа сделана. Если все это подтверждают, результаты утверждаются и становятся постоянными. Если один или несколько процессов отказываются (или до соглашения в них возникают сбои), ситуация возвращается к тому состоянию, которое имело место до начала транзакции, со всеми вытекающими отсюда последствиями для файлов, баз данных и т. д., все изменения в которых волшебным образом исчезают. Такое свойство «все или ничего» сильно упрощает работу программистам.

Использование транзакций в компьютерных системах берет начало в шестидесятых годах. До этого дисков и сетевых баз данных не существовало, все данные хранились на магнитных лентах. Представьте себе супермаркет с автоматизированной системой инвентаризации. Каждый день после закрытия компьютер начинал работу с двумя магнитными лентами. Первая из них содержала полные сведения о товарах на момент открытия утром текущего дня, а вторая — список

изменений за день: продукты, купленные покупателями, и продукты, возвращенные поставщикам. Компьютер считывал информацию с обеих лент и создавал новую основную ленту инвентаризации, как показано на рис. 5.16.

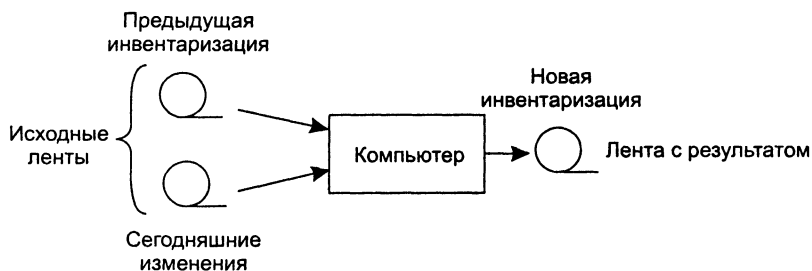


Рис. 5.16. Внесение изменений в основную ленту защищено от сбоев

Основное преимущество этой схемы (несмотря на то, что люди, вынужденные этим заниматься, тогда часто не могли этого оценить) состояло в том, что при любых сбоях все ленты можно было перемотать на начало и начать работу снова без каких-либо проблем. Такие примитивные старые магнитные ленты обладали свойством, характерным для транзакции, — «все или ничего».

Взглянем теперь на современное банковское приложение, которое вносит изменения в сетевую базу данных. Клиент звонит в банк, используя компьютер с модемом, намереваясь снять деньги с одного счета и положить их на другой. Операция осуществляется в два приема.

1. Снять сумму a со счета 1.
2. Положить сумму a на счет 2.

В том случае, если модемное соединение после выполнения первого этапа, но до выполнения второго разорвется, деньги с первого счета будут сняты, но на второй не перечислены. Деньги просто растворятся в воздухе.

Проблему решает объединение этих операций в одну транзакцию. Либо обе они будут выполнены, либо не будет выполнена ни одна. Ключевой, следовательно, является возможность отката к исходному состоянию при невозможности завершить транзакцию. Что нам действительно нужно — так это способ «отмотать к началу» базу данных, как мы это проделывали с магнитной лентой. Эту возможность дает нам транзакция.

Программирование с использованием транзакций требует специальных примитивов, которые могут поддерживаться как базовой распределенной системой, так и исполняющей системой языка программирования. Типичные примеры приводятся в табл. 5.2. Полный список примитивов зависит от того, какие объекты используются в транзакции. В почтовой системе примитивами могут быть отправление, прием и пересылка почты. В банковских системах примитивы могут быть совершенно другими. Однако, как правило, там присутствуют команды READ и WRITE. Обычные инструкции, вызовы процедур и пр. также могут включаться внутрь транзакций.

Таблица 5.2. Некоторые примитивы, используемые в транзакциях

Примитив	Описание
BEGIN_TRANSACTION	Пометить начало транзакции
END_TRANSACTION	Прекратить транзакцию и попытаться завершить ее
ABORT_TRANSACTION	Прервать транзакцию и восстановить прежние значения
READ	Считать данные из файла, таблицы или другого источника
WRITE	Записать данные в файл, таблицу или другой приемник

Для ограничения области действия транзакции используются примитивы BEGIN_TRANSACTION и END_TRANSACTION. Операции, расположенные между ними, формируют тело транзакции. Все эти операции либо выполняются, либо не выполняются. Это могут быть системные вызовы, библиотечные процедуры или инструкции на языке реализации.

Рассмотрим в качестве примера процесс резервирования посадочных мест от Белых равнин, штат Нью-Йорк, до Малинди, Кения, в авиационной системе резервирования мест. Один из возможных маршрутов: из Белых равнин в аэропорт им. Джона Кеннеди, из аэропорта им. Джона Кеннеди в Найроби, из Найроби в Малинди. На листинге 5.1 мы видим, что резервирование билетов на эти три рейса выполняется тремя различными операциями.

Листинг 5.1. Подтверждение транзакции резервирования билетов на три авиарейса

```
BEGIN_TRANSACTION
    зарезервировать WP -> JFK;
    зарезервировать JFK -> Nairobi;
    зарезервировать Nairobi -> Malindi;
END_TRANSACTION
```

Теперь представим, что билеты на первые два рейса зарезервированы, а третий, если судить по документам, оказался переполнен. Транзакция прерывается, и результаты первых двух резервирований отменяются — база данных по билетам на авиарейсы возвращается к тем значениям, которые были в ней до начала транзакции (листинг 5.2). Все выглядит так, будто ничего не происходило.

Листинг 5.2. Транзакция прервана по причине невозможности заказа билета на третий самолет

```
BEGIN_TRANSACTION
    зарезервировать WP -> JFK;
    зарезервировать JFK -> Nairobi;
    Nairobi -> Malindi переполнен =>
ABORT_TRANSACTION
```

Свойство транзакций «все или ничего» — это лишь одно из характерных свойств транзакции. Говоря более конкретно, транзакции:

- ♦ *атомарны (atomic)* — для окружающего мира транзакция неделима;
- ♦ *непротиворечивы (consistent)* — транзакция не нарушает инвариантов системы;

- ♦ *изолированы (isolated)* — одновременно происходящие транзакции не влияют друг на друга;
- ♦ *долговечны (durable)* — после завершения транзакции внесенные ею изменения становятся постоянными.

На эти свойства часто ссылаются по их первым буквам — *ACID*.

Первое ключевое свойство, которое проявляется во всех транзакциях, — *атомарность*. Это свойство гарантирует, что всякая транзакция либо полностью выполняется, либо полностью не выполняется, причем если она выполняется, то выполняется как одна неделимая одновременная операция. Пока транзакция находится в процессе выполнения, другие процессы (независимо от того, вовлечены они сами в транзакцию или нет) не могут наблюдать каких-либо промежуточных состояний.

Представим, например, что транзакция начинается для того, чтобы добавить данные в некий файл с начальной длиной 10 байт. Если этот файл в ходе транзакции пожелает прочитать другой процесс, он увидит только исходные 10 байт независимо от того, сколько байт было добавлено в файл в ходе транзакции. Если транзакция завершится успешно, файл мгновенно вырастет до нового размера, размера на момент завершения без промежуточных состояний независимо от того, сколько операций внутри транзакции привело к его увеличению.

Второе свойство, о котором мы говорим, это *непротиворечивость*. Это значит, что если у системы до начала транзакции имелись некие инварианты, которые она постоянно должна хранить, они будут сохраняться и после ее завершения. Так, например, в банковской системе ключевым инвариантом является закон сохранения денег. После любых внутренних переводов количество денег в банке должно сохраняться таким же, каким оно было до перевода, хотя на краткий момент в ходе транзакции этот инвариант и может нарушаться, но такое нарушение не будет заметно извне.

Третье свойство, о котором мы говорили, — это *изолированность*, или *сериализуемость*. В том случае, если две или более транзакций происходят одновременно, для каждой из них и для других процессов итоговый результат выглядит так же, как если бы все транзакции выполнялись последовательно в некотором (зависящем от системы) порядке. Ниже мы еще вернемся к этому свойству.

Четвертое свойство гласит, что транзакции *долговечны*. Это свойство отражает тот факт, что после завершения транзакций последующие действия не имеют никакого значения, транзакция закончена, ее результаты неизменны. Никакие сбои после завершения транзакции не могут привести к отмене результатов транзакции или их потере. Долговечность подробно обсуждается в главе 7.

5.6.2. Классификация транзакций

Итак, мы считаем транзакцией серию операций, удовлетворяющую свойствам ACID. Этот тип транзакции именуется также *плоской транзакцией (flat transaction)*. Плоские транзакции — это наиболее простой и наиболее часто используемый тип транзакций. Однако плоские транзакции имеют множество ограниче-

ний, которые вынуждают нас заняться поисками альтернативных моделей. Ниже мы обсудим два важных класса транзакций — вложенные транзакции и распределенные транзакции. Другие классы широко рассмотрены в [177].

Ограничения плоских транзакций

Основное ограничение плоских транзакций состоит в том, что они не могут давать частичного результата в случае завершения или прерывания. Другими словами, сила атомарности плоских транзакций является в то же время и их слабостью.

Рассмотрим снова полет из Нью-Йорка в Кению, проиллюстрированный листингами 5.1 и 5.2. Предположим, что билеты на всю поездку продавались в виде относительно дешевого единого пакета, в результате чего три операции были собраны в единую транзакцию. Обнаружив невозможность заказать билеты только на третью часть пути, мы могли бы согласиться на резервирование только первых двух билетов. Однако при этом можем обнаружить, что зарезервировать билет на самолет из аэропорта Кеннеди в Найроби уже невозможно. Прерывание всей транзакции означает, что нам придется предпринять вторую попытку зарезервировать билеты на самолет, которая также может закончиться неудачей. Таким образом, в данном случае нам бы хотелось частично завершить нашу транзакцию. Плоская транзакция не позволит нам этого сделать.

В качестве другого примера рассмотрим web-сайт, гиперссылка в котором двунаправленная. Другими словами, если web-страница W_1 содержит URL страницы W_2 , то W_1 знает, что W_2 ссылается на нее [224]. Теперь представим себе, что страница W перенесена в другое место или заменена другой страницей. В этом случае все гиперссылки на W должны быть обновлены, причем желательно одной атомарной операцией, в противном случае мы можем получить (временно) потерянные ссылки на W . Теоретически здесь можно использовать плоскую транзакцию. Транзакция состоит из внесения изменений в W и серии операций, каждая из которых изменяет одну web-страницу, содержащую гиперссылку на W .

Проблема, однако, состоит в том, что такая транзакция может потребовать для выполнения нескольких часов. Страницы, ссылающиеся на W , могут быть разбросаны по всему Интернету, а таких страниц, нуждающихся в правке, может быть тысячи. Производить правку отдельными транзакциями не слишком хорошо, потому что в этом случае некоторые web-страницы будут содержать правильные ссылки, а некоторые нет. Возможным решением в этом случае было бы завершение изменений с сохранением старой ссылки на W для тех страниц, ссылки на которые еще не изменились.

Вложенные транзакции

Некоторые из тех ограничений, о которых мы говорили выше, могут быть сняты при использовании *вложенных транзакций* (*nested transactions*). Транзакция верхнего уровня может разделяться на дочерние транзакции, работающие параллельно, на различных машинах, для повышения производительности или упрощения программирования. Каждая из этих дочерних транзакций также может со-

стоять из одной или более транзакций или, в свою очередь, делиться на дочерние транзакции.

Вложенные транзакции поднимают перед нами небольшую, но важную проблему. Представьте себе, что транзакция запускает несколько параллельных дочерних транзакций, и одна из них завершается, делая результат видимым для родительской транзакции. В ходе дальнейших вычислений родительская транзакция прерывается, возвращая систему в то состояние, в котором она была до запуска транзакции верхнего уровня. Вместе с другими промежуточными результатами теряются и результаты завершившейся дочерней транзакции. Таким образом, долговечность, о которой мы говорили, применима только к транзакциям верхнего уровня.

Поскольку глубина вложенности транзакций может быть произвольно глубокой, для сохранения правильности необходимо серьезное администрирование. Однако семантика ясна. Когда начинается любая транзакция или вложенная транзакция, создается закрытая копия данных всей системы. Если транзакция прерывается, ее внутренняя вселенная исчезает, как будто ее никогда и не было. Если она завершается, ее внутренняя вселенная замещает родительскую. Таким образом, если дочерняя транзакция завершается и начинается новая дочерняя транзакция, вторая из них получает возможность работать с результатами, созданными первой. Таким же образом, если прерывается объемлющая (верхнего уровня) транзакция, все вложенные в нее дочерние транзакции также прерываются.

Распределенные транзакции

Вложенные транзакции важны для распределенных систем, потому что они предоставляют естественный способ распределения транзакций по нескольким машинам. Однако вложенные транзакции обычно делят работу исходной транзакции *логически*. Например, транзакция, в ходе которой следовало зарезервировать билеты на три различных самолета (см. листинги 5.1 и 5.2), может быть логически разделена на три вложенные транзакции. Каждая из этих вложенных транзакций может управляться отдельно, независимо от остальных двух.

Однако логическое разделение вложенных транзакций на вложенные транзакции не обязательно означает, что все распределение в этом и состоит. Так, вложенная транзакция резервирования места в самолете от Нью-Йорка до Найроби может нуждаться в доступе к двум базам данных, по одной для каждого из этих городов. В этом случае вложенная транзакция может разделяться дальше, на меньшие вложенные транзакции, которые логически уже не существуют, поскольку резервирование само по себе является неделимой операцией.

В этом случае вложенные транзакции (плоские) работают с данными, распределенными по нескольким машинам. Такие транзакции известны под названием *распределенных транзакций* (*distributed transactions*). Разница между вложенными и распределенными транзакциями невелика, но существенна. Вложенные транзакции — это транзакции, которые логически разделяются на иерархически организованные дочерние транзакции. В противоположность им распределенные транзакции логически представляют собой плоские, неделимые транзакции,

которые работают с распределенными данными. Эту разницу иллюстрирует рис. 5.17.

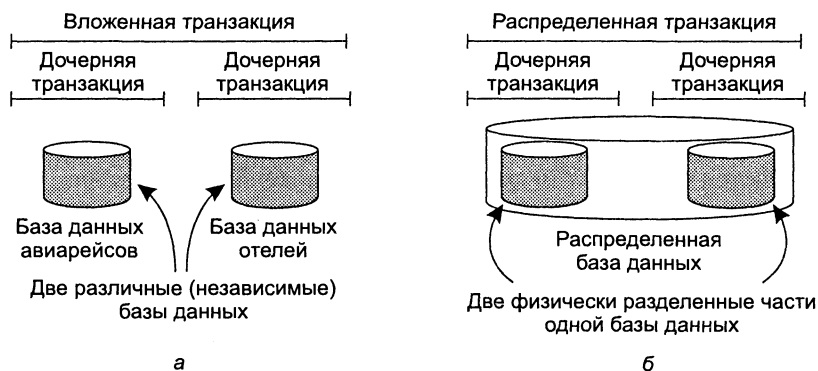


Рис. 5.17. Вложенная транзакция (а), распределенная транзакция (б)

Основная проблема распределенных транзакций в том, что для блокировки данных и подтверждения транзакции необходимы отдельные распределенные алгоритмы. Распределенную блокировку данных мы рассмотрим ниже. Детальное рассмотрение распределенных протоколов подтверждения транзакции мы отложим до главы 7, в которой рассмотрим отказоустойчивость и механизмы восстановления, к которым относится и протокол подтверждения.

5.6.3. Реализация

Транзакции кажутся отличной идеей, но как их реализовать? Этот вопрос мы сейчас и рассмотрим. Чтобы упростить дело, мы ограничимся транзакциями в файловой системе. Должно быть понятно, что если каждый процесс, выполняющий транзакцию, просто обновляет файл там же, где файл используется, транзакция не будет атомарной и при прерывании транзакции все изменения не исчезнут волшебным образом. Понятно, что нужен другой метод. Обычно используются те два метода, которые описаны ниже.

Закрытое рабочее пространство

Концептуально, когда процесс начинает транзакцию, он получает закрытое рабочее пространство, содержащее все файлы, к которым он хочет получить доступ. Пока транзакция не завершится или не прервется, все операции чтения и записи будут происходить не в файловой системе, а в закрытом рабочем пространстве. Это утверждение прямо приводит нас к первому методу реализации — созданию для процесса, в момент начала транзакции, закрытого рабочего пространства.

Проблема этой методики состоит в том, что цена копирования всего подряд в закрытое рабочее пространство такова, что равносильна запрету. Возможна, однако, различная оптимизация. Первый вариант оптимизации основан на том соображении, что если процесс только читает файл, но не изменяет его в создании

закрытой копии нет необходимости. Он может использовать реальный объект (если только тот не меняется в процессе транзакции). Соответственно, когда процесс начинает транзакцию, он должен создать закрытое рабочее пространство, в котором ничего нет, за исключением указателя на рабочее пространство своего родителя. Если это транзакция верхнего уровня, рабочим пространством родителя будет файловая система. Когда процесс открывает файл на чтение, он следует за указателями, пока файл не обнаружится в рабочем пространстве родителя (или более дальнего предка).

Когда файл открывается на запись, он может быть найден таким же образом, как и при чтении, за исключением того, что сначала он будет скопирован в закрытое рабочее пространство. Однако при втором варианте оптимизации удастся отказаться и от большей части копирования. Вместо копирования целого файла в закрытое рабочее пространство можно копировать только индекс файла. Индекс — это блок данных, ассоциированных с каждым файлом и хранящий информацию о том, где на диске расположены его блоки. В UNIX индекс — это индексный узел (inode). При использовании внутреннего индекса файл может быть считан точно так же, поскольку диск, к которому происходит адресация, — это тот же самый диск с исходным расположением блоков. Однако когда блок файла модифицируется, создается копия этого блока, и в индекс вставляется адрес копии, как показано на рис. 5.18, а. После этого блок можно изменять, не оказывая влияния на оригинал. Добавляемые блоки обрабатываются точно так же. Новые блоки иногда называют *теневыми блоками* (*shadow blocks*).

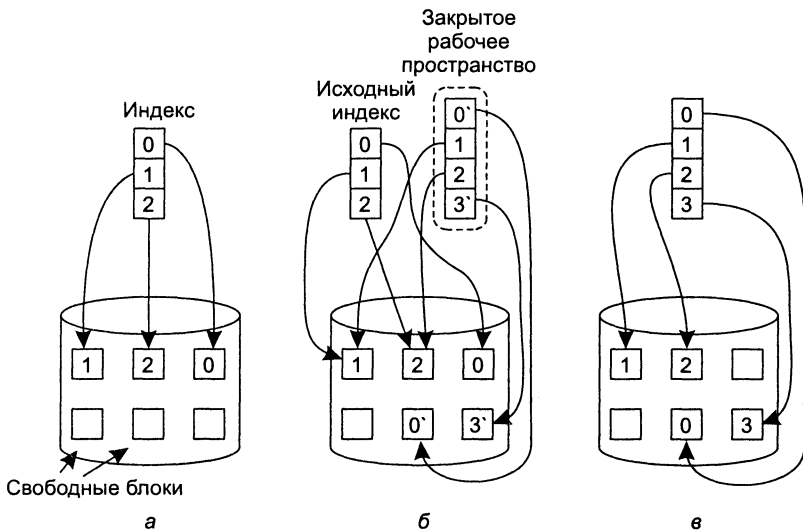


Рис. 5.18. Индекс файла и дисковые блоки для файла из трех блоков (а). Ситуация после того, как транзакция модифицировала блок 0 и добавила блок 3 (б). Ситуация после подтверждения транзакции (в)

Как можно видеть на рис. 5.18, б, процесс, выполняющий транзакцию, видит модифицированный файл, а все остальные процессы продолжают работать с ис-

ходным файлом. В особенно сложных транзакциях закрытое рабочее пространство может содержать не один, а большое количество файлов. Если транзакция прерывается, закрытое рабочее пространство просто удаляется и все закрытые блоки, которые оно содержало, возвращаются в список свободных. Если транзакция завершается, закрытые блоки в ходе атомарной операции перемещаются в рабочее пространство родителя, как показано на рис. 5.18, в. Блоки, которые больше не нужны, возвращаются в список свободных.

Эта схема работает также и для распределенных транзакций. В этом случае аналогичный процесс начинается на каждой машине, содержащей файлы, к которым в ходе транзакции будет производиться доступ. Каждый процесс получает свое собственное закрытое рабочее пространство так, как описано выше. Если транзакция прерывается, все процессы просто отказываются от своих закрытых рабочих пространств. С другой стороны, при подтверждении транзакции производятся локальные обновления, после чего транзакции можно считать полностью завершенными.

Журнал с упреждающей записью

Другой традиционный метод реализации транзакций — это *журнал с упреждающей записью* (*write-ahead log*). Согласно этому методу файлы действительно модифицируются там же, где находятся, но перед тем, как какой-либо блок действительно будет изменен, в журнал заносится запись со сведениями о том, какая транзакция вносит изменения, какой файл и блок изменяются, каковы прежние и новые значения. Только после успешной записи в журнал изменения вносятся в файл.

Ниже описано, как работает этот метод. В листинге 5.3 представлена простая транзакция, которая использует две общие переменные (или другие объекты), x и y , инициализированные нулями.

Листинг 5.3. Транзакция

```
x = 0;
y = 0;
BEGIN_TRANSACTION;
    x = x + 1;
    y = y + 2;
    x = y * y;
END_TRANSACTION;
```

Для каждой из трех инструкций тела транзакции до начала ее выполнения создается запись в журнале, которая содержит прежние и новые значения, разделенные косой чертой:

- ◆ содержимое журнала перед выполнением первой инструкции ($x=x+1$):

[$x=0/1$]

- ◆ содержимое журнала перед выполнением второй инструкции ($y=y+2$):

[$x=0/1$]

[$y=0/2$]

♦ содержимое журнала перед выполнением третьей инструкции ($x=y*u$):

[$x=0/1$]

[$y=0/2$]

[$x=1/4$]

Если транзакция успешна, она подтверждается и в журнал добавляется запись о подтверждении, но структуры данных не изменяются, поскольку они и так изменены. Если транзакция прерывается, журнал используется для восстановления первоначального состояния. Начиная с конца и продвигаясь к началу, из журнала считываются записи и описанные в них изменения отменяются. Это действие называется *откатом* (*rollback*) транзакции.

Как и предыдущая, эта схема также может использоваться для работы с распределенными транзакциями. В этом случае каждая машина поддерживает собственный журнал изменений в локальной файловой системе. Откат в случае прерывания транзакции каждая машина производит по отдельности, восстанавливая свои исходные файлы.

5.6.4. Управление параллельным выполнением транзакций

Итак, мы рассмотрели основные способы достижения атомарности транзакций. Достижение атомарности (и долговечности) транзакций при наличии сбоев — это важнейшая тема, к которой мы вернемся в главе 7, когда будем рассматривать не только транзакции. Такие свойства, как непротиворечивость и изолированность, основаны на возможности управления выполнением параллельных транзакций, то есть транзакций, выполняющихся одновременно и над одним и тем же набором совместно используемых данных.

Цель управления параллельным выполнением транзакций состоит в том, чтобы позволить нескольким транзакциям выполняться одновременно, но таким образом, чтобы набор обрабатываемых элементов данных (например, файлов или записей базы данных) оставался непротиворечивым. Непротиворечивость достигается в результате того, что доступ транзакций к элементам данных организуется в определенном порядке так, чтобы конечный результат был таким же, как и при выполнении всех транзакций последовательно.

Управление параллельным выполнением лучше всего можно понять в терминах трех менеджеров, организованных по уровням, как это показано на рис. 5.19. Нижний уровень предоставляет *менеджера данных* (*data manager*), который физически осуществляет операции чтения и записи данных. Менеджер данных ничего не знает о том, какая транзакция выполняет чтение или запись. На самом деле он вообще ничего не знает о транзакциях.

Средний уровень представлен *планировщиком* (*scheduler*). Он несет основную ответственность за правильность управления параллельной работой, определяя, какой транзакции и в какой момент разрешается передать операцию чтения или записи менеджеру данных. Он также планирует отдельные операции чтения и записи с теми же гарантиями непротиворечивости и изолированности, что и для

транзакций. Позже мы рассмотрим планирование, основанное на использовании блокировок, и планирование, основанное на использовании отметок времени.

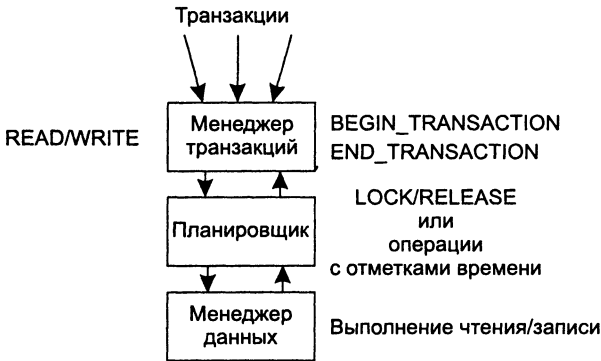


Рис. 5.19. Обобщенная организация менеджеров для управления транзакциями

На самом верхнем уровне находится *менеджер транзакций* (*transaction manager*), который отвечает, прежде всего, за атомарность и долговечность. Он обрабатывает примитивы транзакций, преобразуя их в запросы к планировщику.

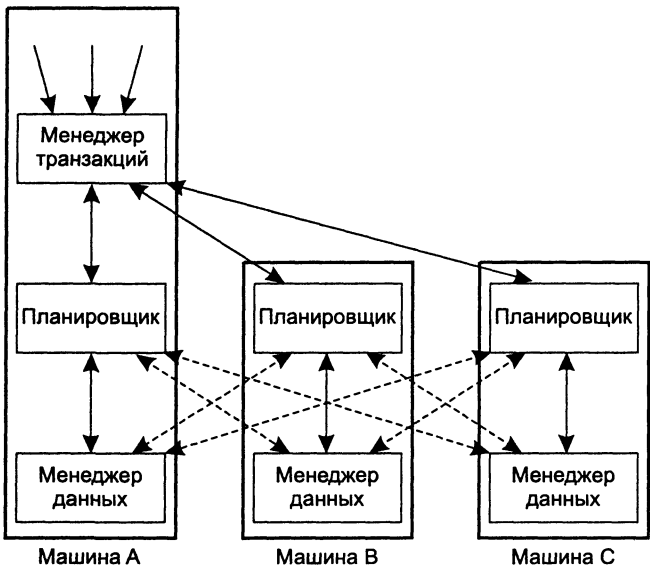


Рис. 5.20. Обобщенная организация менеджеров для управления распределенными транзакциями

Модель, приведенная на рис. 5.19, может быть адаптирована к использованию в распределенных системах, как показано на рис. 5.20. Каждая машина в этом случае имеет своих планировщика и менеджера данных, которые совместно обес-

печивают гарантии непротиворечивости локальных данных. Каждая транзакция обрабатывается одним менеджером транзакций. Последний работает с планировщиками отдельных машин. В зависимости от алгоритма управления параллельным выполнением транзакций планировщик также может работать с удаленными менеджерами данных. Мы вернемся к разговору о распределенном управлении параллельным выполнением чуть ниже.

Изолированность

Основная задача алгоритмов управления параллельным выполнением — гарантировать возможность одновременного выполнения многочисленных транзакций до тех пор, пока они изолированы друг от друга. Это значит, что итоговый результат их выполнения будет таким же, как если бы эти транзакции выполнялись одна за другой в определенном порядке.

В листинге 5.4 представлены три транзакции, которые выполняются одновременно в трех отдельных процессах. Если бы они выполнялись последовательно, итоговыми значениями переменной x были бы 1, 2 или 3, в зависимости от того, какая из транзакций выполнялась бы последней (x может быть совместно используемой переменной, файлом или сущностью другого типа).

Листинг 5.4. Три транзакции T_1 , T_2 и T_3

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 1;
END_TRANSACTION
```

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 2;
END_TRANSACTION
```

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 3;
END_TRANSACTION
```

В табл. 5.3 показаны различные способы упорядочивания, называемые *планами (schedules)*. План 1 реально сериализован. Другими словами, транзакции выполняются строго последовательно, так что они по определению удовлетворяют условию сериализуемости. План 2 не сериализован, но также допустим, поскольку в результате его выполнения значение x , которое будет получено после отработки транзакций, соответствует значению, полученному при строго последовательном выполнении транзакций. Третий план недопустим, потому что в результате его выполнения x становится равным 5, что при последовательном выполнении транзакций невозможно. Системе необходимо убедиться, что отдельные операции перемежаются правильно. Давая системе свободу выбора любой очередности операций и проверяя корректность получаемых результатов, мы избавляем программистов от необходимости реализовывать собственные взаимные исключения, что упрощает их работу.

Таблица 5.3. Возможные планы по очередности выполнения операций

Номер плана	Очередность операций						Правильность плана
1	x=0;	x=x+1;	x=0;	x=x+2;	x=0;	x=x+3;	Правильно
2	x=0;	x=0;	x=x+1;	x=x+2;	x=0;	x=x+3;	Правильно
3	x=0;	x=0;	x=x+1;	x=0;	x=x+2;	x=x+3;	Неправильно

Для того чтобы разобраться в планах и управлении параллельным выполнением, нет необходимости точно знать, как именно переменные будут вычисляться. Другими словами, неважно, будет ли значение x увеличено на 2 или 3. Что действительно важно, так это то, что значение x изменится. Соответственно, мы можем описать транзакцию как наборы операций чтения и записи определенных элементов данных. Так, например, каждая из трех транзакций, T_1 , T_2 и T_3 , приведенных в листинге 5.4, может быть представлена в виде последовательностей:

```
write(Ti,x); read(Ti,x); write(Ti,x)
```

Основная идея управления параллельным выполнением состоит в том, чтобы правильно спланировать *конфликтующие операции* (*conflicting operations*). Две операции конфликтуют, если они работают с одним и тем же элементом данных, и как минимум одна из них — это операция записи. В *конфликте чтения-записи* (*read-write conflict*) запись — это одна из операций. Кроме того, мы имеем дело и с *конфликтом двойной записи* (*write-write conflict*), при этом не имеет значения, принадлежат ли конфликтующие операции к одной транзакции или к разным. Следует отметить, что две операции чтения никогда не конфликтуют между собой.

Алгоритмы управления параллельным выполнением обычно классифицируются по способу синхронизации операций чтения и записи. Синхронизация может производиться посредством механизма взаимного исключения совместно используемых данных (то есть блокировки) или явного упорядочивания операций с помощью отметок времени.

Дальнейшее деление можно провести между оптимистическим и пессимистическим управлением параллельным выполнением. Для *пессимистических подходов* фундаментальным является закон Мерфи, согласно которому если что-нибудь можно сделать неправильно, это что-то будет сделано неправильно. При пессимистических подходах операции синхронизируются до их выполнения, и, таким образом, конфликты разрешаются до того, как они проявятся. В противоположность этому, *оптимистические подходы* основаны на идее о том, что обычно ничего плохого не случается. Поэтому операции без затей выполняются, а синхронизация производится в конце транзакции. Если в этот момент обнаруживается конфликт, одна или более транзакций прерываются. Далее мы рассмотрим два пессимистических и один оптимистический методы. Прекрасный обзор различных подходов можно найти в [46].

Двухфазная блокировка

Самый старый и наиболее широко используемый алгоритм управления параллельным выполнением транзакций — это *блокировка* (*locking*). В своей простей-

шей форме, когда процесс в ходе транзакции нуждается в чтении или записи элемента данных, он просит планировщик заблокировать для него этот элемент данных. Точно так же, когда необходимость в этом элементе данных исчезает, планировщика просят снять блокировку. Задача планировщика состоит в том, чтобы устанавливать и снимать блокировку таким образом, чтобы получать только допустимые планы выполнения. Другими словами, мы нуждаемся в применении алгоритма, предоставляющего нам только сериализуемые планы. Один из таких алгоритмов — двухфазная блокировка.

При *двухфазной блокировке* (*Two-Phase Locking, 2PL*), которая продемонстрирована на рис. 5.21, планировщик сначала, на *фазе подъема* (*growing phase*), устанавливает все необходимые блокировки, а затем, на *фазе спада* (*shrinking phase*), снимает их. Говоря конкретнее, выполняются три правила, описанные в [47].

- ♦ Когда планировщик получает операцию $oper(T, x)$ от менеджера транзакций, он проверяет, не конфликтует ли эта операция с другими операциями, уже получившими блокировку. Если в наличии конфликт, операция $oper(T, x)$ откладывается (и транзакция T вместе с ней). Если конфликта нет, планировщик производит блокировку для элемента данных x и передает операцию менеджеру данных.
- ♦ Планировщик никогда не снимает блокировку с элемента данных x , если менеджер данных уведомляет его, что он осуществляет операцию, в которой участвует этот элемент данных.
- ♦ После того как планировщик снимает с данных блокировку, установленную по требованию транзакции T , он никогда не делает по требованию этой транзакции другую блокировку, при этом неважно, на какой элемент данных транзакция T требует установить блокировку. Любые попытки T потребовать новой блокировки являются ошибкой программирования и приводят к прерыванию транзакции T .

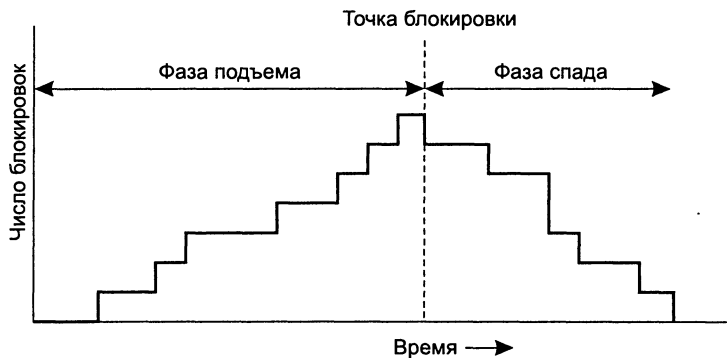


Рис. 5.21. Двухфазная блокировка

Можно доказать [139], что если все транзакции используют двухфазную блокировку, любой план, сформированный путем перекрытия этих транзакций, сериализуем. В этом причина популярности двухфазной блокировки.

Во многих системах фаза спада не начинается до тех пор, пока транзакция не окончится подтверждением или прерыванием, что и приведет к снятию блокировок, как показано на рис. 5.22. Такой режим, именуемый *строгой двухфазной блокировкой* (*strict two-phase locking*), имеет два серьезных преимущества. Во-первых, транзакция всегда считывает значения, записанные подтвержденной транзакцией, поэтому невозможно прерывание транзакции из-за того, что ее вычисления базируются на недоступных данных. Во-вторых, любые наложения и снятия блокировок могут выполняться системой без использования транзакций: любые блокировки устанавливаются при доступе к элементу данных и снимаются по окончании транзакции. При таком поведении устраняются *каскадные прерывания* (*cascaded aborts*): отмена подтвержденной транзакции по той причине, что она получает элемент данных, который получать не должна.

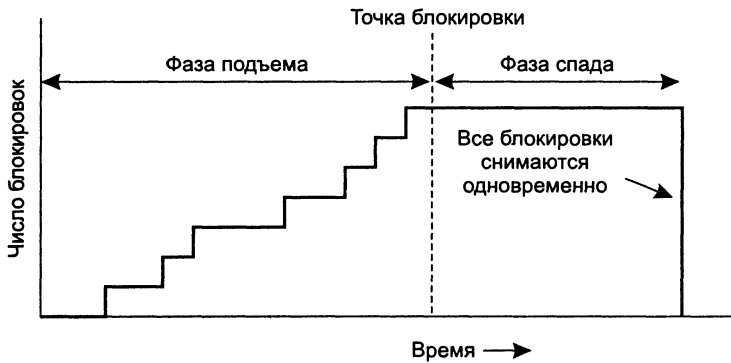


Рис. 5.22. Строгая двухфазная блокировка

Как двухфазная блокировка, так и строгая двухфазная блокировка могут привести к тупикам. Если два процесса пытаются заблокировать одну и ту же пару блоков, но в разном порядке, результатом будет *взаимная блокировка*, или *тупик* (*deadlock*). В этом случае используются традиционные технологии предотвращения тупиков, такие как выполнение блокировок в некотором строго заданном порядке. Также возможно обнаружение взаимных блокировок путем составления полного графа транзакций, показывающего, какой процесс какую блокировку создал и какую хочет создать, с последующей проверкой этого графа на циклы. И наконец, когда заранее известно, что никакая блокировка не может при нормальной работе длиться более t секунд, можно использовать схему с тайм-аутом: если блокировка непрерывно принадлежит процессу более t секунд, это может происходить из-за взаимной блокировки.

Существует несколько способов реализации двухфазной блокировки в распределенных системах. Допустим, что данные, с которыми мы работаем, разбросаны по нескольким машинам. При *централизованной двухфазной блокировке* (*centralized two-phase locking*) за установку и снятие блокировок отвечает одна машина. Все менеджеры транзакций взаимодействуют с одним централизованным менеджером блокировок, который принимает от них запросы на блокиров-

ку. После того как блокировка установлена, менеджер транзакций работает непосредственно с менеджерами данных. Отметим, что при такой схеме элементы данных могут быть также реплицированы на несколько машин. После выполнения операции менеджер транзакций возвращает блокировку менеджеру блокировок.

При *первичной двухфазной блокировке (primary two-phase locking)* с каждого элемента данных делается первичная копия. Блокировки устанавливает и снимает менеджер блокировок той машины, на которой размещена копия. Первичная двухфазная блокировка работает практически так же, как и централизованная, за исключением того, что блокировка может быть распределена по нескольким машинам.

И, наконец, при *распределенной двухфазной блокировке (distributed two-phase locking)* предполагается, что данные могут быть распределены по нескольким машинам. В противоположность первичной и централизованной двухфазной блокировке планировщики каждой из машин отвечают не только за установку и снятие блокировок, но и за пересылку операций менеджерам данных (локальным). В этом смысле распределенная двухфазная блокировка значительно ближе к базовой схеме двухфазной блокировки, но выполняется теперь на всех машинах, содержащих данные.

Классическое исследование двухфазной блокировки для систем баз данных и вообще управления параллельным выполнением можно найти в [47].

Пессимистическое упорядочение по отметкам времени

Абсолютно другой подход к управлению параллельным выполнением состоит в том, чтобы в момент начала каждой из транзакций T присваивать ей отметку времени $ts(T)$. Используя алгоритм Лампорта, мы можем убедиться в уникальности отметок времени, что в данном случае необходимо. Каждая операция, являющаяся частью транзакции T , также получает отметку времени $ts(T)$. Кроме того, каждый элемент данных x в системе получает отметку времени чтения $ts_{RD}(x)$ и отметку времени записи $ts_{WR}(x)$. Отметка времени чтения соответствует отметке времени транзакции, которая последней считывала x , а отметка времени записи соответствует отметке времени транзакции, которая последней записывала x . При использовании упорядочивания по отметкам времени, если две операции вступают в конфликт, менеджер данных выполняет сначала операцию с наименьшей отметкой.

Теперь представим себе, что планировщик получает от транзакции T с отметкой времени ts операцию $read(T, x)$, причем $ts < ts_{WR}(x)$. Другими словами, он уведомляется о том, что операция записи значения x была произведена до начала выполнения T . В этом случае транзакция T просто прерывается. С другой стороны, если $ts > ts_{WR}(x)$, все правильно, и операция чтения выполняется обычным образом. Кроме того, $ts_{RD}(x)$ устанавливается в значение $\max\{ts, ts_{RD}(x)\}$.

Точно таким же образом обстоит дело и при получении планировщиком от транзакции T с отметкой времени ts операции записи $write(T, x)$. Если $ts < ts_{RD}(x)$, нам остается только прервать транзакцию, поскольку текущее значение x было считано предыдущей транзакцией. Транзакция T просто слишком запоздала.

С другой стороны, если $ts > ts_{RD}(x)$, то, поскольку никакая предыдущая транзакция не считывала значения x , нам предстоит изменить его. Таким же образом $ts_{WR}(x)$ устанавливается в $\max\{ts, ts_{WR}(x)\}$.

Для того чтобы лучше разобраться в упорядочении по отметкам времени, рассмотрим следующий пример. Представим себе три транзакции, T_1 , T_2 и T_3 . Транзакция T_1 отработала уже давно и использовала все элементы данных, необходимые транзакциям T_2 и T_3 , так что на всех этих элементах стоят отметки времени чтения и записи $ts(T_1)$. Транзакции T_2 и T_3 начинаются параллельно с отметками $ts(T_2) < ts(T_3)$.

Рассмотрим сначала чтение транзакцией T_2 элемента данных x . Пока транзакция T_3 не подтверждена, обе отметки, $ts_{WR}(x)$ и $ts_{RD}(x)$, будут иметь значение $ts(T_1)$, которое меньше, чем $ts(T_2)$. На рис. 5.23, *а* и *б* мы видим, что $ts(T_2)$ больше, чем $ts_{WR}(x)$ и $ts_{RD}(x)$, так что запись допустима и будет предварительно выполнена (транзакция T_3 еще не подтверждена). Она станет постоянной после завершения T_2 . Теперь у элемента данных будет новая отметка времени записи, отметка времени транзакции T_2 .

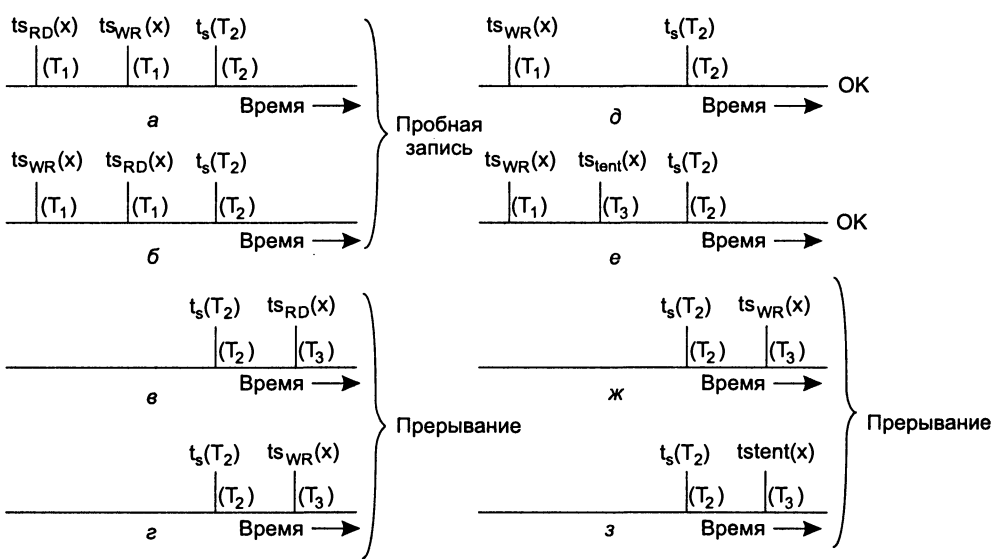


Рис. 5.23. Управление параллельным выполнением транзакций с использованием отметок времени

На рис. 5.23, *в* и *г* транзакции T_2 не везет. Транзакция T_3 производит чтение (рис. 5.23, *в*) или запись (рис. 5.23, *г*) x и подтверждается. Транзакция T_2 прерывается. Однако она может получить новую отметку времени и стартовать снова.

Рассмотрим теперь чтение. На рис. 5.23, *д* конфликтов нет, и чтение может быть произведено немедленно. На рис. 5.23, *е* в середину встывает какой-то бездельник и пытается записать x . Отметка времени этого ловкача меньше, чем у T_2 , и поэтому T_2 просто дожидается, пока тот завершится, после чего считывает новый файл и продолжает работу.

На рис. 5.23, ж транзакция T_3 изменяет значение x и вновь подтверждается. А транзакции T_2 снова приходится прерываться. На рис. 5.23, з транзакция T_3 находится в процессе изменения x до тех пор, пока не подтверждается. Транзакция T_2 снова чересчур запоздала и должна быть прервана.

Свойства отметок времени и блокировки различны. Когда транзакция сталкивается с большими (устаревшими) отметками времени, она прерывается, блокировка же в сходных условиях либо ожидает, либо может быть выполнена немедленно. С другой стороны, при работе с отметками времени не бывает взаимных блокировок, а это большой плюс.

Базовое упорядочение при помощи отметок времени существует в нескольких вариантах, среди которых следует отметить устойчивое упорядочение и многовариантное упорядочение. Подробности можно найти в [177, 337].

Оптимистическое упорядочение по отметкам времени

Третий подход к выполнению нескольких транзакций в одно и то же время — это оптимистическое упорядочение по отметкам времени [244]. Идея, лежащая в основе этого подхода, удивительно проста: идите вперед и делайте то, что вы хотите, не обращая внимания на то, что одновременно происходит что-то еще. Если возникнут проблемы, волноваться об этом мы будем после (этот алгоритм используют также многие политики). На практике конфликты относительно редки, так что большую часть времени все идет гладко.

Конфликты могут быть нечастыми, но они все же случаются, и нам необходим способ их разрешения. Оптимистическое управление отслеживает случаи чтения и записи элементов данных. В момент подтверждения транзакции проверяется, не был ли с момента ее начала изменен какой-то элемент данных, нужный другим транзакциям. Если был, транзакция прерывается, если не был, транзакция подтверждается.

Оптимистическое управление параллельным выполнением транзакций отлично реализуется на основе закрытых рабочих пространств. В этом случае каждая транзакция изменяет свои данные в частном порядке, не пересекаясь с другими. В конце работы новые данные либо принимаются, либо отбрасываются, что приводит нас к относительно простой и понятной схеме работы.

Большое преимущество оптимистического управления параллельным выполнением транзакций состоит в том, что при его использовании достигается максимальный параллелизм и не бывает взаимных блокировок, поскольку никакой процесс никогда не ждет завершения блокировки. Недостаток такого управления в том, что временами оно дает сбой, и нам приходится начинать транзакцию снова. В условиях высокой загрузки вероятность сбоев может существенно возрасти, что делает оптимистическое управление плохим выбором.

Как указывалось в [337], исследования оптимистического управления параллельным выполнением транзакций были сосредоточены в первую очередь на нераспределенных системах. Кроме того, создание коммерческих систем и прототипов сильно затруднено, что делает невозможным сравнение подобных систем с другими.

5.7. Итоги

С взаимодействием между процессами тесно связан вопрос синхронизации процессов в распределенных системах. Синхронизация — это все то, что позволяет нам делать правильные вещи в правильное время. Проблема распределенных систем и вообще компьютерных сетей состоит в том, что для них не существует понятия единых совместно используемых часов. Другими словами, процессы на различных машинах имеют свое собственное мнение о времени.

Существуют различные способы синхронизации часов в распределенных системах, но все они основаны на обмене показаниями часов, а это требует учитывать задержки на посылку и получение сообщений. Точность алгоритмов синхронизации в значительной мере определяют вариации в задержках доступа и способы учета этих вариаций.

Во многих случаях знания абсолютного времени не требуется. Достаточно, чтобы соответствующие события в различных процессах происходили в правильной последовательности. Лампорт показал, что, введя понятие логических часов, можно заставить набор процессов соблюдать общее соглашение относительно правильной очередности событий. В сущности, каждое событие e , такое как посылка или прием сообщения, получает абсолютно уникальную логическую отметку времени $C(e)$, такую, что если событие a происходит раньше b , то $C(a) < C(b)$. Отметки времени Лампорта могут быть расширены до векторных отметок времени: если $C(a) < C(b)$, то мы всегда знаем, что событие a причинно предшествует событию b .

Поскольку в распределенных системах нет понятия совместно используемой памяти, часто трудно определить текущее состояние системы. Определение глобального состояния распределенной системы можно выполнить путем синхронизации всех процессов так, чтобы каждый из них определил и сохранил свое локальное состояние вместе с сообщениями, которые передаются в этот момент. Сама по себе синхронизация может быть выполнена без остановки процессов и записи их состояния. Вместо этого в ходе работы распределенной системы с нее можно сделать мгновенный слепок — распределенный снимок состояния.

Синхронизация между процессами часто требует, чтобы один из процессов выступал в роли координатора. В этом случае если координатор не фиксирован, необходимо, чтобы процессы в распределенных вычислениях могли решить, который из них будет координатором. Это решение принимается посредством алгоритмов голосования. Алгоритмы голосования, как правило, задействуются при сбое или отключении существовавшего координатора.

Важный класс алгоритмов синхронизации — распределенные взаимные исключения. Эти алгоритмы гарантируют, что в распределенном наборе процессов доступ к совместно используемым ресурсам имеет максимум один процесс. Распределенные взаимные исключения можно легко обеспечить с помощью координатора, который будет следить, чья сейчас очередь. Существуют также и полностью распределенные алгоритмы, но их минус — чересчур большая чувствительность к сбоям процессов и связи.

С взаимными исключениями тесно связаны распределенные транзакции. Транзакция состоит из набора операций с совместно используемыми данными, при этом вся транзакция целиком либо полностью выполняется, либо полностью не выполняется. Кроме того, несколько транзакций могут выполняться одновременно, при этом результат оказывается таким, как если бы эти транзакции выполнялись в некоторой заданной очередности. И наконец, транзакции долговечны, в том смысле, что после завершения их результат неизменен.

Вопросы и задания

1. Назовите как минимум три источника задержек между радиостанцией WWV, которая рассылает сигналы точного времени, и процессором в распределенной системе, устанавливающим свои внутренние часы.
2. Рассмотрим поведение двух машин в распределенной системе. Обе они имеют часы, которые считаются выставленными на 1000 тиков в миллисекунду. Одни из часов действительно выдают такую частоту тиков, другие же тикают всего 990 раз в миллисекунду. Если поправки UTC приходят раз в минуту, какое максимальное расхождение может возникнуть между часами?
3. Добавьте к рис. 5.7 новое сообщение одновременно с сообщением *A* так, чтобы ни до сообщения *A*, ни после него ничего не происходило.
4. Является ли абсолютно необходимым подтверждение каждого сообщения для полностью упорядоченной групповой рассылки с отметки времени по Лампорту?
5. Рассмотрим коммуникационный уровень, в котором сообщения доставляются в том же порядке, в котором они были отправлены. Приведите пример, когда даже такое упорядочение излишне строго.
6. Представьте себе, что два процесса одновременно обнаруживают передачу координатора и решают провести голосование по алгоритму забияки. Что произойдет в этом случае?
7. На рис. 5.12 мы видим два одновременно курсирующих сообщения *ГОЛОСОВАНИЕ*. Хотя мы не испытываем проблем с тем, что их два, было бы симпатичнее убрать одно из них. Придумайте алгоритм, который делал бы это, не оказывая влияния на основной алгоритм голосования.
8. Многие распределенные алгоритмы требуют наличия координирующего процесса. В какой степени такие алгоритмы могут считаться распределенными?
9. При централизованном подходе к взаимным исключениям (см. рис. 5.13) после получения сообщения от процесса о том, что он не нуждается больше в исключительном доступе к критической области, которую использовал, координатор обычно разрешает доступ первому процессу в очереди. Приведите другой возможный алгоритм работы координатора.
10. Рассмотрим снова рис. 5.13. Представьте себе, что координатор стал неактивным. Обязательно ли это приведет к сбою системы? Если нет, при каких

условиях это произойдет? Существует ли способ решить проблему и сделать систему устойчивой к потере координатора?

11. Алгоритм, предложенный в [380], имеет недостаток. Если в процессе возникает сбой и он перестает отвечать другим процессам на их запросы на вход в критическую область, отсутствие ответа интерпретируется как запрет доступа. Мы предложили, чтобы ответ на все запросы приходил немедленно, дабы проще было обнаружить сбойные процессы. Существуют ли такие условия, при которых этот метод окажется неприемлемым?
12. Как изменяться элементы в табл. 5.1, если мы предположим, что алгоритмы реализуются в локальной сети, которая поддерживает аппаратную широковещательную рассылку?
13. Распределенные системы могут иметь множество независимых критических областей. Представим себе, что процесс 0 хочет войти в критическую область A , а процесс 1 хочет войти в критическую область B . Приведет ли использование алгоритма, предложенного в [380], к взаимной блокировке? Поясните свой ответ.
14. Рассматривая рис. 5.16, мы говорили об атомарном обновлении списка товаров с использованием магнитных лент. Как вы полагаете, почему, несмотря на простоту эмуляции магнитных лент при помощи диска (в качестве файла), этот способ все же не используется?
15. В табл. 5.3 представлены три плана, два приемлемых и один неприемлемый. Для этих же транзакций (см. листинг 5.4) создайте полный список значений x , которые могут получиться в результате, и укажите, какие из них допустимы, а какие нет.
16. Когда для реализации транзакций над файлами используется закрытое рабочее пространство, может случиться так, что в родительское рабочее пространство придется копировать большое количество индексов файлов. Как это сделать, не вводя условий гонок?
17. Приведите полный алгоритм попытки блокирования файла с успешным и неуспешным результатами. Рассмотрите блокировку на чтение и на запись, а также возможность файла быть в момент попытки неблокированным, заблокированным на чтение, заблокированным на запись.
18. Системы, использующие блокировки для управления параллельным выполнением транзакций, обычно отличают блокировку на чтение от блокировки на запись. Что произойдет, если процесс всегда устанавливал блокировку на чтение, а теперь хочет заменить ее блокировкой на запись? Что произойдет в случае замены блокировки на запись блокировкой на чтение?
19. Используя упорядочение по отметке времени в распределенных системах, предположим, что операция записи $write(T_1, x)$ может быть передана менеджеру данных, поскольку единственная грозящая конфликтом операция $write(T_2, x)$ имеет меньшую отметку. Почему имеет смысл потребовать от планировщика повторно передать операцию $write(T_1, x)$ менеджеру данных после завершения транзакции T_2 ?

20. Является ли оптимистическое управление параллельным выполнением транзакций более (или менее) строгим, чем использование отметок времени? Почему?
21. Гарантирует ли использование отметок времени в управлении параллельным выполнением транзакций сериализуемость процесса? Почему?
22. Мы много раз говорили, что в случае прерывания транзакции мир возвращается в свое первоначальное состояние, как если бы никакой транзакции не было. Мы лгали. Приведите пример, когда вернуть мир в исходное состояние невозможно.

Глава 6

Непротиворечивость и репликация

6.1. Обзор

6.2. Модели непротиворечивости, ориентированные на данные

6.3. Модели непротиворечивости, ориентированные на клиента

6.4. Протоколы распределения

6.5. Протоколы непротиворечивости

6.6. Примеры

6.7. Итоги

Важным вопросом для распределенных систем является репликация данных. Данные обычно реплицируются для повышения надежности и увеличения производительности. Одна из основных проблем при этом — сохранение непротиворечивости реплик. Говоря менее формально, это означает, что если в одну из копий вносятся изменения, то нам необходимо обеспечить, чтобы эти изменения были внесены и в другие копии, иначе реплики больше не будут одинаковыми. В этой главе мы подробно рассмотрим, что в действительности означает непротиворечивость реплицируемых данных и различные способы, которыми она достигается.

Мы начнем с общего обзора, в ходе которого обсудим, для чего используется репликация и как она влияет на масштабируемость. Особое внимание мы уделим репликации на базе объектов, важность которой для распределенных систем в настоящее время растет.

Чтобы добиться высокой производительности в операциях с совместно используемыми данными, разработчики параллельных компьютеров уделяют большое внимание различным моделям непротиворечивости данных в распределенных системах с разделяемой памятью. Эти модели, одинаково успешно применяемые для различных типов распределенных систем, подробно рассматриваются в этой главе.

Реализация моделей непротиворечивости разделяемой памяти для крупномасштабных распределенных систем обычно представляет собой нелегкую задачу.

Однако во многих случаях можно использовать простые модели, которые не сложно реализовать. Один из конкретных классов подобных моделей — клиентские модели непротиворечивости, которые ограничиваются непротиворечивостью с точки зрения одного (возможно, мобильного) клиента. Клиентские модели непротиворечивости рассматриваются в отдельном разделе.

Непротиворечивость — это еще не все. Мы должны также обсудить, как эта непротиворечивость реализуется на практике. В поддержании непротиворечивости реплик играют роль два более или менее независимых аспекта. Первый из них — это реальное распространение обновлений. Сюда входят вопросы размещения реплик и того, как по ним расходятся обновления. Мы опишем и сравним несколько протоколов распространения.

Второй аспект касается поддержания непротиворечивости реплик. В большинстве случаев приложениям необходим жесткий вариант непротиворечивости. Говоря неформально, это означает, что обновления должны расходиться по репликам более или менее быстро. Существует несколько альтернативных реализаций строгой непротиворечивости, которые будут рассмотрены в отдельном разделе. Кроме того, внимание уделяется протоколам кэширования, представляющими собой особую форму протоколов поддержания непротиворечивости.

Мы закончим эту главу двумя примерами приложений, в которых активно используются непротиворечивость и репликация. Первый пример относится к области параллельного программирования и упоминается также при обсуждении распределенных систем на базе объектов в главе 9. Второй пример охватывает вопросы причинной непротиворечивости и того, что называется медленной репликацией.

6.1. Обзор

Этот раздел мы начнем с разговора об основных доводах в пользу репликации данных. Специальное внимание будет уделено реплицируемым объектам, к которым в современных распределенных системах проявляется все больший интерес. И наконец, мы обсудим репликацию, как метод улучшения масштабируемости, и расскажем, почему рассуждения о непротиворечивости столь важны для нас.

6.1.1. Доводы в пользу репликации

В пользу репликации есть два основных довода — надежность и производительность. Во-первых, данные реплицируются для повышения надежности системы. Если файловая система реплицирована, она может продолжать свою работу после сбоя в одной из реплик, просто переключившись на другую. Кроме того, поддерживая несколько копий, легче противостоять сбоям данных. Так, представим себе, что существует три копии некоего файла и операции чтения и записи производятся со всеми тремя. Мы можем защитить себя от единичной неудачной операции записи, считая правильным значение, которое выдают как минимум две копии.

Другой довод в пользу репликации данных — производительность. Репликация повышает производительность, когда распределенную систему приходится масштабировать на множество машин и географических зон. Масштабирование на множество машин имеет место, например, когда возрастает число процессов, требующих доступа к данным, управляемым одним сервером. В этом случае производительность можно повысить путем репликации сервера с последующим разделением общего объема работы между сервером и репликами. Мы уже приводили этот пример в главе 1, когда кратко рассматривали кластеры реплицированных web-серверов.

При масштабировании на увеличившуюся географическую зону мы также нуждаемся в репликации. Основная идея состоит в помещении копии данных поблизости от использующего ее процесса, что ведет к сокращению времени доступа. В результате наблюдаемая производительность процесса возрастает. Этот пример иллюстрирует также и то, что выигрыш в производительности, получаемый благодаря репликации, оценить не просто. Хотя в клиентском процессе может наблюдаться рост производительности, возможна ситуация, в которой для своевременного обновления всех реплик потребуется увеличение пропускной способности сети. Мы вернемся к этой теме при обсуждении протоколов распространения.

Если репликация помогает повысить надежность и производительность, что нам может помешать? К сожалению, это цена, которую нам придется заплатить за репликацию данных. Проблема репликации в том, что наличие множества копий может создать проблемы с непротиворечивостью данных. Каждый раз при изменении копии она начинает отличаться от всех прочих. Соответственно, для сохранения непротиворечивости эти изменения должны быть перенесены и на остальные копии. То, как и когда следует переносить эти изменения, и определяет цену репликации.

Чтобы разобраться в проблеме, рассмотрим постоянно растущее время доступа к web-страницам. Если не предпринимать никаких специальных мер, загрузка страницы с удаленного web-сервера может занимать до нескольких секунд. Для повышения производительности web-браузеры часто локально сохраняют копии загруженных ранее web-страниц (то есть *кэшируют* web-страницы). Если пользователь снова запросит подобную страницу, браузер автоматически вернет ему локальную копию. Скорость доступа, с точки зрения пользователя, будет потрясающей. Однако, если пользователь всегда хочет иметь последнюю версию страницы, его может постигнуть разочарование. Проблема состоит в том, что если в промежутке между обращениями страница будет модифицирована, модификации не распространятся на копии в кэше, что сделает эти копии устаревшими.

Одно из решений проблемы возвращения пользователю актуальных копий состоит в том, чтобы запретить браузеру хранить локальные копии, а задачу обновления полностью возложить на сервер. Однако это решение может привести к увеличению времени доступа, ведь тогда у пользователя не будет реплик. Другое решение — позволить web-серверу объявлять кэшированные копии устаревшими или обновлять их, но тогда серверу придется проверять все кэшированные копии и рассылать им сообщения. Это, в свою очередь, может привести

к снижению общей производительности сервера. Ниже мы еще вернемся к конфликту «производительность против масштабируемости».

6.1.2. Репликация объектов

Чтобы лучше понять роль распределенных систем в том, что касается управления данными (совместно используемыми), полезно рассматривать не отдельные данные, а объекты. Достоинство объектов состоит в том, что они инкапсулируют данные и операции над ними. Это облегчает нам разделение операций, которые зависят от данных, и операций, которые от данных не зависят. Последний тип операций обычно встречается в распределенных системах общего назначения, например тех, которые реализуются посредством многих описанных в этой книге систем промежуточного уровня.

Рассмотрим распределенный удаленный объект, совместно используемый многими клиентами (рис. 6.1). До того как мы начнем думать о репликации удаленного объекта на несколько машин, мы должны решить проблему защиты объекта от одновременного доступа нескольких клиентов. У этой проблемы существуют два основных решения [75].

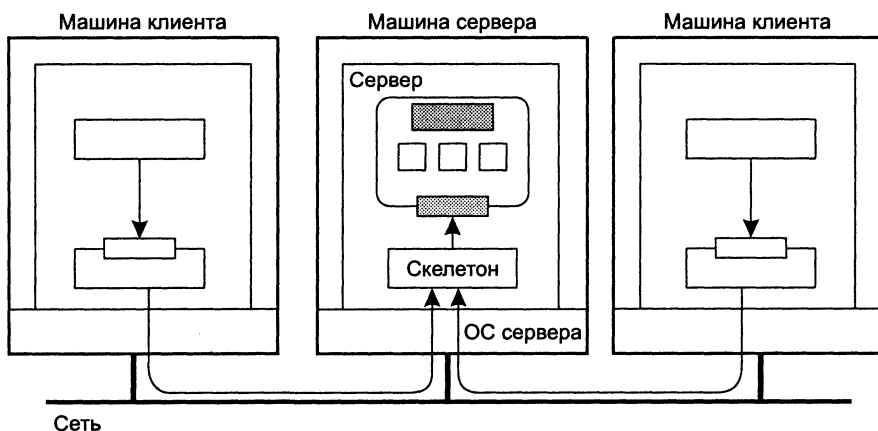


Рис. 6.1. Организация распределенного удаленного объекта, совместно используемого двумя клиентами

Первое из решений состоит в том, что одновременные вызовы обрабатывает сам объект. В качестве примера укажем, что объект Java может быть сконструирован в виде монитора путем объявления его методов *синхронизированными*. Предположим, что два клиента одновременно вызывают метод одного и того же объекта. Это приведет к появлению двух параллельных потоков выполнения на том сервере, на котором находится этот объект. В Java, если методы объекта синхронизированы, выполняться будет только один из потоков выполнения, второй же окажется заблокированным до дальнейших уведомлений. Могут быть созданы различные уровни параллельного выполнения, но основным моментом будет то, что средства обработки параллельных обращений реализует сам объект. Этот принцип иллюстрирует рис. 6.2, а.

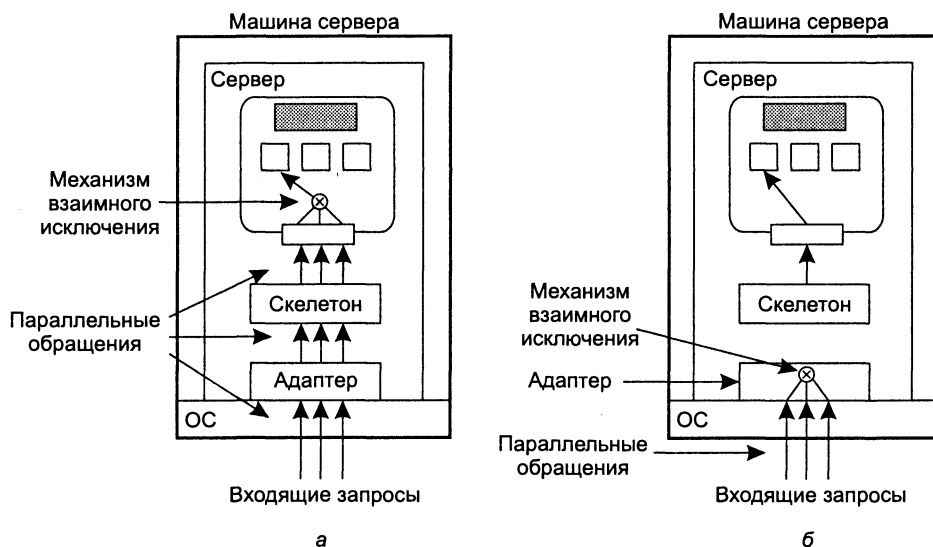


Рис. 6.2. Удаленный объект, способный самостоятельно обрабатывать параллельные обращения (а). Удаленный объект, которому для обработки параллельных обращений необходим адаптер объекта (б)

Второе решение состоит в том, что объект вообще никак не защищается от параллельных обращений, вместо него ответственность за работу с параллельными обращениями несет сервер, на котором расположен объект. В частности, используя соответствующий адаптер объектов, он может обеспечить, чтобы параллельные обращения не повреждали объект. Например, адаптер объектов, использующий для каждого объекта один поток выполнения, может сериализовать весь доступ к каждому из объектов, которыми он управляет (рис. 6.2, б).

Репликация удаленных, совместно используемых объектов без специальных приемов обработки параллельных обращений может привести к проблемам с непротиворечивостью. Эти проблемы связаны с тем, что для параллельных обращений в правильной очередности реплики нуждаются в дополнительной синхронизации. Примером такой синхронизации может быть реплицируемая база данных банковских счетов, описанная в разделе 5.2. И вновь у нас есть два основных способа решения проблемы.

Первый подход основан на осведомленности объекта о том, что он был реплицирован. В этом случае сам объект отвечает за непротиворечивость своей реплики в условиях параллельных обращений. Такой подход весьма напоминает ситуацию с объектами, которые самостоятельно обрабатывают параллельные обращения. Распределенные системы, предлагающие подобные объекты, обычно не нуждаются в централизованной поддержке репликации. Поддержка может быть ограничена предоставлением серверов и адаптеров, помогающих в построении осведомленных о репликации объектов (рис. 6.3, а). Примеры таких систем, которые мы рассмотрим в главе 9, — это SOS [409] и Globe [471]. Преимущество осведомленных о репликации объектов состоит в том, что они могут реализовать специ-

фическую для объекта стратегию репликации, подобно тому, как параллельно работающие объекты могут обрабатывать параллельные обращения каким-нибудь особым образом.

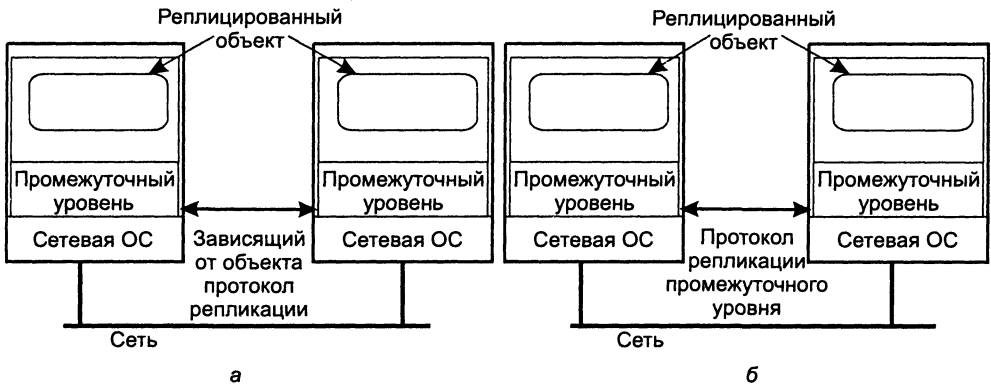


Рис. 6.3. Распределенная система для осведомленных о репликации распределенных объектов (а). Распределенная система, отвечающая за управление репликами (б)

Второй, более общий вариант обеспечения непротиворечивости параллельных объектов — сделать ответственной за управление репликацией распределенную систему, как это показано на рис. 6.3, б. В частности, распределенная система отвечает за то, чтобы параллельные обращения перенаправлялись различным репликам в правильном порядке. Такой подход используется, в частности, в системе Piranha [281], которая предоставляет средства для помехоустойчивых, полностью упорядоченных и причинно упорядоченных обращений к объектам в технологии CORBA. Использование распределенной системы для управления репликацией позволяет упростить разработку приложений. При этом иногда бывает нелегко адаптировать специфические для объектов решения, и в этом недостаток таких систем. Как мы увидим, эти решения часто требуются для масштабирования.

6.1.3. Репликация как метод масштабирования

Репликация и кэширование, увеличивающие производительность, часто используются в качестве способа масштабирования. Масштабирование обычно ведет к проблемам с производительностью. Однако размещение копий данных и объектов неподалеку от использующих их процессов благодаря сокращению времени доступа позволяет повысить производительность и таким образом решить проблемы масштабирования.

Возможно, платой за это будет необходимость сохранения актуальности копий, что потребует дополнительной пропускной способности сети. Рассмотрим процесс P , который обращается к локальной реплике N раз в секунду, в то время как сама реплика обновляется M раз в секунду. Допустим, что обновление полностью изменяет предыдущую версию локальной реплики. Если $N \ll M$, то есть соот-

ношение обращений к обновлениям очень низкое, мы попадаем в такую ситуацию, когда обращений ко многим обновленным версиям со стороны P попросту не будет и задействовать сетевую связь для доставки этих версий бесполезно. В этом случае может быть правильнее не устанавливать локальную реплику вблизи процесса P или применить другую стратегию обновления этой реплики. Ниже мы вернемся к этим вопросам.

Более серьезной проблемой является то, что сохранение актуальности множества копий само по себе может стать серьезной проблемой масштабирования. Интуитивно понятно, что набор копий актуален, если все копии постоянно одинаковы. Это означает, что операция чтения должна давать одинаковые результаты для каждой из копий. Соответственно, при выполнении операции обновления одной из копий обновление должно распространиться на все копии до того, как начнется следующая операция. При этом безразлично, с какой копии началась эта операция.

Такой тип непротиворечивости иногда неформально (и неверно) называют плотной непротиворечивостью, поскольку она поддерживает так называемую синхронную репликацию [78]. (В следующем разделе мы представим точное определение непротиворечивости и введем ряд моделей непротиворечивости.) Ключевой является идея о том, что обновление всех копий проводится как одна атомарная операция или транзакция. К сожалению, реализация атомарности операции, если в нее вовлечено большое число реплик, которые к тому же могут быть разбросаны по узлам крупной сети, изначально затруднена, особенно когда эта операция вдобавок должна производиться за короткое время.

Трудности вытекают из того факта, что мы должны синхронизировать все реплики. Конкретно это означает, что все реплики должны достичь согласия в том, когда точно должно производиться их локальное обновление. Так, например, репликам может потребоваться договориться о глобальном упорядочении операций с использованием механизма отметок времени Лампорта или завести координатора, который будет диктовать им порядок действий. Глобальная синхронизация просто отнимет массу времени на взаимодействие, особенно если реплики разбросаны по глобальной сети.

Итак, мы оказались перед дилеммой. С одной стороны, остроту проблем масштабируемости можно снизить, используя репликацию и кэширование, которые вызовут рост производительности. С другой стороны, чтобы поддерживать все копии в актуальном состоянии, обычно требуется глобальная синхронизация, которая чересчур сильно влияет на производительность. Лекарство может оказаться хуже болезни.

Во многих случаях единственное реальное решение состоит в том, чтобы пожертвовать ограничениями. Другими словами, если мы снимем требование об атомарности операций обновления, мы сможем отказаться от необходимости мгновенной глобальной синхронизации, выиграв при этом в производительности. Ценой за это станет ситуация возможной неодинаковости копий. То, в какой степени мы можем пожертвовать непротиворечивостью, зависит от вариантов доступа к реплицированным данным и вариантов их обновления, а также от задач, в которых используются эти данные.

В следующем разделе мы сначала рассмотрим ряд моделей непротиворечивости, которые позволят нам дать точное определение непротиворечивости. Затем мы продолжим наш разговор о различных способах реализации этих моделей с помощью протоколов распределения и протоколов непротиворечивости. Различные подходы к классификации непротиворечивости и реплицирования можно найти в [176, 492].

6.2. Модели непротиворечивости, ориентированные на данные

По традиции непротиворечивость всегда обсуждается в контексте операций чтения и записи над совместно используемыми данными, доступными в распределенной памяти (разделяемой) или в файловой системе (распределенной). В этом разделе мы задействуем общий термин *хранилище данных* (*data store*). Хранилище данных может быть физически разнесено по нескольким машинам. В частности, каждый из процессов, желающих получить доступ к данным из хранилища, может означать наличие доступной через хранилище локальной (или расположенной неподалеку от него) копии данных. Операции записи распространяются и на другие копии, как это показано на рис. 6.4. Операции над данными называются операциями записи, если они изменяют данные. Все прочие операции считаются операциями чтения.

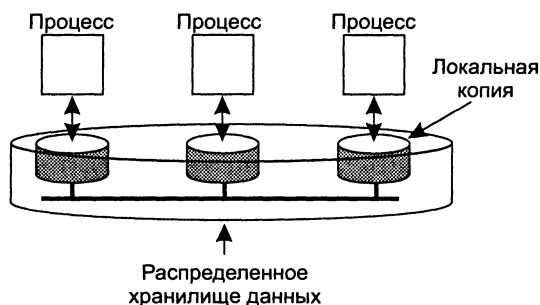


Рис. 6.4. Обобщенная организация логического хранилища данных, физически распределенного и реплицируемого по нескольким процессам

Модель непротиворечивости (consistency model), по существу, представляет собой контракт между процессами и хранилищем данных. Он гласит, что если процессы согласны соблюдать некоторые правила, хранилище соглашается работать правильно. Обычно процесс, выполняющий операцию чтения элемента данных, ожидает, что операция вернет значение, соответствующее результату последней операции записи этих данных.

В отсутствие глобальных часов трудно точно определить, какая из операций записи была последней. Нам нужно ввести другие, альтернативные определения, которые приведут к созданию моделей непротиворечивости. Каждая модель будет

успешно ограничивать набор значений, которые может возвратить операция чтения над элементом данных. Как можно догадаться, модели с минимальным объемом ограничений использовать проще, а модели с максимальными ограничениями — труднее. Плата за это, разумеется, состоит в том, что простые модели работают хуже, чем сложные. Такова жизнь. Дополнительную информацию по моделям непротиворечивости можно почерпнуть в [4, 303].

6.2.1. Строгая непротиворечивость

Наиболее жесткая модель непротиворечивости называется *строгой непротиворечивостью* (*strict consistency*). Она определяется следующим условием: *всякое чтение элемента данных x возвращает значение, соответствующее результату последней записи x .*

Это определение естественно и очевидно, хотя косвенным образом подразумевает существование абсолютного глобального времени (как в классической физике), в котором определение «последней» однозначно. Однопроцессорные системы традиционно соблюдают строгую непротиворечивость, и программисты таких систем склонны рассматривать такое поведение, как естественное. Рассмотрим следующую программу:

```
a = 1; a = 2; print(a);
```

Система, в которой эта программа напечатает 1 или любое другое значение, кроме 2, быстро приведет к появлению толпы возмущенных программистов и массе полезных мыслей.

Для системы, в которой данные разбросаны по нескольким машинам, а доступ к ним имеет несколько процессов, все сильно усложняется. Допустим, что x — это элемент данных, хранящийся на машине B . Представим, что процесс, работающий на машине A , читает x в момент времени $T1$, то есть посылает B сообщение с требованием возвратить x . Чуть позже, в момент времени $T2$, процесс с машины B производит запись x . Если строгая непротиворечивость сохраняется, чтение должно всегда возвращать прежнее значение, которое не зависит от того, где находятся машины и насколько мал интервал между $T1$ и $T2$. Однако если $T2 - T1$ равно, скажем, одной наносекунде, а машины расположены в трех метрах друг от друга, то чтобы запрос на чтение от A к B дошел до машины раньше B отправления запроса на запись, он должен двигаться в 10 раз быстрее скорости света, а это противоречит теории относительности Эйнштейна. Разумно ли программистам требовать, чтобы система была строго непротиворечивой, даже если это требование противоречит законам физики?

Проблема со строгой непротиворечивостью состоит в том, что она завязана на абсолютное глобальное время. В сущности, в распределенной системе невозможно установить на каждую операцию уникальную отметку времени, согласованную с действительным глобальным временем. Мы можем снизить остроту этой ситуации, разделив время на серии последовательных неперекрывающихся интервалов. Каждая операция может происходить в пределах интервала и получает отметку времени, соответствующую этому интервалу. В зависимости от того,

насколько точно можно синхронизировать часы, мы можем оказаться и в ситуации, когда на один такой интервал будет приходиться больше одной операции.

К сожалению, на то, что на один интервал будет приходиться максимум одна операция, нет никаких гарантий. Соответственно, мы вынуждены учитывать возможность наличия в одном интервале нескольких операций. Аналогично ситуации параллельного выполнения распределенных транзакций, две операции в одном и том же интервале приводят к конфликту, если производятся над одними и теми же данными, и одна из них — операция чтения. Важный момент для определения модели непротиворечивости — точно выяснить, какой тип поведения применим при наличии конфликтующих операций.

Для детального изучения непротиворечивости мы можем привести несколько примеров. Чтобы сделать эти примеры точнее, мы нуждаемся в специальной нотации, при помощи которой отобразим операции процессов на ось времени. Ось времени всегда рисуется горизонтально, время увеличивается слева направо. Символы $W_i(x)a$ и $R_i(x)b$ означают соответственно, что выполнены запись процессом P_i в элемент данных x значения a и чтение из этого элемента процессом P_i , возвратившее b . Мы полагаем, что каждый элемент данных первоначально был инициализирован нулем. Если с процессом доступа к данным не было никаких проблем, мы можем опустить индексы у символов W и R .

В качестве примера на рис. 6.5, *а* процесс $P1$ осуществляет запись в элемент данных x , изменяя его значение на a . Отметим, что в принципе эта операция, $W1(x)a$, сначала производит копирование значения в локальное хранилище данных $P1$, а затем распространяет его и на другие локальные копии. В нашем примере $P2$ позже читает значение x из его локальной копии в хранилище и обнаруживает там значение a . Такое поведение характерно для хранилища данных, сохраняющего строгую непротиворечивость. В противоположность ему, на рис. 6.5, *б* процесс $P2$ производит чтение после записи (возможно, лишь на наносекунду позже, но позже) и получает ноль (NIL). Последующее чтение возвращает a . Такое поведение для хранилища данных, сохраняющего строгую непротиворечивость, некорректно.

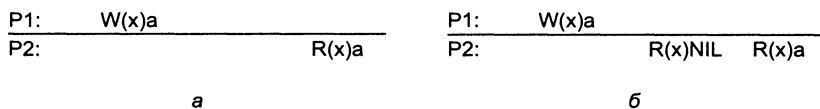


Рис. 6.5. Поведение двух процессов, работающих с одним и тем же элементом данных. Горизонтальная ось — время. Хранилище со строгой непротиворечивостью (*а*). Хранилище без строгой непротиворечивости (*б*)

Итак, когда хранилище данных строго непротиворечиво, все операции записи мгновенно замечаются всеми процессами. Выдерживается абсолютный глобальный порядок по времени. Если элемент данных изменяется, все последующие операции чтения, совершаемые с данными, возвращают новое значение, при этом неважно, как скоро после изменения производится чтение, и не имеет значения, какой процесс производит чтение и где он расположен. Соответственно, если производится чтение, оно возвращает это текущее значение, и неважно, как быстро после этого производится следующая запись.

В следующем пункте мы ослабим требования этой модели, заменив абсолютное время временными интервалами, и точно определим приемлемое поведение для конфликтующих операций.

6.2.2. Линеаризуемость и последовательная непротиворечивость

Хотя строгая непротиворечивость и представляет собой идеальную модель непротиворечивости, реализовать ее в распределенных системах невозможно. Кроме того, опыт показал, что программисты часто могут значительно лучше управлять не столь строгими моделями. Так, например, все книги по операционным системам содержат обсуждение проблем критических областей и взаимных исключений. При этом, как правило, имеется предупреждение о том, что правильно написанные параллельные программы (такие, как программа «производитель—потребитель») не должны делать никаких предположений об относительных скоростях процессов или об очередности выполнения инструкций. Обработка двух событий в одном процессе происходит так быстро, что другой процесс не в состоянии сделать что-нибудь, если между ними обнаружит проблему. Вместо этого читателя учат программировать так, чтобы точный порядок выполнения инструкций (на самом деле, ссылок на память) не имел никакого значения. Если важна очередность событий, следует использовать семафоры или другие операции синхронизации. Принятие этого аргумента означает согласие на модель слабой непротиворечивости.

Последовательная непротиворечивость (sequential consistency) — это менее строгая модель непротиворечивости. Впервые она была определена в [250] в контексте совместно используемой памяти мультипроцессорных систем. В общем, хранилище данных считается последовательно непротиворечивым, если оно удовлетворяет следующему условию: *результат любого действия такой же, как если бы операции (чтения и записи) всех процессов в хранилище данных выполнялись бы в некотором последовательном порядке, причем операции каждого отдельного процесса выполнялись бы в порядке, определяемом его программой.*

Это определение означает, что когда процессы выполняются параллельно на различных (возможно) машинах, любое правильное чередование операций чтения и записи является допустимым, но *все процессы видят одно и то же чередование операций.* Отметим, что о времени никто не вспоминает, то есть никто не ссылается на «самую последнюю» операцию записи объекта. Отметим, что в данном контексте процесс «видит» записи всех процессов, но только свои собственные чтения.

Итак, время роли не играет, что мы и видим на рис. 6.6. Рассмотрим работу четырех процессов с одним элементом данных x . На рис. 6.6, *a* сначала процесс P_1 осуществляет запись $W(x)a$ в элемент x . Позднее (по абсолютному времени) процесс P_2 также осуществляет запись, устанавливая значение x в b . Однако оба процесса, P_3 и P_4 , *сначала* читают значение b и лишь затем — значение a . Другими словами, операция записи процесса P_2 представляется им происходящей раньше записи процесса P_1 .

P1: $W(x)a$				P1: $W(x)a$			
P2: $W(x)b$				P2: $W(x)b$			
P3: $R(x)b$		$R(x)a$		P3: $R(x)b$		$R(x)a$	
P4: $R(x)b$		$R(x)a$		P4: $R(x)a$		$R(x)b$	
a				б			

Рис. 6.6. Хранилище данных с последовательной непротиворечивостью (а).
Хранилище данных без последовательной непротиворечивости (б)

В противоположность происходящему на рис. 6.6, а, на рис. 6.6, б последовательная непротиворечивость нарушается, поскольку не все процессы видят одинаковое чередование операций записи. В частности, для процесса $P3$ дело выглядит так, что элемент данных сначала принимает значение b , а затем — a . С другой стороны, $P4$ полагает, что итоговое значение элемента данных — b .

Модель непротиворечивости, более слабая, чем строгая непротиворечивость, но более сильная, чем последовательная — это *линеаризуемость* (*linearizability*). Согласно этой модели операции получают отметку времени, используя глобальные часы, имеющие конечную точность. Такие часы могут быть реализованы в распределенной системе из предположения, что у процессов имеются слабо синхронизированные часы, которые обсуждались в предыдущей главе. Пусть $ts_{OP}(x)$ — это отметка времени, назначенная операции OP , которая работает с элементом данных x , причем OP — это операция чтения (R) или записи (W). Хранилище данных называется линеаризуемым, если каждая операция имеет отметку времени и соблюдается следующее условие [199]: *результат всякого действия такой же, как если бы все операции (чтения и записи) всех процессов с хранилищем данных выполнялись бы в некотором последовательном порядке, причем операции каждого отдельного процесса выполнялись бы в этой последовательности в порядке, определенном в их программах, и, кроме того, если $ts_{OP1}(x) < ts_{OP2}(x)$, то операция $OP1(x)$ в этой последовательности должна предшествовать операции $OP2(y)$.*

Отметим, что линеаризуемое хранилище данных обладает еще и последовательной непротиворечивостью. Разница между ними состоит в том, что при учете упорядоченности во внимание принимаются также установки синхронизированных часов. На практике линеаризуемость используется в первую очередь для содействия в формальном подтверждении параллельных алгоритмов [199]. Дополнительные ограничения, такие как сохранение упорядоченности в соответствии с отметками времени, делает затраты на реализацию линеаризации выше, чем на реализацию последовательной непротиворечивости, как показано в [20].

Последовательная непротиворечивость напоминает сериализуемость в транзакциях, о которой мы говорили в предыдущей главе. Напомним, что набор параллельно выполняющихся транзакций считается сериализуемым, если итоговый результат может быть получен путем выполнения этих транзакций последовательно с определенной очередностью. Основная разница заключается в степени детализации: последовательная непротиворечивость определяется в понятиях операций записи и чтения, а сериализуемость — в понятиях транзакций, которые объединяют эти операции в одно целое.

Чтобы получить более конкретное представление о последовательной непротиворечивости, рассмотрим три параллельно выполняющихся процесса, *P1*, *P2* и *P3*, приведенные в табл. 6.1 [130]. Элементы данных в этом примере представлены тремя целыми переменными — *x*, *y* и *z*, которые хранятся в последовательно непротиворечивом хранилище данных (возможно, распределенном), предназначенном для совместного доступа. Предположим, что каждая из переменных инициализирована нулем. В этом примере присвоение соответствует операции записи, а инструкция `print` — одновременным операциям чтения обоих аргументов. Все инструкции считаются неделимыми.

Таблица 6.1. Три параллельно выполняющихся процесса

Процесс P1	Процесс P2	Процесс P3
<code>x=1;</code>	<code>y=i;</code>	<code>z=1;</code>
<code>print(y.z);</code>	<code>print(x.z);</code>	<code>print(x.y);</code>

При выполнении возможны различные варианты чередования. Имея шесть независимых выражений, мы получаем 720 (6!) потенциально возможных последовательностей выполнения, хотя некоторые из них нарушают порядок, заданный в программах. Рассмотрим 120 (5!) последовательностей, которые начинаются с *x*=1. В половине из них инструкция `print(x,z)` выполняется раньше инструкции *y*=1, а потому нарушают порядок выполнения инструкций программы. Еще у половины инструкция `print(x,y)` выполняется раньше инструкции *z*=1 и также нарушают порядок выполнения инструкций программы. Только 1/4 из 120 последовательностей являются допустимыми (всего 30). Еще 30 допустимых последовательностей могут начинаться с инструкции *y*=1 и еще 30 — с инструкции *z*=1, давая в сумме 90 допустимых последовательностей выполнения инструкций. Четыре из них приведены в табл. 6.2.

Таблица 6.2. Четыре варианта допустимых последовательностей выполнения процессов

	Вариант			
	1	2	3	4
Инструкции	<code>x=1;</code>	<code>x=1;</code>	<code>y=1;</code>	<code>y=1;</code>
	<code>print(y.z);</code>	<code>y=1;</code>	<code>z=1;</code>	<code>x=1;</code>
	<code>y=1;</code>	<code>print(x.z);</code>	<code>print(x.y);</code>	<code>z=1;</code>
	<code>print(x.z);</code>	<code>print(y.z);</code>	<code>print(x.z);</code>	<code>print(x.z);</code>
	<code>z=1;</code>	<code>z=1;</code>	<code>x=1;</code>	<code>print(y.z);</code>
	<code>print(x.y);</code>	<code>print(x.y);</code>	<code>print(y.z);</code>	<code>print(x.y);</code>
Печать	001011	101011	010111	111111
Сигнатура	001011	101011	110101	111111

В варианте 1 три процесса запускаются в следующем порядке: сначала *P1*, после него *P2*, затем *P3*. В трех других вариантах демонстрируются различные, но также

допустимые чередования выражений во времени. В каждом из этих трех процессов печатаются две переменные. Поскольку единственные значения, которые может принимать каждая из переменных, — это исходное значение (0) или присвоенное значение (1), каждый процесс печатает строку из двух битов. Числа в графе «Печатать» — тот результат, который появится на устройстве вывода.

Если мы объединим результаты работы процессов $P1$, $P2$ и $P3$ в таком порядке, мы получим строку из шести битов, характеризующую частичное чередование инструкций. Эта строка в таблице представлена в графе «Сигнатура». Ниже мы характеризуем каждый из вариантов по его сигнатуре, а не по результатам печати.

Не все из 64 сигнатур допустимы. Тривиальный пример: сигнатура 000000 не разрешена, поскольку означает, что инструкции печати выполнены раньше, чем инструкции присваивания, что нарушает требования о выполнении инструкций в порядке, предусмотренном программой. Более сложный пример — 001001. Первые два бита, 00, означают, что переменные y и z в момент их печати были равны нулю. Такая ситуация может возникнуть только в том случае, если обе инструкции из процесса $P1$ выполняются до начала выполнения процессов $P2$ и $P3$. Следующие два бита, 10, означают, что $P2$ выполняется после начала выполнения $P1$, но до начала выполнения $P3$. Последние два бита, 01, означают, что выполнение $P3$ должно закончиться до начала выполнения $P1$, но мы уже видели, что процесс $P1$ должен выполняться первым. Таким образом, сигнатура 001001 недопустима.

Говоря коротко, 90 допустимых последовательностей выполнения инструкций порождают многообразие различных результатов работы программы (их все-таки менее 64), допустимых по условиям последовательной непротиворечивости. Соглашение между процессами и распределенным хранилищем данных, предназначенным для совместного доступа, состоит в том, что процессы воспринимают все эти результаты как правильные. Другими словами, процессы должны воспринимать четыре результата из табл. 6.2 и все остальные допустимые результаты в качестве правильных ответов и корректно работать в случае получения одного из них. Программа, которая работает с какими-то из этих результатов, а другие воспринимает как ошибочные, нарушает соглашение с хранилищем данных и является неправильной.

Существуют различные способы формального выражения последовательной непротиворечивости (и других моделей). Общий подход изложен в [5, 298]. Каждый процесс P_i имеет ассоциированную с ним *реализацию* (*execution*) E_i , которая представляет собой последовательность операций чтения и записи процессом P_i значений из хранилища данных S . Эта последовательность соответствует порядку, заданному в программе, ассоциированной с процессом P_i .

Так, например, реализация четырех процессов, представленных на рис. 6.6, a , может быть задана следующим образом:

$E1: W1(x)a$

$E2: W2(x)b$

$E3: R3(x)b, R3(x)a$

$E4: R4(x)b, R4(x)a$

Чтобы получить относительный порядок, в котором должны исполняться операции, мы должны объединить строки операций в реализации Ei в единую строку H так, чтобы каждая из операций, входящих в Ei , входила и в H . Строка H называется *историей* (*history*). Интуитивно понятно, что H описывает очередность операций с точки зрения единого централизованного хранилища. Все допустимые значения H обязаны следовать двум ограничениям:

- ♦ должен сохраняться такой же порядок выполнения, как и в программе;
- ♦ должно сохраняться соотношение между данными.

Первое ограничение означает, что если операция записи или чтения $OP1$ в одной из строк реализации Ei следует перед другой операцией $OP2$, то $OP1$ должна следовать перед $OP2$ и в истории H . Если это ограничение для всех пар операций соблюдается, то получившаяся история H не будет содержать операций, нарушающих порядок, заданный программой.

Второе ограничение, которое мы будем называть *связностью данных* (*data coherence*), означает, что чтение $R(x)$ некоторого элемента данных x всегда должно возвращать самое недавнее из записанных значений x , то есть значение v , записанное последней перед $R(x)$ операцией $W(x)v$. Связность данных проверяется путем выделения каждого элемента данных и последовательности операций с ним. Прочие данные при этом остаются без внимания. В противоположность этому, при проверке непротиворечивости внимание обращается на операции записи различных элементов данных и их порядок. Если мы рассматриваем совместно используемую память и локализацию в памяти, не обращая внимания на элементы данных, такой случай связности данных называется *связностью памяти* (*memory coherence*).

Возвращаясь к четырем процессам, представленным на рис. 6.6, *а*, допустимым значением истории H для них может быть следующая строка:

$$H = W1(x)b, R3(x)b, R4(x)b, W2(x)a, R3(x)a, R4(x)a.$$

С другой стороны, для исполнения четырех процессов, представленных на рис. 6.6, *б*, невозможно найти разрешенной истории, поскольку в последовательно непротиворечивой системе невозможно сделать так, чтобы процесс $P3$ сначала выполнял операцию $R3(x)b$, а затем — операцию $R3(x)a$, в то время как процесс $P4$ считывает значения a и b в обратном порядке.

Более сложным примером могут быть несколько допустимых значений H . Поведение программы будет считаться правильным, если ее последовательность операций соответствует одному из разрешенных значений H .

Хотя последовательная непротиворечивость и дает удобную для программистов модель, она имеет серьезные проблемы с производительностью. В [268] было доказано, что при времени чтения r , времени записи w и минимальном времени передачи пакета между узлами t всегда выполняется условие $r + w \geq t$. Другими словами, для всякого последовательно непротиворечивого хранилища изменение протокола для увеличения скорости чтения вызывает падение скорости записи, и наоборот. По этой причине исследователи принялись за поиск новых, еще более слабых моделей. В следующем пункте мы рассмотрим некоторые из них.

6.2.3. Причинная непротиворечивость

Модель *причинной непротиворечивости* (*causal consistency*) [208] представляет собой ослабленный вариант последовательной непротиворечивости, при которой проводится разделение между событиями, потенциально обладающими причинно-следственной связью, и событиями, ею не обладающими. Мы уже говорили о причинно-следственной связи в предыдущей главе, когда обсуждали векторные отметки времени. Если событие B вызвано предшествующим событием A или находится под его влиянием, то причинно-следственная связь требует, чтобы все окружающие наблюдали сначала событие A , а затем B .

Рассмотрим пример с памятью. Предположим, что процесс $P1$ записывает значение переменной x . Таким образом, чтение x и запись y будут потенциально связаны с этим процессом причинно-следственной связью, поскольку вычисление y может зависеть от значения x , которое прочтет $P2$ (то есть от значения, записанного процессом $P1$). С другой стороны, если два процесса спонтанно и одновременно записывают значения двух различных переменных, они не могут иметь причинно-следственную связь. Когда за записью следует чтение, эти два события потенциально имеют причинно-следственную связь. И вообще, чтение связано с записью, которая предоставляет данные для этого чтения, причинно-следственной связью. Операции, не имеющие причинно-следственной связи, называются *параллельными* (*concurrent*).

Для того чтобы хранилище данных поддерживало причинную непротиворечивость, оно должно удовлетворять следующему условию: *операции записи, которые потенциально связаны причинно-следственной связью, должны наблюдаться всеми процессами в одинаковом порядке, а параллельные операции записи могут наблюдаться на разных машинах в разном порядке.*

В качестве примера рассмотрим рис. 6.7. На нем представлена последовательность событий, возможная для хранилища с причинной непротиворечивостью, но запрещенная для хранилища со строгой или последовательной непротиворечивостью. Стоит отметить, что $W2(x)b$ и $W1(x)c$ — параллельные операции, и поэтому их очередность для различных процессов не важна.

P1:	W(x)a	W(x)c		
P2:	R(x)a	W(x)b		
P3:	R(x)a		R(x)c	R(x)b
P4:	R(x)a		R(x)b	R(x)c

Рис. 6.7. Эта последовательность допустима для хранилища с причинной непротиворечивостью, но недопустима для хранилища со строгой или последовательной непротиворечивостью

Рассмотрим теперь второй пример. На рис. 6.8, *а* мы видим, что $W2(x)b$ потенциально зависит от $W1(x)a$, поскольку b может быть результатом вычислений, в которые входит значение, прочитанное операцией $R2(x)a$. Две операции записи связаны причинно-следственной связью, а значит, все процессы должны наблюдать их в одинаковом порядке. Таким образом, рис. 6.8, *а* некорректен. С другой стороны, на рис. 6.8, *б* операция чтения из процесса $P2$ убрана, и теперь

$W1(x)a$ и $W2(x)b$ стали параллельными операциями записи. Хранилище с причинной непротиворечивостью не требует, чтобы параллельные операции записи обладали глобальной упорядоченностью, и потому рис. 6.8, б корректен.

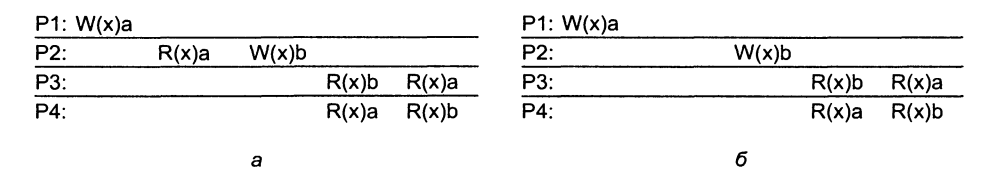


Рис. 6.8. Нарушение причинной непротиворечивости хранилища (а). Правильная последовательность событий в хранилище с причинной непротиворечивостью (а)

Реализация причинно-следственной непротиворечивости требует отслеживать, какие процессы какие записи видели. Это можно успешно проделать, создав и поддерживая граф зависимостей, на котором будет указана взаимная зависимость операций друг от друга. Один из способов сделать это — воспользоваться векторными отметками времени, которые мы обсуждали в предыдущей главе. Позже мы еще вернемся к использованию векторных отметок времени для определения причинно-следственной связи.

6.2.4. Непротиворечивость FIFO

В случае причинно-следственной непротиворечивости допустимо, чтобы на различных машинах параллельные операции записи наблюдались в разном порядке, но операции записи, связанные причинно-следственной связью, должны иметь одинаковый порядок выполнения, с какой бы машины ни велось наблюдение. Следующим шагом в ослаблении непротиворечивости будет освобождение от последнего требования. Это приведет нас к *непротиворечивости FIFO (FIFO consistency)*, которая подчиняется следующему условию: *операции записи, осуществляемые единственным процессом, наблюдаются всеми остальными процессами в том порядке, в котором они осуществляются, но операции записи, происходящие в различных процессах, могут наблюдаться разными процессами в разном порядке.*

Непротиворечивость FIFO в случае распределенных систем с совместно используемой памятью называют также *непротиворечивостью PRAM (PRAM consistency)*. Она описана в [268]. Сокращение PRAM происходит от Pipelined RAM (конвейерная оперативная память), потому что запись, производимая одним процессом, может быть конвейеризована, в том смысле, что процесс не должен терять скорость, ожидая окончания одной операции записи, чтобы начать другую. Сравнение непротиворечивости FIFO с причинной непротиворечивостью иллюстрирует рис. 6.9. Приведенная последовательность событий допустима для хранилища данных с непротиворечивостью FIFO, но запрещена для любой более строгой модели из тех, которые мы рассматривали.

Непротиворечивость FIFO интересна нам из-за простоты ее реализации. В действительности она не дает никаких гарантий того, в каком порядке операции записи будут наблюдаться в различных процессах, исключая порядок прохожде-

ния двух или более операций записи, принадлежащих одному процессу. Если изложить это другими словами, в этой модели все операции записи во всех процессах являются параллельными. Эта модель может быть реализована просто путем именования каждой операции парой (*процесс, очередной номер*) и осуществлением операций записи каждого из процессов в порядке их номеров.

P1:	W(x)a					
P2:		R(x)a	W(x)b	W(x)c		
P3:					R(x)b	R(x)a
P4:					R(x)a	R(x)b

Рис. 6.9. Допустимая последовательность событий для непротиворечивости FIFO

Рассмотрим вновь три процесса из табл. 6.1, опираясь на этот раз на непротиворечивость FIFO вместо последовательной непротиворечивости. При условии непротиворечивости FIFO различные процессы могут наблюдать выполнение инструкций в различном порядке (табл. 6.3). Так, например, в варианте 1 показано, как выглядит последовательность событий с точки зрения процесса *P1*, в варианте 2 — с точки зрения процесса *P2*, а в варианте 3 — с точки зрения процесса *P3* (наблюдаемые результаты создают инструкции, выделенные жирным шрифтом). В условиях последовательной непротиворечивости три различных представления были бы недопустимы.

Таблица 6.3. Очередность выполнения инструкций с точки зрения трех процессов, представленных в табл. 6.1

	Вариант		
	1	2	3
Инструкции	x=1; print(y,z); y=1; print(x,z); z=1; print(x,y);	x=1; y=1; print(x,z); print(y,z); z=1; print(x,y);	y=1; print(x,z); z=1; print(x,y); x=1; print(y,z);
Печать	00	10	01

Если мы объединим результаты трех процессов, то получим итоговый результат 001001, что, как мы говорили раньше, в условиях последовательной непротиворечивости невозможно. Основное различие между последовательной непротиворечивостью и непротиворечивостью FIFO состоит в том, что в первом случае, хотя порядок выполнения инструкций не регламентирован, все процессы, по крайней мере, с ним согласны. Во втором же случае им нет нужды ни с чем соглашаться. Разные процессы могут наблюдать операции в разном порядке.

Иногда непротиворечивость FIFO может приводить к результатам, противоречащим здравому смыслу. В примере, взятом из [174], предполагается, что целые переменные *x* и *y* инициализируются нулями. Кажется, что в ситуации,

представленной в табл. 6.4, возможны три результата: будет прерван процесс $P1$, будет прерван процесс $P2$, ни один из процессов не будет прерван (если первыми произойдут обе операции присваивания). Однако в случае непротиворечивости FIFO могут быть прерваны оба процесса. Такой результат получится в случае, если $P1$ осуществит чтение $R1(y)0$ до обнаружения выполненной процессом $P2$ записи $W2(y)1$, а $P2$ осуществит чтение $R2(x)0$ до обнаружения выполненной процессом $P1$ записи $W1(x)1$. При последовательной непротиворечивости существует шесть допустимых вариантов чередования выражений, и ни один из них не ведет к прерыванию обоих процессов.

Таблица 6.4. Два параллельных процесса

Процесс P1	Процесс P2
<code>x=1;</code>	<code>y=1;</code>
<code>if (y == 0) kill(P2);</code>	<code>if(x==0)kill(P1);</code>

6.2.5. Слабая непротиворечивость

Хотя непротиворечивость FIFO и дает большую производительность, чем более строгие модели непротиворечивости, для многих приложений она остается излишне строгой, поскольку требует, чтобы операции записи одного процесса отовсюду наблюдались в одном и том же порядке. Не все приложения нуждаются даже в том, чтобы наблюдать все операции записи, не говоря уже об их порядке. Рассмотрим случай, когда процесс внутри критической области заносит записи в реплицируемую базу данных. Хотя другие процессы даже не предполагают работать с новыми записями до выхода этого процесса из критической области, система управления базой данных не имеет никаких средств, чтобы узнать, находится процесс в критической области или нет, и может распространить изменения на все копии базы данных.

Наилучшим решением в этом случае было бы позволить процессу покинуть критическую область и затем убедиться, что окончательные результаты разосланы туда, куда нужно, и не обращать внимания на то, разосланы всем копиям промежуточные результаты или нет. Это можно сделать, введя так называемую *переменную синхронизации* (*synchronization variable*). Переменная синхронизации S имеет только одну ассоциированную с ней операцию, $synchronize(S)$, которая синхронизирует все локальные копии хранилища данных. Напомним, что процесс P осуществляет операции только с локальной копией хранилища. В ходе синхронизации хранилища данных все локальные операции записи процесса P распространяются на остальные копии, а операции записи других процессов — на копию данных P .

Используя переменные синхронизации для частичного определения непротиворечивости, мы приходим к так называемой слабой непротиворечивости (*weak consistency*) [130]. Модель слабой непротиворечивости имеет три свойства.

- ◆ Доступ к переменным синхронизации, ассоциированным с хранилищем данных, производится на условиях последовательной непротиворечивости.

- ◆ С переменной синхронизации не может быть произведена ни одна операция до полного и повсеместного завершения предшествующих ей операций записи.
- ◆ С элементами данных не может быть произведена ни одна операция до полного завершения всех операций с переменными синхронизации.

Первый пункт гласит, что все процессы могут наблюдать за всеми операциями над переменными синхронизации, которые с точки зрения этих процессов происходят в одинаковом порядке. Другими словами, если процесс *P1* вызывает операцию `synchronize(S1)` в то же самое время, когда процесс *P2* вызывает операцию `synchronize(S2)`, результат будет точно таким же, как если бы операция `synchronize(S1)` была вызвана раньше операции `synchronize(S2)`, или наоборот.

Второй пункт указывает на то, что синхронизация очищает конвейер. Она ускоряет выполнение всех операций чтения, которые в этот момент происходят, частично завершены или завершены в некоторых локальных копиях, и приводит их к повсеместному завершению. Когда синхронизация заканчивается, гарантировано заканчиваются все предшествовавшие ей операции записи. Производя синхронизацию после изменения совместно используемых данных, процесс переносит новые значения во все локальные копии хранилища.

Третий пункт поясняет, что при доступе к элементам данных (как для записи, так и для чтения) все предшествующие синхронизации должны быть завершены. Производя перед чтением совместно используемых данных синхронизацию, процесс может быть уверен, что получит самые «свежие» значения.

В отличие от предыдущих моделей непротиворечивости, слабая непротиворечивость реализует непротиворечивость групп операций, а не отдельных операций чтения и записи. Эта модель наиболее широко используется, если отдельные процедуры доступа к общим данным редки, а большая часть операций доступа собрана в группы (много операций за короткий срок, а потом ничего в течение долгого времени).

Другое важное отличие от предыдущих моделей непротиворечивости заключается в том, что теперь мы ограничены только *временем* поддержания непротиворечивости, в то время как ранее мы были ограничены *формой* непротиворечивости. На самом деле мы можем сказать, что в случае слабой непротиворечивости соблюдается последовательная непротиворечивость между группами операций, а не между отдельными операциями. Для выделения этих групп используются переменные синхронизации.

В идее о допустимости существования в памяти неверных значений нет ничего нового. Множество компиляторов «плутуют» таким же образом. Для примера рассмотрим фрагмент программы из листинга 6.1, где все переменные инициализируются соответствующими значениями. Оптимизирующий компилятор может производить вычисление переменных *a* и *b* в регистрах, временно сохраняя там результат и не обновляя эти переменные в памяти. Только вызов функции *f* заставляет компилятор поместить текущие значения *a* и *b* обратно в память, поскольку *f* должна ими пользоваться. Это типичный пример оптимизация компилятора.

Листинг 6.1. Хранение в регистрах некоторых переменных

```
int a, b, c, d, e, x, y;           // переменные
int *p, *q;                       // указатели
int f(int *p, int *q);             // прототип функции

a = x * x;                         // a хранится в регистре
b = y * y;                         // b тоже
c = a * a * a + b * b + a * b;    // будет использовано позднее
d = a * a * c;                   // будет использовано позднее
p = &a;                           // p получает адрес a
q = &b;                           // q получает адрес b
e = f(p, q);                      // вызов функции
```

В данном случае неправильные значения в памяти допустимы, потому что компилятор знает, что делает (а программу не волнует, что значения в памяти не актуальны). Ясно, что если будет создан второй процесс, который может читать из памяти без ограничений, эта схема перестанет работать. Так, например, если в ходе присвоения значения *d* второй процесс считает *a*, *b* и *c*, он получит противоречивые данные (старые значения *a* и *b* и новое значение *c*). Для предупреждения хаоса можно представить себе специальную защиту, при которой компилятор сначала должен считывать специальный бит (флаг), сигнализирующий, что память не актуальна. Если доступ к *a* хочет получить другой процесс, ему придется ожидать установки флага. Таким образом, мы получим почти абсолютную непротиворечивость, обеспечиваемую программной синхронизацией и тем, что все стороны выполняют правила.

Обсудим теперь относительно менее отвлеченную ситуацию. На рис. 6.10, *a* мы видим, что процесс *P1* осуществляет две записи значений элементов данных, после чего синхронизируется (показано буквой *S*). Если *P2* и *P3* к этому моменту еще не были синхронизированы, мы не можем дать никаких гарантий по поводу того, что они увидят. Таким образом, эта последовательность событий допустима.

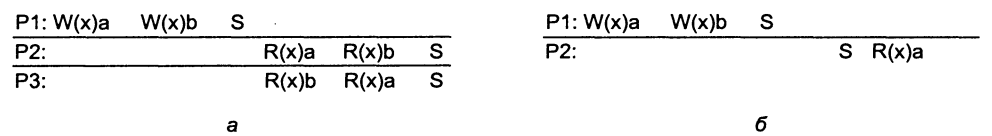


Рис. 6.10. Допустимая при слабой непротиворечивости последовательность событий (а). Недопустимая при слабой непротиворечивости последовательность событий (б)

Ситуация на рис. 6.10, *б* иная. Здесь процесс *P2* синхронизирован. Это означает актуальность его локальной копии хранилища данных. Когда он будет считывать значение *x*, он получит значение *b*. Получение *a*, как видно из рисунка, при слабой непротиворечивости невозможно.

6.2.6. Свободная непротиворечивость

Слабая непротиворечивость имеет проблему следующего рода: когда осуществляется доступ к переменной синхронизации, хранилище данных не знает, то ли это происходит потому, что процесс закончил запись совместно используемых

данных, то ли наоборот начал чтение данных. Соответственно, оно может предпринять действия, необходимые в обоих случаях, например, убедиться, что завершены (то есть распространены на все копии) все локально инициированные операции записи и что учтены все операции записи с других копий. Если хранилище должно распознавать разницу между входом в критическую область и выходом из нее, может потребоваться более эффективная реализация. Для предоставления этой информации необходимо два типа переменных или два типа операций синхронизации, а не один.

Свободная непротиворечивость (release consistency) предоставляет эти два типа [165]. Операция *захвата (acquire)* используется для сообщения хранилищу данных о входе в критическую область, а операция *освобождения (release)* говорит о том, что критическая область была покинута. Эти операции могут быть реализованы одним из двух способов: во-первых, обычными операциями над специальными переменными; во-вторых, специальными операциями. В любом случае программист отвечает за вставку в программу соответствующего дополнительного кода, реализующего, например, вызов библиотечных процедур *acquire* и *release* или процедур *enter_critical_region* и *leave_critical_region*.

В случае свободной непротиворечивости, кроме того, независимо от критических областей можно использовать барьеры. *Барьер (barrier)* — это механизм синхронизации, который предвеляет любой процесс в начале фазы программы под номером $n+1$ до того, как все процессы окончат фазу n . Когда процесс подойдет к барьеру, он должен дожидаться, пока к нему не «подтянутся» и все остальные процессы. Когда последний из процессов подойдет к барьеру, все совместно используемые данные синхронизируются, и процессы продолжают свою работу. Отправление от барьера выполняется по захвату, а приход к барьеру — по освобождению.

Вдобавок к этим операциям синхронизации также возможны чтение и запись совместно используемых данных. Захват и освобождение не могут применяться ко всем данным хранилища. Они могут охранять только отдельные совместно используемые данные, в этом случае только эти данные остаются непротиворечивыми. Совместно используемые данные, сохраняющие свою непротиворечивость, называются *защищенными (protected)*.

Хранилище данных со свободной непротиворечивостью гарантирует, что при захвате процесса хранилище сделает так, что все локальные копии защищенных данных при необходимости будут актуализированы и станут непротиворечивыми относительно своих удаленных копий. Когда произойдет освобождение, измененные защищенные данные будут распространены на другие локальные копии хранилища. Захват не гарантирует, что локальные изменения будут немедленно разосланы другим локальным копиям. Соответственно, освобождение не обязательно приведет к импорту изменений из других копий.

На рис. 6.11 показана допустимая для свободной непротиворечивости последовательность событий. Процесс *P1* производит захват, дважды изменяет элемент данных, а затем производит освобождение. Процесс *P2* производит захват и считывает элемент данных. Он гарантировано получает значение, которое элемент данных имел в момент освобождения, а именно *b* (кроме случая, когда за-

хват $P2$ происходит раньше, чем захват $P1$). Если захват произошел до того, как процесс $P1$ произвел освобождение, захват будет ожидать совершения освобождения. Поскольку процесс $P3$ не сможет осуществить захват до чтения совместно используемых данных, хранилище данных не будет обязано выдать ему текущее значение x , и этому процессу будет возвращено значение a .

P1:	Acq(L)	W(x)a	W(x)b	Rel(L)
P2:			Acq(L)	R(x)b
P3:				Rel(L)

Рис. 6.11. Допустимая последовательность событий при свободной непротиворечивости

Чтобы прояснить свободную непротиворечивость, давайте кратко определим ее возможную реализацию (несложную) в контексте реплицируемой базы данных. Производя захват, процесс посылает сообщение центральному менеджеру синхронизации, запрашивая захват отдельной блокировки. В отсутствии других желающих запрос удовлетворяется и происходит захват. Затем следует произвольная последовательность локальных операций чтения и записи. Ни одна из этих операций не распространяется на другие копии базы. В процессе освобождения модифицированные данные рассылаются другим копиям, которые используют эти данные. После того как каждая копия подтвердит получение этих данных, центральный менеджер синхронизаций уведомляется о произошедшем освобождении. Таким образом, произвольное число операций чтения и записи совместно используемых данных сопровождается фиксированными дополнительными затратами. Захваты и освобождения при разных блокировках происходят независимо друг от друга.

Хотя описанный централизованный алгоритм и решает проблему, это отнюдь не единственный подход. Вообще говоря, распределенное хранилище данных является свободно непротиворечивым при условии выполнения им трех правил.

- ◆ Перед выполнением операций чтения или записи совместно используемых данных все предыдущие захваты этого процесса должны быть полностью закончены.
- ◆ Перед выполнением освобождения все предыдущие операции чтения и записи этого процесса должны быть полностью закончены.
- ◆ Доступ к синхронизируемым переменным должен обладать непротиворечивостью FIFO (последовательная непротиворечивость не требуется).

Если все эти условия выполнены и процессы правильно (то есть попарно) используют захваты и освобождения, результат любого выполнения не будет отличаться от порядка, характерного для последовательно непротиворечивых хранилищ. В результате блокировка операций над совместно используемыми данными будет атомарной благодаря примитивам захвата и освобождения, которые будут препятствовать чередованию.

У свободной непротиворечивости имеется такая реализация, как *ленивая свободная непротиворечивость (lazy release consistency)* [231]. При обычной свободной непротиворечивости, которую мы далее будем называть *энергичной свободной*

непротиворечивостью (*eager release consistency*), чтобы не путать ее с «ленивым» собратом, при освобождении процесс, выполняющий освобождение, рассылает все модифицированные данные всем процессам, которые уже имеют копии этих данных и поэтому потенциально могут быть заинтересованными в их обновленной версии. Не существует способа указать, нужны они на самом деле или нет, и для надежности обновленные данные получают все эти процессы.

Хотя разослать повсюду все данные несложно, обычно это неэффективно. При ленивой свободной непротиворечивости в момент освобождения ничего никому не рассылается. Взамен этого в момент захвата процесс, пытающийся произвести захват, должен получить наиболее свежие данные из процесса или процессов, в которых они хранятся. Для определения того, что эти элементы данных действительно были переданы, используется протокол отметок времени.

Во многих программах критическая область располагается внутри цикла. В случае энергичной свободной непротиворечивости освобождение происходит при каждом проходе цикла, при этом все модифицированные данные рассылаются всем процессам, поддерживающим их копии. Этот алгоритм поглощает пропускную способность каналов и вызывает неизбежные задержки. В варианте ленивой свободной непротиворечивости в момент освобождения не происходит ничего. При следующем захвате процесс определяет, что уже обладает всеми необходимыми данными, а значит, ему не нужно генерировать никаких сообщений. В результате при ленивой свободной непротиворечивости, до тех пор пока другой процесс не произведет захват, сетевой трафик вообще не генерируется. Повторяющиеся пары операций захвата—освобождения, происходящие в одном и том же процессе, в отсутствие попыток доступа к данным извне не вызывают никакой нагрузки на сеть.

6.2.7. Поэлементная непротиворечивость

Еще одна модель непротиворечивости, созданная для применения в критических областях, — поэлементная непротиворечивость (*entry consistency*) [48]. Как и оба варианта свободной непротиворечивости, она требует от программиста (или компилятора) вставки кода для захвата и освобождения в начале и конце критической области. Однако в отличие от свободной непротиворечивости, поэлементная непротиворечивость дополнительно требует, чтобы каждый отдельный элемент совместно используемых данных был ассоциирован с переменной синхронизации — блокировкой или барьером. Если необходимо, чтобы к элементам массива имелся независимый параллельный доступ, то различные элементы массива должны быть ассоциированы с различными блокировками. Когда происходит захват переменной синхронизации, непротиворечивыми становятся только те данные, которые ассоциированы с этой переменной синхронизации. Поэлементная непротиворечивость отличается от ленивой свободной непротиворечивости тем, что в последней отсутствует связь между совместно используемыми элементами данных и блокировками или барьерами, потому что при захвате необходимые переменные определяются эмпирически.

Связывая список совместно используемых элементов данных с переменными синхронизации, мы снижаем накладные расходы на захват и освобождение пере-

менных синхронизации до нескольких синхронизируемых элементов данных. Это также позволяет нам, увеличивая степень параллелизма, иметь несколько одновременно выполняемых критических областей, включающих в себя пересекающиеся группы совместно используемых элементов данных. Цена, которую мы платим за это, — дополнительные усилия и сложность связывания всех разделяемых элементов данных с переменными синхронизации. Программирование в этом случае также сложнее и грозит ошибками.

Переменные синхронизации используются следующим образом. Каждая переменная синхронизации имеет текущего владельца — процесс, который захватил ее последним. Владелец может многократно входить в критические области и выходить из них, не посылая в сеть никаких сообщений. Процесс, не являющийся в настоящее время владельцем переменной синхронизации, но желающий захватить ее, должен послать текущему владельцу сообщение, запрашивая право собственности и текущие значения данных, ассоциированных с переменной синхронизации. Кроме того, несколько процессов могут одновременно владеть переменной синхронизации, но не в эксклюзивном режиме. Это означает, что они могут прочесть ассоциированные с переменной данные, но не записать их.

Формально хранилище данных обеспечивает поэлементную непротиворечивость, если оно удовлетворяет трем условиям [50].

- ✦ Захват процессом доступа к переменной синхронизации невозможен до тех пор, пока не осуществлены все обновления отслеживаемых совместно используемых данных этого процесса.
- ✦ Пока один из процессов имеет эксклюзивный доступ к переменной синхронизации, никакой другой процесс не может захватить эту переменную синхронизации, в том числе и не эксклюзивно.
- ✦ После эксклюзивного доступа к переменной синхронизации не эксклюзивный доступ любого другого процесса к этой переменной синхронизации запрещен, пока это не будет разрешено владельцем этой переменной.

Первое условие гласит, что если процесс производит захват, то захват не может быть выполнен (то есть нельзя передать управление следующей инструкции) до тех пор, пока все контролируемые общие данные не станут непротиворечивыми. Другими словами, при захвате должны быть визуализированы все изменения, сделанные в этих данных удаленными процессами.

Второе условие говорит, что перед обновлением элемента совместно используемых данных процесс должен войти в критическую область в эксклюзивном режиме, чтобы гарантировать, что никакой другой процесс в то же самое время не изменяет эти данные.

Третье условие гласит, что если процесс хочет войти в критическую область в не эксклюзивном режиме, он должен сначала проверить, что владелец переменной синхронизации, отслеживающей эту критическую область, получил самую свежую копию отслеживаемых данных.

Пример поэлементной непротиворечивости иллюстрирует рис. 6.12. Вместо того чтобы работать со всеми совместно используемыми данными, в этом примере мы ассоциируем блокировку с каждым элементом данных. Здесь процесс *P1*

осуществляет захват x , изменяет x , после чего захватывает y . Процесс $P2$ захватывает x , но не y , после чего считывает из x значение для a , но для y может считать только нуль (NIL). Поскольку процесс $P3$ сначала захватывает y , он может прочитать значение b после того, как элемент y будет освобожден процессом $P1$.

P1:	Acq(Lx)	W(x)a	Acq(Ly)	W(y)b	Rel(Lx)	Rel(Ly)
P2:					Acq(Lx)	R(x)a
P3:						R(y)NIL
					Acq(Ly)	R(y)b

Рис. 6.12. Допустимая последовательность событий для поэлементной непротиворечивости

Одна из проблем программирования для поэлементной непротиворечивости относится к правильному связыванию данных с переменными синхронизации. Один из способов решения этой проблемы состоит в применении распределенных совместно используемых объектов. Это делается следующим образом. Каждый распределенный объект имеет ассоциированную с ним переменную синхронизации. Эта переменная предоставляется базовой распределенной системой при создании распределенного объекта, но тем не менее полностью скрыта от клиента.

Когда клиент обращается к методу распределенного объекта, базовая система сначала выполняет захват ассоциированной с объектом переменной синхронизации. В результате самые свежие значения состояния объекта, который можно реплицировать и распределить по нескольким машинам, передаются клиентской копии этого объекта. В этот момент, пока объект остается заблокированным от параллельных операций, и происходит обращение. По окончании обращения следует внутренняя операция освобождения, деблокирующая объект для дальнейших операций.

В результате все обращения к распределенному, совместно используемому объекту последовательно непротиворечивы. К счастью, клиент не должен заниматься переменными синхронизации, поскольку они полностью поддерживаются базовой распределенной системой. В то же время каждый объект оказывается автоматически защищенным от одновременного выполнения параллельных обращений.

Подобный подход реализован в языке программирования Otca [28, 30], который мы обсудим в деталях в этой главе чуть позже. Сходный подход использован в CRL [219], где объекты имеют вид неперекрывающихся областей распределенной общей памяти. Каждая область получает ассоциированную с ней переменную синхронизации, предоставляемую базовой исполняющей системой. За синхронизацию доступа к этим областям также отвечает исполняющая система.

6.2.8. Сравнение моделей непротиворечивости

Хотя мы еще будем рассматривать другие модели непротиворечивости, основные (модели непротиворечивости данных) мы уже обсудили. Они отличаются ограничениями, сложностью реализации, простотой программирования и произво-

дительностью. Строгая непротиворечивость — наиболее ограниченный вариант, но поскольку ее реализация в распределенных системах, в сущности, невозможна, то она никогда в них не применяется.

Линеаризуемость — более слабая модель, основанная на идее синхронизированных часов. В ней делать вывод о корректности параллельных программ проще, но все равно слишком сложно для того, чтобы ее можно было использовать для построения реальных программ. С этой точки зрения лучшей моделью является последовательная непротиворечивость, которая применима, популярна среди программистов и действительно широко используется. Однако у нее есть проблема — низкая производительность. Единственный способ улучшить показатели производительности — это ослабить модель непротиворечивости. Некоторые из возможностей иллюстрирует табл. 6.5. Модели перечислены в приближительном порядке снижения ограничений.

Таблица 6.5. Модели непротиворечивости, не требующие операций синхронизации

Непротиворечивость	Описание
Строгая	Абсолютная упорядоченность во времени всех обращений к совместно используемой памяти
Линеаризуемость	Все процессы наблюдают все обращения к совместно используемой памяти в одном и том же порядке. Обращения упорядочены в соответствии с (неуникальными) глобальными отметками времени
Последовательная	Все процессы наблюдают все обращения к совместно используемой памяти в одном и том же порядке. Обращения не упорядочены по времени
Причинная	Все процессы наблюдают все обращения, связанные причинно-следственной связью, к совместно используемой памяти в одном и том же порядке
FIFO	Все процессы наблюдают операции записи любого процесса в порядке их выполнения. Операции записи различных процессов могут наблюдаться разными процессами в разном порядке

Причинная непротиворечивость и непротиворечивость FIFO представляют собой ослабленные модели, в которых отсутствует глобальный контроль над тем, какие операции в каком порядке выполняются. Разным процессам последовательность операций кажется разной. Эти две модели различаются в том, какие последовательности считаются допустимыми, а какие нет.

Другой подход состоит во введении явных переменных синхронизации, как сделано для слабой непротиворечивости, свободной непротиворечивости и поэлементной непротиворечивости. Эти три модели приведены в табл. 6.6. Когда процесс выполняет операцию с обычным элементом совместно используемых данных, отсутствуют какие-либо гарантии по поводу того, когда его смогут увидеть другие процессы. Изменения распространяются только при явной синхронизации. Эти три модели различаются способом синхронизации, но во всех случаях процесс может производить множественные операции чтения и записи в крити-

ческой области без реального переноса данных. После окончания критической области результат передается другим процессам или хранится в готовом виде, ожидая, пока другой процесс не потребует этих данных.

Таблица 6.6. Модели непротиворечивости, использующие операции синхронизации

Непротиворечивость	Описание
Слабая	Совместно используемые данные могут считаться непротиворечивыми только после синхронизации
Свободная	Совместно используемые данные становятся непротиворечивыми после выхода из критической области
Поэлементная	Совместно используемые данные, относящиеся к данной критической области, становятся непротиворечивыми при входе в эту область

Говоря кратко, слабая, свободная и поэлементная непротиворечивости требуют дополнительных программных конструкций, которые при правильном использовании позволяют программистам представить дело так, будто хранилище данных обладает последовательной непротиворечивостью. В принципе эти три модели, использующие явную синхронизацию, можно задействовать для повышения производительности, однако вполне вероятно, что для разных приложений успешность такого подхода будет разной.

6.3. Модели непротиворечивости, ориентированные на клиента

Модели непротиворечивости, описанные в предыдущем разделе, ориентированы на создание непротиворечивого представления хранилища данных. При этом делалось важное предположение о том, что параллельные процессы одновременно изменяют данные в хранилище и необходимо сохранить непротиворечивость хранилища в условиях этой параллельности. Так, например, в случае поэлементной непротиворечивости на базе объектов хранилище данных гарантирует, что при обращении к объекту процесс получит копию объекта, отражающую все произошедшие с ним изменения, в том числе и сделанные другими процессами. В ходе обращения гарантируется также, что нам не помешает никакой другой объект, то есть обратившемуся процессу будет предоставлен защищенный механизм взаимного исключения доступ.

Возможность осуществлять параллельные операции над совместно используемыми данными в условиях последовательной непротиворечивости является базовой для распределенных систем. По причине невысокой производительности последовательная непротиворечивость может гарантироваться только в случае использования механизмов синхронизации — транзакций или блокировок.

В этом разделе мы рассмотрим специальный класс распределенных хранилищ данных, которые характеризуются отсутствием одновременных изменений

или легкостью их разрешения в том случае, если они все-таки случатся. Большая часть операций подразумевают чтение данных. Подобные хранилища данных соответствуют очень слабой модели непротиворечивости, которая носит название потенциальной непротиворечивости. После введения специальных моделей непротиворечивости, ориентированных на клиента, оказывается, что множество нарушений непротиворечивости можно относительно просто скрыть.

6.3.1. Потенциальная непротиворечивость

Степень, в которой процессы действительно работают параллельно, и степень, в которой действительно должна быть гарантирована непротиворечивость, могут быть разными. Существует множество случаев, когда параллельность нужна лишь в урезанном виде. Так, например, во многих системах управления базами данных большинство процессов не производит изменения данных, ограничиваясь лишь операциями чтения. Изменением данных занимается лишь один, в крайнем случае несколько процессов. Вопрос состоит в том, как быстро эти изменения могут стать доступными процессам, занимающимся только чтением данных.

В качестве другого примера рассмотрим глобальную систему именования, такую как DNS. Пространство имен DNS разделено на домены, каждый домен имеет источник именования, который действует, как владелец этого домена. Только этот источник может обновить свою часть пространства имен. Соответственно, конфликт двух операций, которые хотели бы одновременно обновить одни и те же данные (то есть конфликт двойной записи), невозможен. Единственная ситуация, которую может потребоваться решать, — это конфликт чтения-записи. В том случае, когда он происходит, часто распространение изменений может производиться постепенно, в том смысле, что процессы чтения могут обнаружить произошедшие изменения только через некоторое время после того, как они на самом деле произойдут.

Еще один пример — World Wide Web. Фактически web-страницы всегда изменяются одним человеком — web-мастером или настоящим владельцем страницы. В результате разрешать конфликты двойной записи обычно не требуется. С другой стороны, для повышения эффективности браузеры и прокси-серверы часто конфигурируются так, чтобы сохранять загруженные страницы в локальном кэше и возвращать при следующем запросе именно их. Важный момент обоих типов систем web-кэширования состоит в том, что они могут возвращать устаревшие web-страницы. Другими словами, кэшированные страницы, возвращаемые в ответ на запрос клиента, могут не соответствовать более новым версиям страниц, хранящимся на web-сервере. Но даже при этом многие пользователи могут счесть такое несоответствие приемлемым.

Эти примеры могут рассматриваться как случаи распределенных и реплицируемых баз данных (крупных), нечувствительных к относительно высокой степени нарушения непротиворечивости. Обычно в них длительное время не происходит изменения данных, и все реплики постепенно становятся непротиворечивыми. Такая форма непротиворечивости называется *потенциальной непротиворечивостью* (*eventual consistency*).

Потенциально непротиворечивые хранилища данных имеют следующее свойство: в отсутствие изменений все реплики постепенно становятся идентичными. Как мы будем говорить позднее в этой главе, потенциальная непротиворечивость, в сущности, требует только, чтобы изменения гарантированно расходились по всем репликам. Конфликты двойной записи часто относительно легко разрешаются, если предположить, что вносить изменения может лишь небольшая группа процессов. Поэтому реализация потенциальной непротиворечивости часто весьма дешева. Специфику реализации мы рассмотрим позже в этой главе.

Потенциально непротиворечивые хранилища данных с успехом работают, если клиент всегда осуществляет доступ к одной и той же реплике. Однако при работе с разными репликами возникают проблемы. Они прекрасно иллюстрируются схемой доступа мобильного пользователя к распределенной базе данных, представленной на рис. 6.13.

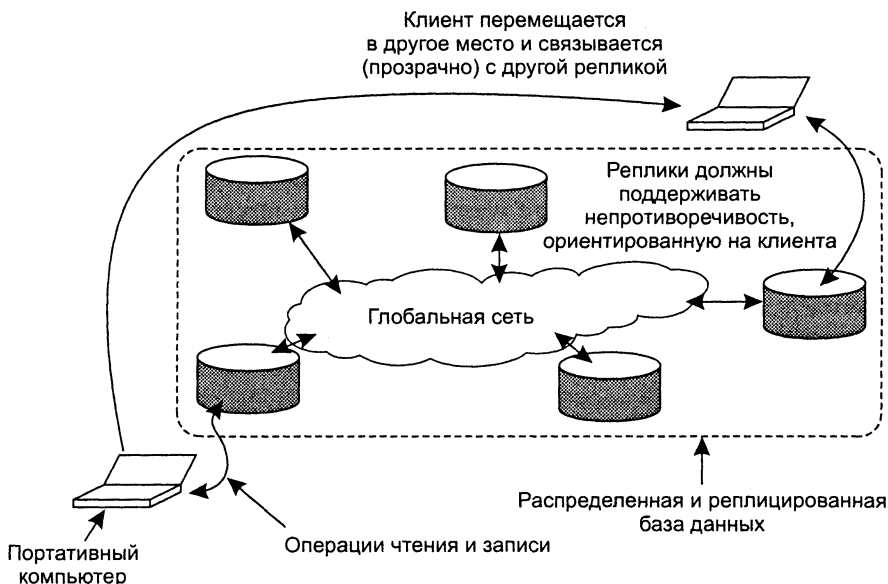


Рис. 6.13. Принцип доступа мобильного пользователя к разным репликам распределенной базы данных

Мобильный пользователь работает с базой данных, прозрачно подсоединяясь к одной из ее реплик. Другими словами, приложение, работающее на портативном компьютере пользователя, не знает, с какой именно репликой оно работает. Допустим, пользователь произвел несколько операций изменения данных и отсоединился от базы. Позже он снова начал работать с базой данных, возможно, после перемещения в другое место или с другого устройства доступа. В этот момент пользователь может подсоединиться к другой нежели раньше реплике, что и показано на рисунке. Однако если изменения, сделанные в базе данных раньше, еще не распространены на эту реплику, пользователь обнаружит противоре-

чивость данных в базе данных. В частности, он будет ожидать увидеть сделанные раньше изменения, а вместо этого обнаружит, что ничего не изменилось.

Этот пример типичен для потенциально непротиворечивых хранилищ данных и вызван тем, что пользователь может иногда работать с различными репликами. Острота проблемы снижается путем введения *непротиворечивости, ориентированной на клиента* (*client-centric consistency*). По существу, непротиворечивость, ориентированная на клиента, предоставляет *одному клиенту* гарантии непротиворечивости при доступе к хранилищу данных. Относительно параллельного доступа других клиентов никаких гарантий не предоставляется.

Моделям непротиворечивости, ориентированным на клиента, дала начало работа над Bayou [453, 454] — системой управления базами данных, разработанной для мобильных вычислений. В этой системе предполагается, что сетевые подключения ненадежны и имеют различные проблемы с производительностью. В эту категорию попадают беспроводные сети и глобальные сети, такие как Интернет.

В системе Bayou различают четыре основных модели непротиворечивости. Для рассмотрения этих моделей мы вновь обсудим хранилище данных, распределенное по нескольким машинам. Когда процесс получает доступ к хранилищу данных, он обычно связывается с локальной (или ближайшей) копией, хотя в принципе подойдет любая копия. Все операции чтения и записи осуществляются с локальной копией. Изменения постепенно распространяются и на другие копии. Для простоты примем, что элементы данных имеют ассоциированного с ними владельца, которым является единственный процесс, имеющий право изменять их. Таким образом, мы избежим конфликтов двойного чтения.

Модели непротиворечивости, ориентированные на клиента, описываются с использованием следующей нотации. Пусть $x_i[t]$ означает версию элемента данных x на локальной копии L_i в момент времени t . Версия $x_i[t]$ — результат серии операций записи, произведенных в L_i после инициализации. Мы обозначим эту серию как $WS(x_i[t])$. Если операции из серии $WS(x_i[t_1])$ были также произведены в локальной копии L_j в более позднее время t_2 , мы запишем это как $WS(X_i[t_1]; X_j[t_2])$. Если временная очередность операций будет ясна из контекста, мы будем опускать индекс у символа времени.

6.3.2. Монотонное чтение

Первая из моделей непротиворечивости, ориентированная на клиента, — *монотонное чтение*. Хранилище данных обеспечивает *непротиворечивость монотонного чтения* (*monotonic-read consistency*), если удовлетворяет следующему условию: *если процесс читает значение элемента данных x , любая последующая операция чтения x всегда возвращает то же самое или более позднее значение*.

Другими словами, непротиворечивость монотонного чтения гарантирует, что если процесс в момент времени t видит некое значение x , то позже он никогда не увидит более старого значения x .

В качестве примера применения монотонного чтения рассмотрим распределенную базу данных электронной почты. В этой базе данных почтовый ящик каж-

дого пользователя может быть распределен по нескольким машинам и реплицирован. Корреспонденция добавляется в почтовые ящики всех реплик. Однако изменения распространяются медленно (по запросу). Данные пересылаются копии базы только в том случае, когда эти данные нужны этой копии для поддержания непротиворечивости. Представим себе, что пользователь читает свою почту в Сан-Франциско. Допустим, что чтение почты не оказывает воздействия на почтовый ящик, то есть сообщения не удаляются, не сохраняются во вложенных каталогах, не помечаются как прочитанные и т. п. Когда пользователь после этого улетает в Нью-Йорк и снова открывает свой почтовый ящик, непротиворечивость монотонного чтения гарантирует, что сообщения, которые он видел в своем почтовом ящике в Сан-Франциско, он увидит в своем почтовом ящике и в Нью-Йорке.

Если использовать нотацию, похожую на ту, которая применялась для описания непротиворечивости, ориентированной на данные, непротиворечивость монотонного чтения может быть графически представлена так, как показано на рис. 6.14. Вдоль вертикальной оси располагаются две различных локальных копии хранилища данных, $L1$ и $L2$. Вдоль горизонтальной оси откладывается время.

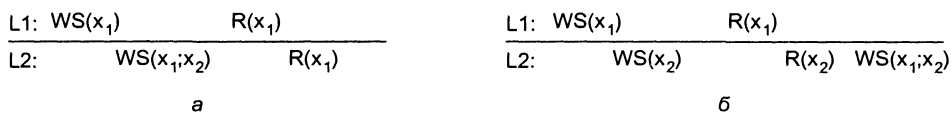


Рис. 6.14. Операции чтения, осуществляемые одиночным процессом P над двумя различными копиями одного хранилища данных. Хранилище с непротиворечивостью монотонного чтения (а). Хранилище без непротиворечивости монотонного чтения (б)

На рис. 6.14, *a* процесс P сначала выполняет операцию чтения x из копии $L1$, которая возвращает значение x_1 (в этот момент). Это значение является результатом операции записи $WS(x_1)$, производимой с копией $L1$. Позднее P выполняет операцию чтения x из хранилища $L2$, обозначаемую как $R(x_2)$. Чтобы гарантировать непротиворечивость монотонного чтения, все операции, сделанные в $WS(x_1)$, должны распространиться на $L2$ до момента второго чтения. Другими словами, мы должны точно знать, что операция $WS(x_1)$ является частью операции $WS(x_2)$ и может быть выражена как $WS(x_1; x_2)$.

В противоположность этому, на рис. 6.14, *б* показана ситуация, в которой непротиворечивость монотонного чтения не гарантируется. После того как процесс P считывает x_1 из $L1$, он производит операцию $R(x_2)$ из $L2$. Однако в $L2$ была выполнена только операция записи $WS(x_2)$, поэтому нет никаких гарантий, что в этом наборе учтены также и операции, содержащиеся в $WS(x_1)$.

6.3.3. Монотонная запись

Во многих ситуациях важно, чтобы по всем копиям хранилища данных в правильном порядке распространялись операции записи. Это можно осуществить при условии *непротиворечивости монотонной записи* (*monotonic-write consistency*). Если хранилище обладает свойством непротиворечивости монотонной записи, для

него соблюдается следующее условие: *операция записи процесса в элемент данных x завершается раньше любой из последующих операций записи этого процесса в элемент x .*

Здесь завершение операции записи означает, что копия, над которой выполняется следующая операция, отражает эффект предыдущей операции записи, произведенной тем же процессом, и при этом не имеет значения, где эта операция была инициирована. Другими словами, операция записи в копию элемента данных x выполняется только в том случае, если эта копия соответствует результатам предыдущей операции записи, выполненных над другими копиями x .

Отметим, что непротиворечивость монотонной записи напоминает одну из моделей непротиворечивости, ориентированной на данные, — непротиворечивость FIFO. Сущность непротиворечивости FIFO состоит в том, что операции записи одного процесса всегда выполняются в правильной очередности. Такое же ограничение на очередность применяется и при монотонной записи, за исключением того, что здесь мы говорим об одном процессе, а не о наборе параллельных процессов.

Актуализация копии x не обязательна, если каждая операция записи полностью изменяет существующее значение x . Однако операции записи часто выполняются только над частью элемента данных. Рассмотрим, например, библиотеку подпрограмм. Во многих случаях обновление библиотеки заключается в замене одной или нескольких функций, а результатом этого является новая версия. При условии непротиворечивости монотонной записи даются гарантии того, что если изменения вносятся в одну из копий библиотеки, сначала будут внесены все предшествующие изменения. Получившаяся библиотека будет реально представлять собой самую последнюю версию и содержать все обновления, имевшиеся в предыдущей версии этой библиотеки.

Непротиворечивость монотонной записи иллюстрирует рис. 6.15. Процесс P осуществляет операцию записи $W(x_1)$ элемента x в локальную копию $L1$ (см. рис. 6.15, *а*). Позднее P выполняет еще одну операцию записи $W(x_2)$, на этот раз над $L2$. Для непротиворечивости монотонной записи необходимо, чтобы предыдущая операция записи в $L1$ уже распространилась на $L2$. Это поясняет наличие операции $W(x_1)$ в $L2$ и то, почему она помещена перед операцией $W(x_2)$.

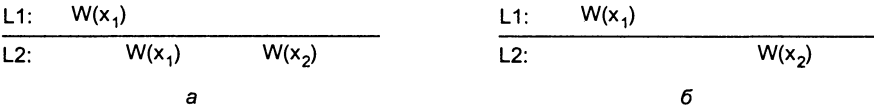


Рис. 6.15. Операции записи, осуществляемые единичным процессом P над двумя различными копиями одного хранилища данных. Хранилище данных с непротиворечивостью монотонной записи (*а*). Хранилище без непротиворечивости монотонной записи (*б*)

В противоположность этому, на рис. 6.15, *б* показана ситуация, в которой непротиворечивость монотонной записи не гарантирована. По сравнению с рис. 6.15, *а* распространение операции $W(x_1)$ на копию $L2$ отсутствует. Другими словами, нет никаких гарантий того, что копия элемента x , над которой производится вто-

рая операция записи, имеет то же самое или более позднее значение, чем полученное во время завершения операции $W(x_1)$ в $L1$.

Отметим, что по определению непротиворечивости монотонной записи операции записи одного процесса выполняются в том же порядке, в котором они были инициированы. В ослабленной форме непротиворечивости монотонного чтения эффект операций чтения наблюдается только в том случае, если все предшествующие операции чтения уже завершились, но не обязательно в том порядке, в котором они начинались. Такая непротиворечивость применяется в тех случаях, когда операции записи коммутативны, то есть их упорядоченность на самом деле не является необходимой. Детали можно найти в [453].

6.3.4. Чтение собственных записей

Существует и еще одна модель непротиворечивости, ориентированная на клиента, весьма схожая с непротиворечивостью монотонной записи. Хранилище данных обладает свойством непротиворечивости чтения собственных записей (*read-your-writes consistency*), если оно удовлетворяет следующему условию: *результат операций записи процесса в элемент данных x всегда виден последующим операциям чтения x этого же процесса*.

Другими словами, операция записи всегда завершается раньше следующей операции чтения того же процесса, где бы ни происходила эта операция чтения.

Отсутствие непротиворечивости чтения собственных записей часто обнаруживается при обновлении web-страниц с последующим просмотром результатов. Операции обновления часто производятся при помощи стандартного или другого текстового редактора, который записывает новую версию в совместно используемую файловую систему web-сервера. Web-браузер пользователя работает с тем же самым файлом, возможно, запрашивая его с локального web-сервера. Однако после первой же загрузки этого файла сервер или браузер кэшируют для последующего доступа его локальную копию. Соответственно, если браузер или сервер возвращает кэшированную копию, а не оригинальный файл, то при обновлении web-страницы пользователь не в состоянии увидеть его результат. Непротиворечивость чтения собственных записей может гарантировать, что если редактор и браузер интегрированы в единую программу, то при обновлении страницы кэш объявляется неактуальным, и в результате загружается и отображается новая версия файла.

Тот же эффект проявляется и при изменении паролей. Так, например, для входа в цифровую библиотеку в Web часто необходимо иметь там учетную запись с соответствующим паролем. Однако при смене пароля для вступления нового пароля в силу может потребоваться несколько минут. В результате эти несколько минут библиотека будет недоступна для пользователей. Задержка может возникнуть из-за того, что для работы с паролями используется отдельный сервер и последующая рассылка пароля (зашифрованного) серверам, на которых находится библиотека, может потребовать времени. Эта проблема уже встречалась в Grapvine, одной из первых распределенных систем, поддерживавших причинно-следственную связь. Помогла решить эту проблему реализация непротиворечивости чтения собственных записей [61].

На рис. 6.16, *а* показано хранилище данных с непротиворечивостью чтения собственных записей. Отметим, что этот рисунок очень похож на рис. 6.14, *а*, за исключением того, что непротиворечивость теперь определяется по последней операции записи процесса *P*, а не по последней операции чтения.

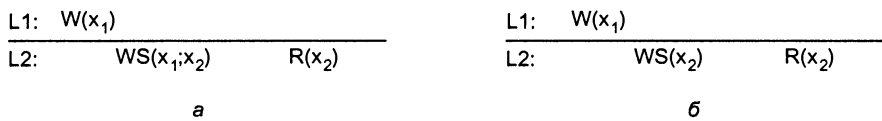


Рис. 6.16. Хранилище данных с непротиворечивостью чтения собственных записей (*а*).
Хранилище данных без непротиворечивости чтения собственных записей (*б*)

На рис. 6.16, *а* процесс *P* выполняет операцию записи $W(x_1)$, а затем операцию чтения другой локальной копии. Непротиворечивость чтения собственных записей гарантирует, что эффект, произведенный операцией записи, будет наблюдаться при последующей операции чтения. На это указывает операция $WS(x_1;x_2)$, означающая, что $W(x_1)$ является частью операции $WS(x_2)$. В противоположность этому, на рис. 6.16, *б* операция $W(x_1)$ остается вне операции $WS(x_2)$, указывая на то, что эффект предыдущей операции записи процесса *P* не распространяется на копию *L2*.

6.3.5. Запись за чтением

Последняя модель непротиворечивости, ориентированная на клиента, — это модель, в которой изменения распространяются как результаты предыдущей операции чтения. В этом случае говорят, что хранилище данных обеспечивает *непротиворечивость записи за чтением* (*writes-follow-reads consistency*), если соблюдается следующее условие: *операция записи в элемент данных x процесса, следующая за операцией чтения x того же процесса, гарантирует, что будет выполняться над тем же самым или более свежим значением x , которое было прочитано предыдущей операцией.*

Иными словами, любая последующая операция записи в элемент данных x , производимая процессом, будет осуществляться с копией x , которая имеет последнее считанное тем же процессом значение x .

Непротиворечивость записи за чтением позволяет гарантировать, что пользователи сетевой группы новостей увидят письма с ответами на некое письмо только позже оригинального письма [453]. Поясним проблему. Представьте себе, что пользователь сначала читает письмо *A*. Затем он реагирует на него, посылая письмо *B*. Согласно требованию непротиворечивости записи за чтением, письмо *B* будет разослано во все копии группы новостей только после того, как туда будет послано письмо *A*. Отметим, что пользователи, которые только читают письма, не нуждаются в какой-либо особой модели непротиворечивости, ориентированной на клиента. Непротиворечивость записи после чтения обеспечит сохранение в локальной копии ответа на письмо только в том случае, если там уже наличествует оригинал.

Эту модель непротиворечивости иллюстрирует рис. 6.17. На рис. 6.17, а процесс читает элемент x из локальной копии $L1$. Операции записи, заменяющие прочитанное значение, присутствуют в наборе операций записи в $L2$, которые позже осуществляет этот процесс (отметим, что другие процессы в $L2$ также наблюдают эти операции записи). В противоположность этому, как показано на рис. 6.17, б, отсутствуют всякие гарантии того, что операция записи в $L2$ выполняется над копией, которая не противоречит копии, только что считанной из $L1$.

L1: WS(x_1)	R(x_1)	L1: WS(x_1)	R(x_1)
L2: WS($x_1; x_2$)	W(x_2)	L2: WS(x_2)	W(x_2)
а		б	

Рис. 6.17. Хранилище данных с непротиворечивостью записи после чтения (а).
Хранилище данных без непротиворечивости записи после чтения (б)

6.3.6. Реализация

Реализация непротиворечивости, ориентированной на клиента, относительно проста, если не принимать во внимание вопросы производительности. Далее мы сначала опишем именно такую реализацию, после чего поговорим о более «жизненных» вариантах реализации.

Примитивная реализация

В примитивной реализации непротиворечивости, ориентированной на клиента, каждой операции записи приписывается глобально уникальный идентификатор. Это делается тем сервером, который первым выполняет операцию. Мы говорим также, что операция *иницируется* этим сервером. Отметим, что генерация глобально уникальных идентификаторов может быть реализована в виде локальной операции (см. задания в конце главы 4). Итак, для каждого клиента мы отслеживаем два набора идентификаторов записи. Набор чтения для клиента состоит из идентификаторов записи, соответствующих операциям чтения, выполненным клиентом. Аналогично, набор записи состоит из идентификаторов операций записи, выполненных клиентом.

Непротиворечивость монотонного чтения реализуется следующим образом. Когда клиент осуществляет операцию чтения с сервера, этот сервер проверяет набор чтения клиента на локальное присутствие результатов всех его операций записи (размер этого набора может породить проблемы производительности, решение которых обсуждается ниже). Если это не так, он связывается с другими серверами, чтобы актуализировать данные до проведения операции чтения. С другой стороны, операции чтения могут быть переданы серверу, на котором производились операции записи. После выполнения операции чтения последующие операции записи, производимые на выбранном сервере и связанные с операциями чтения, добавляются к набору чтения клиента.

Отметим, что точно можно определить лишь место, где происходят операции записи, указанные в наборе чтения. Так, например, идентификатор записи может

включать в себя идентификатор сервера, инициировавшего операцию. Этот сервер требуется, например, для того, чтобы запротоколировать операцию записи в журнале, чтобы ее можно было повторить на другом сервере. Кроме того, операции записи должны осуществляться в том же порядке, в котором они были инициированы. Упорядоченность введена для того, чтобы клиент мог генерировать глобально уникальный последовательный номер, включаемый в идентификатор записи, такой, например, как отметка времени Лампорта. Если каждый элемент данных модифицируется только его владельцем, последний и сможет поддерживать последовательную нумерацию.

Непротиворечивость монотонной записи реализуется аналогично монотонному чтению. Всякий раз при инициировании клиентом новой операции записи на сервере этот сервер просматривает набор записи клиента. (И снова, размер этого набора может быть слишком велик для существующих требований по производительности. Альтернативные решения будут изложены ниже.) Сервер удостоверяется, что указанные операции записи выполнялись первыми и в правильном порядке. После выполнения новой операции идентификатор записи этой операции добавляется к набору записи. Отметим, что актуализация набора записи текущего сервера при помощи набора записи клиента может существенно увеличить время отклика сервера.

Непротиворечивость чтения собственных записей также требует, чтобы сервер, на котором выполняются операции чтения, имел доступ ко всем операциям записи из набора записи клиента. Операции записи можно просто извлекать с других серверов перед выполнением операции чтения, даже если это грозит обернуться проблемами с временем отклика. С другой стороны, клиентское программное обеспечение может само найти сервер, на котором операции записи, указанные в наборе записи клиента, уже выполнены.

И, наконец, непротиворечивость записи за чтением может быть реализована следующим образом. Сначала выбранный сервер актуализируется с помощью операций записи, входящих в набор чтения клиента, а затем в набор записи добавляются идентификатор операции записи и идентификаторы из набора чтения (таким образом, учитывается только что выполненная операция записи).

Повышение производительности

Легко заметить, что наборы чтения и записи, ассоциированные с каждым клиентом, могут стать чересчур большими. Чтобы сохранить управляемость этих наборов, операции чтения и записи клиента могут группироваться в сеансы. Сеансы обычно ассоциируются с приложениями: они открываются при запуске приложения и закрываются по окончании работы с ним. Однако сеансы также могут ассоциироваться и с временно запускаемыми приложениями, такими как пользовательский агент для электронной почты. В любом случае, когда клиент закрывает сеанс, наборы очищаются. Разумеется, если клиент откроет сеанс и никогда его не закроет, ассоциированные с ним наборы чтения и записи могут стать очень большими.

Основная проблема примитивной реализации вытекает из способа представления наборов чтения и записи. Каждый из наборов включает в себя множество

идентификаторов операций записи. Когда клиент передает серверу запрос на запись или чтение, серверу передается и набор идентификаторов — для проверки того факта, что все операции записи соответствуют обрабатываемому на сервере запросу.

Эту информацию удобнее представлять в виде векторных отметок времени следующим образом. Сначала когда сервер принимает требование на проведение новой операции записи W , он описанным выше способом присваивает этой операции глобально уникальный идентификатор WID вместе с отметкой времени $ts(WID)$. Следующая операция записи на этом сервере получает отметку времени с большим значением. Каждый сервер S_i поддерживает векторную отметку времени $RCVD(i)$, где $RCVD(i)[j]$ соответствует отметке времени последней операции записи, инициированной на сервере S_j , которая была получена (и обработана) сервером S_j .

Когда клиент посылает запрос на выполнение операции записи или чтения на определенный сервер, сервер возвращает свою текущую отметку времени вместе с результатом этой операции. Наборы чтения и записи представляются последовательными векторными отметками времени. Для любого такого набора A мы создаем векторную отметку времени $VT(A)$ с набором $VT(A)[i]$, равным отметке времени, максимальной среди всех операций A , инициированных на сервере S_i , что приводит к эффективному представлению этих наборов.

Объединение двух наборов идентификаторов, A и B , обозначается в виде векторной отметки времени $VT(A + B)$, где $VT[A + B][i]$ равно $\max(VT(A)[i], VT(B)[i])$. К тому же, чтобы выяснить, содержится ли набор идентификаторов A в другом наборе B , нам нужно проверить только, выполняется ли для каждого индекса i условие $VT(A)[i] \leq VT(B)[i]$.

Когда сервер возвращает клиенту свою текущую отметку времени, клиент приводит в соответствие с ней векторную отметку времени своего собственного набора чтения или записи (в зависимости от выполняемой операции). Рассмотрим вариант с непротиворечивостью монотонного чтения, при которой клиенту с сервера S возвращается векторная отметка времени. Если векторная отметка набора чтения клиента — $VT(Rset)$, то для каждого индекса j значение $VT(Rset)[j]$ равно $\max\{VT(Rset)[j], RCVD(i)[j]\}$. Набор чтения клиента отражает последнюю известную ему операцию записи. Соответствующая векторная отметка времени будет послана (возможно, на другой сервер) в ходе следующей операции чтения клиента.

6.4. Протоколы распределения

До сих пор мы рассуждали лишь о тех или иных моделях непротиворечивости, стараясь поменьше обращать внимания на их реализацию. В этом разделе мы обсудим различные способы распространения, точнее способы распределения адресованных репликам обновлений независимо от поддерживаемой ими модели непротиворечивости. Протоколы для конкретных моделей непротиворечивости мы рассмотрим в следующем разделе.

6.4.1. Размещение реплик

Основную проблему проектирования распределенных хранилищ данных, которую мы должны решить, — это когда, где и кому размещать копии хранилища (см. также [233]). Различают три различных типа копий, логически организованных так, как показано на рис. 6.18.

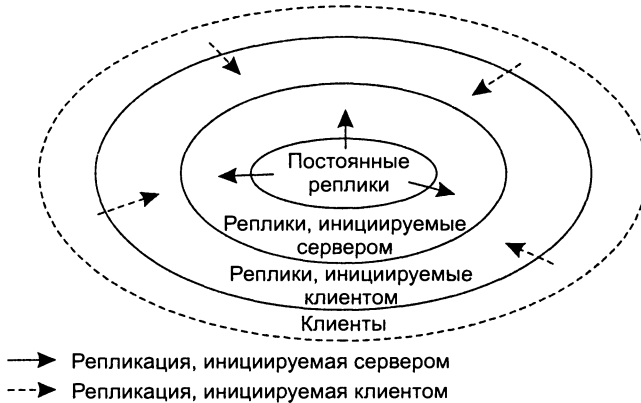


Рис. 6.18. Логическая организация различных типов копий хранилищ данных в виде трех концентрических колец

Постоянные реплики

Постоянные реплики можно рассматривать как исходный набор реплик, образующих распределенное хранилище данных. Во многих случаях число постоянных реплик невелико. Рассмотрим, например, web-сайт. Распределение web-сайтов обычно происходит в одном из двух вариантов. В первом варианте распределения файлы, которые составляют сайт, реплицируются на ограниченном числе серверов одной локальной сети. Когда приходит запрос, он передается одному из серверов, например, с использованием стратегии обхода кольца [96].

Второй тип распределения web-сайтов — это *создание зеркал (mirroring)*. В этом случае web-сайт копируется на ограниченное количество разбросанных по всему Интернету серверов, называемых *зеркальными сайтами (mirror sites)*, или просто *зеркалами*. В большинстве случаев клиенты просто выбирают одно из зеркал из предложенного им списка. Зеркальные web-сайты обычно основаны на технологии кластерных web-сайтов, поддерживающих одну или несколько реплик с возможностью в той или иной степени их статического конфигурирования.

Подобная же статическая организация применяется и для создания распределенных баз данных [337]. Базы данных могут быть реплицированы и распределены по нескольким серверам, образующим вместе *кластер рабочих станций (Cluster Of Workstations, COW)*. О таких кластерах часто говорят как об *архитектуре без разделения (shared-nothing architecture)*, подчеркивая, что ни диски, ни оперативная память не используются процессорами совместно. С другой стороны, ба-

зы данных могут распределяться и возможно реплицироваться по множеству географически разбросанных мест. Такая архитектура нередко применяется при построении федеральных баз данных [416].

Реплики, иницируемые сервером

В противоположность постоянным репликам, реплики, иницируемые сервером, являются копиями хранилища данных, которые создаются для повышения производительности и создание которых иницируется хранилищем данных (его владельцем). Рассмотрим, например, web-сервер, находящийся в Нью-Йорке. Обычно этот сервер в состоянии достаточно быстро обрабатывать входящие запросы, но может случиться так, что внезапно в течение нескольких дней из неизвестного удаленного от сервера места пойдет поток запросов. (Такой поток, как показывает короткая история Web, может быть вызван множеством причин.) В этом случае может оказаться разумным создать в регионах несколько временных реплик, призванных работать с приходящими запросами. Эти реплики известны также [190] под названием *выдвинутых кэшей* (*push caches*).

Совсем недавно проблема динамического размещения реплик была подхвачена службами web-хостинга. Эти службы, которые, в сущности, предлагают набор серверов (более менее статический), разбросанных по всему Интернету, поддерживают и предоставляют доступ к web-файлам, принадлежащим третьим лицам. Для улучшения качества услуг эти службы хостинга могут динамически реплицировать файлы на те серверы, для которых такая репликация вызовет увеличение производительности, то есть поближе к клиентам или группам клиентов.

Одна из проблем реплик, создание которых иницировано сервером, состоит в необходимости принятия решения о том, где и когда будут создаваться и уничтожаться эти реплики. Подход к динамической репликации файлов в случае служб web-хостинга описан в [365]. Алгоритм разработан для поддержки web-страниц и поэтому предусматривает относительно редкие, по сравнению с запросами на чтение, обновления. В качестве модулей данных используются файлы.

Основу функционирования алгоритма динамической репликации составляют два положения. Во-первых, репликация должна производиться для уменьшения нагрузки на сервер. Во-вторых, конкретные файлы с сервера должны перемещаться или реплицироваться на серверы, расположенные как можно ближе к клиентам, от которых исходит множество запросов на эти файлы. Далее мы сосредоточимся только на втором положении. Мы также опустим множество деталей, которые можно найти в [365].

Каждый сервер подсчитывает число обращений к файлам и отслеживает, откуда приходят эти обращения. В частности, это означает, для данного клиента S каждый сервер может определить, какой из серверов службы web-хостинга к нему ближе всего (эта информация может быть получена, например, из баз данных маршрутизации). Если клиент C_1 и клиент C_2 совместно используют один и тот же «ближайший» сервер P , все запросы на доступ к файлу F на сервере Q от C_1 и C_2 будут зарегистрированы на Q в едином счетчике обращений $cnt_Q(P, F)$. Эта картина показана на рис. 6.19.

Когда число запросов на конкретный файл F на сервере S упадет ниже порога удаления $del(S, F)$, файл будет удален с сервера S . Соответственно, число реплик

этого файла уменьшится, что, возможно, приведет к росту нагрузки на другие серверы. Для того чтобы гарантировать наличие в сети хотя бы одной копии файла, принимаются специальные меры.

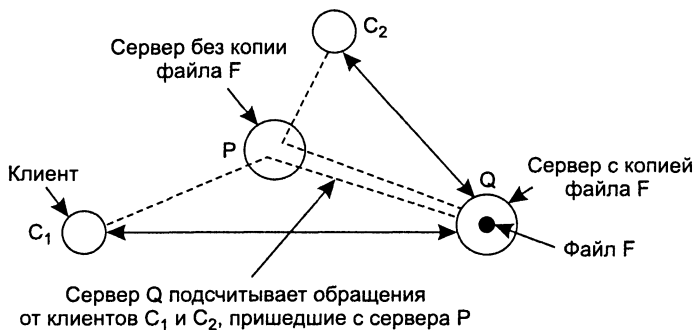


Рис. 6.19. Подсчет числа обращений от различных клиентов

Порог репликации $rep(S, F)$, выбираемый всегда выше порога удаления, указывает на то, что число обращений к конкретному файлу настолько высоко, что его стоит реплицировать на другой сервер. Если число запросов находится где-то между порогами репликации и удаления, файл можно только переместить. Другими словами, в этом случае необходимо как минимум сохранить число реплик этого файла неизменным.

Когда сервер Q решает переопределить местоположение хранимых файлов, он проверяет счетчики обращений для каждого файла. Если общее число обращений к файлу F на Q упало ниже порога удаления $del(Q, F)$, он удаляет F , если только это не последняя его копия. Кроме того, если для какого-либо сервера P значение счетчика $cnt_Q(P, F)$ превышает половину всех запросов к F на Q , сервер P запрашивается на предмет переноса на него копии F . Другими словами, сервер Q будет пытаться перенести файл F на сервер P .

Перенос файла F на сервер P может не быть успешным, например, из-за того, что сервер P и так уже сильно загружен или на нем отсутствует место на диске. В этом случае сервер Q попытается перенести файл F на другой сервер. Разумеется, репликация может производиться только в том случае, если общее число запросов к F на Q будет выше порога репликации $rep(Q, F)$. Сервер Q проверит все остальные серверы службы web-хостинга, начиная с наиболее удаленных. Если для некоторого сервера R значение счетчика $cnt_Q(R, F)$ превзойдет некоторую долю всех запросов к файлу F на сервере Q , будет сделана попытка реплицировать F на R .

Иницируемая сервером репликация постепенно становится все популярнее, особенно в контексте служб web-хостинга, которые были описаны. Отметим, что если мы гарантируем, что каждый элемент данных всегда хранится как минимум на одном сервере, мы можем ограничиться только иницируемой сервером репликацией, вообще отказавшись от постоянных реплик. Тем не менее постоянные реплики продолжают активно использоваться в системах резервного копирования, а также как единственные реплики, гарантирующие сохранение непрото-

речивости данных при внесении изменений. При наличии таких постоянных реплик иницилируемые сервером реплики затем используются для максимального приближения к клиентам копий, предназначенных только для чтения.

Реплики, иницилируемые клиентом

Еще один важный тип реплик — реплики, создаваемые по инициативе клиентов. Иницилируемые клиентом реплики часто называют *клиентскими кэшами* (*client caches*), или просто *кэшами* (*caches*). В сущности, кэш — это локальное устройство хранения данных, используемое клиентом для временного хранения копии запрошенных данных. В принципе управление кэшем полностью возлагается на клиента. Хранилище, из которого извлекаются данные, не делает ничего для поддержания непротиворечивости кэшированных данных. Однако, как мы увидим, существует множество случаев, в которых клиент полагается на то, что хранилище данных известит его об устаревании кэшированных данных.

Клиентский кэш используется только для сокращения времени доступа к данным. Обычно, когда клиент хочет получить доступ к некоторым данным, он связывается с ближайшим хранилищем с их копией и считывает оттуда данные, которые хочет считать, или сохраняет туда данные, которые он только что изменил. Производительность большинства операций, включающих только чтение данных, можно повысить, указав клиенту на необходимость сохранять запрошенные данные в близко расположенном кэше. Такой кэш может располагаться на машине клиента или на отдельной машине в той же локальной сети, что и машина клиента. Когда клиенту в следующий раз потребуются считать те же самые данные, он просто получит их из локального кэша. Эта схема отлично работает, если в промежутке между запросами данные не менялись.

Время хранения данных в кэше обычно ограничивается, например, чтобы предотвратить фатальное устаревание информации или просто освободить место для новых данных. Если запрашиваемые данные могут быть извлечены из локального кэша, говорят, что произошло *кэш-попадание* (*cache hit*). Чтобы повысить число кэш-попаданий, кэш может совместно использоваться несколькими клиентами. Основой для этого служит предположение о том, что данные, запрошенные клиентом C_1 , могут потребоваться также и другому расположенному рядом с ним клиенту C_2 .

Насколько это предположение верно, в значительной степени зависит от типа хранилища данных. Так, например, в традиционных файловых системах файлы вообще редко используются совместно [68, 308], и создание общего кэша бесполезно. С другой стороны, общий кэш web-страниц оказывается очень полезным, хотя вносимый им рост производительности постепенно сходит на нет по причине того, что сайтов, совместно используемых различными клиентами, становится все меньше по сравнению с общим количеством web-страниц [38].

Размещение клиентского кэша — дело относительно несложное. Обычно кэш располагается на той же машине, что и клиент, или на совместно используемой клиентами машине в одной с ними локальной сети. Однако в некоторых случаях системные администраторы применяют дополнительные уровни кэширования,

вводя кэш, совместно используемый множеством отделов или фирм, или даже создавая общий кэш для целых регионов — провинций или стран.

Еще один подход к кэшированию — поместить серверы (кэши) в определенных точках глобальной сети и позволить клиенту найти ближайший. Когда сервер найден, ему посылается запрос на сохранение данных, которые клиент откуда-то получил, как описывается в [318]. Позже в этой главе мы вернемся к кэшированию при описании протоколов непротиворечивости.

6.4.2. Распространение обновлений

Операции обновления в распределенных и реплицируемых хранилищах данных обычно инициируются клиентом, после чего пересылаются на одну из копий. Оттуда обновление распространяется на остальные копии, гарантируя тем самым сохранение непротиворечивости. Как мы увидим далее, существуют различные аспекты проектирования, связанные с распространением обновлений.

Состояние против операций

Один из важных вопросов проектирования заключается в том, что же именно мы собираемся распространять. Существует три основных возможности:

- ◆ распространять только извещение об обновлении;
- ◆ передавать данные из одной копии в другую;
- ◆ распространять операцию обновления по всем копиям.

Распространение извещения производится в соответствии с *протоколами о несостоятельности (invalidation protocols)*. Согласно протоколу о несостоятельности другие копии информируются об имевшем место обновлении, а также о том, что хранящиеся у них данные стали неправильными. Эта информация может определять, какая именно часть хранилища данных была изменена, то есть какая часть данных перестала соответствовать действительности. Важно отметить, что при этом не передается ничего кроме извещения. Независимо от требуемой для неправильной копии операции, обычно сначала нужно обновить эту копию. Конкретные действия по обновлению зависят от поддерживаемой модели непротиворечивости.

Основное преимущество протоколов о несостоятельности в том, что они используют минимум пропускной способности сети. Единственная информация, которую необходимо передавать, — это указание на то, какие данные стали неправильными. Протоколы о несостоятельности обычно прекрасно работают при большем, по сравнению с операциями чтения, количестве операций записи, то есть в условиях, когда отношение операций чтения к операциям записи относительно невелико.

Рассмотрим, например, хранилище данных, в котором обновления распространяются путем рассылки обновленных данных всем репликам. Если размер обновленных данных велик, а обновления, по сравнению с операциями чтения, происходят часто, мы можем столкнуться с ситуацией, когда два обновления происходят одно за другим так, что в промежутке между ними операции чтения

отсутствуют. Соответственно, распространение первого обновления на все реплики практически бесполезно, поскольку его результаты будут стерты вторым обновлением. Вместо этого мы можем послать извещение о том, что данные были обновлены, и это будет значительно быстрее.

Передача обновленных данных между репликами — это вторая альтернатива, она применяется, когда отношение операций чтения к операциям записи относительно велико. В этом случае высока вероятность того, что обновление окажется эффективным в том смысле, что модифицированные данные будут прочитаны до того, как произойдет следующее обновление. Вместо того чтобы распространять обновленные данные, достаточно вести журналы обновлений и для снижения нагрузки на сеть передавать только эти журналы. Кроме того, передачу данных часто можно агрегировать в том смысле, что несколько модификаций можно упаковать в одно сообщение. Это также снизит затраты на взаимодействие.

Третий подход состоит в том, чтобы отказаться от переноса модифицированных данных целиком, а указывать каждой реплике, какую операцию с ней следует произвести. Этот метод, называемый также *активной репликацией* (*active replication*), предполагает, что каждая реплика представлена процессом, способным «активно» сохранять актуальность своих данных путем выполнения операций над ними [403]. Основной выигрыш от активной репликации состоит в том, что обновления часто удается распространять с минимальными затратами на взаимодействие, определяемыми тем, что параметров, ассоциированных с той или иной операцией, относительно немного. С другой стороны, каждой реплике может потребоваться большая процессорная мощность, особенно если операции окажутся сложными.

Продвижение против извлечения

Другой вопрос проектирования состоит в том, следует ли обновления продвигать (*push*) или извлекать (*pull*). В протоколах, основанных на *продвижении* (*push-based protocols*), известных также под названием *серверных протоколов* (*server-based protocols*), обновления распространяются по другим репликам, не ожидая прихода от них запросов на получение обновлений. Подобный подход часто используется в промежутке между постоянными и иницированными сервером репликами, но может применяться также и для передачи обновлений в клиентские кэши. Серверные протоколы применяются, когда в репликах в основном следует поддерживать относительно высокий уровень непротиворечивости. Другими словами, когда реплики должны быть идентичными.

Требование высокой степени непротиворечивости связано с тем фактом, что постоянные и иницированные сервером реплики, так же как и большие совместно используемые кэши, часто разделяются множеством клиентов, которые осуществляют в основном операции чтения. Соответственно, соотношение операций чтения к операциям записи для каждой из реплик относительно высоко. В подобных случаях протоколы, основанные на продвижении, эффективнее, в том смысле, что каждое «продвинутое» обновление будет читаться одним или более пользователями. Кроме того, протоколы продвижения делают непротиворечивые данные доступными немедленно после запроса.

В противоположность им, в подходах, основанных на *извлечении* (*pull-based approach*), сервер или клиент обращается с запросом к другому серверу, требуя отправить ему все доступные к этому моменту обновления. Методы, основанные на извлечении, называемые клиентскими протоколами (*client-based protocols*), часто используются при работе с клиентским кэшем. Так, например, стандартная стратегия, применяемая при работе с кэшами в Web, предполагает предварительную проверку актуальности кэшированных данных. Когда кэш получает запрос на локально кэшированный элемент данных, он обращается к «родному» web-серверу, проверяя, не был ли этот элемент данных модифицирован после его кэширования. В том случае, если модификация имела место, модифицированные данные передаются сначала в кэш, а затем — запросившему их клиенту. Если модификации не было, клиенту сразу передаются кэшированные данные. Другими словами, клиент опрашивает сервер в поисках обновлений.

Метод извлечения эффективен при относительно небольшом отношении количества операций чтения к количеству операций записи. Такая ситуация характерна для клиентских кэшей, с которыми работает только один клиент (не разделяемых). Однако даже в том случае, когда кэш совместно используется множеством клиентов, подход, основанный на извлечении, может быть весьма эффективным — если кэшируемые элементы данных редко используются совместно. Основной недостаток этой стратегии по сравнению с продвижением состоит в том, что в случае *кэш-промаха* (*cache miss*), то есть когда данные в кэше оказываются неактуальными, время отклика увеличивается.

При сравнении подходов, основанных на извлечении и продвижении, можно отметить множество нюансов, часть которых иллюстрирует табл. 6.7. Для простоты ограничимся рассмотрением системы клиент-сервер, состоящей из одного нераспределенного сервера и нескольких клиентских процессов, каждый из которых использует собственный кэш.

Таблица 6.7. Сравнение протоколов извлечения и продвижения для системы с одним сервером и несколькими клиентами

Аспект	Продвижение	Извлечение
Информация о состоянии на сервере	Список клиентских реплик и кэшей	Ничего
Содержание сообщений	Обновление (и, возможно, сначала запрос)	Запрос и обновление
Время отклика для клиента	Немедленно (или, в случае запроса, время получения обновления)	Время получения обновления

Важным для сервера аспектом использования протоколов, основанных на продвижении, является необходимость отслеживать все клиентские кэши. Даже если не учитывать тот факт, что, как упоминалось в главе 3, серверы с фиксацией состояния обычно менее устойчивы к сбоям, отслеживание всех клиентских кэшей может создать серьезную нагрузку на сервер. Например, при подходе, основанном на продвижении, web-серверу просто придется помнить о каждом из десятков тысяч клиентских кэшей. Каждый раз при обновлении содержимого

web-страницы сервер будет вынужден отыскивать в своем списке те клиентские кэши, в которых хранится копия этой страницы, и затем передавать обновления всем найденным клиентам. Более того, если клиент в поисках свободного места на диске выбросит страницу из своего кэша, он должен будет уведомить об этом сервер, что потребует дополнительного обмена сообщениями.

Сообщения, которыми придется обмениваться клиенту и серверу, также различны. При продвижении единственный тип взаимодействия — обновления, передаваемые сервером клиенту. Если в качестве обновлений фигурируют только уведомления о несостоятельности, для получения модифицированных данных клиент будет нуждаться в дополнительном взаимодействии. При извлечении клиент должен опрашивать сервер и в случае необходимости получать модифицированные данные.

И, наконец, время отклика для клиента также будет различным. Когда сервер продвигает модифицированные данные в кэш клиента, понятно, что время отклика для клиента будет равно нулю. Если продвигаются только уведомления о несостоятельности, время отклика будет таким же, как и при извлечении (оно будет определяться временем, необходимым для получения с сервера модифицированных данных).

Недостатки и достоинства обоих подходов приводят нас к смешанной форме распространения обновлений, основанной на аренде. *Аренда (lease)* — это обещание сервера определенное время поставлять обновления клиенту. Когда срок аренды истекает, клиент запрашивает у сервера обновления и при необходимости путем извлечения получает от него модифицированные данные. Клиент может также после окончания предыдущего срока аренды запросить у сервера новый, чтобы и далее получать обновления путем продвижения их с сервера.

Аренда изначально была предложена в [175], где описан удобный механизм динамического переключения между стратегиями продвижения и извлечения. В [131] описана гибкая система аренды, позволяющая динамически адаптировать срок аренды в соответствии с различными критериями. Были выделены следующие три типа аренды (отметим, что во всех случаях до истечения срока аренды обновления продвигаются сервером на клиента):

- ◆ аренда, основанная на «возрасте» элемента данных;
- ◆ аренда, основанная на том, насколько часто клиент требует обновления копии в своем кэше;
- ◆ аренда, основанная на объеме дискового пространства, затрачиваемого сервером на сохранение состояния.

Аренда, основанная на «возрасте» элемента данных, заканчивается для различных элементов данных в соответствии со временем их последней модификации. Идея, которая лежит в основе этого решения, состоит в том, что если данные в течение длительного времени не модифицируются, значит, они не будут модифицироваться еще в течение некоторого времени. Для данных в Web это предположение кажется разумным. Задавая длительные сроки аренды немодифицируемых данных, мы можем значительно снизить число сообщений обновления по сравнению с одинаковыми сроками аренды для всех элементов данных.

Следующий критерий — как часто конкретный клиент требует обновления копии в своем кэше. При такой аренде сервер устанавливает длительные сроки аренды тем клиентам, кэш которых часто обновляется. С другой стороны, клиент, который только от случая к случаю запрашивает конкретный элемент данных, получает на этот элемент данных краткосрочную аренду. В результате такой стратегии сервер, в сущности, отслеживает только тех клиентов, которые активно пользуются его данными и нуждаются в высоком уровне их непротиворечивости.

Последний критерий — объем пространства, затрачиваемый сервером на сохранение состояния клиентов. Когда сервер понимает, что постепенно «переполняется», то сокращает время новой аренды, предоставляемой клиентам. В результате такого подхода серверу приходится отслеживать минимум клиентов — срок аренды истекает быстро. Другими словами, сервер динамически сокращает объем дискового пространства, отводимый на сохранение состояния клиентов (превращаясь в пределе в сервер без фиксации состояния). Это разгружает его, и он начинает работать эффективнее.

Целевая рассылка против групповой

С вопросом о том, как рассылать обновления — продвигая или извлекая, — тесно связан и вопрос о методе рассылки — целевой или групповой. При *целевой рассылке* (*unicasting*) сервер является частью хранилища данных и передает обновления на N других серверов путем отправки N_j отдельных сообщений, по одному на каждый сервер. В случае *групповой рассылки* (*multicasting*) за эффективную передачу одного сообщения нескольким получателям отвечает базовая сеть.

Во многих случаях оказывается дешевле использовать доступные средства групповой рассылки. Предельный случай имеет место, когда все получатели сосредоточены в пределах одной локальной сети и в наличии имеются аппаратные средства *широковещательной рассылки* (*broadcasting*). Тогда затраты на передачу одного сообщения путем широковещательной или групповой рассылки не превышают затрат на сквозную (от точки к точке) передачу этого сообщения. В такой ситуации передача обновлений путем целевой рассылки была бы значительно менее эффективной.

Групповая рассылка часто может быть эффективна в сочетании с распространением обновлений путем продвижения. В этом случае сервер, который хочет «продвинуть» обновление на несколько других серверов, просто использует одну группу групповой рассылки. В противоположность этому, при извлечении обновления копии требуется обычно только одному серверу или одному клиенту. В этом случае целевая рассылка — наиболее эффективное решение.

6.4.3. Эпидемические протоколы

Мы уже упоминали, что для многих хранилищ данных достаточно лишь причинной непротиворечивости. Другими словами, в отсутствии обновлений нам достаточно лишь гарантировать причинную идентичность копий. Распространение обновлений в причинно непротиворечивых хранилищах данных часто реализуется

при помощи класса алгоритмов, известных под названием *эпидемических протоколов* (*epidemic protocols*). Эпидемические алгоритмы не разрешают конфликтов обновления, для этого требуются индивидуальные решения. Вместо этого они ориентированы на то, чтобы распространить обновления при помощи как можно меньшего количества сообщений. Для этой цели несколько обновлений часто объединяются в одно сообщение, которым затем обмениваются два сервера. Эпидемические алгоритмы образуют основу системы Baou, о которой мы говорили ранее, различные схемы распространения обновлений, использованные в ней, описаны в [349].

Чтобы понять основной принцип этих алгоритмов, предположим, что все обновления конкретных элементов данных иницируются одним сервером. Таким образом, мы просто исключаем возможность конфликтов двойной записи. Дальнейшее изложение основано на классической статье по эпидемическим алгоритмам [124].

Модели распространения обновлений

Как следует из названия, эпидемические алгоритмы основываются на теории эпидемий, которая объясняет правила распространения инфекционных заболеваний. Если вместо инфекционных заболеваний взять реплицируемые хранилища данных, они будут распространять обновления. Исследования эпидемий в хранилищах данных преследуют также и совершенно иную цель: в то время как органы здравоохранения считают, что было бы великолепно не допустить активного распространения инфекционных заболеваний, разработчики эпидемических алгоритмов для распределенных хранилищ данных стараются «заразить» все реплики обновлениями как можно быстрее.

Используя терминологию эпидемиологов, сервер, являющийся частью распределенной системы, называется *инфицированным* (*infective*), если на нем хранится обновление, которое он готов распространять на другие серверы. Сервер, который еще не получил обновления, называется *восприимчивым* (*susceptible*). И наконец, сервер, данные которого были обновлены, но не готовый или неспособный распространять обновление дальше, называется *очищенным* (*removed*).

Одна из популярных моделей распространения — *антиэнтропия* (*anti-entropy*). Согласно этой модели сервер P случайным образом выбирает другой сервер Q , после чего обменивается с ним обновлениями. Существует три способа обмена обновлениями:

- ♦ сервер P только продвигает свои обновления на сервер Q ;
- ♦ сервер P только извлекает новые обновления с сервера Q ;
- ♦ серверы P и Q пересылают обновления друг другу (то есть метод «тяги-толкая»).

В отношении скорости распространения обновлений плохим выбором кажется только продвижение обновлений. Интуитивно это можно объяснить следующим образом: Во-первых, отметим, что при чистом продвижении обновления могут распространяться только инфицированными серверами. Однако если инфицировано множество серверов, вероятность того, что каждый из них случайно

выберет восприимчивый сервер, относительно мала (скорее они будут натекать друг на друга). Соответственно, имеется шанс на то, что какой-то из серверов в течение длительного времени будет оставаться восприимчивым просто потому, что он не был выбран инфицированным сервером.

В противоположность этому, извлечение обновлений при большом количестве инфицированных серверов будет работать гораздо лучше. В этом случае распространение обновлений, в сущности, возлагается на восприимчивые серверы. Шансы значительно выше, ведь теперь сам восприимчивый сервер должен связаться с инфицированным сервером, чтобы затем получить от него обновления и самому стать инфицированным.

Можно показать, что если изначально инфицирован лишь один сервер, при использовании любой из форм антиэнтропии обновления потенциально могут распространиться на все серверы. Однако, чтобы убедиться, что обновления распространяются быстро, имеет смысл задействовать дополнительный механизм, при помощи которого более или менее инфицированными немедленно станут как минимум несколько серверов. Обычно в таком случае помогает прямое продвижение обновления на несколько серверов.

Один из специальных вариантов подобного подхода именуется *распространением слухов* (*rumor spreading*), или просто *болтовней* (*gossiping*). Он работает следующим образом. Если на сервере P имеется обновленный элемент данных x , значит, он связывается с другим произвольным сервером Q и пытается продвинуть обновление на Q . Однако возможно, что Q уже получил это обновление с другого сервера. В этом случае P может потерять интерес к дальнейшему распространению обновления, скажем, с вероятностью $1/k$. Другими словами, он становится очищенным.

Болтовня полностью соответствует реальной жизни. Если Боб узнает какую-то «жареную» новость, о которой можно посплетничать, он, скорее всего, позвонит своей подружке Алисе и все ей расскажет. Алиса, как и Боб, будет заинтересована в том, чтобы нести эту новость дальше. Однако она может быть разочарована, если, позвонив приятелю, скажем Чаку, услышит от него, что он уже знает эту новость. Имеется шанс на то, что она перестанет обзванивать своих друзей. Зачем ей это, если все и так знают ее новость?

Болтовня может быть действительно хорошим способом быстрого распространения обновлений. Однако оно не гарантирует, что будут обновлены все серверы [124]. Можно показать, что в случае хранилищ данных, содержащих множество серверов, доля s серверов, которые не получают обновления, то есть останутся восприимчивыми, рассчитывается следующим образом:

$$s = e^{-(k+1)(1-s)}.$$

Например, при $k = 3$ значение s составляет менее 0,02. Это означает, что восприимчивыми останутся менее 2 % серверов. Тем не менее следует принять специальные меры, чтобы гарантировать, что эти серверы также будут обновлены. Комбинирование антиэнтропии с механизмом распространения слухов — одна из таких уловок

Одно из основных преимуществ эпидемических алгоритмов — их масштабируемость, вытекающая из того факта, что синхронизировать друг с другом при-

ходится относительно мало процессов по сравнению с другими методами распространения. В [266] показано, что для глобальных систем для получения лучших результатов имеет смысл учитывать особенности топологии. При этом подходе серверы, соединенные с малым количеством других серверов, контактируют с относительно большей вероятностью. Предполагается, что эти серверы формируют мостик к другим, удаленным частям системы, а поэтому должны связываться друг с другом как можно чаще.

Удаление данных

Эпидемические алгоритмы хорошо подходят для распространения обновлений в причинно непротиворечивых хранилищах данных. Однако у них имеется довольно странный побочный эффект: затрудненное распространение удалений элементов данных. Сущность проблемы заключается в том факте, что удаление элемента данных уничтожает всю информацию об этом элементе. Соответственно, если элемент данных просто изъять с сервера, сервер может, получив старые копии элемента данных, интерпретировать их как обновления, включающие в себя новый элемент.

Хитрость состоит в том, чтобы зафиксировать факт удаления элемента данных в виде очередного обновления и сохранить о нем запись. При этом старые копии будут считаться не новыми элементами, а просто старыми версиями, которые были обновлены при помощи операции удаления. Запись о произведенном удалении выполняется путем рассылки *свидетельств о смерти* (*death certificates*).

Разумеется, остается проблема свидетельств о смерти, состоящая в том, что и они в конечном итоге также должны быть удалены. В противном случае каждый сервер постепенно накопит гигантскую никому не нужную локальную базу данных с исторической информацией об удаленных элементах данных. В [124] предложено использовать так называемые спящие свидетельства о смерти. Каждое свидетельство о смерти получает отметку о времени создания. Если предположить, что обновления распространяются на все серверы за известное конечное время, то по истечении максимального времени распространения свидетельство о смерти можно удалять.

Однако для получения твердой гарантии того, что удаление действительно распространилось на все серверы, только очень немного серверов поддерживают спящие свидетельства о смерти, никогда не передавая их дальше. Предположим, что сервер P имеет подобное свидетельство на элемент данных x . Если из-за какой-то случайности на P придет устаревшее обновление для x , сервер P отреагирует на это событие просто повторной рассылкой свидетельства о смерти этого элемента данных.

6.5. Протоколы непротиворечивости

До сих пор мы в основном обсуждали различные модели непротиворечивости и общие вопросы проектирования протоколов непротиворечивости. В этом разделе мы сосредоточимся на конкретных реализациях моделей непротиворечивости

и рассмотрим несколько реальных протоколов непротиворечивости. *Протокол непротиворечивости (consistency protocol)* описывает реализацию конкретной модели непротиворечивости. Вообще говоря, очень важны и широко используются модели непротиворечивости, в которых операции обладают глобальной сериализуемостью. Имеются в виду такие модели, как последовательная непротиворечивость, слабая непротиворечивость с переменными синхронизации, а также атомарные транзакции. В данном разделе мы обсудим различные способы реализации этих моделей непротиворечивости.

Используя ранний подход, описанный в [440], протоколы последовательной непротиворечивости можно классифицировать по тому, существует ли первичная копия данных, которой должны передаваться все операции записи. Если такой первичной копии не существует, операции записи могут быть инициированы любой из реплик.

6.5.1. Протоколы на базе первичной копии

В протоколах на базе первичной копии каждый элемент данных x , находящийся в хранилище данных, имеет ассоциированный с ним первичный элемент данных, отвечающий за координацию всех операций записи в элемент x . Отличить первичный элемент данных можно по тому, что он постоянно находится на удаленном сервере, или по тому, что после переноса первичного элемента в тот процесс, в котором была инициирована операция записи, эта запись может выполняться локально.

Протоколы удаленной записи

Простейший протокол на базе первичной копии — тот, в котором все операции чтения и записи передаются на удаленный сервер (единственный). В результате данные вообще не реплицируются, они просто размещаются на единственном сервере, с которого их невозможно куда-либо переместить. Эта модель традиционно используется в системах клиент-сервер, причем сервер может быть распределенным. Работу этого протокола иллюстрирует рис. 6.20. Здесь используются следующие обозначения:

- ◆ $W1$ — запрос на запись;
- ◆ $W2$ — пересылка запроса на сервер;
- ◆ $W3$ — подтверждение выполнения записи;
- ◆ $W4$ — подтверждение выполнения записи;
- ◆ $R1$ — запрос на чтение;
- ◆ $R2$ — пересылка запроса на сервер;
- ◆ $R3$ — возвращение ответа;
- ◆ $R4$ — возвращение ответа.

Более интересны с точки зрения непротиворечивости протоколы, которые позволяют процессам выполнять операции чтения с локальной копией, но операции записи должны пересылать на первичную (фиксированную) копию. Подоб-

ные схемы часто называют [76] *протоколами первичного архивирования (primary-backup protocols)*. Как работает протокол первичного архивирования, показано на рис. 6.21. Здесь используются следующие обозначения:

- ♦ $W1$ — запрос на запись;
- ♦ $W2$ — пересылка запроса на первичный сервер;
- ♦ $W3$ — сигнал на обновление резервных копий;
- ♦ $W4$ — подтверждение обновления;
- ♦ $W5$ — подтверждение выполнения записи;
- ♦ $R1$ — запрос на чтение;
- ♦ $R2$ — ответ для чтения.

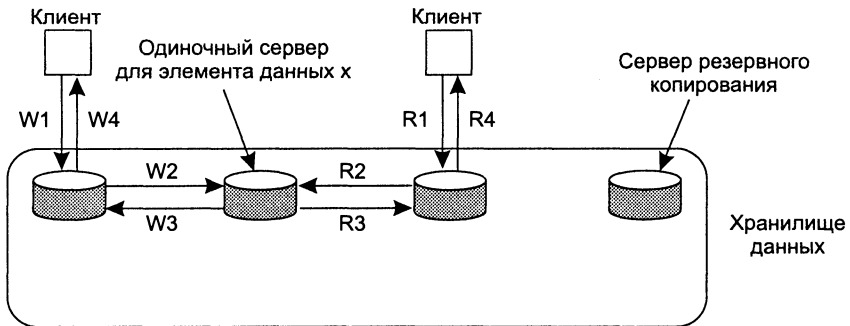


Рис. 6.20. Протокол удаленной записи на базе первичной копии с фиксированным сервером, на который пересылаются все операции записи и чтения

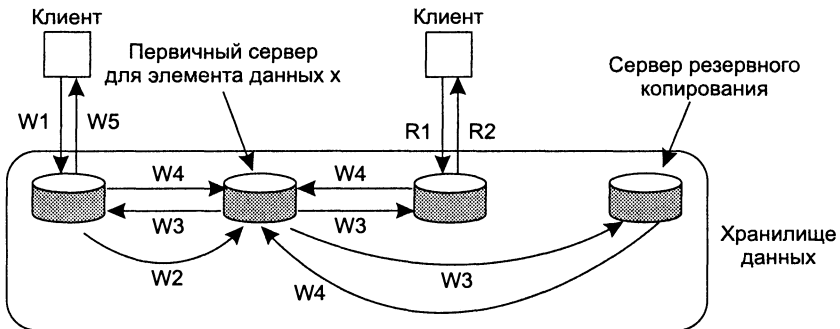


Рис. 6.21. Первичное архивирование данных

Процесс, желающий выполнить операцию записи в элемент данных x , пересылает эту операцию на первичный сервер. Первичный сервер осуществляет обновление своей локальной копии x , после чего пересылает это обновление серверам резервного копирования. Каждый из этих серверов обновляет свои данные и посылает обратно первичному серверу свое подтверждение. После обновления локальных копий всех серверов резервного копирования первичный сервер посылает подтверждение процессу, инициировавшему операцию.

Потенциальные проблемы производительности этой схемы заключаются в том, что в процессе, инициировавшем обновление, может возникнуть относительно длительная задержка между инициированием процесса обновления и его продолжением. То есть обновление реализовано в виде блокирующей операции. Альтернативой является использование неблокирующих методов. Как только первичный сервер вносит обновление в свою локальную копию x , он возвращает подтверждение. После этого он дает сигнал серверам резервного копирования внести необходимые обновления в свои данные. Неблокирующий протокол первичного архивирования обсуждается в [77].

Основная проблема неблокирующего первичного архивирования связана с защитой от сбоев. В схеме с блокировкой процесс клиента осведомлен о том, что операция обновления архивируется на несколько других серверов. В случае схемы без блокировки это не так. Мы вернемся к вопросам защиты от сбоев в следующей главе.

Протоколы первичного архивирования представляют собой простую реализацию последовательной непротиворечивости, в которой первичный сервер упорядочивает все входящие операции записи. Очевидно, что все процессы наблюдают все операции записи в едином порядке, и неважно, какой из серверов резервного копирования они используют для выполнения операций чтения. Кроме того, в блокирующих протоколах процессы видят также и результат последней операции записи (отметим, что для неблокирующих протоколов, если не принимать специальных мер, это не гарантировано).

Протоколы локальной записи

Существует два типа протоколов локальной записи на базе первичной копии. Первый предполагает наличие единственной копии каждого элемента данных x . Другими словами, реплики отсутствуют. Когда процесс собирается выполнить операцию с некоторым элементом данных, единственная копия этого элемента передается в процесс, после чего производится операция. Этот протокол в основном предназначен для полностью распределенных нереплицируемых версий. Непротиворечивость сохраняется просто потому, что существует всего одна копия каждого элемента данных. Работу этого протокола иллюстрирует рис. 6.22. Здесь цифрами показана следующая последовательность действий.

1. Отправка запроса на чтение или запись.
2. Передача запроса на текущий сервер для элемента данных x .
3. Перемещение элемента данных x на сервер клиента.
4. Возвращение результата операции на сервер клиента.

Одна из основных проблем с таким методом полного переноса — отслеживание текущего положения каждого элемента данных. Как упоминалось в главе 4, в решениях для локальных сетей можно воспользоваться встроенными средствами широковещательной рассылки. Альтернативные решения — передача указателей и подходы с использованием базы. Такие решения применяются в распределенных системах с разделяемой памятью [264]. Однако при работе с крупномасштабными и глобальными хранилищами данных необходимы другие

механизмы, например иерархические службы локализации, одна из которых упоминалась в главе 4.

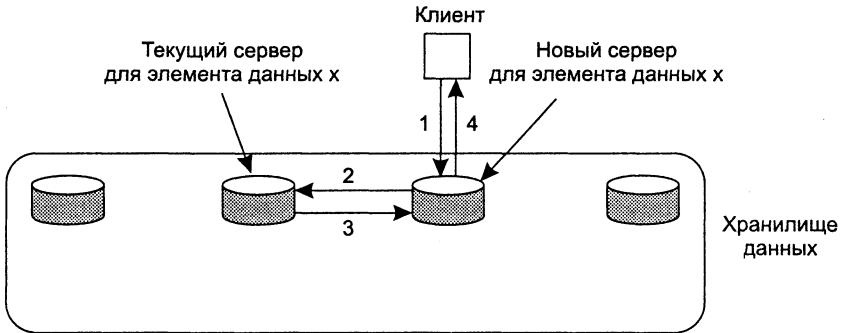


Рис. 6.22. Протокол локальной записи на базе первичной копии, в котором единственная копия данных перемещается между процессами

Одним из вариантов описанного протокола локальной записи является протокол первичного архивирования, в котором первичная копия перемещается между процессами, собирающимися выполнить операцию записи. Как и ранее, если процесс хочет изменить элемент данных x , он находит его первичную копию, после чего перемещает его туда, где находится сам, как это показано на рис. 6.23. Здесь используются следующие обозначения:

- ♦ $W1$ — запрос на запись;
- ♦ $W2$ — перемещение элемента данных x на новый первичный сервер;
- ♦ $W3$ — подтверждение завершения записи;
- ♦ $W4$ — сигнал на обновление резервных копий;
- ♦ $W5$ — подтверждение обновления;
- ♦ $R1$ — запрос на чтение;
- ♦ $R2$ — ответ для чтения.

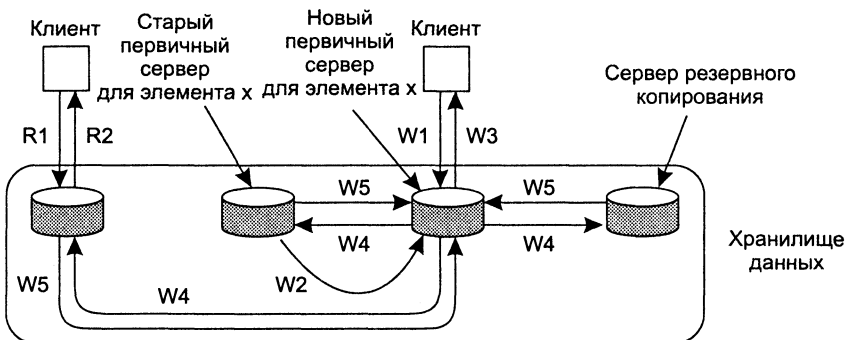


Рис. 6.23. Протокол первичного архивирования, в котором первичная копия перемещается между процессами, которым необходимо изменить данные

Основное преимущество этого подхода состоит в том, что многочисленные последовательные операции записи производятся локально, в то время как процессы чтения по-прежнему могут оперировать своими локальными копиями. Однако это улучшение может быть достигнуто только в том случае, если после обновления первичной копии остальные обновления будут распространяться с использованием описанного ранее неблокирующего протокола. Такой протокол применяется во многих распределенных системах с разделяемой памятью.

Упомянутый протокол первичного архивирования с локальной записью также можно использовать для мобильных компьютеров, способных работать без сети. Перед отсоединением мобильный компьютер становится первичным сервером для всех элементов данных, которые он собирается изменить. Будучи отключен от сети, он локально осуществляет все операции обновления данных, все остальные процессы могут в это время производить чтение, но не изменять данные. Позднее, когда он вновь подсоединится к сети, обновления распространятся с первичного сервера на серверы резервного копирования, вновь приводя хранилище данных в непротиворечивое состояние. Мы вернемся к работе в отключенном от сети состоянии в главе 10, когда будем рассматривать распределенные файловые системы.

6.5.2. Протоколы реплицируемой записи

В протоколах реплицируемой записи операции записи могут осуществляться на многих репликах, а не на одной, как в случае протоколов на базе первичной копии. Их можно разделить на протоколы с активными репликами, в которых операции передаются на все реплики, и протоколы непротиворечивости, основанные на большинстве голосов при голосовании.

Активная репликация

При активной репликации на каждой из реплик выполняется ассоциированный с ней процесс, который осуществляет операции обновления данных. В противоположность другим протоколам, обновления обычно распространяются посредством операции записи, осуществляющей обновление данных. Другими словами, каждой реплике посылается операция записи. Однако, как было показано ранее, можно также посылать и обновления.

С активной репликацией связана одна потенциальная проблема. Она состоит в том, что операции должны осуществляться в одном и том же порядке повсюду. Соответственно, нам необходим полностью упорядоченный механизм групповой рассылки. Подобную групповую рассылку, как показано в предыдущей главе, можно реализовать на базе отметок времени Лампорта. К сожалению, подход с использованием отметок времени Лампорта плохо масштабируется в больших распределенных системах. В качестве альтернативы — полного упорядочения можно достигнуть с помощью центрального координатора, называемого также *секвенсором* (*sequencer*). Один из способов — просто передавать каждую операцию секвенсору, который присвоит ей уникальный последовательный номер и разошлет по всем репликам. Операции будут выполняться в соответствии с их

номерами. Понятно, что такая реализация полностью упорядоченной групповой рассылки очень напоминает протоколы непротиворечивости на базе первичного сервера.

Отметим, что использование секвенсора не решает проблем масштабируемости. На самом деле, если необходима полностью упорядоченная групповая рассылка, ее можно реализовать путем сочетания симметричной групповой рассылки с использованием отметок времени Лампорта и секвенсоров. Подобное решение описано в [384].

Другая проблема, которую следует решить, — что делать с реплицированными обращениями. Рассмотрим объект *A*, обращающийся к другому объекту *B*, как показано на рис. 6.24. Объект *B* при этом обращается к третьему объекту *C*. Если объект *B* реплицируется, каждая реплика *B* будет в принципе обращаться к объекту *C* независимо от других. Проблема в том, что объект *C* получит множество обращений вместо одного. Если вызываемый метод *C* призван выполнить перевод \$100 000 со счета на счет, очевидно, рано или поздно кто-нибудь пожалуется.

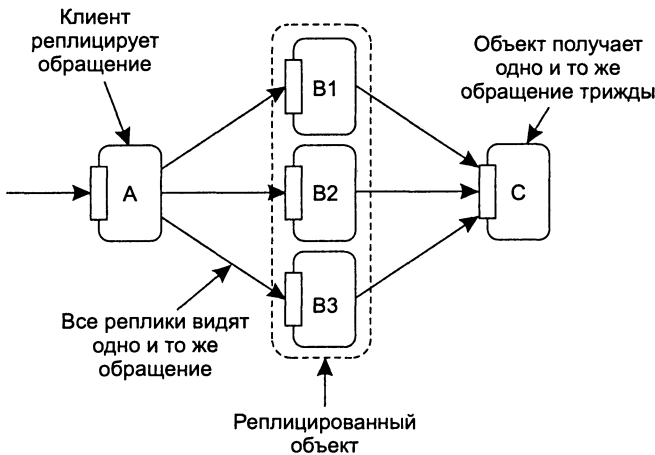


Рис. 6.24. Проблема репликации обращений

Репликация обращений — проблема не только объектов, но и некоторых систем клиент-сервер. Если реплицируемый сервер зависит от других серверов, к которым он отправляет запросы, мы оказываемся в ситуации, когда запросы также будут реплицированы, что может привести к нежелательным эффектам.

Немного существует общих решений проблемы репликации обращений. Одно из решений, описанное в [288], не зависит от правил репликации, то есть от деталей поддержания непротиворечивости реплик. Другое решение является частью системы GARF [158]. Суть его состоит в создании осведомленного о репликах коммуникационного уровня, поверх которого выполняются реплицируемые объекты. Когда реплицированный объект *B* обращается к другому реплицированному объекту *C*, первым делом все реплики объекта *B* присваивают этому обращению один и тот же уникальный идентификатор. После этого координатор реплик *B* передает обращение всем репликам объекта *C*, в то время как осталь-

ные реплики *B* воздерживаются от посылки своих копий обращения, как это показано на рис. 6.25, *а*. В результате каждой реплике *C* посылается только одно обращение.

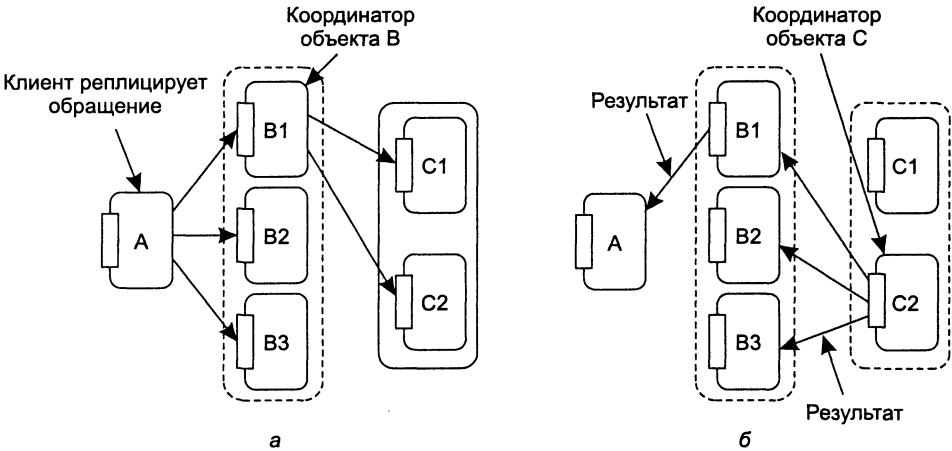


Рис. 6.25. Передача обращения от реплицированного объекта другому реплицированному объекту (а). Возвращение ответа одного реплицированного объекта другому (б)

Такой же механизм используется, чтобы гарантировать получение репликами *B* только одного ответного сообщения. Эта ситуация представлена на рис. 6.25, *б*. Координатор реплик *C* уведомляется, что необходимо разобраться с реплицированными ответными сообщениями, созданными каждой из реплик *C*. Хотя эти сообщения создаются каждой из реплик, только координатор передает ответ репликам объекта *B*, в то время как остальные реплики *C* воздерживаются от посылки своих копий ответного сообщения.

После того как реплика *B* в ответ на запрос, который она передавала объекту *C* или оставляла у себя, получает сообщение с результатом, она, если является координатором, возвращает результат реальному объекту *A*.

В сущности, описанная здесь схема основана на использовании групповой рассылки, но одинаковые сообщения разными репликами не рассылаются. Таким образом, это схема с активным отправителем. Можно сделать и так, чтобы принимающая реплика проверяла многочисленные копии входящих сообщений, относящиеся к одному и тому же обращению, и передавала связанному с ней объекту только одну копию. Детали этой схемы мы оставляем читателю в качестве самостоятельного упражнения.

Протоколы кворума

Другой подход к поддержке реплицированных операций записи основан на использовании *голосования* (*voting*). Изначально он был представлен в [457] и обобщен в [167]. Основная идея состоит в том, чтобы потребовать от клиента до начала чтения или записи данных запрашивать и получать разрешение на это у серверов.

В качестве простого примера работы этого алгоритма рассмотрим распределенную файловую систему и предположим, что файл реплицирован на N серверов. Мы можем создать правило, согласно которому для обновления файла клиент должен сначала связаться как минимум с половиной серверов плюс еще одним (большинством) и попросить их согласия на обновление. Как только они согласятся, файл изменится и с новым файлом будет ассоциирован новый номер версии. Номер версии используется для идентификации версии файла и является одинаковым для всех обновленных файлов.

Для чтения реплицированного файла клиент также должен связаться с половиной серверов плюс еще одним и запросить у них номера версий, ассоциированные с файлом. Если все номера версий согласованы, среди них должна быть и последняя версия, поскольку попытка произвести обновления только на серверах, которым не был направлен запрос, невозможна, — они не составляют большинства.

Так, например, если имеется пять серверов и клиент определил, что три из них имеют файл версии 8, то не может быть, чтобы на остальных двух хранился файл версии 9, поскольку любое обновление версии 8 на версию 9 требует разрешения как минимум трех, а не двух серверов.

Реальная схема более обобщенная, чем мы только что описали. Согласно этой схеме для чтения файла, имеющего N реплик, клиент должен собрать *кворум чтения* (*read quorum*) — произвольный набор любых N_R или более серверов. Соответственно, для модификации файла требуется *кворум записи* (*write quorum*), образованный как минимум N_W серверами. Значения N_R и N_W должны удовлетворять следующим двум условиям:

$$\begin{aligned} N_R + N_W &> N, \\ N_W &> N/2. \end{aligned}$$

Первое ограничение используется для предотвращения конфликтов чтения-записи, а второе — для предотвращения конфликтов двойной записи. Только после того как соответствующее число серверов даст согласие на операцию, файл можно будет прочитать или изменить.

Чтобы рассмотреть, как работает алгоритм, взглянем на рис. 6.26, а, на котором $N_R = 3$, а $N_W = 10$. Представим, что последний кворум записи состоял из 10 серверов, с C по L . Все они получили новую версию файла и новый номер версии. Все последующие кворумы чтения из трех серверов будут содержать как минимум одного члена этого набора. Когда клиент будет просматривать номера версий, он обнаружит последнюю версию и воспользуется ей.

Рисунок 6.26, б иллюстрирует конфликт двойной записи, который может произойти, поскольку $N_W < N/2$. Так, если один из клиентов выберет набор записи $\{A, B, C, E, F, G\}$, а другой — $\{D, H, I, J, K, L\}$, понятно, что мы столкнемся с проблемами, поскольку оба обновления будут приняты, несмотря на то, что между ними явно имеется конфликт.

Ситуация, представленная на рис. 6.26, в, особенно интересна, потому что значение N_R в этом случае равно единице, что делает чтение реплицируемого файла значительно проще. Достаточно найти одну копию и пользоваться ей. Однако все имеет свою цену. За это мы платим тем, что для записи обновлений

нам необходимо согласие всех копий. Такая схема обычно называется *ROWA* (*Read-One, Write-All* — читаем раз, пишем все). Имеются различные варианты протоколов репликации по кворуму. Хороший обзор можно найти в [213].

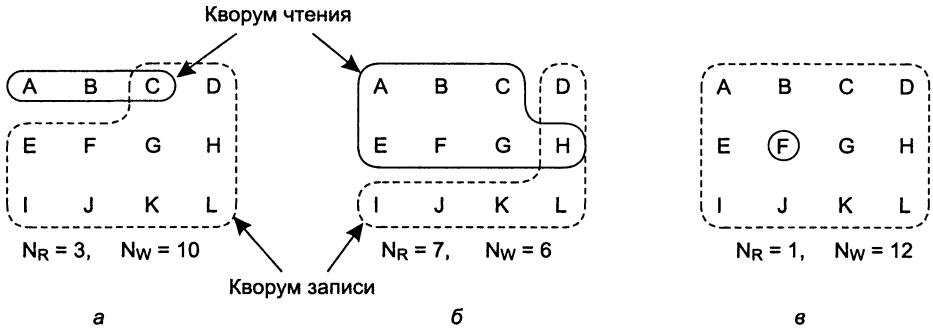


Рис. 6.26. Три примера алгоритма голосования. Правильный выбор наборов чтения и записи (а). Выбор, который может привести к конфликтам двойной записи (б). Правильный выбор, известный под названием ROWA (в)

6.5.3. Протоколы согласования кэшей

Кэши представляют собой особый случай репликации в том смысле, что они обычно находятся под управлением клиентов, а не серверов. Однако протоколы согласования кэшей, которые отвечают за то, чтобы содержимое кэша соответствовало серверным репликам, в принципе не слишком отличаются от ранее обсуждавшихся протоколов непротиворечивости.

Разработке и реализации кэшей, особенно в контексте мультипроцессорных систем с совместно используемой памятью, было посвящено множество работ. Многие решения базируются на аппаратной поддержке, например, на предположении о возможности эффективной широковещательной рассылки или контроля. В контексте распределенных систем промежуточного уровня, построенных на базе операционных систем общего назначения, более интересны программные реализации кэшей. В этом случае для классификации протоколов кэширования используют два различных критерия [265, 296, 452].

В первую очередь, решения для кэшей могут различаться по их *стратегии обнаружения согласованности* (*coherence detection strategy*), то есть по тому, когда реально обнаруживается противоречивость. В статических решениях, как предполагается, необходимый анализ перед исполнением проводит компилятор, определяя, какие данные из-за кэширования могут реально стать противоречивыми. Компилятор просто вставляет инструкции, позволяющие избежать противоречивости. В распределенных системах, изучаемых в этой книге, обычно применяются динамические решения. В них рассогласования обнаруживаются во время исполнения. Проверка может, например, производиться сервером, определяющим, модифицировались ли данные после кэширования.

В случае распределенных баз данных протоколы на базе динамического обнаружения могут быть далее классифицированы по тому, когда именно в ходе

транзакции происходит обнаружение противоречивости. В [151] выделяют три варианта. Первый, когда в ходе транзакции происходит доступ к элементу данных и клиент нуждается в подтверждении непротиворечивости этого элемента данных с той версией, которая хранится на сервере (возможно, реплицируемом). Транзакция не может работать с кэшированной версией, пока непротиворечивость последней не будет подтверждена.

Второй, оптимистический подход состоит в том, чтобы разрешить проведение транзакции в ходе проверки. В этом случае, начиная транзакцию, мы считаем, что кэшированные данные соответствуют действительности. Если позже окажется, что мы ошибались, транзакция будет прервана.

Третий подход заключается в том, чтобы проверять кэшированные данные только перед самым подтверждением транзакции. Этот подход сравним со схемой оптимистического управления параллельным выполнением, которая рассматривалась в предыдущей главе. В результате транзакция просто начинает работать с кэшированными данными, надеясь на лучшее. После завершения всей обработки полученные данные проверяются на непротиворечивость. Если использовались неактуальные данные, транзакция прерывается.

Другой аспект построения протоколов согласования кэша касается *стратегии установления согласованности (coherence enforcement strategy)*, которая определяет, как нужно согласовывать кэш с копиями, хранящимися на серверах. Самое простое решение — вообще запретить кэширование общих данных, а хранить такие данные только на серверах, поддерживающих их непротиворечивость с помощью одного из ранее описанных протоколов первичной копии или реплицируемой записи, а клиентам разрешить кэширование только их внутренних данных. Очевидно, что такое решение дает лишь незначительный выигрыш в производительности.

Если разрешить кэширование общих данных, можно предложить два подхода, позволяющих добиться согласованности кэшей. В первом из них сервер при модификации элемента данных рассылает всем кэшам сообщения о рассогласовании. Второй состоит в простом распространении обновления. В большей части систем кэширования используют одну из этих двух схем. Иногда в базах данных с архитектурой клиент-сервер поддерживается автоматический выбор между рассылкой сообщений о рассогласовании и распространением обновлений [151].

И, наконец, нам следует рассмотреть, что происходит, когда процесс модифицирует кэшированные данные. Для кэшей, ориентированных только на чтение, операции обновления данных могут выполняться исключительно серверами, что требует использования тех или иных распределенных протоколов, обеспечивающих распространение обновлений на кэши. Нередко для этого применяют описанный выше подход, основанный на извлечении данных. В этом случае, когда клиент обнаруживает, что его кэш устарел, он запрашивает на сервере обновление.

Альтернативный подход состоит в том, чтобы позволить клиенту непосредственно модифицировать кэшированные данные и передать обновления на серверы. Такой подход требует *кэшей сквозной записи (write-through caches)*, часто применяемых в распределенных файловых системах. В действительности кэширова-

ние со сквозной записью похоже на протокол локальной записи на базе первичной копии, в котором клиентский кэш выступает в роли временного первичного сервера. Для гарантии непротиворечивости (последовательной) необходимо, чтобы клиент имел исключительные права на запись, в противном случае возможны конфликты двойной записи.

Кэши сквозной записи потенциально дают максимальный выигрыш в производительности по сравнению со всеми остальными схемами, поскольку все операции выполняются локально. Еще одно усовершенствование, которое мы можем сделать, — притормозить распространение обновлений, допуская совершение в промежутках между сеансами связи с сервером нескольких операций записи. Это приведет нас к так называемому *кэшу обратной записи* (*write-back cache*), который также широко применяется в распределенных файловых системах.

6.6. Примеры

Непротиворечивость и репликация находят применение во множестве распределенных систем. В этом разделе мы поближе рассмотрим два очень разных примера реализации непротиворечивости и репликации. Сначала мы обсудим распределенную объектно-ориентированную систему под названием Orca, которая объединяет модель строгой непротиворечивости с физически распределенными объектами, пользователь же, работая с ней, остается в полном неведении о том факте, что объекты распределены и реплицируются. Таким образом, Orca обладает высоким уровнем прозрачности.

Нашим вторым примером станет распределенная база данных, реплики которой обладают причинной непротиворечивостью. Этот пример показателен по множеству причин. Во-первых, он показывает, что распространение обновлений и непротиворечивость действительно можно реализовать более или менее независимо друг от друга. Во-вторых, он иллюстрирует эффективную реализацию причинной непротиворечивости при помощи векторных отметок времени. В-третьих, использование векторных отметок времени в этом примере вплотную подводит нас к тем аспектам реализации виртуальных синхронных систем, которые мы будем рассматривать в следующей главе.

6.6.1. Orca

Orca — это программная система, которая позволяет процессам с различных машин иметь управляемый доступ к совместно используемой распределенной памяти, содержащей защищенные объекты [28, 30]. Система Orca состоит из языка программирования Orca, компилятора и исполняющей системы, которая собственно и управляет распределенными объектами в ходе выполнения. Хотя язык, компилятор и исполняющая система разрабатывались для совместной работы, исполняющая система независима от компилятора и может использоваться с другими языками [51, 53]. После знакомства с языком Orca мы рассмотрим, как исполняющая система реализует совместно используемые объекты, которые могут быть физически разнесены по множеству машин.

Язык программирования Orca

В некоторых отношениях Orca — это традиционный язык программирования, последовательные инструкции которого несколько напоминают нам язык Modula-2. Однако это язык с защитой типов, без указателей и без псевдонимов. Границы массивов проверяются во время исполнения программы (кроме той проверки, которая может производиться в ходе компиляции). Эти и другие подобные черты позволяют не допускать или обнаруживать множество часто встречающихся ошибок программирования, таких как неопределенные области в памяти. Особенности языка были тщательно подобраны, чтобы упростить различные варианты оптимизации.

Две черты Orca, важные для распределенного программирования, — это совместно используемые объекты данных и инструкция `fork`. Объекты инкапсулируют внутренние структуры данных и пользовательские процедуры, называемые *операциями* (*operations*). Операции предназначены для работы с этими структурами данных. Объекты пассивны, то есть не содержат потоков выполнения, которым могли бы посылаться сообщения. Напротив, процессы получают доступ к внутренним данным объектов путем вызова их операций. Объекты не наследуют свойств других объектов, поэтому Orca считается языком на базе объектов, но не объектно-ориентированным языком.

Каждая операция образует список пар (страж, блок инструкций). *Страж* (*guard*) — это логическое выражение без каких-либо сторонних эффектов. Пустой страж означает то же самое, что и значение `true`. При вызове операции все его стражи вычисляются в определенном порядке. Если все они равны значению `false`, вызывающий процесс приостанавливается до того момента, пока один из стражей не получит значение `true`. Когда находится страж, вычисление которого дает результат `true`, вычисляется следующий за ним блок инструкций. В листинге 6.2 показан объект `stack` (стек) с двумя операциями, `push` и `pop`.

Листинг 6.2. Упрощенный объект `stack` в Orca с внутренними данными и двумя операциями

```
OBJECT IMPLEMENTATION stack;
  top: integer; # переменная, идентифицирующая вершину стека
  stack: ARRAY [integer 0..N-1] OF integer; # память для стека

# Эта функция ничего не возвращает
OPERATION push(item: integer);
BEGIN
  GUARD top < N DO
    stack[top] := item; # поместить элемент в стек
    top := top + 1;     # инкремент указателя стека
  OD;
END;

# Эта функция возвращает целое
OPERATION pop(): integer;
BEGIN
  GUARD top > 0 DO      # приостановиться, если стек пуст
    top := top - 1;     # декремент указателя стека
```

```

    RETURN stack[top]: # вернуть элемент с вершины стека
  OD;
END;
BEGIN
  top := 0;           # инициализация
END;
```

После определения объекта `stack` могут быть созданы переменные этого типа:

```
s, t: stack;
```

В данном случае создаются два объекта стека, и в каждом из них переменная `top` инициализируется нулем. Целая переменная `k` может быть помещена в стек `s` с помощью следующей инструкции:

```
s$push(k);
```

Операция изъятия из стека имеет стража, который при попытке достать переменную из пустого стека приостанавливает работу вызвавшего процедуру процесса до тех пор, пока другой процесс не поместит в стек какое-нибудь целое число.

Для создания нового процесса на заданном пользователем процессоре в Orca предусмотрена инструкция `fork`. Новый процесс запускает процедуру, имя которой указано в инструкции `fork`. В новый процесс можно передавать параметры, включая объекты. Именно таким образом объекты распределяются по машинам. Например, следующая инструкция создает по одному новому процессу `foobar` на каждой из машин, от 1 по `n`:

```
FOR i IN 1 .. n DO FORK foobar(s) ON i: OD;
```

Поскольку эти `n` процессов (включая родительский) выполняются параллельно, все они могут добавлять элементы в общий стек `s` и извлекать их оттуда, как если бы они работали на мультипроцессорной системе с общей памятью. Поддерживать иллюзию совместно используемой памяти, хотя на самом деле ее не существует, — это работа исполняющей системы.

Операции над общими объектами атомарны и последовательно непротиворечивы. Система гарантирует, что если несколько процессов почти одновременно совершают операции с одним и тем же совместно используемым объектом, сетевой эффект проявится в том, что операции будут выглядеть происходящими строго последовательно (в некотором неопределенном порядке), при этом ни одна операция не начнется до завершения предыдущей.

Операции представляются для всех процессов в одном и тот же порядке. Так, например, предположим, что мы добавим в объект `stack` из листинга 6.2 новую операцию, `peek`, проверяющую элемент на вершине стека. Тогда в случае, если два независимых процесса одновременно поместят в стек значения 3 и 4 и все процессы позже используют операцию `peek` для исследования вершины стека, система гарантирует, что каждый из процессов увидит там либо 3, либо 4. Ситуация, когда часть процессов видит 3, а остальные — 4, в мультипроцессорной системе с совместно используемой памятью невозможна. Точно так же невозможна она и в Orca. Если поддерживается только одна копия стека, этого эффекта до-

биться легко, но если стек реплицируется на всех машинах, необходимо, как показано ниже, затратить чуть больше усилий.

Огса объединяет синхронизацию и совместно используемые данные примерно так же, как при поэлементной непротиворечивости. Первый вид синхронизации — это синхронизация взаимного исключения, позволяющая не допустить одновременного выполнения двух процессов в одной критической области. В Огса каждая из операций разделяемого объекта весьма напоминает критическую область, поскольку система гарантирует, что итоговый результат будет таким же самым, что и при поочередном (по одной за раз) обработке всех критических областей. В этом отношении объект Огса похож на распределенную форму монитора, обсуждавшегося в главе 1.

Другим видом синхронизации является условная синхронизация, при использовании которой процесс блокируется и ожидает выполнения некоторых условий. В Огса условную синхронизацию обеспечивают стражи. В нашем примере на листинге 6.2 процесс, пытающийся извлечь элемент из пустого списка, будет приостановлен до того момента, когда стек перестанет быть пустым.

Управление совместно используемыми объектами в Огса

Управление объектами в Огса возложено на исполняющую систему. Оно поддерживается как в сетях с широковещательной (и групповой) рассылкой, так и в сетях со сквозной (от точки к точке) передачей. Исполняющая система выполняет репликацию объектов, их пересылку, поддержание непротиворечивости и обращения к операциям.

Каждый объект может быть в одном состоянии из двух — либо в виде единственной копии, либо в виде реплики. Объект в состоянии единственной копии имеется только на одной машине. Реплицированный объект представлен на всех машинах, на которых имеются использующие его процессы. Не требуется, чтобы все объекты находились в одном и том же состоянии, так что некоторые объекты, задействованные процессом, могут быть реплицированы, в то время как остальные представлять собой единственные копии. Объекты в ходе выполнения могут менять свое состояние, переходя от единственной копии к реплицированному состоянию и обратно, что придает им большую гибкость.

Большое преимущество репликации объектов на всех машинах состоит в том, что операции чтения могут выполняться локально, без поглощения сетевого трафика и без задержек. Если объект не реплицирован, ему должны посылаться все операции, а вызывающие его процессы — блокироваться в ожидании ответа. Второе преимущество репликации состоит в том, что она повышает степень параллельности: одновременно может происходить несколько операций чтения. При использовании единственной копии одновременно может производиться только одна операция, что замедляет выполнение. Принципиальный недостаток репликации — затраты на поддержание непротиворечивости всех копий.

Когда программа выполняет операцию над объектом, компилятор вызывает процедуру `invoke_or` исполняющей системы, задающую объект, операцию, параметры и флаг, уведомляющий, будет ли объект при этом модифицироваться (это называется записью) или нет (это называется чтением). Действие, вызываемое

исполнением процедуры `invoke_op`, зависит от того, реплицирован ли объект, имеется ли в наличии его локальная копия, должна ли производиться запись или чтение, поддерживает ли платформа надежную, полностью упорядоченную ширококвещательную рассылку. Различают четыре варианта, как показано на рис. 6.27.

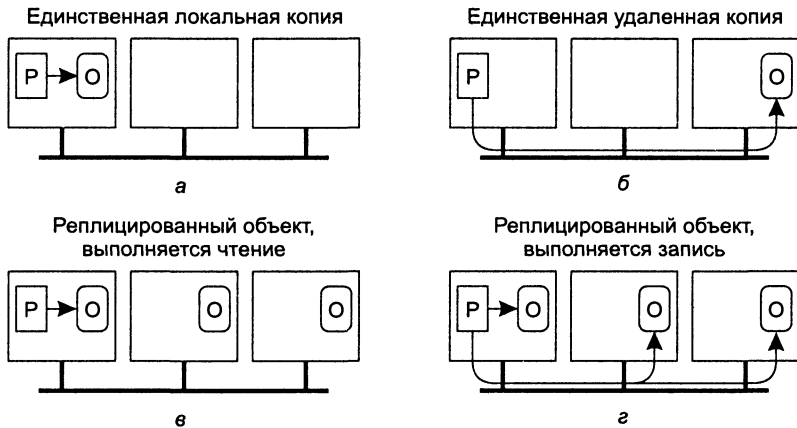


Рис. 6.27. Четыре варианта процесса P , выполняющего операцию над объектом O в Org_a

На рис. 6.27, *а* процесс собирается выполнить операцию над нереплицированным объектом, который расположен на одной машине с процессом. Это требует только блокирования объекта, вызова операции и разблокирования объекта. Смысл блокировки в том, чтобы избежать любых удаленных обращений во время выполнения локальной операции.

На рис. 6.27, *б* мы также имеем дело с единственной копией объекта, но имеется и кое-что еще. Исполняющая система, используя вызов `RPC`, просит удаленную машину выполнить нужную операцию, возможно, с небольшой задержкой, если объект в момент прихода запроса блокирован. Ни в одном из этих двух случаев между чтением и записью разницы нет (исключая тот факт, что запись, если изменит используемые стражами переменные, может активизировать блокированные процессы).

Если объект реплицирован, как показано на рис. 6.27, *в* и *г*, в нашем распоряжении всегда имеется его локальная копия, однако теперь имеет значение, какую операцию мы хотим осуществить — чтения или записи. Чтение можно проделать локально, без использования сети и без дополнительных затрат.

Запись в реплицируемые объекты — более хитрое дело. Если базовая система поддерживает надежные, полностью упорядоченные ширококвещательные рассылки, исполняющая система рассылает имя объекта, операцию и ее параметры, а затем блокируется до завершения ширококвещательной рассылки. Все машины, включая данную, вычисляют новое значение.

Отметим, что примитивы ширококвещательной рассылки должны быть надежны, то есть нижележащие уровни должны автоматически обнаруживать и исправлять потерю сообщений. Система `Amoeba`, на основе которой разрабатыва-

лась система Огса, имеет такие средства. Каждое сообщение, предназначенное для распространения путем широковещательной рассылки, отправляется специальному процессу под названием *секвенсор* (*sequencer*), который присваивает ему последовательный номер, а затем рассылает его, используя ненадежную аппаратную широковещательную рассылку. Когда процесс обнаруживает пропуск в последовательных номерах, он понимает, что потерял сообщение, и предпринимает действия по его восстановлению.

Если система не поддерживает надежной широковещательной рассылки (или не имеет вообще никаких средств рассылки), обновления вносятся при помощи протокола на базе первичной копии. Каждый объект имеет первичную копию. Процесс, выполняющий обновление, посылает сначала сообщение первичной копии объекта, блокирует ее и изменяет. Затем первичная копия посылает индивидуальные сообщения всем остальным машинам, имеющим копии этого объекта, требуя блокировки этих копий. Когда все они подтвердят наличие блокировки, начавший все это процесс переходит во вторую фазу и посылает другое сообщение, предназначенное для того, чтобы провести операцию и разблокировать объект.

Взаимные блокировки при этом невозможны. Если два процесса одновременно захотят изменить один и тот же объект, один из них первым заблокирует первичную копию, а второй будет дожидаться, пока объект снова не освободится. Кроме того, отметим, что в ходе процесса обновления все копии объекта блокируются, так что ни один из прочих процессов не может считать прежние значения. Эта блокировка гарантирует, что все операции будут выполняться в глобально уникальном последовательном порядке.

Теперь кратко рассмотрим алгоритм, с помощью которого мы определяем, в каком состоянии (единственной копии или реплицированном) находится объект. Изначально программа Огса состоит из одного процесса, владеющего всеми объектами. Когда она разветвляется, все остальные машины уведомляются об этом событии и получают текущие копии всех совместно используемых параметров дочерних процессов. Каждая исполняющая система подсчитывает стоимость репликации этого объекта и сравнивает ее со стоимостью отказа от репликации.

Чтобы проделать эти вычисления, необходимо знать ожидаемое соотношение операций чтения и записи. Компилятор оценивает это значение, исследуя программу и принимая во внимание, что вызовы из циклов стоят больше, а вызовы из условных инструкций — меньше, чем обычные вызовы. Также в расчетах учитываются затраты на взаимодействие. Например, объект с соотношением чтение/запись 10 в сети с широковещательной рассылкой следует реплицировать, а объект с соотношением чтение/запись 1 в сети со сквозной (от точки к точке) связью — хранить в состоянии единственной копии, причем эту копию следует располагать на той машине, на которой выполняется большинство операций записи. Более детальное описание этого процесса можно найти в [29].

Поскольку все исполняющие системы проделывают одни и те же вычисления, они приходят к одинаковому решению. Если объект постоянно находится на одной машине, а нужен на всех, он становится распределенным. Если объект реплицирован, а необходимость в этом уже исчезла, все машины, кроме одной,

уничтожают его копии. Используя этот же механизм, объект может перемещаться с машины на машину.

Давайте рассмотрим, как реализуется последовательная непротиворечивость. Для объектов в состоянии единственной копии все операции реально сериализованы, так что последовательную непротиворечивость мы получаем «даром». Для реплицируемых объектов операции записи полностью упорядочены в результате использования либо надежной, полностью упорядоченной широковещательной рассылки, либо алгоритма на базе первичной копии. В любом случае мы имеем глобальное соглашение о порядке выполнения операций записи. Операции чтения локальны и могут случайным образом чередоваться с операциями записи, не нарушая последовательной непротиворечивости.

Возможна различная оптимизация. Так, например, вместо синхронизации после операции ее можно производить в момент начала операции, как в случае элементарной непротиворечивости или слабой свободной непротиворечивости. Преимущество такого нововведения в том, что если процесс выполняет операцию над разделяемым объектом многократно (то есть в цикле), нам не понадобится производить широковещательную рассылку до тех пор, пока какой-либо другой процесс не проявит интерес к этому объекту.

Другая возможная оптимизация — не останавливать вызвавший процесс во время широковещательной рассылки после операции записи, которая не должна возвращать значения (например, как в примере с помещением в стек). Разумеется, такая оптимизация должна быть прозрачна. Информация, предоставляемая компилятором, делает возможными и другие варианты оптимизации.

Подведем итоги. Модель распределенной разделяемой памяти Orca сочетает в себе хорошие традиции разработки программного обеспечения (инкапсулируемые объекты), совместное использование данных, простую семантику и естественную синхронизацию. Кроме того, во многих случаях ее реализация так же эффективна, как может быть разве что свободная непротиворечивость. Она прекрасно работает, если базовое аппаратное обеспечение и операционная система поддерживают эффективную надежную, полностью упорядоченную широковещательную рассылку, а в обращениях приложений к совместно используемым объектам отношение операций чтения к операциям записи изначально велико.

6.6.2. Слабая причинно непротиворечивая репликация

В качестве следующего примера непротиворечивости и репликации, абсолютно непохожего на предыдущий, обсудим схему репликации, реализующую потенциальную непротиворечивость и отслеживающую в то же время причинно-следственные отношения между операциями. Эта схема, описанная в [246], используется в причинно непротиворечивых хранилищах данных, но использует для распространения обновлений слабую форму непротиворечивости.

Модель системы

Для того чтобы понять, как работает слабая причинно непротиворечивая репликация, мы адаптируем модель распределенного хранилища данных, которую не-

давно использовали (см. рис. 6.4). Суть слабой причинно непротиворечивой репликации в том, что при помощи векторных отметок времени устанавливается потенциальная причинно-следственная связь между операциями чтения и записи. Векторные отметки времени мы обсуждали в предыдущей главе. Далее предположим, что хранилище данных является реплицируемым и распределенным по N серверам. Как и ранее, мы считаем, что клиенты соединены с локальными (или ближайшими доступными) серверами, хотя это ограничение не является строгим.

Клиенты могут связываться друг с другом, однако обмениваются информацией они при помощи операций с хранилищем данных. (Поэтому на практике набор клиентов организуется в набор внешних интерфейсов хранилища данных, которые обмениваются информацией, и «чистых» клиентов, которые остаются в полном неведении о том, как именно в хранилище данных поддерживаются непротиворечивость и репликация. Здесь мы не будем делать различия между ними.) Общая структура хранилища данных показана на рис. 6.28.

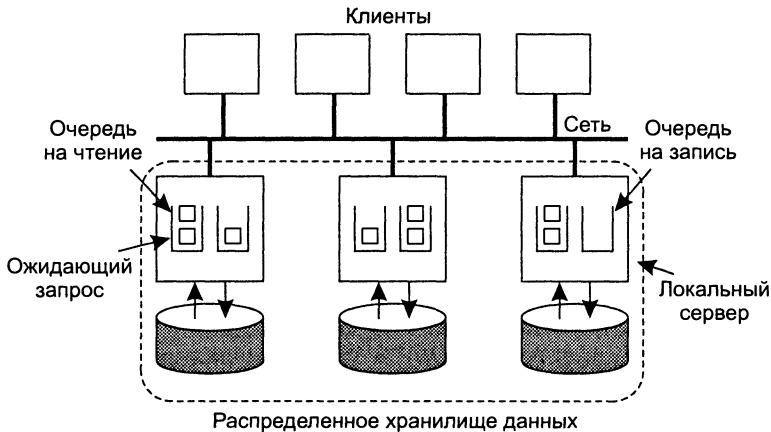


Рис. 6.28. Обобщенная структура распределенного хранилища данных. Предполагается, что клиенты также обслуживают взаимодействие, связанное с поддержкой непротиворечивости

Как показано на рисунке, каждый сервер хранилища содержит локальную базу данных и две очереди операций. Локальная база данных содержит данные, которые должны постоянно храниться на сервере для поддержания глобальной модели непротиворечивости хранилища. Другими словами, она содержит именно те данные, которые отождествляются с программным контрактом между клиентом и хранилищем данных, выраженные в форме модели непротиворечивости, ориентированной на данные. В этом примере контракт побуждает свои локальные копии соответствовать причинной непротиворечивости.

Каждая копия поддерживает две очереди отложенных операций. Очередь на чтение состоит из операций чтения, которые отложены до тех пор, пока локальная база данных не придет в состояние, соответствующее тому, на что рассчитывает операция чтения. Так, например, если в операции чтения сказано, что клиент, инициировавший операцию, обязательно должен увидеть результаты не-

которых операций записи, локальная база данных должна быть обновлена этими операциями до проведения операций чтения. Это требование очень близко к обсуждавшейся ранее модели непротиворечивости, ориентированной на клиента. Что такое на самом деле локальная непротиворечивость, мы обсудим ниже.

Точно так же очередь на запись содержит операции записи, которые отложены до тех пор, пока локальная база данных не придет в состояние, соответствующее тому, на что рассчитывает операция записи. В частности, операция записи может быть произведена только в том случае, если локальная база данных была обновлена всеми предшествующими операциями, от которых причинно зависит эта операция записи. Детали мы также изложим ниже.

Для представления цельного состояния локальной базы данных и точного определения того, какие операции в каком состоянии нуждаются, используются векторные отметки времени. Прежде всего, каждая локальная копия L_i поддерживает два вектора отметок времени. Вектор $VAL(i)$ соответствует текущему состоянию копии L_i . $VAL(i)[i]$ — это общее число запросов на запись, поступивших напрямую от клиента копии L_i и выполненных на этой копии. Кроме того, L_i отслеживает, сколько операций обновления было ею принято (и обработано) от копии L_j . Это число записывается в $VAL(i)[j]$.

Второй вектор отметок времени, $WORK(i)$, отражает, что должно быть сделано копией L_i . В частности, $WORK(i)[i]$ — это общее число операций записи, пришедших от клиента, которые должны быть выполнены (включая уже выполняющиеся), чтобы очередь записи стала пустой. Точно так же $WORK(i)[j]$ отражает общее число обновлений, присланных с L_j , которые должны быть выполнены, чтобы очередь записи стала пустой.

Кроме этих двух векторов отметок времени каждый клиент отслеживает текущее состояние хранилища данных. Для этого клиент C поддерживает вектор отметок времени $LOCAL(C)$, где $LOCAL(C)[i]$ устанавливается равным последнему значению состояния L_i , которое видно из C . (Отметим, что клиент, производя чтение или запись, каждый раз контактирует с разными репликами.) Каждый раз, когда клиент C пересылает запрос на чтение или запись RW локальной копии, в качестве отметки времени $DEP(RW)$ он выставляет $LOCAL(C)$, чтобы показать, от чего зависит RW .

Выполнение операций чтения

Теперь самое время описать, как происходят различные операции. Сначала мы сосредоточимся на операции чтения, различные этапы которой иллюстрирует рис. 6.29. Очередность здесь следующая.

1. Присваивание $DEP(R) := LOCAL(C)$.
2. Проверка $DEP(R) \leq VAL(i)$.
3. Передача данных и значения $VAL(i)$.
4. Присваивание $LOCAL(C) := \max\{LOCAL(C), VAL(i)\}$.

Когда клиент C (в нашем случае, внешний интерфейс) хочет произвести операцию чтения R , отметка времени $DEP(R)$ соответствующего запроса на чтение устанавливается равной $LOCAL(C)$. Эта отметка времени отражает тот факт, что

клиент постоянно знает о хранилище данных. После этого запрос передается одной из копий.

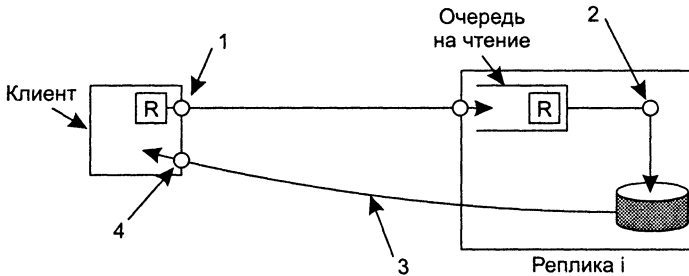


Рис. 6.29. Выполнение операции чтения локальной копии

Входящий запрос на чтение R к копии L всегда помещается в очередь на чтение этой копии. Отметка времени $DEP(R)$ отражает глобальное состояние хранилища данных в тот момент времени, когда был послан запрос R . Для обработки R необходимо, чтобы локальная копия L_i также была осведомлена об этом состоянии. Так, в частности, для каждого значения j справедливо $DEP(R)[j] \leq VAL(i)[j]$.

Как только операция чтения будет произведена, копия L_i возвращает значение запрошенного элемента данных клиенту вместе с $VAL(i)$. После этого клиент приводит в соответствие с ним свой собственный вектор $LOCAL(C)$, устанавливая каждый из его элементов $LOCAL(C)[j]$ в $\max\{LOCAL(C)[j], VAL(i)[j]\}$.

Выполнение операций записи

Выполнение операций записи происходит более или менее аналогично операциям чтения, как показано на рис. 6.30. Очередность этапов выполнения здесь следующая.

1. Присваивание $DEP(W) := LOCAL(C)$.
2. Присваивания $WORK(i)[i] := WORK(i)[i] + 1$, $ts(W)[i] := WORK(i)[i]$ и $ts(W)[i] := DEP(W)[j]$.
3. Передача отметки времени $ts(W)$.
4. Присваивание $LOCAL(C) := \max\{LOCAL(C), ts(W)\}$.
5. Проверка $DEP(W) \leq VAL(i)$.

Когда клиент C собирается выполнить операцию записи, он посылает запрос на запись копии L_i , устанавливая отметку времени $DEP(W)$ равной своему локальному вектору $LOCAL(C)$, как и в прошлый раз.

Когда копия получает запрос на запись W от клиента, она увеличивает $WORK(i)[i]$, оставляя прочие элементы неизменными. Кроме того, в ответ на запрос на запись L_i возвращает отметку времени $ts(W)$, где $ts(W)[i]$ равно $WORK(i)[i]$, а другие элементы $ts(W)[j]$ получили значения $DEP(W)[j]$. Эта отметка времени посылается клиенту как подтверждение, после чего клиент приводит в соответствие с ним собственный вектор $LOCAL(C)$, устанавливая каждый k -й его элемент $LOCAL(C)[k]$ в значение $\max\{LOCAL(C)[k], ts(W)[k]\}$.

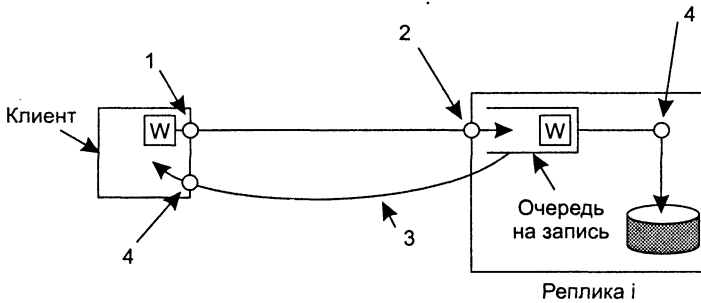


Рис. 6.30. Выполнение операции записи локальной копии

Как и в случае запроса на чтение, запрос на запись W выполняется, только если локальная копия L_i произвела все обновления, от которых зависит W . Это тот случай, когда для каждого элемента $DEP(W)[j] \leq VAL(i)[j]$. В этом случае операция производится и вектор $VAL(i)$ настраивается путем присваивания каждому j -му элементу значения $\max\{VAL(i)[j], ts(W)[j]\}$. Поскольку вектор $WORK(i)[i]$ увеличился на единицу, когда запрос W был получен копией L_i , а затем получил значение метки $ts(W)[i]$, мы легко убедимся, что запрос W обрабатывается копией L_i при выполнении следующих двух условий:

$$ts(W)[i] = VAL(i)[i] + 1,$$

$$ts(W)[j] \leq VAL(i)[j] \text{ для всех } j \neq i.$$

Первое условие гласит, что должны быть выполнены все операции, посланные напрямую копии L_i от других клиентов и предшествующие операции записи W . Второе условие говорит о том, что все обновления, от которых зависит запись W , также должны быть выполнены на L_i . Отметим, что эти два условия полностью аналогичны условиям реализации причинной доставки сообщений, которая обсуждалась в предыдущей главе.

Выполненные операции остаются в очереди, но никогда не выполняются снова. Они сохраняются в очереди потому, что может потребоваться распространить их для выполнения другим копиям. Это распространение обновлений обсуждается далее.

Распространение обновлений

Чтобы закончить тему обработки операций, нам следует обсудить распространение операций обновления среди различных копий. Распространение обновлений выполняется при помощи антиэнтропийных методов, о которых мы упоминали, когда обсуждали эпидемические протоколы. Время от времени копия L_i связывается с другой копией L_j и обменивается с ней операциями, содержащимися в очередях на запись обеих копий.

Если говорить более конкретно, когда копия L_i связывается с другой копией L_j , она пересылает ей все сообщения, содержащиеся в ее очереди на запись. Кроме того, копии L_j пересылается векторная отметка времени $WORK(i)$. В свою очередь, L_j подстраивает свой собственный вектор $WORK(j)$, устанавливая каждый

k -й элемент равным $\max\{WORK(i)[k], WORK(j)[k]\}$. Далее L_j включает операции записи, полученные от L_i , в свою очередь на запись. Если операция W уже содержится в очереди L_j , она просто отбрасывается. В противном случае W добавляется в очередь.

После обмена обновлениями копия L_i просматривает, какие из полученных ею операций могут быть выполнены. Обозначим через U набор всех отложенных операций записи W , для которых для каждого k -го элемента соблюдается условие $DEP(W)[k] \leq VAL(i)[k]$. Копия L_i , соответственно, выбирает операцию W' , не зависящую от любой другой операции записи в наборе U . Другими словами, в U нет больше такой операции W , для которой для каждого k -го элемента выполняется условие $DEP(W)[k] \leq DEP(W')[k]$. После этого операция W' удаляется из U и выполняется. Затем в наборе U выбирается следующая операция.

Существует еще множество нюансов, оставшихся «за кадром». Важно то, что согласно приведенному описанию, очереди записи могут разрастись бесконечно. Понятно, что операция должна удаляться из очереди сразу же, как только мы определили, что она выполнена всеми копиями. Чтобы определить этот момент, необходимо дополнительное администрирование и обмен дополнительной информацией. Детали можно найти в [246].

6.7. Итоги

Существует два основных довода в пользу реплицирования данных — повышение надежности распределенных систем и увеличение их производительности. Репликация порождает проблему противоречивости: при обновлении одной из реплик она становится отличной от остальных. Для сохранения непротиворечивости реплик нам необходимо распространять обновления так, чтобы временная противоречивость оставалась незаметной. К сожалению, это может значительно снизить производительность, особенно в крупных распределенных системах.

Единственное решение этой проблемы — посмотреть, нельзя ли немного ослабить требования к непротиворечивости. В моделях строгой непротиворечивости непротиворечивость определяется для отдельных операций чтения и записи совместно используемых данных. Можно провести разделение этих моделей на строгую непротиворечивость, последовательную непротиворечивость, линейаризуемость, причинную непротиворечивость и непротиворечивость FIFO.

Строгая непротиворечивость предполагает, что операции чтения всегда возвращают последнее записанное значение. Из-за нереальности поддержки глобального времени в распределенных системах реализация строгой непротиворечивости невозможна. Последовательная непротиворечивость и линейаризуемость, в сущности, имеют ту же семантику, которая используется программистами в параллельном программировании: все операции записи должны наблюдаться отовсюду в одинаковом порядке. Линейаризуемость — немного более строгая модель, она упорядочивает операции в соответствии с глобальными часами (с конечной точностью).

Причинная непротиворечивость отражает тот факт, что операции, потенциально зависящие друг от друга, происходят в порядке, определяемом этой зави-

симостью. Непротиворечивость FIFO просто определяет, что операции одного процесса происходят в порядке, определяемом этим процессом.

Модели слабой непротиворечивости относятся к последовательностям операций чтения и записи. В частности, они предполагают, что каждая последовательность сопровождается сопутствующими операциями с переменными синхронизации, такими, как блокировки. Хотя это требует дополнительных усилий со стороны программистов, эффективно реализовать модели слабой непротиворечивости обычно проще, чем строгой.

В противовес этим моделям, ориентированным на данные, исследователи распределенных баз данных, предназначенных для мобильных пользователей, предложили множество моделей непротиворечивости, ориентированных на клиента. В этих моделях игнорируется тот факт, что данные могут совместно использоваться различными пользователями, вместо этого обсуждение сконцентрировано на уровне непротиворечивости, необходимом отдельному клиенту. В основу этих моделей легло соображение о том, что клиент время от времени может подсоединяться к различным репликам, при этом различия между ними должны быть незаметны. В сущности, модели непротиворечивости, ориентированные на клиента, обеспечивают такое поведение системы, что при соединении клиента с новой репликой эта реплика быстро приводится в соответствие с данными, с которыми этот клиент работал раньше и которые могут находиться в других репликах.

Для распространения обновлений применяются различные технологии. Разделение можно провести по тому, *что* на самом деле распространяется, *куда* распространяются обновления и *кто* инициирует распространение. Мы можем рассмотреть распространение уведомлений, операций или состояния. Кроме того, не все реплики следует всегда обновлять немедленно. Какая реплика и в какой момент будет обновлена, зависит от протокола распределения. И наконец, следует решить, могут ли обновления продвигаться другим репликам или каждая реплика сама должна извлекать обновление у других реплик.

Протоколы непротиворечивости описывают конкретные реализации моделей непротиворечивости. Для последовательной непротиворечивости и ее вариантов выбор лежит между протоколами первичной копии и реплицируемой записи. В протоколах первичной копии все операции обновления передаются первичной копии, которая затем убеждается, что обновления правильно упорядочены и переданы. В протоколах реплицируемой записи обновление одновременно передается нескольким репликам. В этом случае правильное упорядочение операций часто затруднено.

Вопросы и задания

1. Доступ к совместно используемым объектам в Java можно сериализовать, объявив их методы синхронизируемыми. Достаточно ли этого, чтобы гарантировать сериализацию при репликации этих объектов?
2. Рассмотрим монитор, описанный в главе 1. Если потоки выполнения в реплицированном мониторе можно блокировать, что нам нужно сделать для гарантии правильного срабатывания условной переменной?

3. Опишите своими словами, какова главная причина создания моделей слабой непротиворечивости.
4. Опишите, как реализована репликация в DNS и почему она так хорошо работает на практике.
5. В ходе обсуждения моделей непротиворечивости мы часто ссылались на контракт между программой и хранилищем данных. Зачем нужен такой контракт?
6. Линеаризуемость предполагает существование глобальных часов. Однако, как мы видели, подобное требование делает нереальным поддержание в большинстве распределенных систем строгой непротиворечивости. Можно ли реализовать линеаризуемость в физически распределенных хранилищах данных?
7. Мультипроцессорная система имеет одну шину. Можно ли реализовать в такой системе память со строгой непротиворечивостью?
8. Почему последовательность $W1(x)a R2(x)NIL R3(x)a$ по отношению к представленному на рис. 6.5, б хранилищу данных запрещена?
9. Рассмотрим рис. 6.7. Является ли допустимым результатом для распределенной разделяемой памяти, поддерживающей только непротиворечивость FIFO, значение 001110? Поясните свой ответ.
10. Рассмотрим рис. 6.8. Является ли значение 001110 допустимым результатом для последовательно непротиворечивой памяти? Поясните свой ответ.
11. В конце пункта 6.2.2 мы обсуждали формальную модель, которая гласит, что любой набор операций в последовательно непротиворечивом хранилище данных может быть промоделирован строкой истории H , из которой могут быть выведены все индивидуальные последовательности процессов. Для процессов $P1$ и $P2$, показанных на рис. 6.9, приведите все возможные значения H . Игнорируйте процессы $P3$ и $P4$ и не включайте в H их операции.
12. Последовательно непротиворечивая память допускает шесть вариантов чередования инструкций, представленных в табл. 6.4. Перечислите их.
13. Часто утверждается, что модели слабой непротиворечивости — это дополнительная «головная боль» для программистов. В какой степени это высказывание соответствует действительности?
14. Во многих реализациях свободной непротиворечивости в распределенных системах с разделяемой памятью общие переменные синхронизируются при освобождении, а не при захвате. Зачем для них вообще нужен захват?
15. Какую непротиворечивость обеспечивает система Огса, последовательную или поэлементную? Поясните свой ответ.
16. Способна ли полностью упорядоченная групповая рассылка, организуемая секвенсором и предназначенная для сохранения непротиворечивости при активной репликации, повредить аргумент, передаваемый от точки к точке, в архитектуре системы?
17. Какой тип непротиворечивости вы будете использовать при реализации проекта электронной биржи? Поясните свой ответ.

18. Рассмотрим персональный почтовый ящик мобильного пользователя, реализованный как часть глобальной распределенной базы данных. Какой тип непротиворечивости, ориентированной на клиента, будет для него наиболее подходящим?
19. Опишите простую реализацию непротиворечивости чтения своих записей для отображения только что обновленных web-страниц.
20. Приведите пример, когда непротиворечивость, ориентированная на клиента, может легко привести к конфликтам двойной записи.
21. Необходимо ли, чтобы в условиях аренды часы клиента и сервера были точно синхронизированы?
22. Рассмотрим неблокирующий протокол первичного архивирования, используемый для гарантированного соблюдения последовательной непротиворечивости в распределенном хранилище данных. Будет ли это хранилище всегда обеспечивать непротиворечивость чтения своих записей?
23. Обычно для работы активной репликации необходимо, чтобы все операции выполнялись всеми репликами в одном и том же порядке. Всегда ли это необходимо?
24. Один из методов реализации полностью упорядоченной групповой рассылки с помощью секвенсора — сначала передать операцию секвенсору, который присвоит ей уникальный номер и произведет групповую рассылку операции. Предложите два альтернативных метода и сравните эти три решения.
25. Файл реплицирован на 10 серверах. Перечислите все комбинации кворумов чтения и записи, которые допускает алгоритм голосования.
26. В тексте мы описывали ориентированную на отправителя схему, предотвращающую реплицирование обращений. В схеме, ориентированной на получателя, реплика, получающая сообщение, распознает копии входящих сообщений, относящихся к одному и тому же обращению. Опишите, как можно предотвратить реплицирование обращений в схеме, ориентированной на получателя.
27. Рассмотрим причинно непротиворечивую медленную репликацию. Когда операции могут быть удалены из очереди?
28. В этом упражнении вам предстоит реализовать простую систему, поддерживающую групповые вызовы RPC. Мы предполагаем, что имеется несколько реплицированных серверов и каждый клиент работает с сервером посредством RFC. Однако в условиях репликации клиент должен послать RFC каждой реплике. Программируя клиента, помните, что для приложения это должно выглядеть посылкой единственного вызова RFC. Считайте, что репликация требуется для повышения производительности, а серверы подвержены сбоям.

Глава 7

Отказоустойчивость

- 7.1. Понятие отказоустойчивости
- 7.2. Отказоустойчивость процессов
- 7.3. Надежная связь клиент-сервер
- 7.4. Надежная групповая рассылка
- 7.5. Распределенное подтверждение
- 7.6. Восстановление
- 7.7. Итоги

Характерной чертой распределенных систем, которая отличает их от единичных машин, является возможность частичного отказа. Частичный отказ происходит при сбое в одном из компонентов распределенной системы. Этот отказ может нарушить нормальную работу некоторых компонентов, в то время как другие компоненты это никак не затронет. В противоположность отказу в распределенной системе отказ в нераспределенной системе всегда является глобальным, в том смысле, что он затрагивает все ее компоненты и легко может привести к неработоспособности всего приложения.

При создании распределенной системы очень важно добиться, чтобы она могла автоматически восстанавливаться после частичных отказов, незначительно снижая при этом общую производительность. В частности, когда бы ни случился отказ, распределенная система в процессе восстановления должна продолжать работать приемлемым образом, то есть быть устойчивой к отказам, сохраняя в случае отказов определенную степень функциональности.

В этой главе мы познакомимся со способами обеспечения отказоустойчивости распределенной системы. После изложения определенных базовых сведений об отказоустойчивости мы рассмотрим вопросы отказоустойчивости процессов и надежной групповой рассылки. Под отказоустойчивостью процессов мы понимаем методы, при помощи которых отказ одного или более процессов проходит для остальной части системы почти незаметно. С этим вопросом связана проблема надежной групповой рассылки, при которой передача сообщений набору процессов производится с гарантией доставки. Надежная групповая рассылка часто необходима для поддержания синхронности процессов.

Как мы уже говорили в главе 5, атомарность — это свойство, важное для многих приложений. Так, например, в распределенных транзакциях необходимо гарантировать, что все операции, входящие в транзакцию, либо происходят, либо нет. Фундаментальным для атомарности в распределенных системах является понятие распределенных протоколов подтверждения, которые обсуждаются в отдельном разделе этой главы.

И, наконец, мы рассмотрим, как восстанавливать систему после отказов. В частности, мы обсудим, когда и как следует сохранять состояние распределенной системы на тот случай, если позже это состояние потребуется восстанавливать.

7.1. Понятие отказоустойчивости

Исследованию отказоустойчивости посвящено большое количество трудов. Этот раздел мы начнем с рассмотрения базовых концепций обработки отказов, а далее обсудим модели отказов. Основа всех методик ликвидации последствий отказов — избыточность, о ней мы тоже поговорим. Дополнительную информацию общего характера по отказоустойчивости можно найти, например, в [213].

7.1.1. Основные концепции

Чтобы понять роль отказоустойчивости в распределенных системах, сначала необходимо выяснить, что для распределенных систем означает «быть отказоустойчивыми». Отказоустойчивость тесно связана с понятием *надежных систем* (*dependable systems*). Надежность — это термин, охватывающий множество важных требований к распределенным системам [241], включая:

- ✦ доступность (availability);
- ✦ безотказность (reliability);
- ✦ безопасность (safety);
- ✦ ремонтпригодность (maintainability).

Доступность — это свойство системы находиться в состоянии готовности к работе. Обычно доступность показывает вероятность того, что система в данный момент времени будет правильно работать и окажется в состоянии выполнить свои функции, если пользователи того потребуют. Другими словами, система с высокой степенью доступности — это такая система, которая в произвольный момент времени, скорее всего, находится в работоспособном состоянии.

Под *безотказностью* имеется в виду свойство системы работать без отказов в течение продолжительного времени. В противоположность доступности безотказность определяется в понятиях временного интервала, а не момента времени. Система с высокой безотказностью — это система, которая, скорее всего, будет непрерывно работать в течение относительно долгого времени. Между безотказностью и доступностью имеется небольшая, но существенная разница. Если система отказывает на одну миллисекунду каждый час, она имеет доступность порядка 99,9999 %, но крайне низкую безотказность. С другой стороны, система,

которая никогда не отказывает, но каждый август отключается на две недели, имеет высокую безотказность, но ее доступность составляет всего 96 %. Эти две характеристики — не одно и то же.

Безопасность определяет, насколько катастрофична ситуация временной неспособности системы должным образом выполнять свою работу. Так, многие системы управления процессами, используемые, например, на атомных электростанциях или космических кораблях, должны обладать высокой степенью безопасности. Если эти управляющие системы даже временно, на короткий срок, перестанут работать, результат может быть ужасен. Множество примеров происшедших в прошлом событий показывают, как тяжело построить безопасную систему (и может быть еще больше таких примеров ожидают нас в будущем).

И, наконец, *ремонтпригодность* определяет, насколько сложно исправить неполадки в описываемой системе. Системы с высокой ремонтпригодностью могут также обладать высокой степенью доступности, особенно при наличии средств автоматического обнаружения и исправления неполадок. Однако, как мы увидим позже в этой главе, говорить об автоматическом исправлении неполадок гораздо проще, чем создавать способные на это системы.

Часто надежные системы требуют также повышенного уровня защиты, особенно когда дело доходит до такого вопроса, как непротиворечивость. Мы поговорим о защите в следующей главе.

Говорят, что система *отказывает* (*fail*), если она не в состоянии выполнять свою работу. В частности, если распределенная система создавалась для предоставления пользователям некоторых услуг, то система будет считаться находящейся в состоянии отказа в том случае, если она не сможет предоставлять все или некоторые услуги. *Ошибкой* (*error*) называется такое состояние системы, которое может привести к ее неработоспособности. Так, например, при передаче пакетов по сети может случиться, что некоторые пакеты, пришедшие к получателю, окажутся поврежденными. Повреждения в данном случае будут означать, что получатель может неверно прочесть значения битов (например, 1 вместо 0) или оказаться не в состоянии определить сам факт прихода пакета.

Причиной ошибки является *отказ* (*fault*). Понятно, что найти причину ошибки очень важно. Так, например, вызвать повреждение пакетов вполне может неисправная или некачественная среда передачи. В этом случае устранить отказ относительно легко. Однако ошибки передачи в беспроводных сетях могут быть вызваны, например, плохой погодой. Изменение погоды с целью предупредить возникновение ошибок нам пока не под силу.

Построение надежных систем тесно связано с управлением отказами. Управление в данном случае означает нечто среднее между предотвращением, исправлением и предсказанием отказов [256]. Для нашей цели наиболее важным вопросом является *отказоустойчивость* (*fault tolerance*), под которой мы будем понимать способность системы предоставлять услуги даже при наличии отказов.

Отказы обычно подразделяются на проходные, перемежающиеся и постоянные. *Проходные отказы* (*transient faults*) происходят однократно и больше не повторяются. Если повторить операцию, они не возникают. Птица, пролетевшая через луч микроволнового передатчика, в некоторых сетях может привести к по-

тере битов. Если передатчик, выждав положенную паузу, повторит отправку, то по всей вероятности передача пройдет как положено.

Перемежающиеся отказы (intermittent faults) появляются и пропадают, «когда захотят», а потом появляются снова и т. д. Перемежающиеся отказы нередко бывают вызваны потерей контакта в разъеме. Из-за трудностей в диагностике перемежающиеся отказы часто вызывают сильное раздражение. Обычно когда приходит ремонтник, система работает просто прекрасно.

Постоянные отказы (permanent faults) — это отказы, которые продолжают свое существование до тех пор, пока отказавший компонент не будет заменен. Примерами постоянных отказов могут быть сгоревшие микросхемы, ошибки в программном обеспечении или сместившиеся головки дисков.

7.1.2. Модели отказов

Отказавшая система не в состоянии корректно выполнять ту работу, для которой она была создана. Если рассматривать распределенную систему как набор серверов, взаимодействующих друг с другом и с клиентами, то некорректное выполнение работы будет означать, что серверы или коммуникационные каналы, а возможно и оба этих компонента, не в состоянии делать то, для чего они, как предполагалось, предназначены. Однако сами по себе нефункционирующие серверы не всегда приводят к отказу системы, как мы его понимаем. Если сервер для корректной работы нуждается в услугах других серверов, причину ошибки, может быть, следует искать в другом месте.

Таких зависимостей в распределенных системах великое множество. Отказавший диск осложняет работу файлового сервера, который разрабатывался для реализации файловой системы с высокой степенью доступности. Если этот файловый сервер является частью распределенной базы данных, под угрозой находится работа всей базы, поскольку доступной оказывается только часть данных.

Чтобы лучше понимать, насколько серьезен на самом деле конкретный отказ, были разработаны различные схемы классификации. Одна из таких схем, приведенная в табл. 7.1, основана на выкладках из [114, 191].

Таблица 7.1. Различные типы отказов

Тип отказа	Описание
Поломка	Сервер перестал работать, хотя до момента отказа работал правильно
Пропуск данных пропуск приема пропуск передачи	Сервер неправильно реагирует на входящие запросы Сервер неправильно принимает входящие запросы Сервер неправильно отправляет сообщения
Ошибка синхронизации	Реакция сервера происходит не в определенный интервал времени
Ошибка отклика ошибка значения ошибка передачи состояния	Отклик сервера неверен Сервер возвращает неправильное значение Сервер отклоняется от верного потока управления
Произвольная ошибка	Сервер отправляет случайные сообщения в случайные моменты времени

Поломка (crash failure) имеет место при внезапной остановке сервера, при этом до момента остановки он работает нормально. Важная особенность поломки состоит в том, что после остановки сервера никаких признаков его работы не наблюдается. Типичный пример поломки — полное зависание операционной системы, когда единственным решением проблемы является перезагрузка. Многие операционные системы персональных компьютеров претерпевают поломки настолько часто, что люди уже начинают полагать, что для них это обычное дело. В этом смысле перенос кнопки Reset с задней части корпуса компьютера на переднюю панель был, несомненно, оправдан.

Пропуск данных (omission failure) возникает в том случае, когда сервер неправильно реагирует на запросы. Эту ошибку могут вызывать различные причины. В случае *пропуска приема (receive omission)* сервер может, например, не получать запросов. Отметим, что такая ошибка может произойти, в частности, и в том случае, когда соединение между клиентом и сервером установлено совершенно правильным образом, но на сервере не запущен процесс для приема входящих запросов. Пропуск приема обычно не влияет на текущее состояние сервера, но сервер остается в неведении о посланных ему сообщениях.

Похожая ошибка — *пропуск передачи (send omission)* — происходит, когда сервер выполняет свою работу, но по каким-либо причинам не в состоянии послать ответ. Подобная ошибка может произойти, например, при переполнении буфера передачи, если сервер не готов к подобной ситуации. Отметим, что в противоположность пропуску приема в данном случае сервер может перейти в состояние, соответствующее полному выполнению услуги для клиента. Впоследствии, если обнаружится, что имел место пропуск передачи, сервер, вероятно, должен быть готов к тому, что клиент повторно пошлет свой последний запрос.

Другие типы пропусков не имеют отношения к взаимодействию и могут быть вызваны ошибками в программе, такими как бесконечные циклы или некорректная работа с памятью, которые способны «подвесить» сервер.

Следующий класс ошибок связан с синхронизацией. *Ошибки синхронизации (timing failures)* возникают при ожидании ответа дольше определенного временного интервала. Как мы говорили во время обсуждения изохронных потоков данных в главе 2, слишком раннее предоставление данных легко может вызвать у принимающей стороны проблемы, связанные с отсутствием места в буфере для хранения получаемых данных. Чаще, однако, сервер отвечает слишком поздно, в этом случае говорят, что произошла *ошибка производительности (performance failure)*.

Еще один важный тип ошибок — *ошибки отклика (response failures)*, при которых ответы сервера просто неверны. Существует два типа ошибок отклика. В случае *ошибки значения (value failure)* сервер дает неверный ответ на запрос. Так, например, эту ошибку демонстрирует поисковая машина, систематически возвращающая адреса web-страниц, не связанных с запросом пользователя.

Другой тип ошибок отклика — *ошибки передачи состояния (state transition failures)*. Этот тип ошибок характеризуется реакцией на запрос, не соответствующей ожиданиям. Так, например, если сервер получает сообщение, которое он не в состоянии распознать, и никаких мер по обработке подобных сообщений

не предусмотрено, возникает ошибка передачи состояния. В частности, сервер может неправомерно осуществить по умолчанию некие действия, производить которые в данном случае не следовало бы.

Весьма серьезны *произвольные ошибки* (*arbitrary failures*), известные также под названием *византийских ошибок* (*Byzantine failures*). Когда случается произвольная ошибка, клиент должен приготовиться к самому худшему. Например, может оказаться, что сервер генерирует сообщения, которые он в принципе не должен генерировать, но система не опознает их как некорректные. Хуже того, неправильно функционирующий сервер может, участвуя в работе группы серверов, приводить к появлению заведомо неверных ответов. Эта ситуация показывает, почему для надежных систем очень важна защита. Термин «византийские» восходит к Византийской империи (Балканы и современная Турция) времен 330–1453 годов, когда бесконечные заговоры, интриги и ложь считались в правящих кругах обычным делом. Византийские ошибки впервые были проанализированы в [252, 347]. Ниже мы вернемся к разговору об ошибках такого рода.

Произвольные ошибки похожи на поломки. Поломка — наиболее распространенная причина остановки сервера. Поломки известны также под названием *ошибок аварийной остановки* (*fail-stop failures*). В действительности аварийно остановленный сервер просто прекращает генерировать исходящие сообщения. По этому признаку его остановка обнаруживается другими процессами. Например, по настоящему дружественный сервер может предупредить нас о том, что находится на грани поломки.

Разумеется, в реальной жизни серверы, останавливаясь по причине пропуска данных или поломок, не настолько дружелюбны, чтобы оповестить нас о надвигающейся остановке. Другие процессы должны сами обнаружить «безвременную кончину» сервера. Однако в подобных *системах остановки без уведомления* (*fail-silent systems*) другие процессы могут сделать неверный вывод об остановке сервера. Сервер может просто медленно работать, то есть может иметь место ошибка производительности.

И, наконец, возможно, что сервер производит случайные сообщения, которые другие процессы считают абсолютным мусором. В этом случае мы имеем дело с наиболее простым случаем произвольной ошибки. Подобные ошибки называют *безопасными* (*fail-safe*).

7.1.3. Маскирование ошибок при помощи избыточности

Если система считается отказоустойчивой, она должна пытаться маскировать факты ошибок от других процессов. Основной метод маскирования ошибок — использование *избыточности* (*redundancy*). Возможно применение трех типов избыточности — информационной избыточности, временной избыточности и физической избыточности [218]. В случае информационной избыточности к сообщению добавляются дополнительные биты, по которым можно произвести исправление сбойных битов. Так, например, можно добавить к передаваемым

данным код Хемминга для восстановления сигнала в случае зашумленного канала передачи.

При временной избыточности уже выполненное действие при необходимости осуществляется еще раз. В качестве примера к этому способу рассмотрим транзакции (описанные в главе 5). Если транзакция была прервана, ее можно без каких-либо опасений повторить. Временная избыточность особенно полезна, если мы имеем дело с проходным или перемежающимся отказом.

В случае физической избыточности мы добавляем в систему дополнительное оборудование или процессы, которые делают возможной работу системы при утрате или неработоспособности некоторых компонентов. Физическая избыточность, таким образом, может быть как аппаратной, так и программной. Так, например, можно добавить к системе дополнительные процессы, так что при крахе некоторых из них система продолжит функционировать правильно. Другими словами, посредством репликации достигается высокая степень отказоустойчивости. Ниже мы вернемся к этому типу программной избыточности.

Физическая избыточность — это широко распространенный способ добиться отказоустойчивости. Она используется в биологии (млекопитающие имеют по два глаза, по два уха, по два легких и т. д.), самолетостроении (у Боинга-747 четыре двигателя, но он может лететь и на трех) и спорте (несколько судей на тот случай, если один чего-нибудь не заметит). Она также много лет используется для обеспечения отказоустойчивости в радиосхемах. Рассмотрим, например, схему, показанную на рис. 7.1, а. На ней сигнал проходит последовательно через устройства А, В и С. Если одно из них неисправно, результат, вероятно, будет неверен.

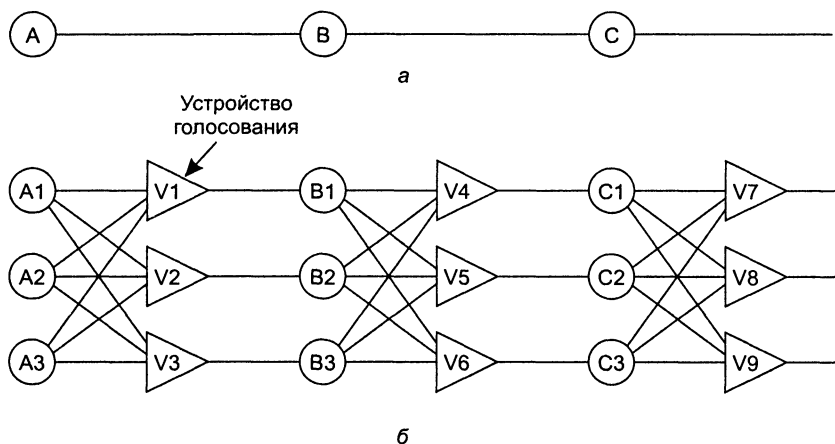


Рис. 7.1. Тройное модульное резервирование

На рис. 7.1, б каждое из устройств присутствует в трех экземплярах. Переход к следующему участку схемы определяется тройным голосованием. Устройство голосования — это схема с тремя входами и одним выходом. Если два или три входных сигнала совпадают, выходной сигнал равен входному. Если все три входных сигнала различны, выходной сигнал не определен. Такая схема известна под названием *тройного модульного резервирования (Triple Modular Redundancy, TMR)*.

Допустим, элемент $A2$ отказал. Каждое из устройств голосования, $V1$, $V2$ и $V3$, получает два правильных (идентичных) входных сигнала и один неправильный, и каждое из них передает на второй участок цепи правильное значение. В результате эффект отказа $A2$ оказывается полностью замаскированным, а значит, входные сигналы элементов $B1$, $B2$ и $B3$ абсолютно такие же, как если бы никакого отказа не было.

Рассмотрим теперь, что будет, если в придачу к $A2$ откажут также элементы $B3$ и $C1$. Эффект их отказа также будет замаскирован, и все три выходных сигнала окажутся правильными.

Прежде всего, непонятно, зачем на каждом этапе нужно использовать три устройства голосования. Вообще-то определить и донести до нас мнение большинства может и одно устройство голосования. Однако устройство голосования — это тоже компонент, который тоже может отказаться. Рассмотрим, например, отказ $V1$. Входящий сигнал $B1$ в этом случае будет неверным, но до тех пор, пока все остальное работает, $B2$ и $B3$ будут давать одинаковые выходные сигналы, и устройства $V4$, $V5$ и $V6$ образуют правильный результат для третьего этапа. Отказ $V1$ практически ничем не будет отличаться от отказа $B1$. В обоих случаях выходной сигнал от $B1$ будет неверным, и в любом случае при голосовании он окажется в меньшинстве.

Хотя не все отказоустойчивые распределенные системы используют TMR, эта технология является очень распространенной и помогает яснее понять, что такое отказоустойчивая система и чем она отличается от системы, составленной из высоконадежных компонентов, но не обладающей отказоустойчивой структурой. Разумеется, TMR можно применять и рекурсивно. Например, можно повышать надежность микросхем, встраивая в них механизмы TMR. Для разработчиков, использующих микросхемы, это останется неизвестным.

7.2. Отказоустойчивость процессов

Теперь, когда мы обсудили основные понятия отказоустойчивости, сосредоточимся на том, как она реализуется в распределенных системах на практике. Первой темой, которую мы обсудим, будет отказоустойчивость процессов, которая организуется путем репликации процессов в группах. Далее мы рассмотрим общие вопросы разработки групп процессов и поговорим о том, что собой представляет отказоустойчивая группа. Кроме того, мы увидим, как достигается согласие в группе процессов в том случае, если обнаружится, что один или более процессов дают неправильные ответы.

7.2.1. Вопросы разработки

Основной подход к защите от последствий отказа процессов — объединить несколько идентичных процессов в группу. Основное свойство всех подобных групп состоит в том, что когда сообщение посылается группе, его получают все члены этой группы. Таким образом, если один из процессов группы перестает работать, можно надеяться на то, что его место займет другой [187].

Группы процессов могут быть динамическими. Могут создаваться новые группы и ликвидироваться старые. В ходе системной операции процесс может войти в группу или покинуть ее. Процесс может входить в несколько групп одновременно. Таким образом, нам необходимы механизмы для управления группами и членством в них.

Группы отдаленно напоминают общественные организации. Алиса может быть членом клуба книголюбов, теннисного клуба и общества «зеленых». В определенные дни она может получать письма (сообщения), извещающие о новой книге «Выпечка для юбилеев» из клуба книголюбов, о ежегодном теннисном турнире, посвященном празднику 8 Марта, из теннисного клуба и из общества защиты природы о начале кампании в защиту южных сурков. В любой момент она может покинуть любой из них или все эти клубы, или вступить в другие.

Цель группировки состоит в том, чтобы перейти от рассмотрения отдельных процессов к рассмотрению новой абстракции — группы процессов. Так, процесс может посылать сообщения группе серверов, не зная ничего о том, сколько их там и где они находятся, причем состав группы серверов при каждом вызове может быть разным.

Одноранговые и иерархические группы

Все группы можно разделить в соответствии с их внутренней структурой. В некоторых группах все процессы равны между собой. Никаких начальников нет, и все решения принимаются коллективно. В других группах существует нечто вроде иерархии. Так, например, один из процессов — координатор, а все остальные — простые исполнители. В такой модели при появлении запроса, созданного где-то вне процесса или одним из внутренних рабочих процессов, этот запрос посылается координатору. Координатор решает, который из исполнителей лучше всего справится с запросом и передает ему этот запрос. Разумеется, возможны также и более сложные иерархические отношения. Эти способы взаимодействия иллюстрирует рис. 7.2.

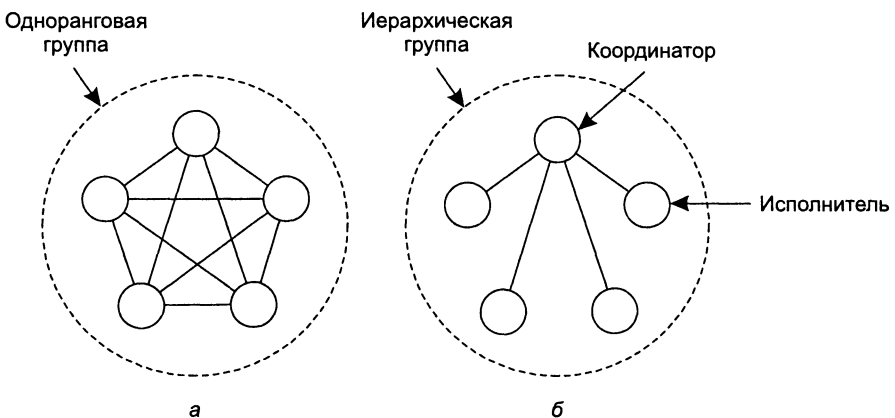


Рис. 7.2. Взаимодействие в одноранговой (а) и иерархической (б) группах

Любая из этих организаций имеет свои преимущества и недостатки. Одно-ранговая группа симметрична и не имеет единичной точки отказа. Если в одном из процессов обнаруживается ошибка, группа просто становится меньше, но продолжает существовать. Недостаток одноранговых групп состоит в том, что процесс принятия решений более сложен. Так, например, для того чтобы договориться о чем-то, необходимо проводить голосование, что влечет за собой определенную задержку и необходимость дополнительных действий.

Иерархическая группа обладает противоположными свойствами. Потеря координатора влечет за собой остановку работы всей группы, но пока он находится в рабочем состоянии, принимает решения сам, никого при этом не беспокоя.

Членство в группе

При групповом взаимодействии необходимы специальные методы создания и уничтожения групп, а также добавления процессов в группу и их удаления из нее. Один из возможных методов — создание *сервера групп* (*group server*), к которому должны направляться соответствующие запросы. Сервер групп будет поддерживать полную базу данных по группам и членству в них. Этот метод прост, эффективен и легко реализуем. К сожалению, для него характерен основной недостаток всех централизованных методов — единственная точка отказа. Если сервер групп выйдет из строя, всякое управление группами будет потеряно. Вероятно, из получившихся обломков большую часть групп удастся воссоздать, но при этом всякая деятельность, которая в них происходила, будет прервана.

Другой возможный метод — распределенное управление членством. Так, например, если доступна надежная групповая рассылка, внешний участник взаимодействия может послать сообщение всем членам группы, объявив о своем желании войти в состав этой группы.

В идеале, для того чтобы покинуть группу, ее члену достаточно разослать всем членам группы «прощальное письмо». В контексте отказоустойчивости использование семантики аварийной остановки, вообще говоря, запрещено. Проблема состоит в том, что существует нормальный способ объявить о добровольном прекращении членства в группе, но нет способа объявить об аварийной потере одного из членов. Другие члены группы должны определить это экспериментально, заметив, что некий член группы длительное время никому не отвечает. Как только становится понятно, что член группы действительно находится в нерабочем состоянии (а не просто медленно работает), он должен быть удален из группы.

Еще одна сложная проблема состоит в том, что удаление и добавление членов должны происходить синхронно с обработкой сообщений с данными. Другими словами, начиная с момента добавления процесса в группу, он должен получать все сообщения, направляемые группе. Соответственно, как только процесс покидает группу, он больше не должен получать отправленные в группу сообщения, и другие члены группы не должны получать сообщений от него. Один из способов гарантировать, что удаление или добавление в группу попало в правильное место потока сообщений — конвертировать эту операцию в последовательность сообщений, отправляемых всей группе.

Еще один вопрос относительно членства в группах состоит в том, что делать при отказе сразу нескольких машин, в результате которого группа не может продолжать функционирование. Необходим какой-то протокол повторной сборки группы. И вновь некий процесс должен проявить инициативу. Но что произойдет, если это одновременно сделают два или три процесса? Протокол должен быть в состоянии решить и эту проблему.

7.2.2. Маскировка ошибок и репликация

Группы процессов предлагают решение части задачи построения отказоустойчивых систем. В частности, группа идентичных процессов позволяет замаскировать наличие в этой группе одного или более отказавших процессов. Другими словами, мы можем реплицировать процессы и организовать их в группу, заменяя одиночный (уязвимый) процесс отказоустойчивой группой. Как мы говорили в предыдущей главе, имеется два способа проведения подобной репликации — с использованием протоколов на основе первичной копии или протоколов реплицируемой записи.

В случае отказоустойчивости репликация на основе первичной копии обычно применяется в форме протокола первичного архивирования. В этом случае группа процессов организуется в иерархию, в которой первичная копия координирует все операции записи. На практике первичная копия фиксирована, хотя при необходимости ее роль может взять на себя одна из архивных копий. В действительности при ошибке в первичной копии архивные копии, используя определенный алгоритм голосования, выбирают новую первичную копию.

Как мы видели в предыдущей главе, протоколы реплицируемой записи используются в форме активной репликации или протоколов на основе кворума. Эти решения и применяются для организации набора идентичных процессов в одноранговую группу. Ее главное преимущество состоит в том, что такая группа не имеет единой точки отказа, а ценой за это преимущество является распределенная координация.

Важный вопрос использования групп процессов для повышения отказоустойчивости состоит в том, насколько значительной должна быть репликация. Чтобы упростить наши рассуждения, давайте рассмотрим только систему реплицируемой записи. Система будет называться устойчивой к k отказам, если она останется работоспособной после отказа k компонентов. Если компоненты, называемые также процессами, останавливаются без уведомления, то наличия $k + 1$ процессов достаточно, чтобы обеспечить устойчивость к k отказам. Если k из них просто прекратят работу, будет использован ответ от одного оставшегося.

С другой стороны, если процесс, в котором произошла византийская ошибка, продолжает работать, рассылая ошибочные или просто случайные сообщения, то для того, чтобы обеспечить устойчивость к k отказам, нам необходимо как минимум $2k + 1$ процессов. В наихудшем случае ошибки в k процессах могут случайно (или даже намеренно) породить одинаковые результаты. Однако остальные $k + 1$ процессов также дадут одинаковые результаты, так что клиент или устройство голосования смогут все же поверить большинству.

Разумеется, в теории легко говорить, что система устойчива к k отказам просто потому, что $k + 1$ одинаковых результатов перевесят k одинаковых результатов, но на практике трудно представить себе обстоятельства, в которых можно с определенностью сказать, что k процессов могут ошибаться, а $k + 1$ процессов — не могут. Таким образом, даже в отказоустойчивой системе необходимо что-то вроде статистического анализа.

Подразумеваемое предусловие для этой модели состоит в том, что все запросы приходят ко всем серверам в одинаковом порядке. Это предусловие известно под названием *проблемы атомарной групповой рассылки (atomic multicast problem)*. На самом деле это условие может быть немного ослаблено, поскольку операции чтения нас не интересуют, а значит, некоторые операции записи могут быть завершены и во время чтения, но общая проблема остается. Атомарная групповая рассылка детально рассматривается в следующем разделе.

7.2.3. Соглашения в системах с ошибками

Организация реплицируемых процессов в группы помогает повысить отказоустойчивость. Как мы говорили, если клиент в состоянии принимать решения на основе процесса голосования, мы можем выдержать, даже если k из $2k + 1$ процессов будут выдавать неверные результаты. При этом предполагается, что эти процессы не сговорились нас обмануть.

Если мы нуждаемся в том, чтобы группа процессов соблюдала некое соглашение, дело обычно сильно усложняется. Соглашения необходимы во многих случаях. Вот несколько примеров: выборы координатора, решение о том, завершать транзакцию или нет, разделение задач между рабочими процессами или синхронизация. Если линии связи и процессы находятся в полном порядке, соблюдать соглашение обычно довольно просто, но когда это не так, возникают проблемы.

Основная задача алгоритмов распределенного соглашения состоит в том, чтобы привести все безошибочные процессы к согласию по определенным вопросам и к тому, чтобы достичь этого согласия за конечное число шагов. В зависимости от системных параметров, в частности наличия или отсутствия надежной связи или семантики отказа процессов, возможны различные варианты таких алгоритмов.

Перед тем как обсуждать процессы с ошибками, давайте рассмотрим «простой» случай идеальных процессов, линии связи между которыми могут терять сообщения. Это известнейшая проблема, называемая *проблемой двух армий (two-army problem)*, которая иллюстрирует, насколько трудно двум даже идеальным процессам достичь соглашения относительно 1 бита информации. Армия «зеленых», из 5000 человек, стоит лагерем в долине. Две армии «синих», каждая численностью по 3000 человек, расположены на окружающих долину холмах, с которых просматривается долина. Если эти две армии «синих» в состоянии скоординировать свою атаку на «зеленых», они победят. Однако если каждая из них атакует по отдельности, они будут разбиты. Задача армий «синих» — заключить соглашение об одновременной атаке. Проблема состоит в том, что для связи друг с другом у них имеется только ненадежный канал: они могут посылать сообщения с посыльным, которого «зеленые» могут взять в плен.

Представим себе, что командир одной из армий «синих», генерал Александер, посылает сообщение командиру второй армии «синих», генералу Бонапарту, в котором пишет: «У меня есть план — давайте атакуем завтра на рассвете». Посыльный добирается до места назначения, и Бонапарт отправляет его назад с ответом: «Отличная идея, Алекс. Увидимся завтра, как рассветет». Посыльный невредимым возвращается назад, и Александер приказывает войскам подготовиться к бою на рассвете.

Однако позже в этот же день Александер понимает, что Бонапарт не знает, добрался ли посыльный назад, а не зная этого, может и не решиться на атаку. В результате Александер приказывает посыльному пойти и сказать Бонапарту, что его (Бонапарта) сообщение доставлено и битва состоится.

Посыльный добирается до армии Бонапарта и доставляет подтверждение. Но теперь Бонапарт начинает беспокоиться о том, что Александер не знает, дошло ли до него это подтверждение. По мнению Бонапарта, Александер будет рассуждать так: «Если Бонапарт подумает, что посыльный был взят в плен, он не будет уверен в моих (Александера) планах и может не рискнуть атаковать». И Бонапарт вновь посылает гонца назад.

Сколько бы посыльный ни бегал туда-сюда, нетрудно понять, что Александер и Бонапарт никогда не договорятся между собой, сколько бы соглашений они друг другу не пересылали. Допустим, что это — некий протокол, который ограничен конечным числом шагов. Уберем все дополнительные шаги в конце обмена, чтобы получить минимальный работающий протокол. Какое-то сообщение станет теперь последним, и оно будет абсолютно необходимым для достижения этого соглашения (поскольку это минимальный протокол). Если это сообщение не будет доставлено, война прекратится.

Однако отправитель последнего сообщения не знает, доставлено ли это сообщение. Если оно не доставлено, протокол не завершается и второй генерал не начинает атаку. Таким образом, отправитель последнего сообщения не может узнать, планируется война или нет, а следовательно, не может отдать безопасной команды своим войскам. Поскольку получатель последнего сообщения знает, что отправитель не уверен в результате, он также не хочет идти на верную смерть, и соглашение не заключается. Даже при условии безошибочных процессов (генералов) соглашение между двумя процессами при условии ненадежного взаимодействия невозможно.

Теперь предположим, что связь идеальна, а процессы — нет. Эта классическая задача известна под названием *проблемы византийских генералов* (*Byzantine generals problem*). Согласно условиям этой задачи армия «зеленых» по-прежнему стоит лагерем в долине, а n армий синих под командованием своих генералов расположились за рядом стоящими холмами. Между каждой парой из них налажена телефонная связь, она мгновенна и идеальна. Однако m генералов — предатели (дефектные процессы), которые активно стараются помешать лояльным генералам заключить соглашение, сообщая им заведомо неверную и противоречивую информацию (моделируя неверно функционирующие процессы). Вопрос в том, смогут ли лояльные генералы договориться, невзирая на противодействие.

В целях общности в данном случае мы определим соглашение немного иначе. Каждый генерал знает, сколько у него солдат. Нашей целью будет обмен сведениями о силе войск, то есть после окончания работы алгоритма каждый генерал должен будет получить вектор длины n , соответствующий всем армиям. Если генерал i лоялен, то элемент i содержит численность его группировки, в противном случае он будет не определен.

Рекурсивный алгоритм, предложенный в [252], при определенных условиях решает эту проблему. Рисунок 7.3 иллюстрирует работу этого алгоритма при $n = 4$ и $m = 1$. При таких параметрах алгоритм срабатывает за четыре шага. На шаге 1 каждый генерал посылает (надежным путем) каждому из прочих генералов сообщение, объявляя о численности своей армии. Лояльные генералы говорят правду, предатели могут лгать каждому из генералов по-разному. На рис. 7.3, а мы видим, что генерал 1 сообщает о 1К солдат, генерал 2 — о 2К солдат, генерал 3 лжет каждому из них, указывая x, y и z соответственно, а генерал 4 сообщает о 4К солдат. На шаге 2 результаты, объявленные на шаге 1, собираются вместе в виде векторов (рис. 7.3, б).

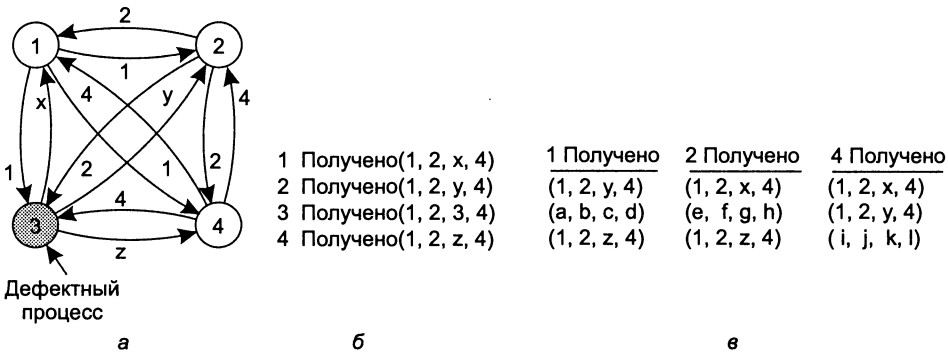


Рис. 7.3. Задача византийских генералов с тремя лояльными генералами и одним предателем. Генералы объявляют численность своих армий в kilosолдатах (а). Векторы, которые собрал каждый из генералов на основе полученной информации (б). Векторы, полученные генералами на шаге 3 (в)

На шаге 3 каждый из генералов посылает свой вектор всем остальным генералам. И снова генерал 3 лжет, придумывая 12 новых значений, от a до l . Результаты шага 3 показаны на рис. 7.3, в. И наконец, на шаге 4 каждый генерал проверяет i -й элемент каждого из присланных ему векторов. Если одно из значений встречается в большинстве присланных векторов, оно помещается в итоговый вектор. Если ни одно из значений не имеет преимущества, соответствующий элемент в итоговом векторе помечается как *UNKNOWN*. На рис. 7.3, в мы видим, что генералы 1, 2 и 4 пришли к соглашению о том, что следующий итоговый вектор является правильным:

(1, 2, UNKNOWN, 4)

Предатель не смог повредить информацию лояльных генералов; он оказался не в состоянии испортить их работу.

Рассмотрим теперь ту же задачу при $n = 3$ и $m = 1$, то есть при двух лояльных генералах и одном предателе, как показано на рис. 7.4. Как мы видим на рис. 7.4, а, ни один из лояльных генералов не может получить большинства значений по элементу 1, элементу 2 или элементу 3, а значит, все они помечаются как *UNKNOWN*. Алгоритм не в состоянии привести группу к соглашению.

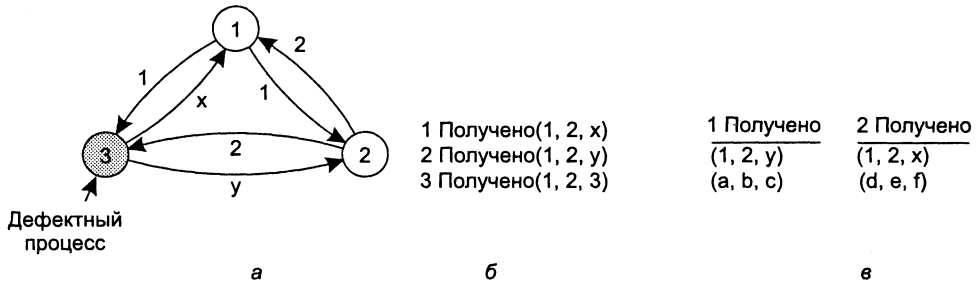


Рис. 7.4. Задача византийских генералов с двумя лояльными генералами и одним предателем

В [252] доказано, что в системе, где m дефектных процессов, соглашение может быть достигнуто только при наличии $2m + 1$ правильно функционирующих процессов, а в сумме процессов должно быть $3m + 1$. Если говорить проще, соглашение достижимо только в том случае, если правильно функционируют более двух третей процессов.

Имеется и другой способ представления этой проблемы. Фактически нам нужно достичь того, чтобы при голосовании большинство принадлежало группе лояльных генералов, несмотря на то, что среди них имеются и предатели. Если имеется m предателей, нам нужно гарантировать, что их голоса вместе с голосами тех лояльных генералов, которых им удалось сбить с толку, не могли перевесить голосов лояльных генералов. При наличии $2m + 1$ лояльных генералов это можно сделать, потребовав, чтобы соглашение было заключено только в том случае, если две трети значений голосов одинаковы. Другими словами, если более двух третей всех генералов пришли к одинаковому решению, это решение соответствует значению, за которое проголосовало большинство группы лояльных генералов.

К сожалению, вариант соглашения может быть еще хуже. В [145] показано, что в распределенных системах невозможно гарантировать доставку сообщений за известное конечное время, а значит, если один из процессов дефектный (особенно если этот процесс прекратил работу без оповещения), то никакое соглашение невозможно. Проблема таких систем состоит в том, что очень медленные процессы неотличимы от неработающих. Существует множество теоретических рассуждений о том, когда подобное соглашение возможно, а когда нет. Отчет об этих результатах можно найти в [37, 465].

7.3. Надежная связь клиент-сервер

Во многих случаях при описании вопросов отказоустойчивости в распределенных системах основное внимание уделяется дефектным процессам. Однако необ-

ходимо также учитывать и дефекты взаимодействия. Большинство из обсуждавшихся ранее моделей отказов одинаково успешно могут быть применены и к каналам связи. Так, в частности, в каналах связи могут иметь место поломки, пропуски, ошибки синхронизации и произвольные ошибки. На практике при строительстве надежных каналов связи основные усилия направлены на маскирование поломок и пропусков. Произвольные ошибки часто имеют вид повторяющихся сообщений, порождаемых в результате того, что в компьютерных сетях сообщения могут надолго «застрять» в промежуточных буферах и вновь попадать в сеть уже после того, как исходный отправитель произвел повторную отправку того же сообщения [446].

7.3.1. Сквозная передача

Во многих распределенных сетях надежная сквозная (point-to-point) передача реализуется путем использования надежного транспортного протокола, такого как TCP. TCP маскирует пропуски, проявляющиеся в виде потери сообщений, с помощью механизма подтверждений и повторных посылок. Эти ошибки остаются абсолютно незамеченными клиентом TCP.

Однако поломки (обрывы) связи часто оказывается невозможно замаскировать. Поломка может произойти, когда по каким-либо причинам соединение TCP внезапно прерывается, так что сообщения по этому каналу больше передаваться не могут. Во многих случаях клиент уведомляется о поломке канала путем возбуждения исключения. Единственный способ замаскировать ошибки такого рода состоит в том, чтобы позволить распределенной системе автоматически установить новое соединение.

7.3.2. Семантика RPC при наличии ошибок

Давайте взглянем поближе на взаимодействие между клиентом и сервером с использованием высокоуровневых коммуникационных механизмов, таких как удаленный вызов процедур (Remote Procedure Call, RPC) или удаленное обращение к методам (Remote Method Invocation, RMI). Далее мы сосредоточимся на RPC, но наше обсуждение также будет приложимо и к связи с удаленными объектами.

Назначение RPC — скрыть сам факт взаимодействия путем вызовов удаленных процедур, которые выглядят так же, как локальные вызовы. С некоторыми исключениями до сих пор мы были к этому очень близки. В самом деле, пока и клиент и сервер работают без ошибок, механизм RPC отлично справляется со своим делом. Проблемы возникают, когда начинаются ошибки. Причина проблем кроется в том, что при наличии ошибок скрыть разницу между локальными и удаленными вызовами гораздо сложнее.

Чтобы систематизировать обсуждение, разделим ошибки, которые могут возникнуть в системах RPC, на пять классов следующим образом:

- ◆ клиент не в состоянии обнаружить сервер;
- ◆ потеря сообщения с запросом от клиента к серверу;

- ◆ поломка сервера после получения запроса;
- ◆ потеря ответного сообщения от сервера к клиенту;
- ◆ поломка клиента после получения ответа.

Ошибки каждого из этих классов ставят перед нами различные задачи, которые требуют различных способов решения.

Клиент не в состоянии обнаружить сервер

Начнем с того, что происходит, если клиент не в состоянии обнаружить подходящий сервер, который может быть, например, отключен. С другой стороны, клиент может быть скомпилирован с некоей конкретной версией клиентской заглушки. Если исполняемый файл длительное время не используется, за это время сервер может обзавестись новой версией интерфейса, могут быть созданы и запущены новые заглушки. В этом случае после запуска клиента может оказаться, что он не соответствует серверу, вызывая сообщение об ошибке. Поскольку подобный механизм используется для защиты клиента от попыток обмена с «неправильным» сервером (сервером, с которым клиент не может договориться об используемых параметрах или решаемых задачах), нужно определиться, что делать с такой ошибкой.

Одно из возможных решений состоит в том, чтобы заставить ошибку возбуждать *исключение (exception)*. В некоторых языках (например, в Java) программисты могут писать специальные процедуры, которые вызываются в случае определенных ошибок, например деления на ноль. Для этой цели можно использовать обработчики сигналов. Другими словами, мы можем определить новый тип сигнала и потребовать его обработки наравне с любыми другими сигналами.

Этот подход, разумеется, имеет свои минусы. Для начала, не все языки поддерживают такие конструкции, как исключения или сигналы. Кроме того, написание процедуры обработки исключения или сигнала нарушит прозрачность, которой мы так старались добиться. Представьте, что вы — программист и ваш босс требует от вас написать процедуру суммирования `sum`. Вы улыбаетесь и говорите ему, что она будет написана, протестирована и задокументирована за пять минут. После этого босс добавляет, что вы должны также написать обработчик исключения на тот случай, если процедуры вдруг не окажется «на месте». В этих условиях очень трудно поддерживать иллюзию, что удаленные процедуры ничем не отличаются от локальных, ведь написание обработчика исключения под условным названием «невозможно найти сервер» — достаточно необычно для однопроцессорной системы. Прозрачность этого «не переживет».

Потеря сообщения с запросом

Второй элемент в нашем списке касается потери сообщений с запросами. С этим справиться проще всего: отправляя сообщение, операционная система или клиентская заглушка просто должна включать таймер. Если таймер переполнится, а ответ или подтверждение так и не будет получен, сообщение посылается повторно. Если сообщение действительно пропало, сервер не увидит разницы между повторной посылкой и оригиналом и все произойдет так, как нужно. Разумеется,

если пропадет такое количество сообщений, что клиент откажется от своей затеи и ошибочно решит, что сервер не работает, мы снова вернемся к ситуации «невозможно найти сервер». Если запрос не пропал окончательно, то нам следует сделать так, чтобы сервер был в состоянии обнаружить, что имеет дело с повторной посылкой. К сожалению, как мы увидим при обсуждении утерянных ответов, сделать это не так-то легко.

Поломка сервера

Следующая ошибка в списке — поломка сервера. Нормальную последовательность событий на сервере иллюстрирует рис. 7.5, а. Запрос приходит, обрабатывается и посылается ответ. Рассмотрим теперь рис. 7.5, б. Запрос приходит и обрабатывается, как и в предыдущем случае, но на сервере происходит поломка до того, как он успевает отправить ответное сообщение. И наконец, посмотрим на рис. 7.6, в. Снова приходит запрос, но на этот раз сервер ломается еще до начала обработки.

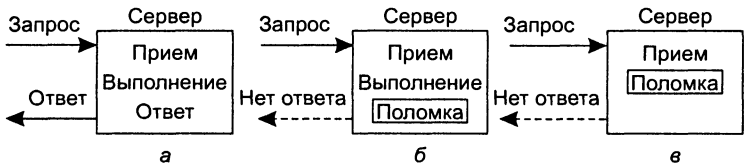


Рис. 7.5. Сервер при взаимодействии клиент-сервер. Нормальная работа (а). Поломка после обработки запроса (б). Поломка до обработки запроса (в)

Досадно, что правильные действия в случаях (б) и (в) различны. В случае (б) система должна передать клиенту сообщение об ошибке (например, возбудить исключение), в то время как в случае (в) она может просто послать запрос повторно. Проблема состоит в том, что операционная система клиента не в состоянии понять, что именно произошло. Ее таймер переполнился — вот все, что ей известно.

Вопрос о том, что делать в данном случае, разные школы решают по-разному [428]. Одна из методик состоит в том, чтобы ожидать перезагрузки сервера (или связаться с новым сервером) и повторить операцию. Идея — повторять попытки до тех пор, пока сервер не выдаст ответ, который дойдет до клиента. Подобный прием называется *семаantikой «минимум однажды»* (*at least once semantics*) и гарантирует, что вызов RPC будет произведен как минимум один раз, а возможно и больше.

Другая методика состоит в том, чтобы немедленно отказаться от дальнейших попыток и вернуть сообщение об ошибке. Этот прием называется *семаantikой «максимум однажды»* (*at most once semantics*) и гарантирует, что вызов RPC будет произведен максимум один раз, а возможно и ни разу.

Третья из методик не гарантирует ничего. Когда на сервере происходит поломка, клиент не получает никакой помощи и никаких гарантий. Вызов RPC может быть не произведен ни разу или произведен любое количество раз. Основное достоинство этой схемы — в простоте ее реализации.

Ни одна из этих идей не выглядит слишком привлекательной. Нам очень понравилась бы *семантика «только однажды»* (*exactly once semantics*), но обычно ее невозможно реализовать. Рассмотрим операцию удаленной печати текста с посылкой сервером сообщения о завершении работы клиенту после завершения печати. Также предположим, что в ответ на посылку запроса клиент получает подтверждение доставки этого запроса на сервер. Сервер может действовать двумя способами. Он может посылать клиенту сообщение об окончании работы еще до того, как на самом деле отправит задание на принтер, или после того, как текст будет напечатан.

Предположим, что сервер был сломан, а затем восстановился. Он оповещает всех клиентов о том, что был сломан, но теперь работает снова. Проблема в том, что клиенты не знают, были или нет обработаны их запросы на печать текста.

Клиент может выбирать одну стратегию из четырех. Во-первых, клиент может никогда не повторять запроса, невзирая на риск, что текст не будет напечатан. Во-вторых, он может всегда повторять запрос, но это может привести к тому, что текст будет напечатан дважды. В-третьих, клиент может решить повторять запрос только в том случае, если он не получал подтверждения доставки запроса на печать на сервер. В этом случае клиент считает, что сервер сломался до получения запроса на печать. Четвертая и последняя стратегия состоит в том, чтобы повторить запрос только в случае получения подтверждения о доставке запроса на печать.

Имея две стратегии у сервера и четыре у клиента, можно составить восемь комбинаций. К сожалению, ни одна из них нас не устраивает. Чтобы объяснить, почему это так, рассмотрим три события, которые должны происходить на сервере, — посылка сообщения о завершении работы (M), печать текста (P) и поломка (C). Эти события могут происходить в шести различных последовательностях.

1. $M \rightarrow P \rightarrow C$: Поломка происходит после отсылки сообщения и печати текста.
2. $M \rightarrow C(\rightarrow P)$: Поломка происходит после отсылки сообщения, но до печати текста.
3. $P \rightarrow M \rightarrow C$: Поломка происходит после отсылки сообщения и печати текста.
4. $P \rightarrow C(\rightarrow M)$: Печатается текст, после чего происходит поломка (до отправки сообщения).
5. $C(\rightarrow P \rightarrow M)$: Поломка происходит до того, как сервер успевает что-либо сделать.
6. $C(\rightarrow M \rightarrow P)$: Поломка происходит до того, как сервер успевает что-либо сделать.

Скобками отделены события, которые не успевают произойти, поскольку сервер к этому времени уже сломан. Рисунок 7.6 иллюстрирует все возможные комбинации. Как легко можно убедиться, не существует такой комбинации стратегий клиента и сервера, которая бы корректно работала при всех возможных последовательностях событий. И последний штрих: клиент никогда не знает, сломался сервер до или после печати текста.

Клиент	Сервер					
	Стратегия М → Р			Стратегия Р → М		
	MPC	MC(P)	C(MP)	MPC	MC(P)	C(MP)
Стратегия повторных посылок						
Всегда	DUP	OK	OK	DUP	DUP	OK
Никогда	OK	ZERO	ZERO	OK	OK	ZERO
Если есть подтверждение	DUP	OK	ZERO	DUP	OK	ZERO
Если нет подтверждения	OK	ZERO	OK	OK	DUP	OK
OK = Текст печатается один раз						
DUP = Текст печатается дважды						
ZERO = Текст вообще не печатается						

Рис. 7.6. Различные комбинации стратегий клиента и сервера при наличии поломок на сервере

Говоря кратко, возможность поломки сервера радикально изменяет природу RPC и проводит четкую грань между однопроцессорными и распределенными системами. В случае однопроцессорной системы поломка сервера подразумевает поломку клиента, причем его восстановление при неработающем сервере бессмысленно. В случае распределенной системы можно и нужно предпринимать соответствующие действия.

Потеря ответного сообщения

С потерей ответных сообщений справиться также нелегко. Очевидное решение — снова положиться на таймер, установленный операционной системой клиента. Если за разумное время не было получено ответа, можно просто послать запрос еще раз. Проблема этого решения состоит в том, что клиент на самом деле не уверен в причине отсутствия ответа. Потерялся запрос, потерялся ответ или просто сервер слишком медленный? Между этими вариантами имеется существенная разница.

Некоторые операции можно без проблем повторять столько раз, сколько нужно, и это не вызовет никаких нарушений. Так, например, запрос на чтение первых 1024 байтов файла не имеет побочных эффектов и может производиться так часто, как это необходимо без каких-либо проблем. Запрос, обладающий такими свойствами, называется *идемпотентным (idempotent)*.

Теперь рассмотрим запрос к банковскому серверу, требующий перевода миллиона долларов с одного счета на другой. Если запрос пришел и был выполнен, но ответное сообщение потерялось, клиент ничего не знает о выполнении запроса и посылает его повторно. Банковский сервер считает, что это новый запрос, и выполняет его. Переводятся два миллиона долларов. Вся надежда на то, что небеса не позволят ответному сообщению потеряться 10 раз. Перевод денег не является идемпотентной операцией.

Единственный способ справиться с этой проблемой — попытаться строить все запросы так, чтобы они были идемпотентными. В действительности, однако,

многие запросы (например, перевод денег) не идемпотентны по своей природе, а значит, необходимы какие-то другие меры. Другой способ состоит в том, что клиент присваивает каждому запросу последовательный номер. Если заставить сервер сохранять номер последнего принятого сообщения каждого из клиентов, работавших с этим сервером, он сможет обнаружить разницу между оригинальным и повторным запросами. Тогда сервер откажется выполнять запрос во второй раз, но сможет повторно послать клиенту ответ. Отметим, что подобный подход требует, чтобы сервер занимался отслеживанием всех клиентов. Дополнительной защитой послужит специальный бит в заголовке сообщения, позволяющий отличать исходные запросы от повторных передач (мы предполагаем, что выполнение исходного запроса всегда безопасно, тогда как его повторение требует осторожности).

Поломка клиента

Последний пункт в списке ошибок — поломка клиента. Что произойдет, если клиент пошлет серверу запрос на некие действия и сломается, прежде чем сервер ответит ему? В этом случае вычисления будут произведены, но у нас не окажется заказчика, ожидающего результата. Такие не имеющие заказчика вычисления называются *сиротами* (*orphans*).

Сироты могут породить разнообразные проблемы. Как минимум, они вызовут излишнюю трату процессорного времени. Они могут также блокировать файлы или как-то иначе связывать полезные ресурсы. И наконец, если клиент перезагрузится и вновь выполнит вызов RPC, а сразу после этого к нему придет ответ от процесса-сироты, может произойти немалый беспорядок.

Что делать с сиротами? В [313] предлагаются четыре решения. В соответствии с решением 1, перед тем как клиентская заглушка пошлет вызов RPC, она создает запись в журнале с описанием того, что происходит. Журнал хранится на диске или другом устройстве долговременного хранения информации, способном пережить перезагрузку. После перезагрузки клиента журналы проверяются и все сироты уничтожаются. Это решение называется *истреблением сирот* (*extermination*).

Недостатки этого сценария в том, что он требует для каждого вызова RPC записи на диск огромного объема информации. Однако он может и не сработать, если сироты сами по себе способны делать вызовы RPC, создавая *внучатых сирот* (*grandorphans*) или сирот еще большей степени, которых трудно или вообще невозможно обнаружить. И наконец, сеть может быть разделена на фрагменты, например, сломавшимся шлюзом, что сделает невозможным истребление сирот, даже если их удастся найти. Все показывает, что это не слишком многообещающий подход.

Согласно решению 2, именуемому *реинкарнацией* (*reincarnation*), все эти проблемы могут быть решены без записи на диск. Способ, которым это делается, предполагает разбиение времени на последовательно пронумерованные эпохи. При перезагрузке клиента он путем широковещательной рассылки отправляет всем машинам сообщение, объявляющее о начале новой эпохи. Когда эта рассылка приходит на сервер, все удаленные вычисления, производимые там по за-

казу этого клиента, прекращаются. Разумеется, если сеть разделена на части, некоторые сироты могут выжить. Однако в ответах от них будет содержаться номер прошедшей эпохи, что предельно упростит их обнаружение.

Решение 3 — это вариант предыдущей идеи, но не настолько драконовский. Он называется *мягкой реинкарнацией* (*gentle reincarnation*). Когда приходит сообщение о смене эпох, каждая машина проверяет, происходят ли на ней какие-либо удаленные вычисления, и если да, пытается найти их владельца. Вычисления прекращаются только в том случае, если владельца найти не удалось.

И, наконец, у нас есть решение 4, *истечение срока* (*expiration*). В этом решении каждому вызову RPC приписывается стандартная продолжительность работы T . Если он не закончил работу, то должен явным образом затребовать следующий срок, что создает определенные неудобства. С другой стороны, если после поломки клиента до его перезагрузки проходит время T , все сироты точно успевают умереть. Проблема решается путем выбора разумного времени T для вызовов RPC с сильно различающимися требованиями.

На практике неприменим ни один из этих методов. Что еще хуже, истребление сирот может привести к непредвиденным последствиям. Предположим, например, что процесс-сирота заблокировал один или более файлов или записей базы данных. Если сироту внезапно истребить, эта блокировка сохранится. Кроме того, сирота может к этому времени породить различные удаленные запросы на запуск других процессов. Таким образом, при истреблении процесса-сироты может оказаться невозможно устранить все следы его деятельности. Истребление сирот подробно рассматривается в [340].

7.4. Надежная групповая рассылка

Мы уже говорили о том, насколько важна устойчивость процессов, достигаемая путем репликации. Не станет неожиданностью и тот факт, что надежные службы групповой рассылки также имеют важное значение. Подобные службы гарантируют, что сообщения будут доставлены всем процессам в группе. К сожалению, реализовать надежную групповую рассылку не просто. В этом разделе мы вплотную займемся вопросами, относящимися к надежной доставке сообщений группе процессов.

7.4.1. Базовые схемы надежной групповой рассылки

Хотя большинство систем транспортного уровня предоставляют в наше распоряжение надежные сквозные каналы, средства для надежного взаимодействия с набором процессов встречаются значительно реже. Самое лучшее, что системы транспортного уровня в состоянии сделать, — это позволить любому процессу создать сквозное соединение с любым другим процессом, с которым ему необходимо связаться. Очевидно, что подобная организация связи не слишком эффективна, поскольку она требует слишком большого расхода пропускной способно-

сти сети. Однако, если число процессов мало, надежная групповая рассылка через несколько надежных сквозных каналов представляет собой самое простое решение, которое часто используется на практике.

Чтобы двигаться дальше, нам следует точно определить, что такое *надежная групповая рассылка* (*reliable multicasting*). Интуитивно мы понимаем, что имеется в виду. Сообщение, отправленное группе процессов, должно быть гарантированно доставлено всем членам этой группы. Однако что произойдет, если процесс станет членом группы непосредственно в момент рассылки? Должен ли этот процесс также получить сообщение? Точно так же мы должны определить, что произойдет, если отправляющий сообщение процесс в процессе взаимодействия откажет.

Чтобы разрешить и эти ситуации, следует провести границу между надежной связью в присутствии ошибочно функционирующих процессов и надежной связью с корректно работающими процессами. В первом случае групповая рассылка считается надежной, если можно гарантировать получение сообщения всеми правильно работающими членами группы. Сложность в том, что вдобавок ко всем ограничениям, связанным с организацией процессов в группе, необходимо выяснить, какие процессы входили в группу до получения сообщения. Мы еще вернемся к этому вопросу, когда будем обсуждать атомарную групповую рассылку.

Ситуация немного прояснится, если допустить, что существует соглашение о том, кто является членом группы. Так, в частности, если предположить, что процессы в ходе сеанса связи не отказывают, не входят в группу и не покидают ее, то надежная групповая рассылка будет просто означать, что любое сообщение доставляется всем текущим членам группы. В простейшем случае между членами группы нет договоренности о доставке сообщений всем членам группы в определенном порядке, но иногда подобное условие необходимо.

Подобная наиболее слабая форма надежной групповой рассылки, относительно просто реализуемая, применяется в том случае, если число приемников мало. Рассмотрим случай, когда один передатчик должен сделать групповую рассылку для нескольких приемников. Предположим, что базовая система связи обеспечивает только ненадежную групповую рассылку, то есть групповые сообщения могут быть частично потеряны, а частично доставлены, но не всем потенциальным потребителям.

Простой способ реализации надежной групповой рассылки в таких условиях иллюстрирует рис. 7.7. Передающий процесс приписывает каждому рассылаемому сообщению последовательный номер. Предположим, что сообщения принимаются в том же порядке, в котором рассылаются. В этом случае получатель с легкостью обнаружит пропажу сообщения. Каждое посылаемое сообщение сохраняется отправителем в локальном буфере истории. Полагая, что получатели знают, кто отправлял сообщение, отправитель просто сохраняет сообщение в буфере до получения подтверждений его приема от всех получателей. Если получатель обнаружит пропажу сообщения, он возвращает отправителю отрицательное подтверждение, запрашивая повторную передачу. С другой стороны, отправитель, не получивший подтверждения в течение заданного срока, может произвести повторную передачу автоматически.

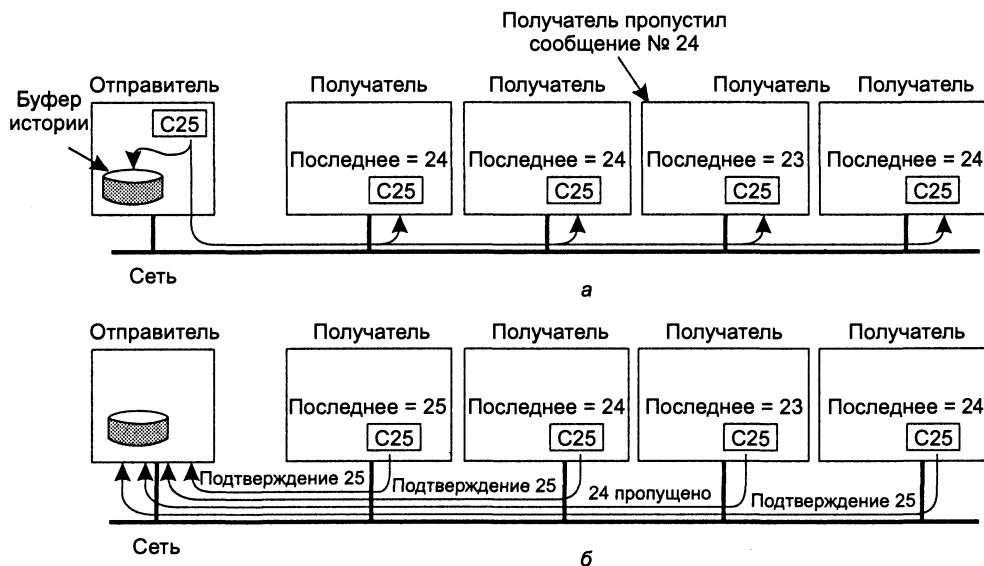


Рис. 7.7. Простой способ получения надежной групповой рассылки в том случае, если предполагается, что получатели известны и безотказны. Передача сообщений (а). Получение подтверждений приема (б)

При разработке могут быть использованы различные хитрости. Так, например, чтобы сократить число возвращаемых отправителю сообщений, подтверждения могут вкладываться в другие сообщения. Кроме того, повторная передача может производиться как по сквозному каналу с каждым из затребовавших ее процессов, так и путем групповой рассылки одного сообщения всем процессам. Дополнительные сведения по этой теме можно найти, например, в [90].

7.4.2. Масштабируемость надежной групповой рассылки

Основная проблема описанной выше схемы групповой рассылки состоит в том, что она не в состоянии работать с большим количеством получателей. При наличии N получателей отправитель должен быть готов обрабатывать как минимум N подтверждений. Если получателей будет много, отправитель может быть просто погребен под ответами. Этот эффект известен под названием *обратный удар* (*feedback implosion*). Кроме того, мы должны принимать во внимание и тот факт, что получатели могут быть разбросаны по всей глобальной сети.

Одно из решений проблемы состоит в том, чтобы запретить получателям подтверждать прием сообщения. Вместо этого получатель должен посылать сообщение отправителю только при потере сообщения. Если возвращать только негативные подтверждения, серьезность проблем с масштабированием будет значительно снижена [462], но твердые гарантии того, что обратный удар никогда не произойдет, по-прежнему отсутствуют.

Другая проблема с получением только негативных подтверждений состоит в том, что отправитель теоретически может быть вынужден вечно хранить сообщения в буфере истории. Поскольку отправитель не в состоянии узнать, было ли сообщение доставлено всем получателям, он должен всегда быть готов к тому, что один из получателей потребует повторно послать ему какое-нибудь древнее сообщение. Практически отправитель должен удалять сообщения из своего буфера истории по истечении определенного срока, что предохранит буфер от переполнения. Однако удаление сообщения всегда чревато тем, что один из запросов на повторную передачу не будет удовлетворен.

Существуют определенные требования к масштабируемым надежным системам групповой рассылки. Сравнение различных схем можно найти в [262]. Сейчас мы кратко разберем два диаметрально противоположных подхода, которые соответствуют многим из существующих решений.

Неиерархическое управление обратной связью

Ключевая задача при создании масштабируемых решений для надежной групповой рассылки — уменьшение числа откликов, получаемых отправителем от получателей. В некоторых глобальных приложениях используется популярная модель *подавления откликов* (*feedback suppression*), которая лежит в основе протокола масштабируемой надежной групповой рассылки (Scalable Reliable Multicasting, SRM), предложенного в [146] и работающего следующим образом.

Прежде всего, в SRM получатель никогда не подтверждает успешного приема сообщения, посылая отклик только в случае потери сообщения. Как приложение отслеживает потерю сообщения, зависит от него самого. Итак, в качестве отклика возвращаются только негативные подтверждения. Когда получатель обнаруживает, что он потерял сообщение, он, пользуясь групповой рассылкой, посылает свой отклик остальным членам группы.

Посланный путем групповой рассылки отклик заставляет других членов группы отказаться от посылки своих собственных сообщений. Допустим, что сообщение m не дошло до нескольких членов группы. Каждый из них должен послать отправителю, S , негативное подтверждение, чтобы тот повторно послал сообщение m . Однако если, как мы говорили, повторная посылка всегда выполняется средствами групповой рассылки, понятно, что для повторной посылки будет вполне достаточно, если S получит один-единственный запрос.

По этой причине получатель R , не получивший сообщения m , планирует посылку отклика с некоторой случайной задержкой. То есть запрос на повторную посылку отправляется только по прошествии некоторого случайного интервала времени. Если, однако, в это время к R приходит другой запрос на повторную посылку, R отменит посылку собственного отклика, зная, что сообщение m скоро и так будет послано повторно. Таким образом, в идеале до S дойдет только одно сообщение, которое, в свою очередь, вызовет повторную посылку m . Описанную схему иллюстрирует рис. 7.8.

Подавление отклика приводит к значительному повышению масштабируемости и используется в качестве базового механизма для множества кооперативных Интернет-приложений, таких как совместно используемые доски объявле-

ний. Однако такой подход порождает множество серьезных проблем. Во-первых, чтобы гарантировать, что отправитель получит только один запрос на повторную посылку, необходимо очень точно планировать отклик каждого из получателей. В противном случае множество получателей могут послать свои отклики одновременно. Установка таймеров в разбросанных по глобальной сети группах процессов — задача непростая.

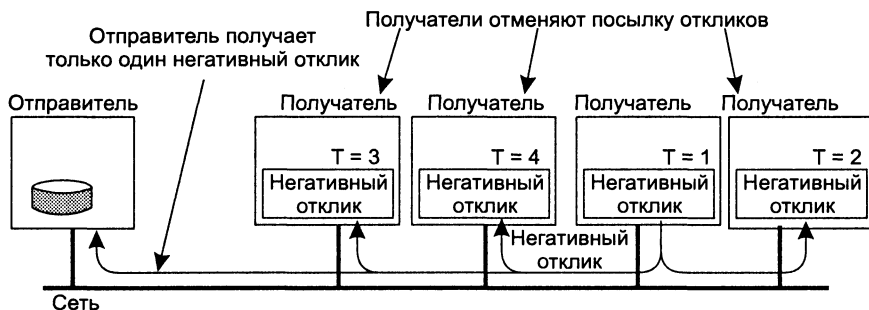


Рис. 7.8. Посылку запросов на повторную передачу планируют несколько получателей, но первый запрос на повторную передачу приводит к отмене остальных

Другая проблема состоит в том, что групповая рассылка отклика прерывает работу и тех процессов, которые успешно получили сообщения. Другими словами, получатели вынуждены принимать и обрабатывать ненужное им сообщение. Единственное решение этой проблемы — выделить получателей, оставшихся без сообщения m , в отдельную группу для групповой рассылки, как было предложено в [226]. К сожалению, это решение требует очень эффективного управления группами, что в глобальных системах трудноосуществимо. Наилучшим подходом поэтому было бы объединить получателей, имеющих тенденцию к пропуску одинаковых сообщений, в группу и использовать один и тот же канал групповой рассылки как для откликов, так и для повторной посылки сообщений. Детали такого подхода можно найти в [270].

Для повышения масштабируемости SRM можно потребовать, чтобы получатели осуществляли локальное восстановление. Так, если получатель, который успешно получил сообщение m , получает затем запрос на повторную посылку, он сам еще до того, как запрос дойдет до истинного отправителя, может решить разослать это сообщение m . Дополнительные подробности можно найти в [146, 270].

Иерархическое управление обратной связью

Описанный выше механизм подавления откликов в основе своей не является иерархическим. Однако чтобы добиться масштабируемости в очень больших группах получателей, необходимо применить иерархический подход. Основы иерархического метода организации групповой рассылки иллюстрирует рис. 7.9.

Для простоты будем считать, что рассылкой сообщений большой группе получателей занимается только один отправитель. Группа получателей разбита на

множество подгрупп, которые организованы в виде дерева. Подгруппа, содержащая отправителя, образует корень дерева. Внутри каждой из подгрупп может использоваться любая схема групповой рассылки, подходящая для малой группы.

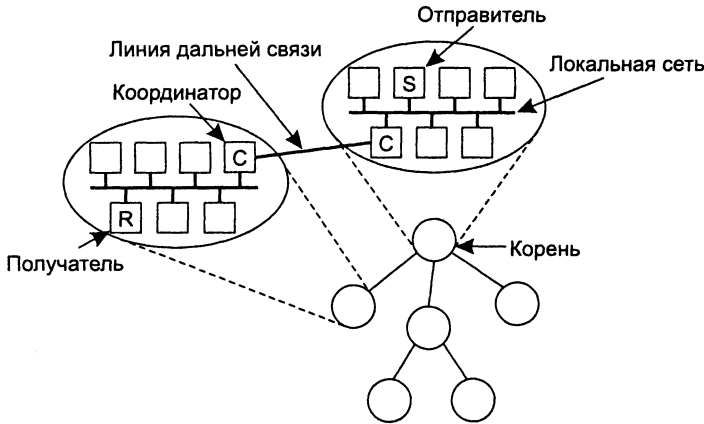


Рис. 7.9. Сущность иерархической надежной групповой рассылки. Каждый локальный координатор пересылает сообщения своим потомкам, а затем обрабатывает запросы на повторную передачу

В каждой подгруппе определяется локальный координатор, который отвечает за обработку запросов на повторную передачу, отправляемых получателями, входящими в эту подгруппу [201]. Локальный координатор поддерживает для этой цели собственный буфер истории. Если сам координатор пропускает сообщение m , он запрашивает повторную рассылку этого сообщения у координатора родительской подгруппы. В схеме с подтверждениями локальный координатор, приняв сообщение, посылает подтверждение своему родителю. Если координатор получил подтверждения приема сообщения m от всех членов своей подгруппы, а также от всех их потомков, он может удалить m из своего буфера истории.

Основная проблема в иерархическом подходе — построение дерева. Во многих случаях дерево должно строиться динамически. Один из способов сделать это — использовать дерево групповой рассылки базовой сети, если оно существует. В принципе для этого достаточно расширить задачу маршрутизаторов групповой рассылки сетевого уровня таким образом, чтобы они могли выполнять функции локальных координаторов. К сожалению, адаптировать таким образом существующую компьютерную сеть не так-то легко.

Мы можем заключить, что построение надежных схем групповой рассылки, которые можно было бы масштабировать на варианты систем с большим числом получателей, распределенных по глобальной сети, представляет собой сложную проблему. Не существует какого-либо наилучшего решения, а каждое из существующих порождает новые проблемы. Таким образом, в этой области предстоит еще немало работы.

7.4.3. Атомарная групповая рассылка

Давайте снова вернемся к ситуации, когда нам требовалась надежная групповая рассылка при наличии ошибок в процессах. Часто в распределенных системах требуется гарантия доставки сообщения либо всем процессам в системе, либо ни одному из них. Кроме того, обычно требуется также, чтобы сообщения доставлялись всем процессам в определенном порядке. Эта проблема известна под названием *проблемы атомарной групповой рассылки (atomic multicast problem)*.

Чтобы понять, почему атомарность так важна, рассмотрим реплицируемую базу данных, разработанную в виде надстройки над распределенной системой. Распределенная система поддерживает механизмы надежной групповой рассылки. В частности, она позволяет создавать группы процессов, которым можно гарантировано посылать сообщения. Реплицируемая база данных, таким образом, построена в виде группы процессов, по одному на реплику. Операция изменения всегда пересылается путем групповой рассылки всем репликам, а затем выполняется локально. Другими словами, мы считаем, что используется протокол активной репликации.

Предположим теперь, что производится серия изменений и в ходе выполнения одного из них случается поломка реплики. Соответственно, обновление этой реплики не происходит, в то время как обновления других реплик происходят успешно.

Естественно, реплика восстанавливается в том виде, в котором она была перед поломкой, то есть в ней не будут учтены обновления, имевшие место после поломки. Поэтому важно привести ее в соответствие с остальными репликами. Приведение реплики в соответствие с другими репликами требует полной информации о том, какие операции она пропустила и в каком порядке эти операции выполнялись.

Теперь предположим, что базовая распределенная система поддерживает атомарную групповую рассылку. В этом случае операция изменения, разосланная всем репликам перед тем, как произошла поломка одной из них, будет выполнена на всех корректно работающих репликах или ни на одной из них. В сущности, в случае атомарной групповой рассылки операция может быть выполнена всеми рабочими репликами, только если они заключат новое соглашение о членстве. Другими словами, изменение данных произойдет только в том случае, если оставшиеся реплики согласятся не считать в дальнейшем сломанную реплику членом группы.

Когда сломанная реплика будет восстановлена, она должна будет снова войти в группу. Она не будет получать операций изменения, пока снова не зарегистрируется в качестве члена группы. Вход в группу требует, чтобы ее состояние было приведено в соответствие с состоянием остальных членов группы. Соответственно, атомарная групповая рассылка гарантирует, что корректно работающие процессы сохраняют базу данных непротиворечивой и способны привести в такое же состояние восстановленную реплику, вновь вошедшую в группу.

Виртуальная синхронность

Надежная групповая рассылка при наличии ошибок в процессах может быть точно определена в терминах групп процессов и изменения членства в группе. Далее

мы определим разницу между *доставкой* и *приемом* сообщений. В частности, мы примем модель, согласно которой распределенная система включает в себя коммуникационный уровень. Схема этой модели приведена на рис. 7.10. Сообщения посылаются и принимаются коммуникационным уровнем распределенной системы. Принятое сообщение помещается в локальный буфер коммуникационного уровня, а затем доставляется приложению, которое логически находится на более высоком уровне.

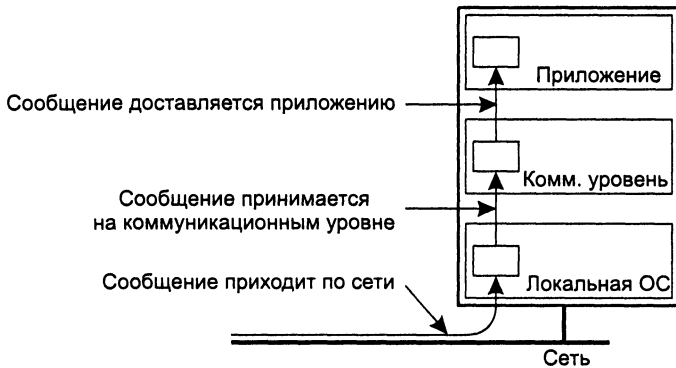


Рис. 7.10. Логическая организация распределенной системы с разделением средств получения и доставки сообщения

Основная идея атомарной групповой рассылки состоит в том, что рассылка членам группы сообщения m однозначно ассоциируется со списком процессов-получателей. Список рассылки соответствует *представлению группы (group view)*, а точнее, представлению составляющих группу набора процессов, с которым работает отправитель в момент рассылки сообщения m . Важно заметить, что каждый процесс в этом списке имеет одинаковое представление. Другими словами, все они согласны с тем, что сообщение m будет доставлено каждому из них, а больше никому.

Теперь предположим, что m рассылается в тот момент, когда отправитель обладает представлением группы G . Кроме того, допустим, что во время рассылки некий процесс входит в группу или покидает ее. Это изменение состава группы объявляется всем процессам, входящим в G . Иначе говоря, происходит *изменение представления (view change)* путем групповой рассылки сообщения vc о входе процесса в группу или выходе из нее. Мы имеем два рассылаемых сообщения, которые находятся в пути одновременно: m и vc . Нам необходимо гарантировать, что m либо будет доставлено всем процессам в G до того, как любой из них получит сообщение vc , либо вообще не будет доставлено. Отметим, что это требование — что-то вроде полностью упорядоченной групповой рассылки, которую мы обсуждали в главе 5.

На ум приходит следующий вопрос: если m не получит ни один из процессов, как мы вообще можем говорить о надежном протоколе групповой рассылки? В принципе имеется только один случай, когда доставка m допускает отказ: когда состав группы изменяется в результате поломки отправителя сообщения m . В этом случае либо все члены G должны получить сообщение об аварийном завер-

шении процесса добавления нового элемента, либо ни один из них. С другой стороны, сообщение m может быть просто проигнорировано всеми членами группы, что будет соответствовать ситуации поломки отправителя до начала рассылки m .

Эта наиболее жесткая форма надежной групповой рассылки гарантирует, что сообщение, рассылаемое в соответствии с представлением группы G , будет доставлено каждому нормально функционирующему процессу, входящему в G . Если отправитель сообщения в ходе рассылки откажет, сообщение будет либо принято, либо проигнорировано всеми оставшимися процессами. Надежная групповая рассылка с такими свойствами [56, 57] носит название *виртуально синхронной* (*virtually synchronous*).

Рассмотрим четыре процесса, представленные на рис. 7.11. В некоторый момент времени процесс P_1 вступает в группу, которая после этого состоит из процессов P_1, P_2, P_3 и P_4 . После рассылки нескольких сообщений процесс P_3 отказывает. До поломки он успевает отправить очередное сообщение процессам P_2 и P_4 , но не P_1 . Однако виртуальная синхронность гарантирует, что сообщение вообще не будет доставлено, успешно имитируя такое положение, как если бы сообщение перед поломкой P_3 вообще не отправлялось.

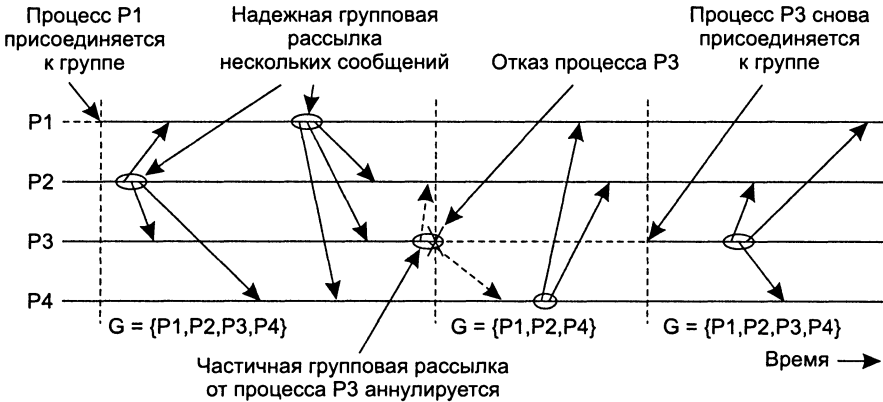


Рис. 7.11. Принцип виртуально синхронной групповой рассылки

После того как процесс P_3 будет удален из группы, взаимодействие между оставшимися в группе членами продолжится. Позже, когда процесс P_3 будет восстановлен, его можно будет вновь включить в группу, а затем актуализировать его состояние.

Принцип виртуальной синхронности вытекает из того факта, что все групповые рассылки происходят в промежутках между изменениями представления. Другими словами, изменения представлений служат барьерами, через которые рассылки пройти не в состоянии. В этом смысле этот принцип соответствует использованию переменных синхронизации в распределенных хранилищах данных, которые мы обсуждали в предыдущей главе. Перед тем как изменение представления вступит в силу, завершаются все групповые рассылки, продолжающиеся во время изменения представления. Реализация виртуальной синхронности, как мы далее увидим, — задача нетривиальная.

Упорядочение сообщений

Виртуальная синхронность позволяет разработчику приложений считать, что групповые рассылки происходят в различные эпохи, отделенные друг от друга изменениями в составе группы. Однако мы ничего не говорили о порядке следования групповых рассылок. В общем можно выделить четыре варианта групповых рассылок с разным порядком следования:

- ◆ неупорядоченные групповые рассылки;
- ◆ групповые рассылки в порядке FIFO;
- ◆ причинно упорядоченные групповые рассылки;
- ◆ полностью упорядоченные групповые рассылки.

Надежная неупорядоченная групповая рассылка (reliable unordered multicast) — это виртуально синхронная групповая рассылка, не дающая никаких гарантий порядка прихода сообщений к различным процессам. Чтобы понять это определение, рассмотрим надежную групповую рассылку, реализованную при помощи библиотеки, в которой имеются примитивы отправки и получения (табл. 7.2). Операция отправки блокирует вызвавший ее процесс до передачи ему сообщения.

Таблица 7.2. Три взаимодействующих в группе процесса

Процесс P1	Процесс P2	Процесс P3
Отправка m1	Получение m1	Получение m2
Отправка m2	Получение m2	Получение m1

Как показано в таблице, пусть процесс *P1* рассылает группе два сообщения, а два других процесса, входящие в группу, ожидают прихода этих сообщений. Будем считать, что ни один из процессов в ходе этой рассылки не ломается и не покидает группу. Может случиться, что коммуникационный уровень процесса *P2* сначала примет сообщение *m1*, а затем *m2*. Поскольку на порядок доставки сообщений не наложено никаких ограничений, сообщения могут быть доставлены *P2* в порядке их приема. С другой стороны, коммуникационный уровень *P3* может сначала принять сообщение *m2*, а затем *m1* и доставить их *P3* в том же порядке.

В случае *надежной групповой рассылки в порядке FIFO (reliable FIFO-ordered multicast)* коммуникационный уровень обязан доставлять входящие сообщения каждому из процессов в том же порядке, в котором они были отправлены. Рассмотрим взаимодействие в группе из четырех процессов, которую иллюстрирует табл. 7.3. При упорядоченности FIFO имеет значение только то, что *m1* всегда доставляется перед *m2*, а *m3*, соответственно, перед *m4*. Это правило соблюдается всеми процессами группы. Другими словами, если коммуникационный уровень *P2* примет сначала сообщение *m2*, он будет ожидать получения *m1*, чтобы принять и доставить его первым.

Однако какие-либо ограничения на доставку сообщений, посланных *разными* отправителями, отсутствуют. Другими словами, если процесс *P2* принимает *m1* перед *m3*, эти сообщения могут быть доставлены ему именно в таком порядке. В то же время процесс *P3* может принять *m3* перед приемом *m1*. Упорядоченность FIFO предполагает, что *m3* может быть доставлено *P3* раньше, чем *m1*, несмотря на то, что этот порядок доставки отличается от порядка доставки для *P2*.

Таблица 7.3. Четыре процесса в группе с двумя разными отправителями, и возможный порядок доставки сообщений при надежной рассылке в порядке FIFO

Процесс P1	Процесс P2	Процесс P3	Процесс P4
Отправка m1	Получение m1	Получение m3	Отправка m3
Отправка m2	Получение m2	Получение m1	Отправка m4
	Получение m3	Получение m2	
	Получение m4	Получение m4	

И, наконец, при *надежной причинно упорядоченной групповой рассылке (reliable causally-ordered multicast)* сообщения доставляются в порядке потенциальной причинной связи между ними. Другими словами, если сообщение *m1* причинно предшествует сообщению *m2*, независимо от того, посылались они одним отправителем или разными, коммуникационный уровень каждого из получателей всегда будет доставлять *m2* после того, как примет и доставит *m1*. Отметим, что причинно упорядоченная групповая рассылка может быть реализована с использованием обсуждавшихся в главе 5 векторных отметок времени.

Кроме этих трех вариантов упорядочения можно ввести дополнительные ограничения, определяющие полную упорядоченность сообщений. *Полностью упорядоченная групповая рассылка (total-ordered multicast)* означает, что независимо от того, как именно упорядочена доставка сообщений (причинно, по FIFO или не упорядочена вовсе), дополнительно требуется, чтобы сообщения доставлялись всем членам группы в одинаковом порядке.

Так, например, в комбинации упорядоченной по FIFO и полностью упорядоченной рассылки процессы *P2* и *P3* (см. табл. 7.3) могут оба получить сначала сообщение *m3*, а затем — сообщение *m1*. Однако если *P2* получает сначала *m1*, а потом *m3*, а *P3* — сначала *m3*, а потом *m1*, то ограничения полной упорядоченности будут нарушены, хотя упорядоченность FIFO будет по-прежнему соблюдена. Другими словами, сообщение *m2* должно быть доставлено после *m1*, а *m4*, соответственно, — после *m3*.

Виртуально синхронная надежная групповая рассылка, поддерживающая полностью упорядоченную доставку, носит название *атомарной групповой рассылки (atomic multicasting)*. Вместе с рассмотренными ранее тремя различными вариантами упорядоченности сообщений это дает нам шесть видов надежной групповой рассылки, которые перечислены в табл. 7.4 [191].

Таблица 7.4. Шесть видов виртуально синхронной надежной групповой рассылки

Групповая рассылка	Базовая упорядоченность сообщений	Полная упорядоченность доставки
Надежная групповая рассылка	Отсутствует	Нет
Групповая рассылка в порядке FIFO	Доставка в порядке FIFO	Нет
Причинно упорядоченная групповая рассылка	Причинно упорядоченная доставка	Нет
Атомарная групповая рассылка	Отсутствует	Да

Групповая рассылка	Базовая упорядоченность сообщений	Полная упорядоченность доставки
Атомарная рассылка в порядке FIFO	Доставка в порядке FIFO	Да
Атомарная причинно упорядоченная рассылка	Причинно упорядоченная доставка	Да

Реализация виртуальной синхронности

Обсудим теперь реализацию виртуально синхронной надежной групповой рассылки. Пример такой реализации имеется в Isis, отказоустойчивой распределенной системе, которая несколько лет использовалась на практике. Давайте рассмотрим некоторые вопросы реализации, описанные также в [58].

Надежная групповая рассылка в Isis основана на использовании имеющихся в базовой сети, в частности в сети TCP, механизмов надежной сквозной связи. Рассылка сообщения m группе процессов реализована путем надежной отправки сообщения m каждому члену группы. Таким образом, хотя любая передача гарантированно успешна, нет никаких гарантий, что сообщение m будет получено всеми членами группы. В частности, отказ отправителя может иметь место до того, как он передаст всем членам группы сообщение m .

Кроме надежной сквозной коммуникации в Isis предполагается, что сообщения из одного источника получаются коммуникационным уровнем в том же порядке, в котором источник их отправляет. На практике это требование разрешается путем использования для сквозной коммуникации соединений TCP.

Основная проблема — гарантировать, что все сообщения, посланные представлению G , будут доставлены всем правильно работающим процессам из G до очередного изменения состава группы. Для этого в первую очередь требуется, чтобы каждый процесс из представления G получил все отправленные туда сообщения. Отметим, что поскольку отправитель сообщения m в представление G может отказать до завершения рассылки, в G могут найтись процессы, которые никогда не получают m . Поскольку отправитель отказал, эти процессы должны получить m откуда-то еще. Далее мы рассмотрим, как процесс обнаруживает потерю сообщения.

Решение этой проблемы состоит в том, чтобы каждый процесс в G сохранял m до тех пор, пока он не будет точно знать, что это сообщение получено всеми членами G . Если сообщение m получено всеми членами представления G , m называют *устойчивым* (*stable*) сообщением. Доставлять разрешается только устойчивые сообщения. Чтобы гарантировать устойчивость, достаточно выбрать в G случайный рабочий процесс и потребовать от него отправки сообщения m всем остальным процессам.

Для большей конкретности предположим, что текущее представление — G_i , но имеется необходимость установить новое представление, G_{i+1} . Без потери общности можно считать, что G_i и G_{i+1} различаются максимум одним процессом. Процесс P узнает об изменении представления, получая соответствующее сообщение. Это сообщение может прийти от процесса, желающего войти в группу или покинуть ее, или от процесса, обнаружившего отказ одного из процессов,

входящих в G . Сбойный процесс после этого должен быть удален, как показано на рис. 7.12, а.

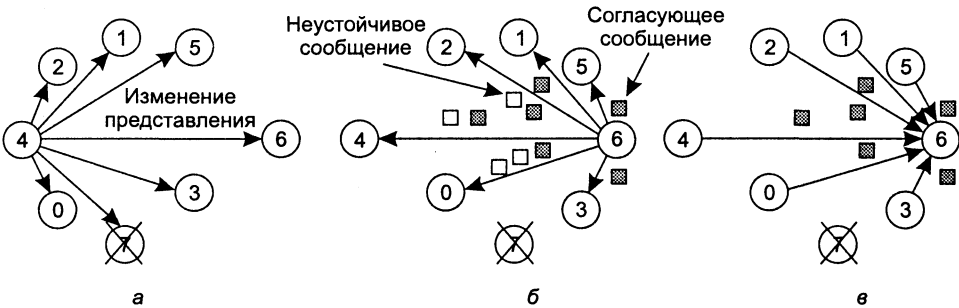


Рис. 7.12. Процесс 4 извещает, что процесс 7 отказал, рассылая сообщение об изменении представления (а). Процесс 6 рассылет все свои неустойчивые сообщения, сопровождая их согласующим сообщением (б). Процесс 6 устанавливает новое представление, получив согласующее сообщение от кого-либо еще (в)

Когда процесс P получает сообщение об изменении представления на G_{i+1} , он в первую очередь пересылает в G копии всех неустойчивых сообщений, чтобы они дошли до каждого процесса, входящего в G , после чего помечает их как устойчивые. Напомним, что в Isis сквозные взаимодействия предполагаются надежными, так что пересылаемые сообщения никогда не теряются. Таким образом, при пересылке гарантируется, что все сообщения, отправленные в G и полученные хотя бы одним процессом, будут получены всеми правильно работающими процессами в G . Отметим, что для пересылки неустойчивых сообщений достаточно выбрать одного координатора.

Чтобы обозначить, что процесс P больше не имеет неустойчивых сообщений и готов установить представление G_{i+1} , как только это смогут сделать другие процессы, он рассылет в представление G_{i+1} *согласующее сообщение* (*flush message*), как показано на рис. 7.12, б. После того как P примет согласующее сообщение для представления G_{i+1} от какого-либо другого процесса, он может благополучно установить новое представление, как показано на рис. 7.12, в.

Если процесс Q будет полагать, что текущее представление — G_i , то, получив сообщение m , посланное в соответствии с G_i , он доставит его, принимая во внимание все дополнительные ограничения на порядок сообщений. Если окажется, что он уже принимал m , он сочтет это сообщение повтором и пропустит его.

Поскольку процесс Q в конце концов примет сообщение о смене представления на G_{i+1} , он также сначала перешлет все свои неустойчивые сообщения, после чего объявит об окончании этой работы посылкой согласующего сообщения членам представления G_{i+1} . Отметим, что согласно порядку сообщений, который установлен базовым коммуникационным уровнем, согласующее сообщение всегда принимается после неустойчивого сообщения, присланного тем же процессом.

Главный недостаток описанного протокола состоит в том, что он не в состоянии сделать что-либо со сбоями процессов в момент объявления нового изменения представления. Таким образом, мы предполагаем, что до тех пор, пока новое представление G_{i+1} не будет установлено каждым членом представления G_{i+1} , ни

один процесс из G_{i+1} не откажет (отказ процесса приведет к возникновению нового представления G_{i+2}). Эта проблема решается путем объявления об изменении представлений для любого представления G_{i+k} еще до того, как предыдущие изменения будут установлены всеми процессами. Детали мы оставляем читателю в качестве самостоятельного упражнения.

7.5. Распределенное подтверждение

Задача атомарной групповой рассылки, которую мы обсуждали в предыдущей главе, — это пример более общей задачи, известной под названием *распределенного подтверждения* (*distributed commit*). Задача распределенного подтверждения включает в себя операции, производимые либо с каждым членом группы процессов, либо ни с одним из них. В случае надежной групповой рассылки операцией будет доставка сообщения. В случае распределенных транзакций операцией будет подтверждение транзакции на одном из сайтов, задействованных в транзакции. Другие примеры распределенного подтверждения и способы их разрешения обсуждаются в [451].

Распределенное подтверждение часто организуется при помощи координатора. В простой схеме координатор сообщает всем остальным процессам, которые также участвуют в работе (они называются участниками), в состоянии ли они локально осуществить запрашиваемую операцию. Эта схема известна под названием *протокола однофазного подтверждения* (*one-phase commit protocol*). Он обладает одним серьезным недостатком. Если один из участников на самом деле не может осуществить операцию, он не в состоянии сообщить об этом координатору. Так, например, в случае распределенных транзакций локальное подтверждение может оказаться невозможным из-за того, что оно будет нарушать ограничения управления параллельным выполнением.

Для практического применения необходима более сложная схема. Обычно используется протокол двухфазного подтверждения, который детально рассматривается ниже. Основной его недостаток состоит в том, что он не в состоянии эффективно справляться с ошибками координатора. Поэтому был разработан протокол трехфазного подтверждения, который мы также будем рассматривать.

7.5.1. Двухфазное подтверждение

Первоначально *протокол двухфазного подтверждения* (*Two-phase Commit Protocol, 2PC*) был описан в [178]. Рассмотрим распределенную транзакцию, предполагающую участие множества процессов, каждый из которых работает на отдельной машине. Если предположить, что ошибки отсутствуют, протокол состоит из следующих двух фаз, каждая из которых включает в себя два шага [47].

1. Координатор рассылает всем участникам сообщение *VOTE_REQUEST*.
2. После того как участник получит сообщение *VOTE_REQUEST*, он возвращает координатору либо сообщение *VOTE_COMMIT*, указывая, что он готов ло-

кально подтвердить свою часть транзакции, либо сообщение *VOTE_ABORT* в противном случае.

3. Координатор собирает ответы участников. Если все участники проголосовали за подтверждение транзакции, координатор начинает осуществлять соответствующие действия и посылает всем участникам сообщение *GLOBAL_COMMIT*. Однако если хотя бы один участник проголосовал за прерывание транзакции, координатор принимает соответствующее решение и рассылает сообщение *GLOBAL_ABORT*.
4. Каждый из участников, проголосовавших за подтверждение, ожидает итогового решения координатора. Если участник получает сообщение *GLOBAL_COMMIT*, он локально подтверждает транзакцию. В случае же получения сообщения *GLOBAL_ABORT* транзакция локально прерывается.

Первая фаза (фаза голосования) состоит из шагов 1 и 2, вторая (фаза решения) — из шагов 3 и 4. Эти четыре шага показаны на диаграмме конечного автомата (рис. 7.13).

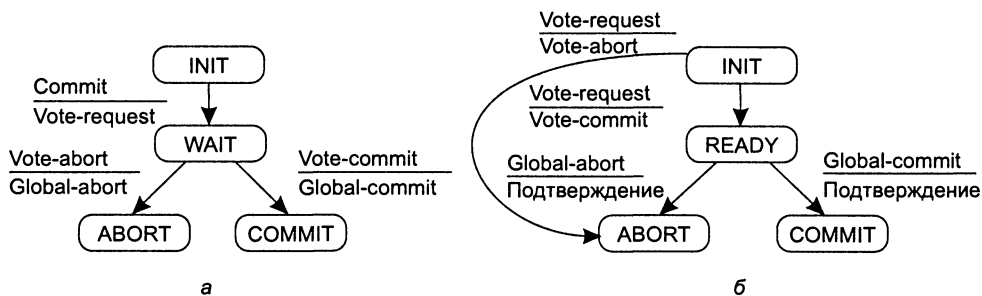


Рис. 7.13. Координатор в протоколе 2PC в виде конечного автомата (а). Участник в виде конечного автомата (б)

Когда при использовании базового протокола 2PC в системе появляются ошибки, возникают определенные проблемы. В первую очередь, отметим, что как координатор, так и участники имеют такие состояния, в которых они блокируются и ожидают прихода сообщений. Соответственно, если процесс отказывает, работа протокола легко может быть нарушена, поскольку другие процессы будут бесконечно ожидать от него сообщения. По этой причине вводятся механизмы тайм-аута.

Глядя на конечные автоматы, представленные на рисунке, можно заметить, что всего существует три состояния, в которых координатор или участник могут, будучи заблокированными, ожидать прихода сообщения. Во-первых, участник может, находясь в начальном состоянии *INIT*, ожидать сообщения *VOTE_REQUEST* от координатора. Если в течение некоторого времени этого сообщения нет, участник приходит к выводу о необходимости локального прерывания транзакции и посылает координатору сообщение *VOTE_ABORT*.

Точно так же координатор, заблокированный в состоянии *WAIT*, ожидает прихода голосов всех участников. Если в течение некоторого времени не будут

собраны все голоса, координатор решает, что транзакцию необходимо прервать, и рассылает всем участникам сообщение *GLOBAL_ABORT*.

И, наконец, участник может быть заблокирован в состоянии *READY*, ожидая глобального объявления результатов голосования, рассылаемого координатором. Если это сообщение за определенный период времени не приходит, участник не может просто решить прервать транзакцию. Вместо этого он должен определить, какое же сообщение на самом деле посылал координатор. Самым простым решением этого вопроса будет блокировка участника до момента восстановления координатора.

Более правильным решением будет разрешить участнику *P* контакт с другим участником, *Q*, чтобы он мог по текущему состоянию *Q* решить, что же ему делать дальше. Так, например, пусть *Q* перешел в состояние *COMMIT*. Это возможно только в том случае, если координатор до поломки послал *Q* сообщение *GLOBAL_COMMIT*. Видимо, до *P* это сообщение не дошло. Соответственно, *P* теперь может прийти к выводу о необходимости локального подтверждения. Точно так же если *Q* находится в состоянии *ABORT*, *P* может уверенно прерывать свою часть транзакции.

Теперь предположим, что *Q* пребывает в состоянии *INIT*. Такая ситуация может произойти, если координатор разошлет всем участникам сообщения *VOTE_REQUEST* и это сообщение дойдет до *P* (который ответит на него посылкой сообщения *VOTE_COMMIT*), но не дойдет до *Q*. Другими словами, координатор может отказать во время рассылки сообщения *VOTE_REQUEST*. В этом случае правильно будет прервать транзакцию: и *P*, и *Q* могут осуществить переход в состояние *ABORT*.

Наиболее сложную ситуацию мы получим, если *Q* также будет находиться в состоянии *READY*, ожидая ответа от координатора. Так, если окажется, что все участники находятся в состоянии *READY*, конкретное решение принять невозможно. Проблема состоит в том, что все участники готовы подтвердить транзакцию, но для этого им необходим голос координатора. Соответственно, протокол блокируется до момента восстановления координатора.

Различные варианты иллюстрирует табл. 7.5.

Таблица 7.5. Действия, предпринимаемые участником *P* в состоянии *READY* и контакте с участником *Q*

Состояние участника <i>Q</i>	Действие участника <i>P</i>
COMMIT	Перейти в состояние COMMIT
ABORT	Перейти в состояние ABORT
INIT	Перейти в состояние ABORT
READY	Связаться с другим участником

Чтобы гарантировать, что процесс действительно восстановился, нужно чтобы он сохранял свое состояние при помощи средств длительного хранения данных (как сохранять защищенным от отказов способом, мы поговорим позже в этой главе). Так, например, если участник находился в состоянии *INIT*, то после восстановления он может решить локально прервать транзакцию и сообщить об

этом координатору. Точно так же если он уже принял решение и в момент поломки находился в состоянии *COMMIT* или *ABORT*, то при восстановлении он снова придет в выбранное состояние и повторно сообщит свое решение координатору.

Проблемы возникают, когда участник отказывает, находясь в состоянии *READY*. В этом случае при восстановлении он не может определить на основании собственной информации, что делать дальше, завершать или прерывать транзакцию. Соответственно, чтобы выяснить, что делать этому процессу, его следует заставить связаться с другими участниками, как это проделывалось в ситуации истечения тайм-аута в состоянии *READY*, описанной ранее.

Координатор имеет только два критических состояния, за которыми надо следить. Когда он начинает протокол 2PC, он должен сохранить сведения о том, что он находится в начальном состоянии *WAIT*, чтобы иметь возможность при необходимости после восстановления повторно разослать сообщение *VOTE_REQUEST* всем участникам. Кроме того, если в ходе второй фазы он принял решение, следует сохранить это решение, чтобы после восстановления координатора его также можно было разослать повторно.

Схема действий координатора иллюстрирует листинг 7.1. Координатор начинает с групповой рассылки всем участникам сообщения *VOTE_REQUEST* для того, чтобы собрать их голоса. Он записывает это действие, после чего входит в состояние *WAIT*, в котором ожидает прихода голосов участников.

Листинг 7.1. Схема действий, предпринимаемых координатором согласно протоколу двухфазного подтверждения

ДЕЙСТВИЯ КООРДИНАТОРА:

```

записать START_2PC в локальный журнал;
разослать VOTE_REQUEST всем участникам;
пока собраны не все голоса {
    ожидать прихода голосов;
    если время вышло {
        записать GLOBAL_ABORT в локальный журнал;
        разослать GLOBAL_ABORT всем участникам;
        выход;
    }
    записать голос;
}
если все участники прислали VOTE_COMMIT и координатор голосует COMMIT {
    записать GLOBAL_COMMIT в локальный журнал;
    разослать GLOBAL_COMMIT всем участникам;
} в противном случае {
    записать GLOBAL_ABORT в локальный журнал;
    разослать GLOBAL_ABORT всем участникам;
}

```

Если собраны не все голоса, а недостающие в течение заданного времени не присланы, координатор приходит к выводу, что один или более участников находятся в нерабочем состоянии. Это означает, что он должен прервать транзакцию и разослать остальным участникам сообщение *GLOBAL_ABORT*.

Если не произошло никаких отказов, координатор в конечном итоге соберет все голоса. Если все участники, так же как и координатор, проголосовали за под-

тверждение, сообщение *GLOBAL_COMMIT* будет сначала записано в журнал, а потом разослано всем процессам. В противном случае координатор разошлет сообщение *GLOBAL_ABORT* (предварительно также записав его в локальный журнал).

В листинге 7.2 приведены действия, осуществляемые участником. В начале процесс ожидает запроса на голосование от координатора. Отметим, что это ожидание выполняется в отдельном потоке выполнения, выполняющемся в адресном пространстве процесса. Если сообщение не поступает, транзакция просто прерывается. По всей вероятности, причина отсутствия сообщения состоит в том, что координатор отказал.

Листинг 7.2. Схема действий, предпринимаемых процессом-участником протокола 2PC

ДЕЙСТВИЯ УЧАСТНИКА:

```
записать состояние INIT в локальный журнал;
ожидать VOTE_REQUEST от координатора;
если время вышло {
    записать GLOBAL_ABORT в локальный журнал;
    выход;
}
если участник голосует за COMMIT {
    записать VOTE_COMMIT в локальный журнал;
    послать VOTE_COMMIT координатору;
    ожидать решение от координатора;
    если время вышло {
        разослать DECISION_REQUEST другим участникам;
        ожидать прихода решения; /*процесс блокирован*/
        записать решение в локальный журнал;
    }
    если решение == GLOBAL_COMMIT
        записать GLOBAL_COMMIT в локальный журнал;
    иначе если решение == GLOBAL_ABORT
        записать GLOBAL_ABORT в локальный журнал;
} если {
    записать VOTE_ABORT в локальный журнал;
    послать VOTE_ABORT координатору;
}
```

После получения запроса на голосование участник может решить голосовать за подтверждение транзакции, для чего ему следует сначала записать это решение в локальный журнал, а затем проинформировать о нем координатора путем отправки ему сообщения *VOTE_COMMIT*. После этого участник должен дожидаться глобального решения. Получив это решение (которое опять-таки должно прийти от координатора), он записывает его в локальный журнал, а затем выполняет.

Однако если участник ожидает решения координатора весь отведенный на это срок и не получает его, он выполняет протокол прекращения ожидания, сначала рассылая остальным процессам сообщения *DECISION_REQUEST*, а затем блокируясь в ожидании ответа. Когда приходит ответ (возможно, от координатора, который уже успел восстановиться), участник записывает решение в локальный журнал и поступает в соответствии с ним.

Каждый участник должен быть готов к тому, что другой участник запросит у него глобальное решение. Для этого каждый из участников должен запустить отдельный поток выполнения параллельно с главным потоком выполнения участника, как показано в листинге 7.3. Этот поток блокируется до получения запроса о решении. Он может ответить другим процессам, только если ассоциированный с ним участник уже принял конечное решение. Другими словами, если в локальный журнал было записано сообщение *GLOBAL_COMMIT* или *GLOBAL_ABORT*, можно быть уверенным, что координатор послал свое решение как минимум этому процессу. Кроме того, как мы говорили ранее, что если ассоциированный с потоком участник находится в состоянии *INIT*, он может решить послать сообщение *GLOBAL_ABORT*. Во всех остальных случаях этот поток ничем не сможет помочь задавшему вопрос процессу, и пославший запрос участник останется без ответа.

Листинг 7.3. Схема обработки входящих запросов о решении

ДЕЙСТВИЯ ПО ОБРАБОТКЕ ЗАПРОСА РЕШЕНИЯ:

```
/*выполняются в отдельном потоке выполнения*/
пока истина {
    ожидать любого сообщения DECISION_REQUEST: /*процесс блокирован*/
    считать последнее записанное состояние из локального журнала
    если состояние == GLOBAL_COMMIT
        послать GLOBAL_COMMIT запросившему решение участнику
    иначе если состояние == INIT или состояние == GLOBAL_ABORT
        послать GLOBAL_ABORT запросившему решение участнику
    иначе
        пропустить /*участник блокирован*/
}
```

Как можно заметить, возможна ситуация, когда участнику придется установить блокировку до восстановления координатора. Такая ситуация может возникнуть, если все участники получают и обработают сообщение координатора *VOTE_REQUEST* и в ходе этой работы координатор сломается. В этом случае участники не смогут совместно решить, какое действие им предпринять. По этой причине протокол 2PC также называют *протоколом блокирующего подтверждения* (*blocking commit protocol*).

Существует несколько способов избежать блокировки. Одно из них, описанное в [22], состоит в использовании примитива групповой рассылки, когда получатель немедленно рассылает ответное сообщение всем остальным процессам. Можно показать, что подобный подход позволяет участнику принять итоговое решение даже в том случае, если координатор еще не восстановился. Другое решение — использование протокола трехфазного подтверждения, который является последней темой этого раздела.

7.5.2. Трехфазное подтверждение

Проблема протокола двухфазного подтверждения состоит в том, что при поломке координатора участники могут оказаться не в состоянии прийти к итоговому решению. В результате участникам приходится дожидаться восстановления коор-

динатора, находясь в заблокированном состоянии. В [422] был предложен вариант протокола 2PC, названный *протоколом трехфазного подтверждения (Three-phase Commit Protocol, 3PC)*, который предотвращает блокировку процессов при появлении ошибок аварийной остановки. Хотя протокол 3PC часто упоминается в литературе, на практике он используется редко, потому что редко возникают условия, в которых блокируется 2PC. Мы рассмотрим этот протокол, поскольку он представляет собой дальнейшее развитие решений проблем отказоустойчивости в распределенных системах.

Как и 2PC, 3PC формулируется в терминах координатора и набора участников. Соответствующие им конечные автоматы показаны на рис. 7.14. Сущность протокола состоит в том, что состояния координатора и любого из участников удовлетворяют двум условиям.

- ♦ Не существует такого состояния, из которого может быть осуществлен прямой переход как в состояние *COMMIT*, так и в состояние *ABORT*.
- ♦ Не существует такого состояния, в котором невозможно принять итоговое решение, но возможен переход в состояние *COMMIT*.

Можно показать, что эти два условия необходимы и достаточны для того, чтобы протокол подтверждения был неблокирующим [424].

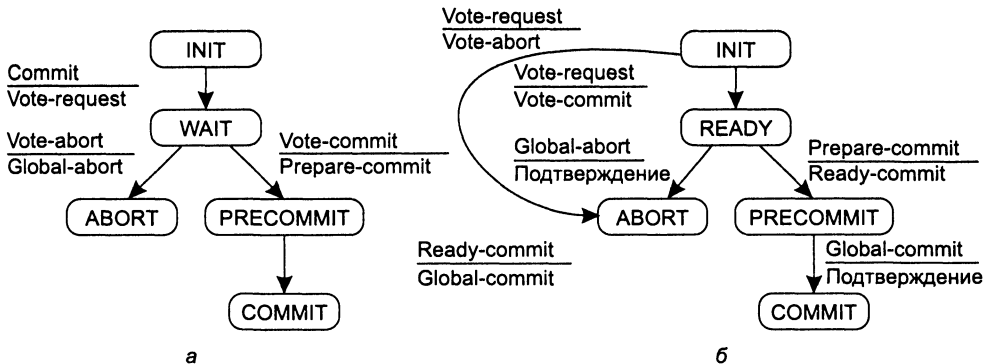


Рис. 7.14. Координатор в протоколе 3PC в виде конечного автомата (а). Участник в виде конечного автомата (б)

Координатор 3PC начинает с рассылки всем участникам сообщения *VOTE_REQUEST*, после чего ожидает прихода ответов. Если хотя бы один участник голосует за прерывание транзакции, это становится итоговым решением и координатор рассылает участникам сообщение *GLOBAL_ABORT*. Однако если транзакция может быть подтверждена, рассылается сообщение *PREPARE_COMMIT*. Только после того, как все участники подтвердят свою готовность к подтверждению, координатор посылает итоговое сообщение *GLOBAL_COMMIT*, в результате которого транзакция действительно подтверждается.

И снова существует лишь несколько ситуаций, в которых процесс блокируется и ожидает сообщений. Во-первых, когда участник, находясь в состоянии *INIT*, ожидает прихода запроса на голосование, он может в итоге перейти в состояние

ABORT, что будет означать поломку координатора. Эта ситуация идентична ситуации для протокола 2PC. Также аналогично координатор может находиться в состоянии *WAIT*, ожидая голосов участников. По истечении заданного времени координатор решает, что участник отказал, и обрывает транзакцию путем рассылки сообщения *GLOBAL_ABORT*.

Рассмотрим теперь блокировку координатора в состоянии *PRECOMMIT*. В случае истечения заданного времени он приходит к выводу, что один из участников отказал, но прочие участники знают, что они голосовали за подтверждение транзакции. Соответственно, координатор приказывает работающим участникам подтвердить транзакцию путем рассылки сообщения *GLOBAL_COMMIT*. Кроме того, он полагает, что протокол восстановления отказавшего участника поможет подтвердить его часть транзакции после того, как этот участник будет восстановлен.

Участник *P* может блокироваться в состоянии *READY* или *PRECOMMIT*. После окончания отведенного участнику *P* на ожидание времени, *P* в состоянии понять, что координатор отказал и нужно решать, что делать дальше. Как и в 2PC, если *P* при контакте с другим участником обнаружит, что он находится в состоянии *COMMIT* (или *ABORT*), *P* также переходит в это состояние. Кроме того, если все участники находятся в состоянии *PRECOMMIT*, транзакция может быть без проблем подтверждена.

Опять-таки, аналогично схеме 2PC, если другой участник *Q* по-прежнему находится в состоянии *INIT*, транзакция может быть совершенно спокойно прервана. Важно отметить, что *Q* может находиться в состоянии *INIT* только в том случае, если ни один из участников не находится в состоянии *PRECOMMIT*. Участник может перейти в состояние *PRECOMMIT*, только если координатор перед сбоем сам перешел в состояние *PRECOMMIT*, а значит, получил голоса всех участников. Другими словами, участник не может оставаться в состоянии *INIT*, когда другой участник переходит в состояние *PRECOMMIT*.

Если каждый из участников, с которым может связаться *P*, находится в состоянии *READY* (и вместе они образуют большинство), транзакция должна быть прервана. Здесь следует отметить, что если один из участников отказывает, позже он будет восстановлен. Однако ни *P*, ни любой другой активный участник не знают, в каком состоянии будет находиться отказавший участник после восстановления. Если процесс будет восстановлен в состоянии *INIT*, решение о прерывании транзакции — единственно верное. В наихудшем случае процесс может восстановиться в состоянии *PRECOMMIT*, но и в этом случае он не сможет помешать прерыванию транзакции.

Этим ситуация сильно отличается от протокола 2PC, где отказавший участник может восстановиться в состоянии *COMMIT*, тогда как все прочие участники будут находиться в состоянии *READY*. В этом случае оставшиеся рабочие процессы не смогут принять итогового решения и вынуждены будут ожидать восстановления отказавшего процесса. В 3PC, если какой-то из рабочих процессов находится в состоянии *READY*, отказавшие процессы не смогут восстановиться ни в каком другом состоянии, кроме *INIT*, *ABORT* или *PRECOMMIT*. По этой причине оставшиеся невредимыми процессы всегда смогут прийти к итоговому решению.

И, наконец, если процессы, такие как P , могут достичь состояния *PRECOMMIT* (и они составляют большинство), то можно без проблем подтвердить транзакцию. И снова мы можем видеть, что все остальные процессы либо находятся в состоянии *READY*, либо, как минимум, если они отказали, будут восстановлены в состоянии *READY*, *PRECOMMIT* или *COMMIT*.

Дополнительные подробности по ЗРС можно найти в [47, 103].

7.6. Восстановление

Теперь мы рассмотрим алгоритмы, позволяющие противостоять отказам. Следует понимать, что процесс, в котором произошел отказ, можно восстановить, приведя его в корректное состояние. Из этого следует, что мы сосредоточимся сначала на том, что такое «привести в корректное состояние», а уж затем — где и как в случае распределенных систем это состояние может быть зафиксировано и как восстановить его, используя контрольные точки и протоколирование сообщений.

7.6.1. Основные понятия

Основа отказоустойчивости — исправление после ошибок. Напомним, что ошибкой считается отказ части системы. Основная идея механизма исправления ошибок состоит в замене ошибочного состояния безошибочным. Существует два основных способа восстановления после ошибок.

При *обратном исправлении* (*backward recovery*) основная задача состоит в возвращении системы из текущего ошибочного состояния к предыдущему безошибочному состоянию. Чтобы сделать это, необходимо время от времени записывать состояние системы и восстанавливать ее в предыдущем состоянии, если дело пойдет плохо. При каждой записи текущего состояния системы (или его части) говорят, что создается *контрольная точка* (*checkpoint*).

Другой вариант исправления ошибок — *прямое исправление* (*forward recovery*). В этом случае при входе системы в ошибочное состояние вместо отката назад, к предыдущей контрольной точке, делается попытка перевести систему в новое корректное состояние, в котором она могла бы продолжать работать. Основная проблема механизмов прямого исправления ошибок состоит в том, что для них необходимо заранее знать о возможных ошибках. Только в этом случае удастся исправить эти ошибки и перейти в новое состояние.

Разницу между прямым и обратным механизмами исправления ошибок легко понять на примере реализации надежной связи. Основной способ восстановления потерянного пакета — попросить отправителя послать этот пакет повторно. В действительности повторная передача пакета означает отход назад, к предыдущему правильному состоянию, а именно к состоянию, когда начинал передаваться потерянный пакет. Таким образом, надежная связь, организуемая посредством повторной отправки пакетов, — это пример обратного исправления ошибок.

Альтернативный подход состоит в использовании так называемого метода коррекции разрушенной информации. В этом случае потерянный пакет создает-

ся из других, успешно доставленных пакетов. Так, например, при использовании блочного кодирования от потерь (n, k) набор из k исходных пакетов преобразуется в набор n декодированных пакетов, так что любой набор из k декодированных пакетов подходит для реконструкции k исходных пакетов. Типичные значения $k = 16$ и 32 , причем $k < n \leq 2k$ [383]. Если доставленных пакетов оказывается недостаточно, отправителю придется продолжать передавать пакеты до тех пор, пока не удастся восстановить ранее потерянный пакет. Коррекция разрушенной информации представляет собой типичный пример прямого исправления ошибки.

Вообще говоря, для исправления ошибок в распределенных системах в качестве основного механизма чаще используются методы обратного исправления. Важное преимущество обратного исправления ошибок состоит в том, что это общий метод, который не зависит от конкретной системы или процесса. Другими словами, он может быть интегрирован в распределенную систему, например в ее промежуточный уровень, в виде службы общего назначения.

Однако обратное исправление ошибок также создает определенные проблемы [421]. Во-первых, восстановление системы или процесса в предыдущем состоянии — дело достаточно затратное с точки зрения производительности. Как мы увидим далее, обычно восстановление, например, отказавшего процесса или сайта, обычно требует большого объема работы.

Во-вторых, поскольку механизмы обратного исправления ошибок не зависят от конкретного распределенного приложения, в котором они используются, невозможно дать какие-либо гарантии того, что после исправления ошибок та же самая или похожая ошибка не возникнет вновь. Если подобные гарантии необходимы, обработка ошибок обычно выполняется в цикле. Другими словами, применение механизмов обратного исправления ошибок не позволяет добиться полностью прозрачного исправления ошибок.

И, наконец, хотя при обратном исправлении ошибок используются контрольные точки, в некоторые состояния вернуться просто невозможно. Например, если некий человек (возможно, злоумышленник) из-за неисправного банкомата внезапно получил 1000 долларов, шанс на то, что эти деньги удастся вернуть, очень невелик. Также невозможно и восстановление в предыдущее состояние большинства UNIX-систем после ввода следующей команды и выбора неправильного каталога:

```
/bin/rm -fr *
```

Некоторые вещи просто необратимы.

Контрольные точки позволяют вернуться в предыдущее правильное состояние. Однако создание контрольной точки часто вызывает значительные затраты само по себе, что оборачивается значительными потерями производительности. Поэтому многие отказоустойчивые распределенные системы сочетают создание контрольных точек с *протоколированием сообщений (message logging)*. В этом случае после создания контрольной точки процесс записывает свои сообщения перед отсылкой в журнал, что означает *протоколирование отправителем (sender-based logging)*. Альтернативное решение — позволить принимающему процессу протолировать приходящие сообщения перед их доставкой выполняемому

приложению. Эта схема известна под названием *протоколирования получателем* (*receiver-based logging*). Если в принимающем процессе возникает отказ, он должен восстановить состояние последней сохраненной контрольной точки, после чего может повторно запустить все принятые после нее сообщения. Таким образом, комбинация из контрольных точек и протоколирования сообщений позволяет восстановить значительно более позднее состояние, чем состояние, сохраненное в последней контрольной точке, без дополнительных затрат на создание контрольных точек.

Есть еще одно важное различие между контрольными точками и схемами, основанными на применении журналов. В системе, использующей только контрольные точки, процесс может быть восстановлен только в состояние контрольной точки. Но его поведение в этом состоянии может отличаться от поведения перед ошибкой. Так, например, если периоды между сеансами заданы не жестко, сообщения теперь могут приходить в ином порядке, вызывая другую реакцию получателя. В противоположность этому в случае протоколирования сообщений происходит повторение всех событий, имевших место с момента последней контрольной точки. Это повторение облегчит взаимодействие системы с внешним миром.

Так, например, рассмотрим случай, когда ошибка была вызвана неверными данными, вводимыми пользователем. Если использовать исключительно контрольные точки, максимум, что может система, — это вернуться в состояние контрольной точки, предшествовавшей приему введенных пользователем данных. При протоколировании сообщений можно использовать более старую контрольную точку, после чего, повторяя события, выйти к точке, где пользователь должен вводить данные. На практике комбинация небольшого количества контрольных точек и протоколирования сообщений используется гораздо чаще, чем схемы с множеством контрольных точек.

Устойчивые хранилища

Чтобы иметь возможность восстановить предыдущее состояние системы, требуется хранить нужную для восстановления информацию в безопасности. «В безопасности» в данном случае означает, что хранимая информация должна выдерживать отказы системы и сайтов, а возможно, и различные повреждения хранилища. Устойчивые хранилища играют важную роль в восстановлении распределенных систем. Здесь мы кратко обсудим этот вопрос.

Хранилища делятся на три категории. Первая — это стандартная оперативная память, стирающаяся при отключении питания или перезагрузках машины. Следующая категория — дисковые хранилища, переживающие ошибки процессора, но неспособные выжить при поломке головок диска.

И, наконец, существуют так называемые *устойчивые хранилища* (*stable storage*), разработанные для выживания в любых «передрягах», кроме катастроф вроде наводнений или землетрясений. Устойчивые хранилища могут быть реализованы в виде пары стандартных дисков, как это показано на рис. 7.15, а. Каждый блок устройства 2 является точной копией соответствующего блока устрой-

ства 1. Если блок диска изменяется, то сначала изменяется и проверяется блок устройства 1, а затем та же операция производится и с устройством 2.

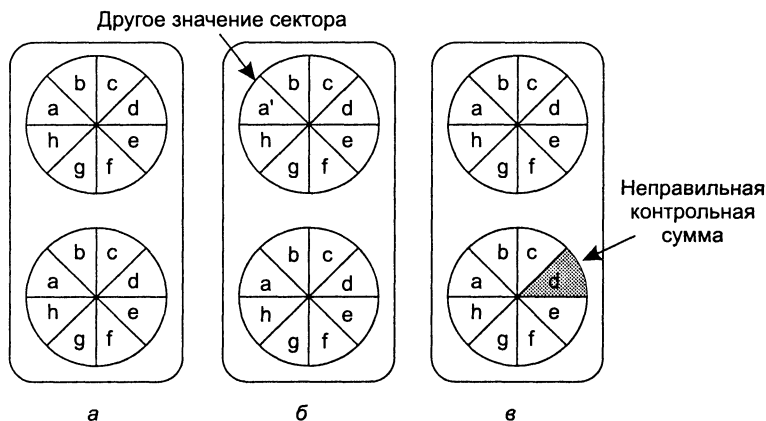


Рис. 7.15. Устойчивое хранилище (а). Отказ после обновления диска 1 (б). Плохой участок диска (в)

Представим себе, что система отказала после изменения содержимого устройства 1, но до изменения содержимого устройства 2, как показано на рис. 7.15, б. Во время исправления ошибок диски должны сравниваться поблочно. Если два соответствующих блока различны, это может означать, что правилен блок на устройстве 1 (поскольку устройство 1 всегда обновляется раньше устройства 2), а значит, новый блок копируется с устройства 1 на устройство 2. По завершении процесса восстановления оба диска снова будут идентичны.

Другая потенциальная проблема — самопроизвольное повреждение блока. Частицы пыли или безжалостное время способны внезапно, без каких-либо предварительных признаков, вызвать ошибку контрольной суммы нормального ранее блока, как показано на рис. 7.15, в. Если произойдет подобная ошибка, потерянные блоки можно будет восстановить по соответствующим блокам другого диска.

Как видно из этого описания, устойчивые хранилища хорошо подходят приложениям, требующим высокой степени отказоустойчивости, таким как атомарные транзакции. При записи данных в устойчивое хранилище с последующим чтением их оттуда с целью проверки правильности записи шанс на то, что впоследствии они окажутся потерянными, очень незначителен.

Ниже мы подробнее рассмотрим вопросы, связанные с контрольными точками и протоколированием сообщений. В [137] имеется обзор метода контрольных точек и протоколирования в распределенных системах. Различные детали алгоритмов можно найти в [103].

7.6.2. Создание контрольных точек

В отказоустойчивых распределенных системах обратное исправление ошибок требует, чтобы система регулярно записывала свое состояние в устойчивое хранилище данных. Понятие состояния распределенных систем обсуждалось в гла-

ве 5. В частности, мы делали упор на необходимость записи непротиворечивого глобального состояния, называемого также *распределенным снимком состояния* (*distributed snapshot*). Если в распределенном снимке состояния процесс P был сохранен в момент получения сообщения, должен также быть и процесс Q , сохраненный в момент передачи сообщения. Правда, в виде исключения сообщение может прийти и со стороны.

В схемах обратного исправления ошибок каждый процесс время от времени записывает свое состояние в локальное устойчивое хранилище. Для восстановления после системной ошибки необходимо из этих локальных состояний воссоздать непротиворечивое общее состояние. В частности, лучше всего восстановить последний распределенный снимок состояния, именуемый также *границей восстановления* (*recovery line*). Другими словами, граница восстановления соответствует последнему непротиворечивому срезу, что и показано на рис. 7.16.

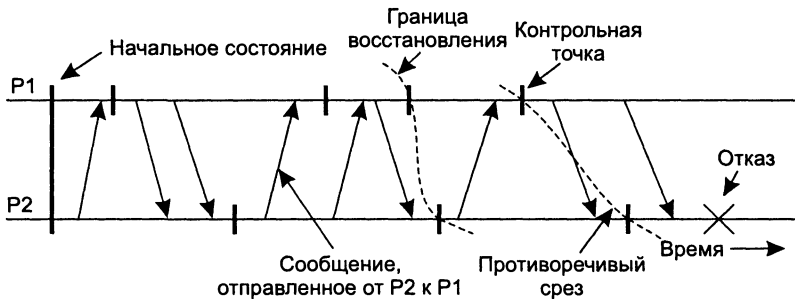


Рис. 7.16. Граница восстановления

Независимое создание контрольных точек

К сожалению, распределенная природа контрольных точек, когда каждый процесс просто время от времени записывает свое локальное состояние, никак не координируя свои действия с другими, может сильно затруднить поиски границы восстановления. Для нахождения границы восстановления необходимо, чтобы каждый из процессов откатился к последнему записанному состоянию. Если эти локальные состояния не образуют распределенного снимка состояния, потребуется новый откат и т. д. Этот процесс каскадного отката может привести к *эффекту домино* (*domino effect*), как показано на рис. 7.17.

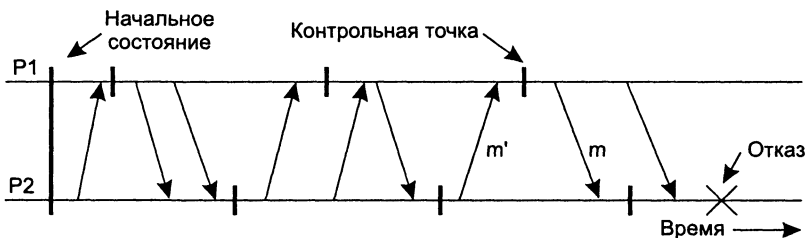


Рис. 7.17. Эффект домино

При ошибке в процессе $P2$ необходимо восстановить его состояние, каким оно было записано в последней контрольной точке. Процесс $P1$ также придется откатывать. К сожалению, два последних записанных локальных состояния не образуют глобального непротиворечивого состояния: состояние, записанное в $P2$, указывает на прием сообщения m , но ни один процесс не может быть назван его отправителем. Соответственно, $P2$ нужно откатить к еще более раннему состоянию.

Однако следующее состояние, в котором будет восстановлен процесс $P2$, также не может использоваться в качестве части распределенного снимка состояния. В этом случае процесс $P1$ записан принимающим сообщение m' , а событие его отправления записано снова не будет. Таким образом, появится необходимость откатить к более раннему состоянию также и процесс $P1$. В этом примере получается, что границей восстановления на самом деле является исходное состояние системы.

Поскольку процессы создают локальные контрольные точки независимо друг от друга, этот метод называется также методом *независимого создания контрольных точек* (*independent checkpointing*). Альтернативой ему является глобально координированное создание контрольных точек, которое мы обсудим ниже. Однако координация требует глобальной синхронизации, которая может создать проблемы с производительностью. Другой недостаток независимого создания контрольных точек состоит в том, что любое локальное хранилище нужно время от времени чистить, например, запуская специальную программу «сборки мусора». Однако самый серьезный недостаток — это необходимость расчета границы восстановления.

Независимое создание контрольных точек требует записи зависимостей, чтобы процессы можно было откатить вместе, в непротиворечивое глобальное состояние. Для этого обозначим m -ю контрольную точку процесса P как $CP[i](m)$. Пусть $INT[i](m)$ обозначает интервал между контрольными точками $CP[i](m-1)$ и $CP[i](m)$.

Когда процесс P_i отправляет сообщение в интервале $INT[i](m)$, он добавляет к нему пару (i, m) , пересылая ее принимающему процессу. Когда процесс P_j получает сообщение в интервале $INT[j](n)$ вместе с парой (i, m) , он записывает зависимость $INT[i](m) \rightarrow INT[j](n)$. Таким образом, когда P_j создает контрольную точку $CP[j](n)$, он дополнительно вместе с прочей информацией для восстановления, входящей в $CP[j](n)$, записывает в локальное хранилище данных и эту зависимость.

Предположим теперь, что в некоторый момент процесс $P1$ необходимо откатить в контрольную точку $CP[i](m-1)$. Для того чтобы гарантировать глобальную непротиворечивость, нам следует убедиться, что все процессы, получавшие от $P1$ сообщения, посланные в интервале $INT[i](m)$, откачены назад к состоянию, предшествующему получению этих сообщений. Так, например, процесс P_j в нашем примере следует откатить как минимум до контрольной точки $CP[j](n-1)$. Если $CP[j](n-1)$ не приведет систему в глобально непротиворечивое состояние, следует произвести дальнейший откат.

Расчет границы восстановления требует анализа интервальных зависимостей, сохраненных каждым из процессов при создании контрольных точек. Если даже

не входить в дополнительные детали, это делает такие вычисления достаточно сложными и не убеждает нас в преимуществах независимого создания контрольных точек по сравнению с координированным созданием. Кроме того, как оказалось, часто основным фактором, влияющим на производительность, оказывается не координация процессов, а затраты, вызываемые хранением состояния в локальных устойчивых хранилищах (см., например, [136]). Поэтому координированное создание контрольных точек, которое значительно проще их независимого создания, становится все более популярным.

Координированное создание контрольных точек

В соответствии с названием, при *координированном создании контрольных точек* (*coordinated checkpointing*) все процессы синхронизированы для того, чтобы запись их состояний в локальные устойчивые хранилища происходила одновременно. Основное преимущество координированного создания контрольных точек состоит в том, что записанное состояние автоматически получается глобально непротиворечивым, что позволяет избежать каскадного отката, приводящего к эффекту домино. Для координированного создания контрольных точек может использоваться алгоритм распределенного снимка состояния, который мы обсуждали в главе 5. Этот алгоритм представляет собой пример неблокирующей координации контрольных точек.

Самым простым решением считается использование двухфазного протокола блокировки. Сначала координатор рассылает всем процессам сообщение *CHECKPOINT_REQUEST*. Когда процесс получает это сообщение, он создает локальную контрольную точку, начинает выстраивать в очередь все последующие сообщения, приходящие к нему от работающего приложения, и отправляет координатору подтверждение создания контрольной точки. После того как координатор получит подтверждения от всех процессов, он рассылает сообщение *CHECKPOINT_DONE*, позволяющее заблокированным процессам продолжить выполнение.

Легко заметить, что такой подход также приводит к записи глобально непротиворечивого состояния, поскольку никакие входящие сообщения не становятся частью состояния контрольной точки. Это происходит потому, что никакие сообщения, следующие за запросом на создание контрольной точки, не считаются частью локальной контрольной точки. В то же время исходящие сообщения (посылаемые процессом, для которого создается контрольная точка, работающим приложениям) сохраняются в локальной очереди до прихода сообщения *CHECKPOINT_DONE*.

Усовершенствованием этого алгоритма является групповая рассылка запроса на создание контрольных точек только тем процессам, которые зависят от восстановления координатора. Процесс зависит от координатора, если он получает сообщение, которое прямо или косвенно причинно связано с сообщением, отправленным координатором с момента создания предыдущей контрольной точки. Это приводит нас к понятию *инкрементного снимка состояния* (*incremental snapshot*).

Для получения инкрементного снимка состояния координатор рассылает запрос на создание контрольных точек только тем процессам, которым он с момента создания предыдущей контрольной точки посылал сообщения. Когда процесс P получает такой запрос, он рассылает его всем прочим процессам, которым с момента создания предыдущей контрольной точки посылал сообщения он сам и т. д. Процесс рассылает запрос только один раз. Когда будут определены все процессы, производится вторая групповая рассылка, чтобы закончить создание контрольных точек и запустить выполнение процессов с того места, где они были остановлены.

7.6.3. Протоколирование сообщений

Мы пришли к выводу, что создание контрольных точек — довольно затратная операция, особенно если учесть необходимость записи состояния в устойчивое хранилище. Поэтому следует применять технологии сокращения числа контрольных точек, не ухудшающие качества исправления ошибок. Одна из важнейших технологий такого типа в распределенных системах — это *протоколирование сообщений* (*message logging*).

Основная идея, лежащая в основании протоколирования сообщений, состоит в том, что если удастся *воспроизвести* повторную передачу сообщений, мы также получим глобально непротиворечивое состояние, не восстанавливая его из устойчивого хранилища. В этом случае, начиная от состояния контрольной точки, просто воспроизводятся отправка, прием и обработка всех сообщений, которыми обменивались процессы после создания контрольной точки.

Этот способ хорошо работает при соблюдении условий так называемой *кусочно детерминированной модели* (*piecewise deterministic model*). Согласно этой модели считается, что выполнение каждого процесса разбивается на последовательность интервалов, во время которых происходят события. Это те же события, которые мы обсуждали в главе 5 в контексте отношения «происходят раньше» по Лампорту. Так, например, событием может быть выполнение инструкции, отправка сообщения и т. п. Каждый интервал в кусочно детерминированной модели считается начинающимся с недетерминированного события, такого как получение сообщения. Однако с этого момента выполнение процесса полностью детерминировано. Концом интервала считается последнее событие из цепочки детерминированных перед новым недетерминированным событием.

В результате интервал можно повторить с известным результатом, то есть абсолютно детерминированным образом, начиная с повторения того же самого недетерминированного события. Соответственно, если мы запишем в журнал все недетерминированные события модели, то получим возможность полностью, детерминированным образом повторить все выполнение процесса.

Если полагать, что журнал сообщений при аварии процесса необходимо восстановить для восстановления глобально непротиворечивого состояния, то для нас важно точно знать время записи сообщений в журнал. В соответствии с подходом, описанным в [10], обнаруживается, что можно легко характеризовать

множество схем протоколирования сообщений, сосредоточившись на том, как они работают с процессами-сиротами.

Осиротевший процесс (orphan process) — это процесс, состояние которого противоречит состоянию другого, сначала отказавшего, а затем восстановленного процесса. В качестве примера рассмотрим ситуацию, представленную на рис. 7.18. Процесс Q получает от процессов P и R сообщения $m1$ и $m2$ соответственно, после чего посылает сообщение $m3$ процессу R . Однако в отличие от других сообщений сообщение $m2$ в журнал не записывается. Если происходит отказ процесса Q , для его восстановления задействуются только записанные в журнал сообщения, в нашем примере — $m1$. Поскольку сообщение $m2$ не сохранено в журнале, его передача не будет воспроизведена, а значит, передачи сообщения $m3$ также не произойдет, что и показано на рисунке.

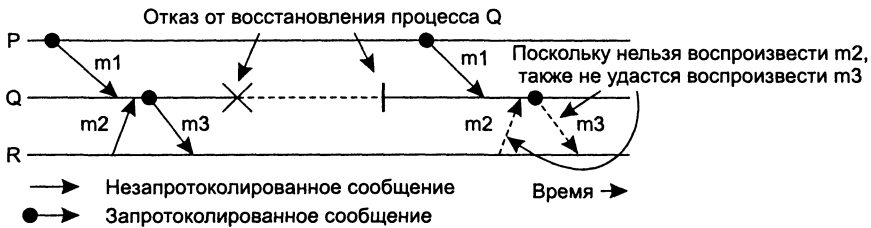


Рис. 7.18. Неправильное воспроизведение сообщений после восстановления ведет к появлению осиротевших процессов

Таким образом, ситуация после восстановления процесса Q противоречит ситуации до его восстановления. Так, в частности, процесс R хранит сообщение ($m3$), которое было передано перед отказом, но получения и доставки которого при воспроизведении событий, имевших место перед сбоем, не происходит. Понятно, что такого противоречия нужно избегать.

Характеристические схемы протоколирования сообщений

Чтобы характеризовать различные схемы протоколирования сообщений, мы используем подход, описанный в [10]. Предполагается, что каждое сообщение m имеет заголовок, содержащий всю информацию, необходимую для повторной передачи этого сообщения и правильной его обработки. Так, например, каждый заголовок определяет отправителя и получателя, а также последовательный номер для распознавания дубликатов. Кроме того, к заголовку может добавляться входящий номер, чтобы точно определить момент обработки сообщения получившим его приложением.

Сообщение называется *устойчивым (stable)*, если оно не может быть потеряно, например, после записи в устойчивое хранилище. Устойчивые сообщения могут использоваться для восстановления путем воспроизведения их передачи.

Каждое сообщение m отвечает за набор процессов $DEP(m)$, которые зависят от доставки m . В частности, $DEP(m)$ состоит из тех процессов, которым должно

прийти это сообщение. Кроме того, если другое сообщение, m' , причинно зависящее от доставки m , доставляется процессу Q , то Q также входит в набор $DEP(m)$. Отметим, что причинная зависимость m' от доставки m означает, что оно посылается тем же самым процессом, который ранее получил m или другое сообщение, причинно зависящее от доставки m .

Набор $COPY(m)$ состоит из тех процессов, которые содержат копию m , но (пока) не в своих локальных хранилищах. Когда процесс Q получает сообщение m , он также становится членом $COPY(m)$. Отметим, что $COPY(m)$ состоит из тех процессов, которые могут передавать копию m для воспроизведения передачи. Если все эти процессы отказали, очевидно, что воспроизведение передачи m оказывается невозможным.

Используя эту нотацию, легко определить, что же такое осиротевший процесс. Предположим, что в распределенной системе отказали несколько процессов. Пусть Q — один из сохранившихся процессов. Процесс Q является сиротой, если существует такое сообщение m , что Q оказывается в наборе $DEP(m)$, в то время как все процессы из $COPY(m)$ отказали. Другими словами, осиротевший процесс появляется в том случае, если он зависит от сообщения m , однако нет никакой возможности повторить передачу этого сообщения.

Чтобы предотвратить появление осиротевших процессов, необходимо гарантировать, что в $DEP(m)$ не будет выживших процессов, если откажут все процессы в $COPY(m)$. Система оказывается в таком состоянии, если каждый процесс, являющийся членом набора $DEP(m)$, также является и членом набора $COPY(m)$. Другими словами, всякий процесс, зависящий от доставки сообщения m , должен хранить копию этого сообщения.

Существует два основных подхода, которым мы можем следовать. Первый из них представлен так называемыми *протоколами пессимистического протоколирования* (*pessimistic logging protocols*). В этих протоколах предполагается, что для каждого неустойчивого сообщения m имеется хотя бы один зависящий от него процесс. Другими словами, протоколы пессимистического протоколирования предполагают, что любое неустойчивое сообщение m доставляется как минимум в один процесс. Отметим, что как только m доставляется, скажем, в процесс P , этот процесс становится членом набора $COPY(m)$.

Самое плохое, что может произойти, — процесс P откажет еще до того, как будет записано сообщение m . В случае пессимистического протоколирования процессу P после получения сообщения m нельзя рассылать какие-либо сообщения, пока m не будет записано в устойчивое хранилище. Таким образом, пока не будет гарантирована возможность повторной отправки сообщения m , не сможет появиться ни одного процесса, зависящего от доставки m процессу P . Таким образом, мы полностью предотвращаем появление осиротевших процессов.

В противоположность этому в *протоколе оптимистического протоколирования* (*optimistic logging protocol*) реальная работа начинается после отказа процесса. В частности, допустим, что для некоторого сообщения m отказали все процессы из набора $COPY(m)$. Согласно оптимистическому подходу любой осиротевший процесс из $DEP(m)$ откатывается в такое состояние, когда он перестает входить в набор $DEP(m)$.

Как было указано в [137], пессимистическое протоколирование значительно проще оптимистического, так что этот способ протоколирования сообщений при практической разработке распределенных систем предпочтительнее.

7.7. Итоги

Отказоустойчивость — это важная область построения распределенных систем. Отказоустойчивость определяется как способность системы маскировать появление ошибок и исправлять их. Другими словами, система отказоустойчива, если она продолжает функционировать при наличии отказов.

Существуют разные типы отказов. Поломка означает простую остановку системы. Пропуск данных наблюдается в том случае, если процесс не реагирует на входящие запросы. Если процесс отвечает слишком быстро или слишком медленно, мы говорим об ошибке синхронизации. Отклик на входящий запрос с ошибкой — пример ошибки отклика. Сложнее всего обрабатывать ситуации, когда в процессе может случиться ошибка любого типа. Такие ошибки называют произвольными или византийскими.

Избыточность — это стандартный способ обеспечения отказоустойчивости. Если избыточность применяется к процессам, становится важным понятие группы процессов. Процессы в группе тесно взаимодействуют между собой для предоставления некоторой услуги. В отказоустойчивых группах один или более процессов могут отказаться, не оказывая при этом заметного влияния на доступность услуги, реализуемой группой. Часто необходимо, чтобы взаимодействие внутри группы было высоконадежным и в плане обеспечения отказоустойчивости поддерживало свойства строгой упорядоченности и атомарности.

Надежное групповое взаимодействие, именуемое также надежной групповой рассылкой, может существовать в различных формах. До тех пор пока группы относительно малы, достижение надежности вполне возможно. Однако при необходимости поддерживать очень большие группы масштабирование надежной групповой рассылки становится проблематичным. Ключевым моментом в реализации масштабируемости становится снижение числа откликов, возвращающих отчет об удачном (или неудачном) приеме разосланного сообщения.

Проблемы усложняются, когда требуется соблюсти атомарность. В протоколах атомарной групповой рассылки важно, чтобы каждый член группы имел одинаковое представление о том, каким элементам группы доставляется сообщение. Атомарная групповая рассылка должна быть точно сформулирована в понятиях модели виртуально синхронного выполнения. В частности, эта модель вводит границы, внутри которых членство в группах не меняется, а сообщения передаются надежно. Сообщения никогда не пересекают границ.

Механизм изменения членства в группе — это пример того, что каждый процесс должен согласиться с единым списком членов. Такое соглашение может быть заключено при помощи протоколов подтверждения, самым распространенным из которых является протокол двухфазного подтверждения. В условиях протокола двухфазного подтверждения координатор сначала проверяет готов-

ность всех процессов к выполнению одной и той же операции (то есть к ее подтверждению), а на втором этапе рассылает результат этого голосования. Протокол трехфазного подтверждения используется для того, чтобы справиться с отказом координатора без необходимости блокировать все процессы, для достижения соглашения дожидаясь восстановления координатора.

Восстановление в отказоустойчивых системах неизменно организуется на основе регулярного сохранения информации о состоянии системы. Работа с контрольными точками полностью распределена. К сожалению, создание контрольной точки — довольно затратная операция. Для повышения производительности множество распределенных систем сочетают контрольные точки с протоколированием сообщений. Благодаря протоколированию взаимодействия между процессами после отказа системы можно воспроизвести ход ее функционирования.

Вопросы и задания

1. Надежные системы часто требуют обеспечения высокой степени защиты. Почему?
2. Что делает модель аварийной остановки трудной для реализации в случае поломок?
3. Рассмотрим web-браузер, возвращающий устаревшие страницы из кэша, а не обновленные с сервера. Является ли это ошибкой, и если да, какой это тип ошибки?
4. Может ли модель тройного модульного резервирования, описанная в тексте, справиться с византийскими ошибками?
5. Сколько отказавших элементов (обычных устройств и устройств голосования) может быть обработано схемой, представленной на рис. 7.1? Дайте пример ошибки, которая может быть замаскирована.
6. Можно ли обобщить тройное модульное резервирование на пять элементов в группе вместо трех? Если да, какие свойства будет оно иметь?
7. Для какого из следующих приложений, по вашему, наилучшим образом подойдет семантика «минимум однажды» или «максимум однажды»? Дайте развернутое объяснение.
 - ✦ Чтение и запись файлов на файловый сервер.
 - ✦ Компиляция программы.
 - ✦ Домашний банк.
8. В случае асинхронных вызовов RPC клиент блокируется до момента получения его запроса сервером (см. рис. 7.12). В какой степени наличие ошибок влияет на семантику асинхронных вызовов RPC?
9. Приведите пример, когда для группового взаимодействия вообще не нужно никаких синхронизирующих сообщений.
10. Всегда ли в надежных групповых рассылках есть необходимость в том, чтобы копии сообщений сохранялись на коммуникационном уровне с целью повторной отправки?

11. В какой степени важна масштабируемость для атомарных групповых рассылок?
12. В тексте мы предложили, что при атомарной групповой рассылке можно сохранять дату осуществления изменения согласного на это набора процессов. В какой степени мы можем гарантировать, что каждое из изменений действительно было сделано?
13. Виртуальная синхронность аналогична слабой непротиворечивости распределенных хранилищ данных с изменениями представления групп в качестве точек синхронизации. В этом контексте что будет аналогом строгой непротиворечивости?
14. Каков разрешенный порядок доставки для комбинации FIFO и полностью упорядоченной групповой рассылки (см. табл. 7.3)?
15. Адаптируйте протокол для установки следующего представления G_{i+1} в случае виртуальной синхронности, чтобы он мог противостоять ошибкам в процессах.
16. Почему в протоколе двухфазного подтверждения невозможно полностью исключить блокировку, даже в случае выбора участниками нового координатора?
17. Из нашего объяснения механизма трехфазного подтверждения можно понять, что завершение транзакции определяется большинством голосов. Так ли это?
18. В случае кусочно детерминированной модели достаточно ли заносить в журнал только сообщения? Или также требуется протоколировать другие события?
19. Опишите, как для восстановления после отказов можно использовать в определенных транзакциях журнал с упреждающей записью.
20. Нуждается ли в создании контрольных точек сервер без фиксации состояния?
21. Протоколирование сообщений получателем обычно считается более правильным, чем протоколирование отправителем. Почему?

Глава 8

Защита

8.1. Общие вопросы защиты

8.2. Защищенные каналы

8.3. Контроль доступа

8.4. Управление защитой

8.5. Пример — Kerberos

8.6. Пример — SESAME

8.7. Пример — электронные платежные системы

8.8. Итоги

Последняя составная часть распределенных систем, которую мы рассмотрим, — это защита. Хотя мы рассматриваем защиту последней, по значимости она далеко не последняя. Вряд ли кто-нибудь может поспорить с тем, что это одна из наиболее сложных частей, ведь защита должна пронизывать всю систему целиком. Единственная брешь в системе защиты может сделать бесполезными все принимаемые меры обеспечения безопасности. В этой главе мы рассмотрим различные механизмы, которые обычно применяются для защиты данных в распределенных системах.

Мы начнем с разговора о базовых понятиях защиты. Построение механизмов защиты всех типов в системе на самом деле бессмысленно, если неизвестно, как именно должны использоваться эти механизмы и чему они будут противостоять. Для этого нужно знать реализуемые правила защиты. Сначала мы рассмотрим понятие правил защиты, а также дадим некоторое общее представление используемых механизмов реализации этих правил. Кроме того, мы кратко коснемся вопросов криптографической защиты информации.

Системы защиты в распределенных системах можно разделить на две независимые части. Одна из них — это связь между пользователями или процессами, возможно, расположенными на различных машинах. Принципиальный способ гарантировать защиту взаимодействия — это защищенный канал. Защищенные каналы, и более конкретно, идентификация, целостность сообщений и конфиденциальность, обсуждаются в отдельном разделе.

Другая часть систем защиты — это авторизация, которая позволяет гарантировать, что процессы получают только те возможности доступа к ресурсам распределенной системы, на которые имеют право. Авторизацию и контроль доступа можно рассматривать совместно. В добавление к традиционным механизмам контроля доступа мы рассмотрим также контроль доступа при работе с мобильным кодом, например с агентами.

Защищенные каналы и средства контроля доступа нуждаются в механизмах для работы с криптографическими ключами, а также в механизмах добавления пользователей в систему и удаления их из нее. Эти вопросы входят в то, что называется управлением защитой. В отдельном разделе мы рассмотрим вопросы, связанные с управлением криптографическими ключами, управлением защитой в группах и обработкой сертификатов, удостоверяющих право владельца на доступ к определенным ресурсам.

И, наконец, мы конкретизируем наше обсуждение, разобрав два примера реализации средств защиты в распределенных системах. SESAME — это законченная система, которая может встраиваться в распределенные системы для обеспечения их защиты. Чтобы продемонстрировать абсолютно иной подход к защите распределенных систем, мы кратко рассмотрим электронные платежные системы, позволяющие пользователям и коммерсантам, находящимся в различных местах, безопасным образом инициировать транзакции, включающие заказ и оплату товаров.

8.1. Общие вопросы защиты

Мы начнем наш разговор о защите распределенных систем с рассмотрения общих вопросов защиты. В начале необходимо определить, что же такое защищенная система. Затем мы отделим *правила* защиты от *механизмов* защиты и обсудим глобальную систему Globus, правила защиты которой сформулированы явным образом. Нашей второй темой станет обсуждение общих вопросов разработки систем защиты. И наконец, мы кратко обсудим некоторые криптографические алгоритмы, играющие ключевую роль в разработке протоколов защиты.

8.1.1. Угрозы, правила и механизмы

Защита в компьютерных системах жестко связана с понятием *надежности* (*dependability*). Говоря неформально, надежной компьютерной системой считается система, службам которой мы оправданно доверяем [256]. Как мы говорили в главе 7, надежность подразумевает доступность, безотказность, безопасность и ремонтпригодность. Однако, если мы собираемся доверять компьютерной системе, следует также принимать во внимание конфиденциальность и целостность. Под *конфиденциальностью* (*confidentiality*) мы понимаем свойство компьютерной системы, благодаря которому доступ к информации в ней ограничен кругом доверенных лиц. *Целостность* (*integrity*) — это характеристика, указывающая на то, что изменения могут быть внесены в систему только авторизованными лицами или процессами. Другими словами, незаконные изменения в защищенной

компьютерной системе должны обнаруживаться и исправляться. Основные части компьютерной системы — это аппаратура, программы и данные.

Другой способ взглянуть на защиту в компьютерных системах — считать, что мы стараемся защитить службы и данные от *угроз защите* (*security threats*). Мы выделяем четыре типа угроз защите [351]:

- ◆ перехват (*interception*);
- ◆ прерывание (*interruption*);
- ◆ модификация (*modification*);
- ◆ подделка (*fabrication*).

Перехватом мы называем такую ситуацию, когда неавторизованный агент получает доступ к службам или данным. Типичный пример перехвата — когда связь между двумя агентами подслушивает кто-то третий.

Примером прерывания может служить повреждение или потеря файла. Обычно прерывание связывают с такой ситуацией, когда службы или данные становятся недоступными, уничтожаются, их невозможно использовать и т. п. В этом смысле атаки типа «отказ в обслуживании» (*denial of service*), при которых кто-то злонамеренно старается сделать определенную службу недоступной для других, — это угроза защите, классифицируемая как прерывание.

Модификации включают в себя неавторизованные изменения данных или фальсификацию служб с тем, чтобы они не соответствовали своему оригинальному предназначению. Примеры модификации включают перехват сообщений с последующим изменением передаваемых данных, фальсификацию входов в базы данных и изменение программ с тем, чтобы скрытно отслеживать деятельность пользователей.

Подделке соответствует ситуация, когда создаются дополнительные данные или осуществляется деятельность, невозможная в нормальных условиях. Так, например, злоумышленник может попытаться добавить записи в файл паролей или базу данных. Кроме того, иногда удастся войти в систему, воспроизведя отправку ранее посланного сообщения. Мы рассмотрим подобные примеры чуть позже.

Отметим, что прерывание, модификация и подделка могут рассматриваться как формы фальсификации данных.

Просто констатировать, что система должна быть в состоянии противостоять всевозможным угрозам защите — это не метод построения защищенных систем. Прежде всего, следует описать требования к защите, то есть правила защиты. *Правила защиты* (*security policy*) точно описывают разрешенные и запрещенные действия для системных сущностей. В понятие «системные сущности» входят пользователи, службы, данные, машины и т. п. После составления правил защиты можно сосредоточиться на *механизмах защиты* (*security mechanisms*), посредством которых реализуются эти правила. Наиболее важные из них:

- ◆ шифрование (*encryption*);
- ◆ аутентификация (*authentication*);
- ◆ авторизация (*authorization*);
- ◆ аудит (*auditing*).

Шифрование — фундамент компьютерной защиты. Шифрование трансформирует данные в нечто, чего злоумышленник не в состоянии понять. Другими словами, шифрование — это средство реализации конфиденциальности. Кроме того, шифрование позволяет нам проверить, не изменялись ли данные, давая возможность контролировать целостность данных.

Аутентификация используется для проверки заявленного имени пользователя, клиента, сервера и пр. В случае с клиентом основная идея заключается в том, что до начала работы службы с клиентом служба должна определить подлинность клиента. Обычно пользователи аутентифицируют себя при помощи пароля, однако существуют и другие способы аутентификации клиента.

После того как клиент аутентифицирован, необходимо проверить, имеет ли он право на проведение запрашиваемых действий. Типичным примером является доступ к базе данных с медицинской информацией. В зависимости от того, кто работает с базой, ему может быть разрешено читать записи, модифицировать определенные поля записей или добавлять и удалять записи.

Средства аудита используются для контроля за тем, что делает клиент и как он это делает. Хотя средства аудита не защищают от угроз защите, журналы аудита постоянно используются для анализа «дыр» в системах защиты с последующим принятием мер против нарушителей. Поэтому нарушители всеми силами стараются не оставлять каких-либо следов, которые в конце концов могут привести к их раскрытию. До известной степени именно благодаря протоколированию доступа хакерство является весьма рискованным занятием.

Пример — архитектура защиты Globus

Понятие о правилах защиты и роли, которую механизмы защиты играют в соблюдении этих правил, часто лучше объяснять на конкретном примере. Рассмотрим правила защиты, определенные в глобальной системе Globus [100]. Globus — это система поддержки распределенных вычислений, в которой для производства вычислений одновременно используется множество хостов, файлов и других ресурсов. Такие системы часто называют вычислительными сетями [148]. Ресурсы в таких сетях нередко расположены в различных административных доменах, которые могут находиться в разных частях света.

Поскольку пользователи и ресурсы велики числом и рассеяны по множеству административных доменов, важность защиты резко возрастает. Чтобы разработать и правильно использовать механизмы защиты, необходимо понять, что на самом деле нужно защищать и какие допущения по этому поводу можно сделать. Опуская некоторые подробности, мы можем сказать, что правила защиты в Globus вытекают из следующих восьми умозаключений [147].

- ◆ Среда состоит из множества административных доменов.
- ◆ Локальные операции (то есть операции, протекающие в пределах одного домена) обеспечиваются исключительно локальными мерами защиты.
- ◆ Глобальные операции (то есть операции, в которые включается несколько доменов) требуют инициатора, известного во всех вовлеченных в операцию доменах.

- ✦ Для операций между сущностями в различных доменах необходима обоюдная идентификация.
- ✦ Глобальная аутентификация стоит выше локальной.
- ✦ Контроль доступа к ресурсам относится к компетенции локальной идентификации.
- ✦ Пользователи могут делегировать свои права процессам.
- ✦ Группа процессов в одном домене может использовать свои идентификаторы совместно.

В системе Globus предполагается, что среда включает в себя множество административных доменов, каждый из которых имеет свои локальные правила защиты. Предполагается, что локальные правила не изменятся только из-за того, что система входит в Globus. Глобальные правила Globus, кроме того, не изменяют действия локальных правил защиты. Соответственно, глобальная защита в Globus ограничена лишь операциями, в которые вовлечены несколько доменов.

В соответствии с этим моментом в Globus считается, что операции, целиком происходящие внутри одного домена, подчиняются только локальным правилам защиты этого домена. Другими словами, если операция инициирована и происходит в пределах одного домена, все действия производятся с использованием только локальных мер защиты. Globus не предпринимает на этот счет никаких дополнительных действий.

Правила защиты Globus определяют, что запросы на операцию должны инициироваться — глобально или локально. Инициатор, которым является пользователь или процесс, работающий от имени пользователя, должен быть известен во всех доменах, участвующих в операции. Так, например, пользователь может задействовать глобальное имя, которое отображается в локальные имена в конкретных доменах. Как именно осуществляется это отображение, зависит от домена.

Один из важных принципов состоит в том, что операции между сущностями в разных доменах требуют обоюдной аутентификации. Это означает, например, что если пользователь из одного домена захочет воспользоваться службой, расположенной в другом домене, домену, в котором находится служба, необходимо провести аутентификацию пользователя. Не менее важно, чтобы пользователь был уверен в том, что он использует именно тот сервер, который собирался использовать. Мы более подробно обсудим вопросы аутентификации чуть позже.

Эти два правила комбинируются в следующее требование защиты. Если личность пользователя подтверждена и пользователь локально известен в данном домене, то в этом домене он может считаться прошедшим аутентификацию. То есть Globus полагает, что при получении доступа к ресурсам удаленного домена общесистемным средствам аутентификации достаточно установить тот факт, что пользователь уже аутентифицирован в этом удаленном домене (если он там известен). Дополнительная аутентификация в этом удаленном домене уже не нужна.

Как только пользователь (или процесс, работающий по его заданию) аутентифицирован, ему необходимо подтвердить соответствующие права на доступ к ре-

сурсам. Так, например, пользователь, желающий изменить файл, должен сначала пройти аутентификацию, после чего следует проверить, действительно ли этот пользователь имеет право изменять этот файл. Правила защиты Globus предполагают, что контроль доступа осуществляется исключительно локально, внутри домена, в котором находятся используемые ресурсы.

Чтобы разобраться в правиле, согласно которому пользователи могут делегировать свои права процессам, рассмотрим мобильного агента системы Globus, который выполняет свою задачу, иницилируя одну за другой некоторые операции в различных доменах. Подобному агенту для выполнения своей работы может потребоваться длительный срок. Чтобы существовала возможность поддерживать связь между агентом и пользователем, по заданию которого работает этот агент, Globus требует, чтобы пользователи могли делегировать часть своих прав процессам. Соответственно, при аутентификации агента с последующей проверкой его прав система Globus должна иметь возможность разрешить агенту иницилировать операции, не связываясь с владельцем агента.

В качестве последнего правила Globus требует, чтобы группы процессов, выполняемые в одном домене и работающие по заданию одного пользователя, могли совместно использовать один набор полномочий. Как мы покажем далее, наборы полномочий необходимы для аутентификации. Это утверждение, в сущности, открывает дорогу масштабируемым решениям аутентификации за счет отказа от поддержки каждым из процессов своего собственного уникального набора полномочий.

Правила защиты Globus позволяют ее разработчикам сосредоточиться на решении общих проблем защиты. Предполагая, что каждый домен поддерживает собственные правила защиты, разработчики Globus сосредотачивают свои усилия только на угрозах защите на междоменном уровне. В частности, правила защиты показывают, что важными вопросами построения считаются представление пользователя в удаленных доменах и выделение ресурсов удаленного домена пользователю или его представителю. В первую очередь Globus нуждается в механизмах междоменной аутентификации и объявления пользователя в удаленных доменах.

Для этой цели вводятся два типа представителей. *Заместитель пользователя (user proxy)* представляет собой процесс, которому дано право действовать от лица пользователя в течение ограниченного времени. Ресурсы представляются в виде заместителей ресурсов. *Заместитель ресурса (resource proxy)* — это процесс, работающий в определенном домене и используемый для трансляции глобальных операций с ресурсами в локальные операции, удовлетворяющие требованиям локальных правил защиты. Так, например, заместитель пользователя обычно применяется для связи с заместителем ресурса в процессе доступа к необходимым ресурсам.

Архитектура защиты системы Globus в основном состоит из различных сущностей, таких как пользователи, заместители пользователей, заместители ресурсов и процессы общего назначения. Эти сущности размещены в доменах и взаимодействуют друг с другом. В частности, архитектура защиты определяет четыре различных протокола взаимодействия, показанных на рис. 8.1 [148].

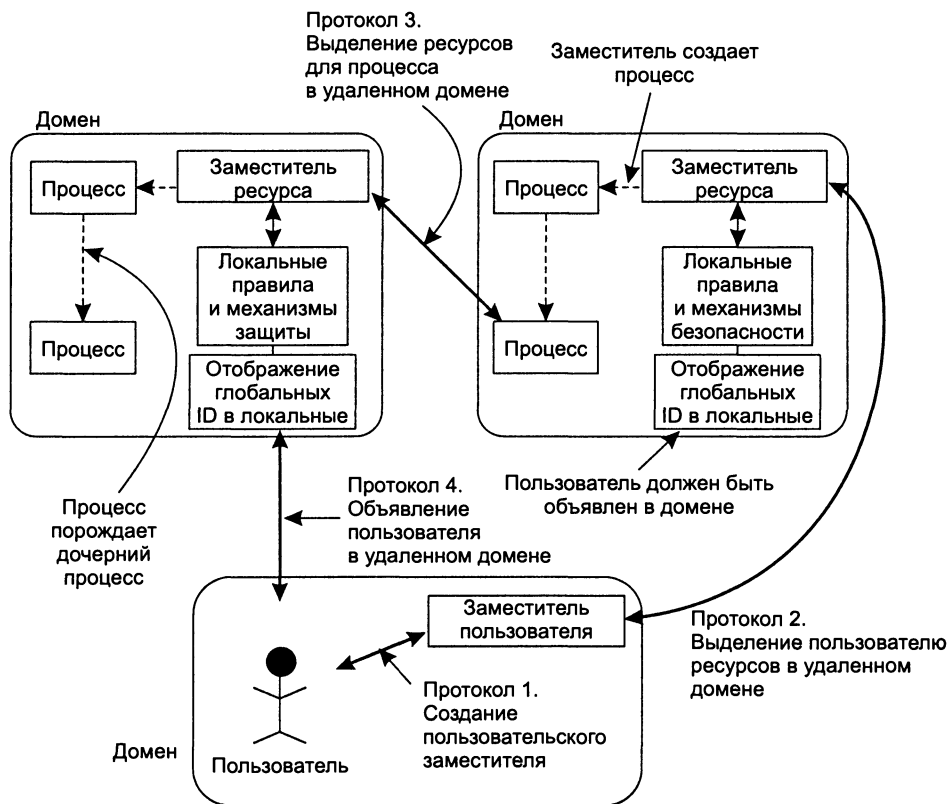


Рис. 8.1. Архитектура защиты Globus

Первый протокол детально описывает создание пользователем своего заместителя и делегирование этому заместителю своих прав. В частности, чтобы позволить заместителю действовать по заданию пользователя, этот пользователь передает ему соответствующий набор полномочий.

Второй протокол определяет, как заместитель пользователя может запросить ресурсы в удаленном домене. Коротко говоря, протокол указывает, что заместитель ресурса создает процесс в удаленном домене после проведения обоюдной аутентификации. Этот процесс представляет пользователя (как это ранее делал заместитель пользователя), но работает в том же домене, что и запрошенные ресурсы. Процесс имеет доступ к ресурсам и подчиняется правилам контроля доступа, принятым в этом домене.

Процесс, созданный в удаленном домене, может инициировать дополнительные вычисления в других доменах. Соответственно, необходим протокол выделения ресурсов в удаленных доменах, которое инициируется процессом, не являющимся заместителем. В Globus такое выделение ресурсов производится через заместителя пользователя. Процессу просто разрешается иметь собственный заместитель пользователя, который может запрашивать выделение ресурсов в соответствии со вторым протоколом.

Четвертый и последний протокол в архитектуре защиты Globus — это способ, посредством которого пользователь может объявить о себе в удаленном домене. Если считать, что пользователь имеет учетную запись в конкретном домене, нам необходимо, чтобы системный набор прав, представленный заместителем пользователя, автоматически конвертировался в права для этого домена. Протокол предписывает, как зарегистрировать отображение глобальных прав пользователя в локальные права в локальной таблице отображений домена.

Конкретные подробности каждого из протоколов описаны в [148]. Самый важный момент во всем этом заключается в том, что архитектура защиты Globus соответствует описанным ранее правилам защиты. Механизмы реализации этой архитектуры, в частности рассмотренные здесь протоколы, едины для многих распределенных систем и будут обсуждаться далее в этой главе. Трудность разработки защищенных распределенных систем связана не столько с механизмами защиты, сколько с тем, как использовать эти механизмы для обеспечения защиты. В следующем подразделе мы обсудим некоторые из вариантов.

8.1.2. Вопросы разработки

В распределенную систему, да впрочем и в любую компьютерную систему, должны быть встроены механизмы защиты, при помощи которых можно будет реализовать различные правила защиты. При реализации служб защиты общего назначения следует учитывать несколько моментов. Ниже мы рассмотрим три таких важных момента: фокус управления, многоуровневую организацию механизмов защиты и простоту [172].

Фокус управления

Организуя защиту приложений (в том числе и распределенных), можно использовать три основных подхода, которые иллюстрирует рис. 8.2. Первый вариант — это защита непосредственно ассоциированных с приложением данных. Под «непосредственно» мы имеем в виду, что независимо от того, какие операции могут производиться с элементами данных, основная задача этого типа защиты — сохранение целостности данных. Обычно такая защита используется в системах баз данных, в этом случае заранее определяются различные ограничения целостности, автоматически проверяемые затем при каждой модификации элемента данных [466].

Второй вариант — это защита путем точного указания того, кто и как может использовать операции доступа к данным или ресурсам. В этом случае фокус управления тесно связан с механизмами контроля доступа, о которых мы поговорим подробнее чуть позже. Так, например, в системе, построенной на основе объектов, для каждого из методов, доступ к которым мы открываем клиентам, можно указать, какой именно клиент имеет право к нему обратиться. С другой стороны, методы контроля доступа могут применяться к интерфейсу, предоставляемому объектом, или собственному объекту. Этот подход также используется с целью детализации управления доступом.

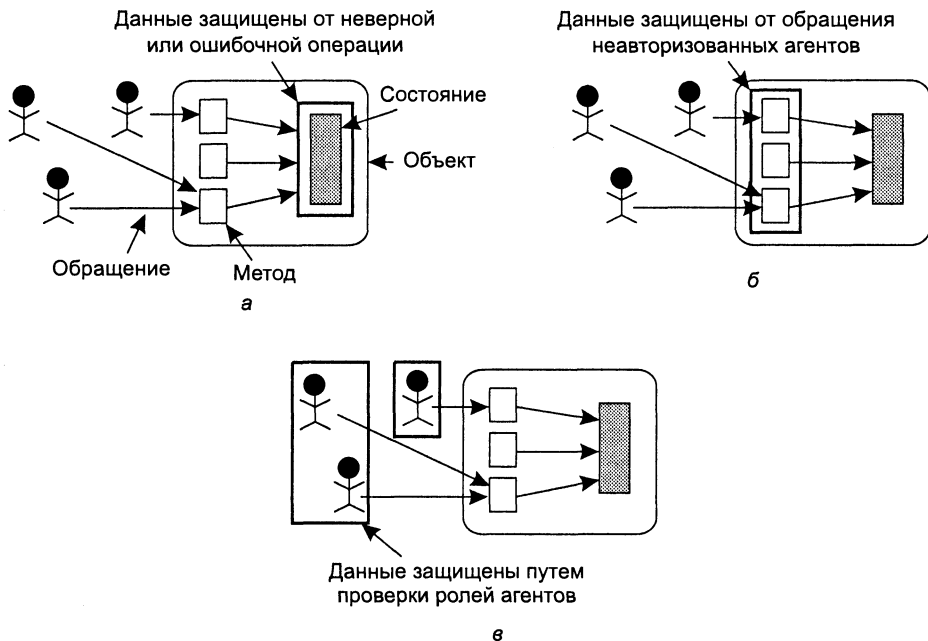


Рис. 8.2. Три подхода к противодействию угрозам защите. Защита от неверных операций (а). Защита от неавторизованных обращений (б). Защита от неавторизованных пользователей (в)

Третий вариант — сосредоточить внимание непосредственно на пользователе, приняв меры, чтобы доступ к приложению имели только определенные люди, независимо от операций, которые они собираются производить. Так, например, банковская база данных может быть защищена путем закрытия доступа к ней всем, кроме высшего управленческого персонала. Другой пример: во многих университетах доступ к определенным данным и приложениям разрешен лишь преподавателям и персоналу, студентам же доступ к ним закрыт. В действительности управление сведено к определению *ролей* (roles) пользователей. После подтверждения соответствующей роли ей предоставляется или запрещается доступ к соответствующим ресурсам. Процесс разработки системы защиты состоит, в частности, в том, чтобы определить роли, которые могут потребоваться пользователям, и предоставить механизмы управления доступом на основе списков ролей. Мы еще поговорим о ролях в следующей главе.

Многоуровневая организация механизмов защиты

Важным моментом при разработке систем защиты является решение о том, сколько уровней должны иметь механизмы защиты. Уровень в данном контексте соответствует логической организации системы. Так, например, компьютерные сети часто построены из нескольких уровней, в соответствии с некоторой эталонной моделью, о чем мы говорили в главе 2. В главе 1 мы вводили понятие о такой структуре распределенных систем, в которой имеются отдельные уровни для

приложений, задач промежуточного уровня, служб и ядра операционной системы. Комбинируя многоуровневые структуры компьютерных сетей и распределенных систем, мы получаем схему, представленную на рис. 8.3.



Рис. 8.3. Логическая многоуровневая организация распределенных систем

В сущности, на рисунке службы общего назначения отделены от коммуникационных служб. Это разделение очень важно для понимания распределения по уровням механизмов защиты распределенных систем и, в частности, для представления о доверии. Разница между доверием и защитой очень важна. Система может быть или не быть защищенной, особенно принимая во внимание различные случайности, но мнение клиента о том, что система защищена — это вопрос доверия [351]. Уровень, на котором размещается механизм защиты, зависит от доверия клиента к защите служб этого уровня.

В качестве примера рассмотрим организацию, расположенную в нескольких географически удаленных друг от друга местах, которые соединены коммуникационной службой, такой как коммутируемая мультимегабитная служба данных (Switched Multi-megabit Data Service, SMDS). Сеть SMDS можно рассматривать как базовую сеть канального уровня, соединяющую локальные сети, в том числе и разнесенные в пространстве, как это показано на рис. 8.4 (дополнительную информацию по SMDS можно найти в [238]).

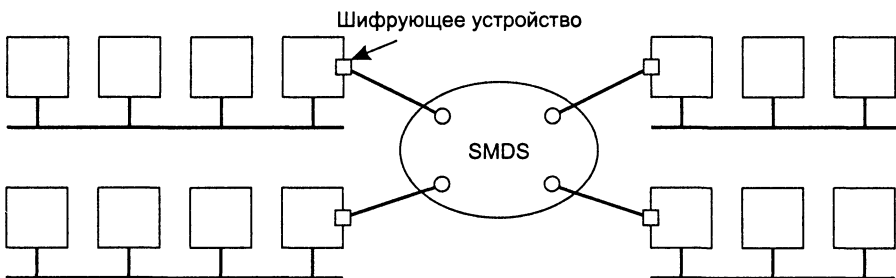


Рис. 8.4. Несколько сайтов, связанных глобальной службой магистральной

Защиту можно обеспечить путем установки шифрующего устройства на каждом маршрутизаторе SMDS, как показано на рисунке. Эти устройства автомати-

чески зашифровывают и расшифровывают пересылаемые пакеты, но не предоставляют никаких средств для организации безопасной связи в пределах одной локальной сети. Если Алиса из сети А посылает сообщение Бобу в сеть Б и беспокоится о том, что это письмо кто-нибудь перехватит, она должна быть уверена по крайней мере в том, что шифрование трафика с удаленными сетями работает успешно. Это означает, например, что она должна доверять системным администраторам обеих локальных сетей и полагать, что они принимают меры против вмешательства в работу шифрующих устройств.

Допустим теперь, что Алиса не доверяет защите трафика с удаленными сетями. Она может предпринять собственные меры, например, использовать службу защиты транспортного уровня, такую как служба *SSL (Secure Socket Layer — уровень защищенных сокетов)*, которая служит, в частности, для защищенной пересылки почты по соединениям TCP. Мы более подробно рассмотрим SSL в главе 11, когда будем обсуждать защиту в Web. Важно отметить, что SSL позволит Алисе установить защищенное соединение с Бобом. Все сообщения транспортного уровня будут зашифрованы — и на уровне SMSD тоже, но Алисе это не важно. В данном случае Алиса доверяет своей службе SSL. Другими словами, она верит, что SSL ее защитит.

В распределенных системах механизмы защиты часто размещаются на промежуточном уровне. Если Алиса не доверяет SSL, она может пожелать использовать локальную службу защиты RPC. И опять-таки она должна доверять этой службе, полагая, что та работает корректно, а в данном случае не допускает утечек информации и верно идентифицирует клиенты и серверы.

Службам защиты, размещаемым на промежуточном уровне распределенных систем, можно доверять только в том случае, если те службы, на которые они полагаются, также защищены. Так, например, если служба защиты RPC реализована частично на основе SSL, доверие к службе RPC зависит от того, насколько вы доверяете SSL. Если вы не доверяете защите SSL, то не можете доверять и уровню защиты, предоставляемому службой RPC.

Распределение механизмов защиты

Зависимости между службами, требующими доверия, приводят к понятию *доверенной вычислительной базы (Trusted Computing Base, TSB)*. TSB — это набор всех механизмов защиты (распределенной) компьютерной системы, необходимых для осуществления правил защиты. Чем меньше TSB, тем лучше. Если распределенная система построена на базе промежуточного уровня (надстройки над существующей сетевой операционной системой), ее защита может зависеть от защиты базовых локальных операционных систем. Другими словами, TSB в распределенной системе может включать в себя локальные операционные системы различных хостов.

Рассмотрим файловый сервер в распределенной файловой системе. Этот сервер может быть завязан на многие механизмы защиты, предоставляемые локальной операционной системой. В перечень этих механизмов входят не только средства защиты файлов от доступа к ним посторонних процессов, не относящихся к файловому серверу, но и механизмы защиты сервера от злонамеренного повреждения.

Распределенные системы, привязанные к промежуточному уровню, требуют поэтому доверия к локальной операционной системе, под управлением которой они работают. Если такого доверия нет, часть функциональности локальной операционной системы должна быть встроена в саму распределенную систему. Рассмотрим операционную систему с микроядром, в которой большинство служб операционной системы выполняется в виде обычных пользовательских процессов. В этом случае, например, стандартная файловая система может быть полностью заменена другой файловой системой, особо приспособленной к специфическим нуждам распределенной системы, в том числе и по части различных механизмов защиты. Отметим, что, используя подобный подход, можно отчасти заменить распределенную систему промежуточного уровня распределенной операционной системой, о которой мы говорили в главе 1.

Этому подходу будет соответствовать отделение служб защиты от прочих видов служб путем распределения этих служб по различным машинам в соответствии с той степенью защиты, которая им необходима. Так, например, для защищенной распределенной файловой системы можно отделить файловый сервер от клиентов путем установки сервера на машину с доверенной операционной системой, на которой, возможно, установлена также отдельная защищенная файловая система. Клиенты и их приложения могут быть размещены на машинах, не вызывающих доверия.

Подобное разделение эффективно уменьшает размер TCB до относительно небольшого числа аппаратных и программных компонентов. В ходе последующей защиты этих машин от внешних атак общее доверие к защите распределенной системы может быть еще увеличено. Предотвращение прямого доступа клиентов и их приложений к критически важным службам производится путем предоставления *ограниченного интерфейса к защищенным системным компонентам* (*Reduced Interfaces for Secure System Components, RISSC*), как описано в [316]. Согласно этому подходу, любой критичный к уровню защиты сервер помещается на отдельную машину, изолированную от систем конечных пользователей, и использует низкоуровневые защищенные сетевые интерфейсы, как показано на рис. 8.5. Клиенты и их приложения работают на различных машинах и могут взаимодействовать с защищенным сервером только через эти сетевые интерфейсы.

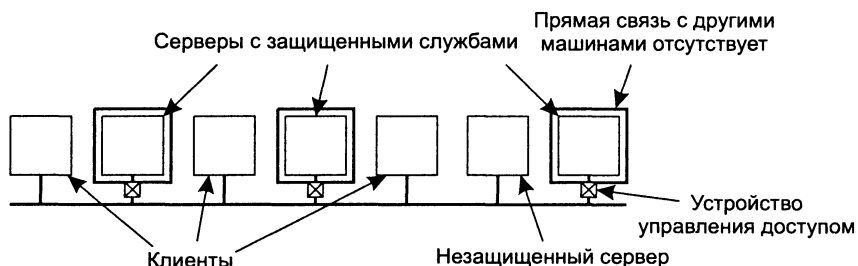


Рис. 8.5. Применение принципов RISSC к защищенным распределенным системам

Простота

Другой важный момент проектирования, касающийся уровня, на котором следует разместить механизмы защиты, — это простота. Разработка защищенных компьютерных систем обычно считается сложным делом. Соответственно, самой лучшей будет система, использующая небольшое количество простых механизмов, в которых легко разобраться и правильную работу которых легко гарантировать.

К сожалению, простых механизмов для реализации правил защиты часто недостаточно. Рассмотрим снова эпизод, когда Алиса посылает письмо Бобу. Мы обсуждали его чуть раньше. Шифрование на канальном уровне — это простой и легко понятный механизм защиты от перехвата глобального трафика. Однако если Алиса хочет, чтобы ее письма мог получить только Боб, ей необходимо кое-что еще. Она нуждается в службах аутентификации пользовательского уровня, кроме того, чтобы доверять им, Алисе может потребоваться знать о том, как они работают. Поэтому аутентификация пользовательского уровня может потребовать как минимум представления о криптографических ключах и знания определенных механизмов, таких как сертификация, даже несмотря на то, что многие службы защиты полностью автоматизированы и прозрачны для пользователя.

В других случаях само приложение может быть достаточно сложным, а введение защиты дополнительно усложняет его. Примером приложений, в которые включаются сложные протоколы защиты, являются цифровые платежные системы, которые мы рассмотрим далее в этой главе. Сложность цифровых платежей часто связана с тем, что для совершения платежа необходимо взаимодействие нескольких действующих лиц. В этих случаях важно, чтобы базовые механизмы, используемые для реализации протоколов, были относительно простыми и легко понятными. Простота будет способствовать доверию пользователей, работающих с приложением, и что более важно, сможет убедить разработчиков в отсутствии «прорех» в системе защиты.

8.1.3. Криптография

В защите распределенных систем очень важную роль играют приемы криптографии. Основная идея этих приемов проста. Рассмотрим отправителя S , который хочет переслать сообщение m получателю R . Чтобы обезопасить сообщение от угроз защиты, отправитель сначала шифрует его, превращая в непонятное сообщение m' , после чего посылает это m' получателю R . R , в свою очередь, должен расшифровать полученное сообщение и получить оригинал, то есть сообщение m .

Шифрование и расшифровка осуществляются путем применения криптографических методов с использованием ключей, как показано на рис. 8.6. Исходная форма посылаемого сообщения, называемая *простым текстом* (*plaintext*), обозначена на рисунке как P , его зашифрованный вариант, известный как *шифрованный текст* (*ciphertext*), обозначен как C .

Чтобы описать различные протоколы, используемые для построения служб защиты распределенных систем, полезно будет ввести обозначения для обычного текста, шифрованного текста и ключей. В соответствии с общими соглашениями

ниями мы будем использовать выражение $C = E_K(P)$ для описания того факта, что шифрованный текст C был получен путем шифрования простого текста P с использованием ключа K . Точно так же $P = D_K(C)$ обозначает операцию расшифровки шифрованного текста C с использованием ключа K , приводящую к получению простого текста P .

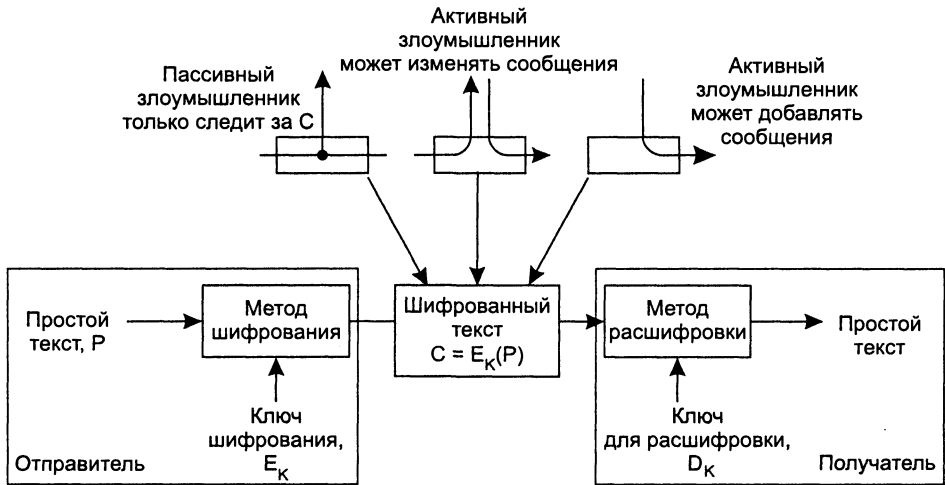


Рис. 8.6. Проникновение и подслушивание при взаимодействии

Возвращаясь к нашему примеру, при передаче сообщения в виде шифрованного текста C мы должны защищаться от трех различных типов атак, и в этом нам прекрасно помогает шифрование. Во-первых, злоумышленник может перехватить сообщение так, что об этом не узнают ни отправитель, ни получатель. Разумеется, если передаваемое сообщение зашифровано таким образом, что его невозможно расшифровать, не имея соответствующего ключа, перехват бесполезен: злоумышленник сможет увидеть только непонятные данные. (Кстати говоря, самого факта передачи сообщений может иногда быть достаточно для злоумышленника, чтобы сделать определенные выводы. Так, например, если в ходе мирового кризиса объем входящего трафика Белого дома внезапно падает почти до нуля, а объем трафика определенной точки в Скалистых горах, штат Колорадо, соответственно вырастает, это может оказаться очень полезной информацией.)

Второй тип атак, который нам следует рассмотреть, это изменение сообщений. Изменить простой текст очень легко, изменить грамотно зашифрованный текст гораздо сложнее, поскольку для внесения в него осмысленных изменений злоумышленник должен сначала расшифровать сообщение. Кроме того, он должен быть в состоянии корректно зашифровать его, в противном случае получатель может заметить, что сообщение подделано.

Третий тип атак — добавление злоумышленником шифрованного сообщения в систему коммуникации с попыткой заставить R поверить, что это сообщение пришло от S . И снова, как мы увидим далее в этой главе, шифрование может помочь нам защититься от подобных атак. Отметим, что если злоумышленник в со-

стоянии изменить сообщения, он может и добавлять в систему собственные сообщения.

Существует фундаментальное разделение криптографических систем на системы с одинаковыми и различными ключами шифрования и дешифровки. В случае *симметричной криптосистемы* (*symmetric cryptosystem*) для шифрования и расшифровки сообщения используется один и тот же ключ. Иными словами:

$$P = D_K(E_K(P)).$$

Симметричные криптосистемы также именуются системами с секретным, или общим, ключом, поскольку отправитель и получатель должны совместно использовать один и тот же ключ, и, чтобы гарантировать защиту, этот общий ключ должен быть секретным. Никто другой этот ключ видеть не должен. Мы будем использовать для такого ключа, разделяемого A и B , обозначение $K_{A,B}$.

В *асимметричной криптосистеме* (*asymmetric cryptosystem*) ключи шифрования и расшифровки различны, но вместе образуют уникальную пару. Другими словами, существует отдельный ключ шифрования K_E и отдельный ключ расшифровки, K_D , так что:

$$P = D_{K_D}(E_{K_E}(P)).$$

Один из ключей асимметричной криптосистемы является закрытым, другой — открытым. По этой причине асимметричные криптосистемы также называют *системами с открытым ключом* (*public-key systems*). Поэтому мы будем использовать обозначение K_A^+ для обозначения открытого ключа, принадлежащего A , а K_A^- — для соответствующего ему закрытого ключа.

Предваряя подробное обсуждение протоколов защиты, скажем, что какой из ключей (шифрующий или дешифрующий) будет сделан открытым, зависит от того, как эти ключи используются. Например, если Алиса хочет послать Бобу конфиденциальное сообщение, она будет использовать для шифрования сообщения открытый ключ Боба. Поскольку Боб — единственный, кто обладает закрытым ключом расшифровки, он будет также единственным, кто сможет расшифровать сообщение.

С другой стороны, предположим, что Боб хочет быть уверен, что сообщение, которое он только что получил, действительно отправлено Алисой. В этом случае Алиса может использовать для шифрования сообщения свой закрытый ключ. Если Боб в состоянии успешно расшифровать сообщение, используя открытый ключ Алисы (и расшифрованный текст сообщения содержит достаточно информации, чтобы убедить Боба), он знает, что сообщение было послано Алисой, поскольку ключ расшифровки однозначно связан с ключом шифрования. Далее мы рассмотрим этот алгоритм подробнее.

Еще одно применение криптографии в распределенных системах — это *хэш-функции* (*hash functions*). Хэш-функция H принимает на входе сообщение m произвольной длины и создает строку битов h фиксированной длины:

$$h = H(m).$$

Хэш h можно сравнить с дополнительными битами, добавляемыми к сообщению в коммуникационной системе для обнаружения ошибок, например, при циклическом избыточном кодировании (CRC).

Хэш-функции, используемые в криптографических системах, имеют определенные свойства. Во-первых, это *односторонние функции* (*one-way functions*). То есть вычисление входного сообщения m по известному результату их работы h невозможно. С другой стороны, вычислить h по m достаточно несложно. Во-вторых, они обладают свойством *слабой устойчивости к коллизиям* (*weak collision resistance*), означающим, что если заданы исходное сообщение m и соответствующий ему результат $h = H(m)$, невозможно вычислить другое сообщение m' , такое, что $H(m) = H(m')$. И наконец, криптографическая хэш-функция также обладает свойством *сильной устойчивости к коллизиям* (*strong collision resistance*). Это значит, что если дана только хэш-функция H , невозможно вычислить любые два различных входных значения m и m' , такие что $H(m) = H(m')$.

Подобные свойства приложимы к любой функции шифрования E и используемым ключам. Кроме того, для любой функции шифрования E не должно быть возможности вычислить используемый ключ K , имея простой текст P и соответствующий ему зашифрованный текст $C = E_K(P)$. Также аналогично устойчивости к коллизиям, при наличии простого текста P и ключа K невозможно вычислить другой подходящий ключ K' , такой, что $E_K(P) = E_{K'}(P)$.

Искусство и наука разработки алгоритмов для криптографических систем имеют длинную историю [222], и построение защищенных систем часто оказывается неожиданно трудной или даже невозможной задачей [405]. В задачи этой книги не входит детальное рассмотрение подобных алгоритмов. Однако для того, чтобы дать определенное представление о криптографии в компьютерных системах, мы кратко представим три показательных криптографических алгоритма. Подробную информацию по этим и другим криптографическим алгоритмам можно найти в [228, 290, 404].

Перед тем как мы погрузимся в детали различных протоколов, обобщим используемую нотацию и аббревиатуры (табл. 8.1).

Таблица 8.1. Нотация, используемая в этой главе

Нотация	Описание
$K_{A,B}$	Секретный ключ, совместно используемый A и B
K_A	Открытый ключ A
K_A	Закрытый ключ A

Симметричные криптосистемы — DES

Нашим первым примером криптографического алгоритма будет *стандарт шифрования данных* (*Data Encryption Standard, DES*), который используется в симметричных криптосистемах. Алгоритм DES создавался для работы с 64-битовыми блоками данных. Блок преобразуется в зашифрованный (64-битный) блок за 16 шагов, на каждом из которых используется собственный 48-битный ключ шифрования. Каждый из этих 16 ключей порождается из 56-битного главного ключа, как показано на рис. 8.7, а. Перед тем как над исходным блоком начнут выполняться его 16 циклов шифрования, проводится его первичное преобразование, а после шифрования производится преобразование, обратное первичному. Результатом является зашифрованный блок.

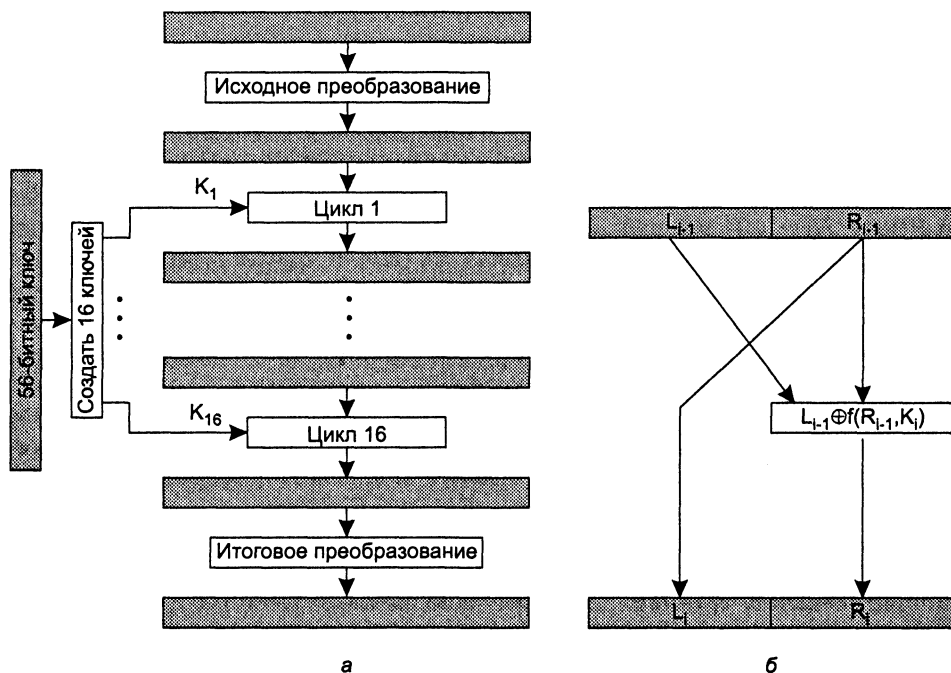


Рис. 8.7. Алгоритм DES (а). Схема одного цикла шифрования (б)

Каждый цикл шифрования i берет в качестве исходных данных 64-битный блок с предыдущего цикла шифрования $i - 1$, как показано на рис. 8.7, б. 64 бита разбиваются на левую часть L_{i-1} и правую часть R_{i-1} , по 32 бита каждая. Правая часть в следующем цикле используется в качестве левой, и наоборот.

Главная работа выполняется в искажающей функции f . Эта функция принимает в качестве исходных данных 32-битный блок R_{i-1} и 48-битный ключ K_i , а затем генерирует 32-битный блок, который подвергается операции «исключающее ИЛИ» (XOR) с блоком L_{i-1} , порождая R_i . Искажающая функция сначала расширяет R_{i-1} до 48 бит, а затем проводит над ним операцию «исключающее ИЛИ» с ключом K_i . Результат делится на восемь частей по шесть бит. Каждая часть протягивается затем через различные S -боксы (S -boxes), которые представляют собой операции замены каждой из возможных 64 6-битных комбинаций на одну из 16 возможных 4-битных комбинаций. Восемь получившихся кусков по 4 бита объединяются в одно 32-битовое значение и преобразуются далее.

48-битный ключ K_i порождается из 56-битного главного ключа следующим образом. Сначала главный ключ преобразуется и делится на две 28-битных половины. На каждом цикле первая половина сдвигается на один или два бита влево, после чего из нее выделяются 24 бита. Вместе с 24 битами из второй сдвинутой половины они образуют 48-битный ключ. Детально один цикл шифрования иллюстрирует рис. 8.8.

Алгоритм DES достаточно прост, но его нелегко взломать с использованием аналитических методов. «Грубая сила» (простой подбор ключа) значительно

проще. В настоящее время надежно защищает от взлома «тройное» использование алгоритма DES в специальном режиме шифрование — расшифровка — шифрование с разными ключами.

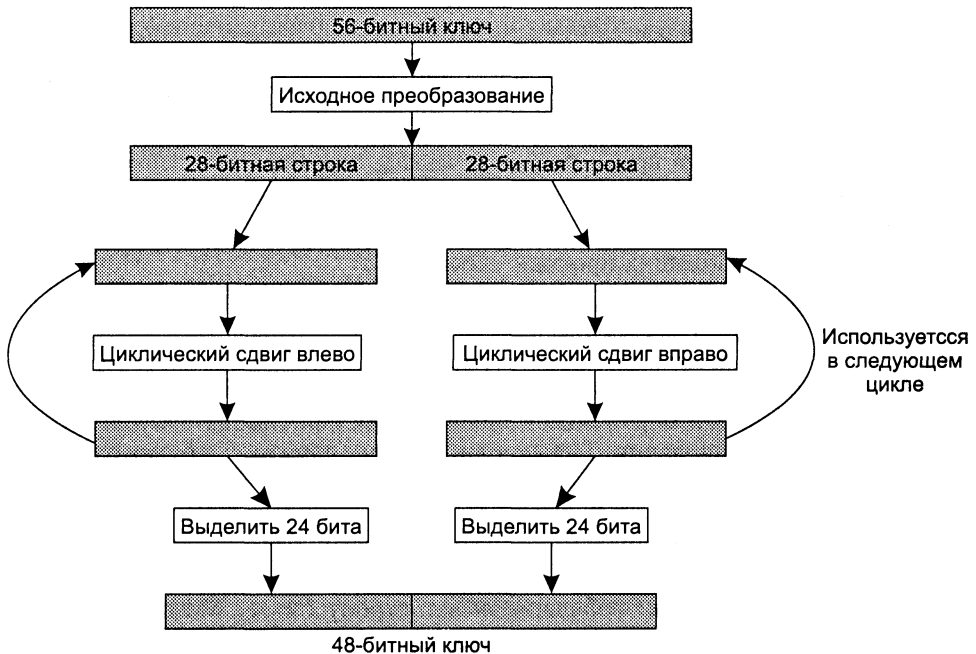


Рис. 8.8. Техника циклической генерации ключа в алгоритме DES

Аналитические атаки на DES сильно усложняют тот факт, что структура этого алгоритма никогда не была полностью описана в открытой документации. Так, например, можно показать, что применение нестандартных S-боксов значительно упрощает взлом алгоритма (краткий анализ алгоритма DES см. в [351]). Рекомендации по структуре и использованию S-боксов были опубликованы только после новых «моделей» атак, разработанных в девяностых годах. Алгоритм DES доказал свою устойчивость к этим атакам, и его разработчики открыли, что они знали об этих «новых» моделях еще в 1974 году, когда разрабатывали DES [111].

DES уже много лет используется в качестве стандартной технологии шифрования, но в настоящее время происходит процесс его замены алгоритмом Риндаля (Rijndael) с блоками длиной 128 бит. Также имеются варианты с большими ключами и более длинными блоками. Разработанный алгоритм достаточно быстр, чтобы его можно было реализовать даже в смарт-картах. Важность этой области применения криптографии растет с каждым днем.

Криптосистемы с открытым ключом — RSA

В качестве второго примера криптографического алгоритма рассмотрим широко используемую систему с открытым ключом — RSA [382], названную в честь ее изобретателей — Ривеста (Rivest), Шамира (Shamir) и Альдемана (Aldeman).

Защита RSA вытекает из того факта, что в настоящее время не существует эффективного метода нахождения простых множителей больших чисел. Можно показать, что любое целое число можно записать в виде произведения простых чисел. Так, например, 2100 может быть записано как:

$$2100 = 2 \times 2 \times 3 \times 5 \times 5 \times 7.$$

Таким образом, 2, 3, 5 и 7 являются простыми множителями числа 2100. В RSA открытый и закрытый ключи создаются из очень больших простых чисел (содержащих сотни десятичных цифр). Взлом RSA эквивалентен обнаружению этих чисел. В настоящее время подобная задача неразрешима, несмотря на то, что математики работают над ней уже несколько столетий.

Создание открытых и закрытых ключей происходит в четыре этапа.

1. Выбираются два больших простых числа, p и q .
2. Вычисляется их произведение $n = p \times q$ и $z = (p - 1) \times (q - 1)$.
3. Выбирается число d , взаимно простое с z .
4. Вычисляется число e , такое, что $e \times d = 1 \bmod z$.

Одно из чисел, например d , может впоследствии использоваться для расшифровки, а e — для шифрования. Только одно из этих двух чисел будет открытым, какое именно — зависит от используемого алгоритма.

Рассмотрим вариант, когда Алиса хочет, чтобы сообщение, которое она посылает Бобу, было конфиденциально. Другими словами, она хочет быть уверена в том, что никто кроме Боба не сможет перехватить и прочитать ее сообщение. RSA рассматривает любое сообщение m как строку битов. Каждое сообщение сначала разбивается на блоки фиксированной длины, и каждый очередной блок m_i представляется в виде числа в двоичном виде, лежащего в интервале $0 \leq m_i < n$.

Для шифрования сообщения m отправитель вычисляет для каждого блока m_i значение $c_i = m_i^e \bmod n$, которое и отправляется получателю. Расшифровка на стороне получателя производится путем вычисления $m_i = c_i^d \bmod n$. Отметим, что для шифрования необходимы e и n , в то время как для расшифровки — d и n .

Сравним RSA с симметричными криптосистемами, такими как DES. Для RSA характерен недостаток — сложность вычислений, которая приводит к тому, что расшифровка сообщений, зашифрованных по алгоритму RSA, занимает в 100–1000 раз больше времени, чем расшифровка сообщений, зашифрованных по алгоритму DES. Точный показатель зависит от реализации. В результате многие криптографические системы используют RSA только для безопасного обмена общими секретными ключами, избегая применять этот способ для шифрования «обычных» данных. Позже мы рассмотрим примеры комбинированного использования этих двух технологий.

Хэш-функции — MD5

В качестве последнего примера широко используемого криптографического алгоритма мы рассмотрим MD5 [381]. MD5 — это хэш-функция для вычисления 128-битных *дайджестов сообщений* (*message digests*) фиксированной длины из двоичных исходных строк произвольной длины. Сначала исходная строка дополняется до общей длины в 448 бит (по модулю 512), после чего к ней добавляется

длина исходной строки в виде 64-битового целого числа. В результате исходные данные преобразуются в набор 512-битных блоков.

Структура алгоритма приведена на рис. 8.9. Начиная с определенного постоянного 128-битового значения, алгоритм включает в себя k фаз, где k — число 512-битных блоков, получившихся из дополненного согласно алгоритму сообщения. В ходе каждой из фаз из 512-битного блока данных и 128-битного дайджеста, вычисленного на предыдущей фазе, вычисляется новый 128-битный дайджест. Фаза алгоритма MD5 состоит из четырех циклов вычислений, на каждом из которых используется одна из следующих четырех функций:

$$\begin{aligned} F(x, y, z) &= (x \text{ AND } y) \text{ OR } ((\text{NOT } x) \text{ AND } z), \\ G(x, y, z) &= (x \text{ AND } z) \text{ OR } (y \text{ AND } (\text{NOT } z)), \\ H(x, y, z) &= x \text{ XOR } y \text{ XOR } z, \\ I(x, y, z) &= y \text{ XOR } (x \text{ OR } (\text{NOT } z)). \end{aligned}$$

Каждая из этих функций работает с 32-битными переменными x, y и z . Чтобы проиллюстрировать, как используются эти функции, рассмотрим 512-битный блок b дополненного сообщения на фазе k . Блок b разделяется на 16 32-битных подблоков b_0, b_1, \dots, b_{15} . Как показано на рис. 8.10, в ходе первого цикла для изменения четырех переменных (назовем их p, q, r и s соответственно) в ходе 16 итераций используется функция F . Эти переменные переносятся на очередной цикл, а после окончания этой фазы передаются на следующую. Всего существует 64 заранее определенных констант C_i . Для указания циклического сдвига влево используется запись $x \lll n$: биты в x сдвигаются на n позиций, причем биты, ушедшие за левую границу числа, добавляются справа.

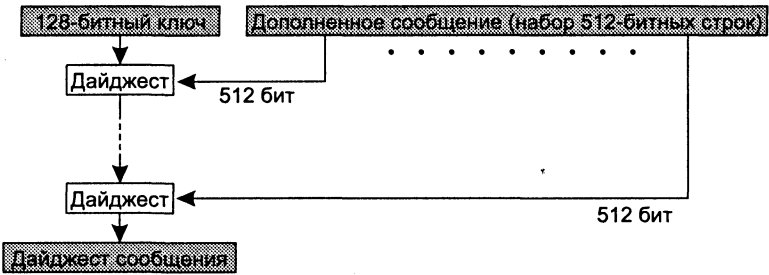


Рис. 8.9. Структура MD5

Итерации 1–8	Итерации 19–16
$p \leftarrow (p + F(q, r, s) + b_0 + C_1) \lll 7$	$p \leftarrow (p + F(q, r, s) + b_8 + C_9) \lll 7$
$s \leftarrow (s + F(p, q, r) + b_1 + C_2) \lll 12$	$s \leftarrow (s + F(p, q, r) + b_9 + C_{10}) \lll 12$
$r \leftarrow (r + F(s, p, q) + b_2 + C_3) \lll 17$	$r \leftarrow (r + F(s, p, q) + b_{10} + C_{11}) \lll 17$
$q \leftarrow (q + F(r, s, p) + b_3 + C_4) \lll 22$	$q \leftarrow (q + F(r, s, p) + b_{11} + C_{12}) \lll 22$
$p \leftarrow (p + F(q, r, s) + b_4 + C_5) \lll 7$	$p \leftarrow (p + F(q, r, s) + b_{12} + C_{13}) \lll 7$
$s \leftarrow (s + F(p, q, r) + b_5 + C_6) \lll 12$	$s \leftarrow (s + F(p, q, r) + b_{13} + C_{14}) \lll 12$
$r \leftarrow (r + F(s, p, q) + b_6 + C_7) \lll 17$	$r \leftarrow (r + F(s, p, q) + b_{14} + C_{15}) \lll 17$
$q \leftarrow (q + F(r, s, p) + b_7 + C_8) \lll 22$	$q \leftarrow (q + F(r, s, p) + b_{15} + C_{16}) \lll 22$

Рис. 8.10. 16 итераций первого цикла MD5

Во втором цикле подобным же образом используется функция G , а H и I — соответственно в третьем и четвертом циклах. Каждый шаг, таким образом, включает в себя 64 итерации, и в очередной фазе используются вычисленные на предыдущей фазе значения p , q , r и s .

8.2. Защищенные каналы

В предыдущих главах в качестве стандартного способа организации распределенных систем мы часто упоминали модель клиент-сервер. В этой модели серверы могут быть распределенными, реплицироваться, а также выступать в качестве клиентов по отношению к другим серверам. При обсуждении вопросов защиты в распределенных системах можно вновь вести разговор на примере клиентов и серверов. В частности, построение защищенной распределенной системы на практике сводится к двум основным моментам. Первый из них — построение защищенной связи между клиентом и сервером. Защищенная связь требует аутентификации общающихся сторон, кроме того, она гарантирует целостность сообщений, а возможно, и их конфиденциальность. Связь серверов внутри группы также можно рассматривать в качестве частного случая этой задачи.

Второй момент — способ авторизации. Как серверу, получившему запрос от клиента, распознать факт авторизации клиента для передачи этого запроса на обработку? Авторизация связана с проблемой управляемого доступа к ресурсам, который мы будем углубленно рассматривать в следующем разделе. Здесь мы сосредоточимся на защите связи в распределенных системах.

Идею о защите связи между клиентами и серверами можно изложить в понятиях организации между общающимися сторонами *защищенного канала* (*secure channel*) [479]. Защищенные каналы оберегают отправителя и получателя от перехвата, модификации и подделки сообщения. Обычно нет необходимости вводить защиту от прерывания связи. Защита сообщений от перехвата выполняется путем гарантированной конфиденциальности: защищенные каналы гарантируют, что сообщения в них не могут быть подсмотрены злоумышленниками. Защита сообщений от модификации или подделки производится при помощи протоколов взаимной аутентификации и целостности сообщений. Далее мы сначала обсудим различные протоколы аутентификации на основе криптосистем как с симметричными, так и с открытыми ключами. Детальный разбор логических построений, лежащих в основе аутентификации, можно найти в [254]. Мы рассмотрим отдельно вопросы конфиденциальности сообщений и их целостности.

8.2.1. Аутентификация

Перед тем как мы углубимся в детали различных протоколов аутентификации, следует заметить, что аутентификация и целостность сообщений не существуют отдельно друг от друга. Рассмотрим, например, распределенную систему, которая поддерживает аутентификацию двух общающихся между собой агентов, но не предоставляет механизмов проверки целостности сообщений. В такой системе

Боб может быть уверен, что сообщение m отправила Алиса. Однако, если Боб не может получить гарантии, что сообщение m в ходе пересылки не изменялось, что ему толку в знании того факта, что исходную версию сообщения отправила именно Алиса?

Представим таким же образом поддержку одной только целостности сообщений без механизма аутентификации. Когда Боб получает сообщение, в котором говорится, что он выиграл 1 000 000 долларов в лотерею, стоит ли ему радоваться, если он не в состоянии проверить, действительно ли это письмо прислано организаторами лотереи?

Таким образом, аутентификация и целостность сообщений должны идти «рука об руку». Во многих протоколах их комбинация работает примерно следующим образом. Предположим, что Алисе и Бобу снова не терпится пообщаться и Алиса берет инициативу по организации канала в свои руки. Она начинает этот процесс с посылки сообщения Бобу или доверенному третьему лицу, которое поможет устанавливать канал. Когда канал установлен, Алиса уверена, что она общается с Бобом, а Боб уверен, что общается с Алисой. Теперь они могут начать обмениваться сообщениями.

Чтобы в дальнейшем обеспечить целостность сообщений, которыми они будут обмениваться после аутентификации, обычно используется криптография с секретным ключом с использованием сеансовых ключей. Сеансовый ключ (session key) — это общий ключ, используемый для шифрования сообщений с целью соблюдения их целостности, а также, возможно, конфиденциальности. Этот ключ обычно требуется только в течение времени существования канала. После закрытия канала соответствующий сеансовый ключ отменяется (или уничтожается с соблюдением мер защиты). Позже мы еще вернемся к сеансовым ключам.

Аутентификация на основе общего секретного ключа

Начнем с того, что рассмотрим протокол аутентификации на основе секретного ключа, который используется Алисой и Бобом постоянно. Как эти двое могут в реальности безопасным образом получить общий ключ, мы обсудим чуть позже. В описании протокола мы сократим Алису и Боба соответственно до A и B , а совместно используемый ими ключ обозначим как $K_{A,B}$. Обычно запрос одной из сторон вызывает ответ другой, который будет верен только в том случае, если эта другая сторона знает общий секретный ключ. Такие решения известны под названием *протоколов запрос-ответ* (challenge-response protocols).

В случае аутентификации на базе общего секретного ключа протокол работает так, как показано на рис. 8.11. Сначала Алиса посылает Бобу свой идентификатор (сообщение 1), показывая, что она хочет установить между ними канал связи. После этого Боб посылает Алисе запрос R_B , обозначенный как сообщение 2. От Алисы требуется зашифровать запрос секретным ключом $K_{A,B}$, который она использует совместно с Бобом, и вернуть этот запрос Бобу. Ответ, показанный на рисунке в виде сообщения 3, содержит $K_{A,B}(R_B)$.

Когда Боб получит ответ $K_{A,B}(R_B)$ на свой запрос R_B , он должен расшифровать сообщение, снова применяя общий ключ, чтобы убедиться, что в сообщении содержится запрос R_B . Если это так, то он считает, что на другой стороне канала

находится Алиса, иначе кто же еще мог зашифровать запрос R_B с использованием ключа $K_{A,B}$? Другими словами, Боб теперь уверен, что действительно общается с Алисой. Однако отметим, что Алиса пока еще не уверена в том, что на другой стороне канала действительно Боб. Поэтому она посылает запрос R_A (сообщение 4), на который Боб отвечает, возвращая $K_{A,B}(R_A)$, сообщение 5. Когда Алиса расшифрует его при помощи ключа $K_{A,B}$ и обнаружит в нем запрос R_A , она поймет, что общается с Бобом.

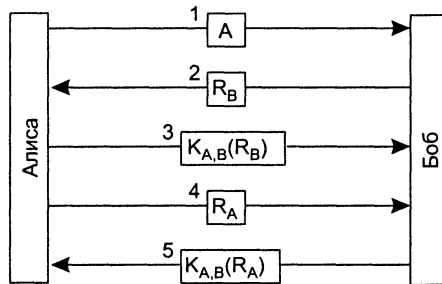


Рис. 8.11. Аутентификация на базе совместно используемого секретного ключа

Один из наиболее сложных моментов в обеспечении защиты — создание действительно работающих протоколов. Чтобы показать, как простые решения могут привести к ошибке, рассмотрим «оптимизацию» протокола аутентификации, в которой число сообщений сокращено с пяти до трех, как показано на рис. 8.12. Основная идея состоит в том, что если Алиса, в конце концов, все равно захочет отправить Бобу запрос, она вполне может отправить его вместе со своим уведомлением при создании канала. Точно так же и Боб, возвращая свой ответ на этот запрос, объединяет его в одно сообщение с собственным запросом.

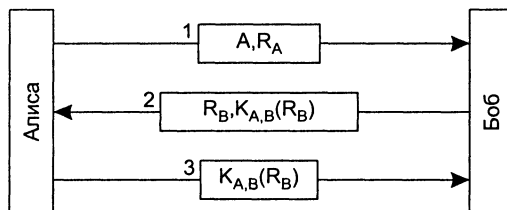


Рис. 8.12. Аутентификация на базе общего секретного ключа с тремя, а не пятью сообщениями

К сожалению, долго этот протокол не проработает. Он с легкостью будет вскрыт при помощи так называемой *атаки на отражении (reflection attack)*. Чтобы понять, что представляет из себя эта атака, предположим, что злоумышленник зовут Чак, мы будем обозначать его в нашем протоколе как C . Цель Чака — установить канал связи с Бобом и сделать так, чтобы Боб поверил, что разговаривает с Алисой. Чак может сделать это, если ему удастся правильно ответить на запрос Боба, например, вернув зашифрованную версию числа, присланного Бобом.

Поскольку он не знает ключа $K_{A,B}$, это шифрование может произвести только сам Боб, и это именно то, на чем Чак должен надуть Боба.

Атаку иллюстрирует рис. 8.13. Чак начинает с отправки сообщения, содержащего уведомление A якобы от Алисы вместе с запросом R_C . Боб возвращает свой запрос R_B и ответ $K_{A,B}(R_C)$ в одном сообщении. После этого Чак должен показать, что он знает секретный ключ, вернув Бобу $K_{A,B}(R_B)$. К сожалению, у него нет ключа $K_{A,B}$. Но он может установить второй канал связи с Бобом, чтобы тот сам зашифровал ему свой запрос.

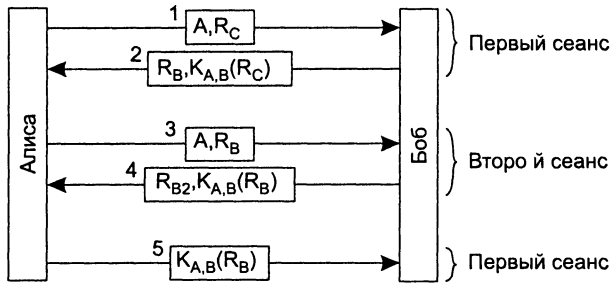


Рис. 8.13. Атака на отражении

Таким образом, Чак отправляет A и R_B в одном сообщении, как и раньше, но теперь притворяется, что ему нужен второй канал. На рис. 8.13 этому соответствует сообщение 3. Боб, не понимая, что он сам использовал ранее R_B в качестве запроса, отвечает, посылая $K_{A,B}(R_B)$ и другой запрос, R_{B2} , как это показано в сообщении 4. В этот момент Чак получает $K_{A,B}(R_B)$ и заканчивает первый сеанс посылкой сообщения 5, содержащего отклик $K_{A,B}(R_B)$, который и является ответом на запрос, присланный ему в сообщении 2.

Как было показано в [228], одна из ошибок, сделанных при адаптации оригинального протокола, состояла в том, что две стороны в новой версии протокола могут использовать одинаковые запросы в двух разных версиях протокола. Лучше было бы, если бы инициатор и его корреспондент использовали разные запросы. Так, например, если Алиса всегда использует нечетные числа, а Боб — четные, Боб смог бы догадаться, что задумал какой-то хитрец, приславший ему в качестве запроса в сообщении 3 R_B (см. рис. 8.13). К сожалению, как показано в [404], такое решение неустойчиво к другим типам атак, например, так называемым атакам посредника. Вообще, позволять двум сторонам, вовлеченным в организацию защищенного канала, проделывать какие-то вещи одинаковым образом — не лучшая идея.

Другой момент, который делает адаптированный протокол непригодным, — то, что Боб отправляет важную информацию в виде отклика, $K_{A,B}(R_B)$, не будучи уверенным в том, кому он ее посылает. Этот момент в исходном протоколе отсутствует, ведь Алиса должна сначала подтвердить себя, и только после этого Боб пошлет ей зашифрованную информацию.

Существуют и другие принципы, к пониманию которых постепенно, за много лет, пришли разработчики криптографических систем. Мы рассмотрим некото-

рые из них при обсуждении других протоколов. Один из важных уроков состоит в том, что разрабатывать протоколы защиты, которые делают то, на что они рассчитаны, часто значительно труднее, чем кажется. Итак, как мы только что видели, упрощая существующие протоколы с целью поднятия их производительности, можно легко сделать эти протоколы непригодными. Дополнительную информацию о принципах разработки протоколов можно получить из [1].

Аутентификация с использованием центра распространения ключей

Одна из проблем, возникающих при работе с общим секретным ключом, — это проблема масштабируемости. Если распределенная система включает в себя N хостов и каждый хост должен использовать секретный ключ совместно с другими $N - 1$ хостами, система в целом вынуждена содержать $N(N - 1)/2$ ключей, а каждый хост — поддерживать $N - 1$ из них. При больших N это становится проблемой. Альтернативой является подход с использованием *центра распространения ключей* (*Key Distribution Center, KDC*). KDC позволяет каждому из хостов применять собственный ключ, при этом ни одной паре хостов специальный секретный ключ не требуется. Другими словами, при наличии KDC требуется поддерживать только N ключей вместо $N(N - 1)/2$, а это уже явное улучшение.

Когда Алиса хочет установить защищенный канал связи с Бобом, она может сделать это при помощи доверенного центра KDC. Основная идея состоит в том, что KDC поддерживает ключи Алисы и Боба, которые они могут использовать при обмене сообщениями, что и показано на рис. 8.14.

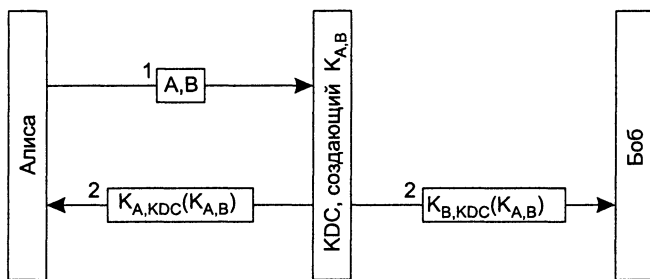


Рис. 8.14. Принцип использования KDC

Сначала Алиса посылает в центр KDC сообщение о том, что она хочет связаться с Бобом. KDC отвечает Алисе, посылая сообщение, содержащее общий секретный ключ $K_{A,B}$, который она может задействовать для этой цели. Сообщение зашифровано секретным ключом $K_{A,KDC}$, который Алиса использует совместно с KDC. Кроме того, KDC и Бобу посылает ключ $K_{A,B}$, но зашифрованный теперь уже секретным ключом $K_{B,KDC}$, который используется центром совместно с Бобом.

Основной минус такого подхода состоит в том, что Алиса может захотеть начать организацию защищенного канала связи с Бобом раньше, чем Боб получит их общий ключ от KDC. Кроме того, KDC может потребовать поставить Боба

в цикл ожидания для получения им ключа. Эти проблемы можно обойти, если KDC просто вернет $K_{B,KDC}(K_{A,B})$ Алисе и возложит на нее всю ответственность за дальнейшую организацию связи с Бобом. Это приведет нас к протоколу, показанному на рис. 8.15. Сообщение $K_{B,KDC}(K_{A,B})$ также известно под названием *талона* (*ticket*). Дело Алисы передать этот талон Бобу. Отметим, что Боб — единственный, кто может осмысленно использовать этот талон, поскольку он — единственный, кроме центра KDC, кто знает, как расшифровать это сообщение и извлечь из него полезную информацию.

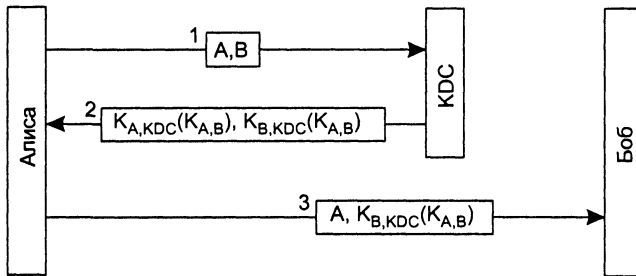


Рис. 8.15. Использование талона и передача Алисе права на установление контакта с Бобом

Протокол, показанный на рис. 8.15, на самом деле является одним из вариантов протокола аутентификации с использованием KDC, известного под названием *протокола аутентификации Нидхема—Шредера* (*Needham—Schroeder authentication protocol*), названного так по имени его создателей [311]. Другой вариант этого протокола используется в системе Kerberos, которую мы рассмотрим чуть позднее. Протокол Нидхема—Шредера — это многоканальный протокол запрос/ответ, работающий следующим образом (рис. 8.16).

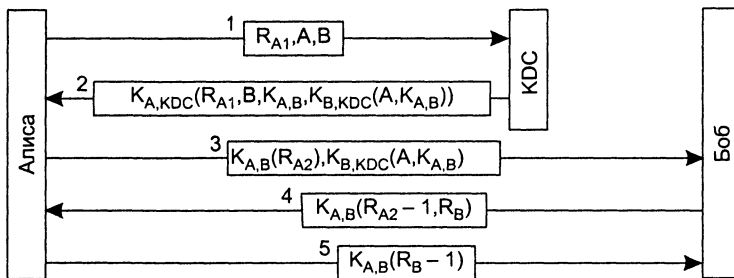


Рис. 8.16. Протокол аутентификации Нидхема—Шредера

Когда Алиса собирается организовать защищенный канал связи с Бобом, она посылает KDC сообщение, содержащее запрос R_A , вместе со своим идентификатором A и, разумеется, идентификатором Боба. KDC отвечает на этот запрос, выдавая ей талон $K_{B,KDC}(K_{A,B})$ вместе с секретным ключом $K_{A,B}$, который она далее будет использовать совместно с Бобом.

Запрос R_{A1} , который Алиса посылает KDC вместе со своим требованием на создание канала связи с Бобом, также известен под названием *nonce* и представляет собой случайное число, которое используется только однажды. Основная задача этого «одноразового» случайного числа — единственным образом привязать одно сообщение из пары к другому, в данном случае — связать сообщения 1 и 2. В частности, при включении R_{A1} в сообщение 2 Алиса будет твердо знать, что сообщение 2 отправлено в ответ на сообщение 1 и не является, например, ответом на другое, более старое сообщение.

Чтобы как следует понять проблему, допустим, что мы не используем случайное число, а Чак ухитрился стянуть один из старых ключей Боба, назовем его $K_{B,KDC}^{old}$. Кроме того, Чак перехватил старый ответ $K_{A,KDC}(B, K_{A,B}, K_{B,KDC}^{old}(A, K_{A,B}))$, который вернул центр KDC в ответ на предыдущий запрос Алисы на организацию связи с Бобом. Тем временем Боб договорился с KDC об использовании нового общего секретного ключа. Однако Чак терпеливо ожидает того момента, когда Алиса вновь начнет организовывать защищенный канал связи с Бобом. В этот момент он воспроизводит старый ответ и обманывает Алису, заставляя ее поверить, что она действительно общается с Бобом, поскольку ее талон расшифрован, а значит, он доказал, что знает их общий секретный ключ $K_{A,B}$.

Включив в сообщение случайное число, мы делаем подобную атаку невозможной, поскольку воспроизведение старого ответа будет немедленно раскрыто. В частности, случайное число ответа не совпадет со случайным числом исходного сообщения.

Сообщение 2 также содержит B , идентификатор Боба. Путем включения B в сообщение центр KDC защищает Алису от очередной атаки. Предположим, что идентификатор B в сообщении 2 отсутствует. Чак может модифицировать сообщение 1, заменив идентификатор Боба своим идентификатором, скажем, C . После этого KDC решит, что Алиса хочет создать защищенный канал с Чаком и ответит в соответствии с этим предположением. Как только Алиса захочет пообщаться с Бобом, Чак будет перехватывать ее сообщения и обманывать Алису, заставляя ее поверить, что она связана с Бобом. При копировании идентификатора из сообщения 1 в сообщение 2 Алиса немедленно обнаружит, что ее запрос был изменен.

После того как KDC передаст Алисе талон, можно организовывать защищенный канал между Алисой и Бобом. Алиса начинает этот процесс с послышки сообщения 3, которое содержит талон для Боба и запрос R_{A2} , зашифрованный их общим ключом $K_{A,B}$, который только что был создан KDC. Боб расшифровывает талон, чтобы получить общий ключ, и возвращает отклик, $R_{A2} - 1$, вместе с запросом R_B к Алисе.

Следует сделать по поводу сообщения 4 следующее замечание. Вообще говоря, возвращая отклик $R_{A2} - 1$, а не просто R_{A2} , Боб не только доказывает, что он знает общий секретный ключ, но также и то, что он действительно расшифровал запрос. И вновь это связывает сообщение 4 с сообщением 3, таким же образом, как случайное число R_A связывало сообщение 2 с сообщением 1. Таким образом, протокол становится более защищенным от повторного воспроизведения сообщений.

Однако в этом конкретном случае может быть достаточно вернуть $K_{A,B}(R_{A2}, R_B)$ по той простой причине, что это сообщение нигде ранее в протоколе не используется. Отклик $K_{A,B}(R_{A2}, R_B)$ всегда доказывает, что Боб в состоянии расшифровать запрос, присланный ему в сообщении 1. Сообщение 4, так как оно приведено на рис. 8.16, сохранено по историческим причинам.

У приведенного здесь алгоритма Нидхема—Шредера имеется одно слабое место. Если Чаку каким-то образом удастся завладеть старым ключом $K_{A,B}$, он может просто повторить посылку сообщения 3, требуя от Боба установки канала. Боб может поверить, что он общается с Алисой, хотя на самом деле на другом конце канала будет Чак. В этом случае мы должны соотнести сообщение 3 с сообщением 1, то есть сделать так, чтобы ключ зависел от исходного запроса, посылаемого Алисой для организации канала связи с Бобом. Это решение представлено на рис. 8.17.

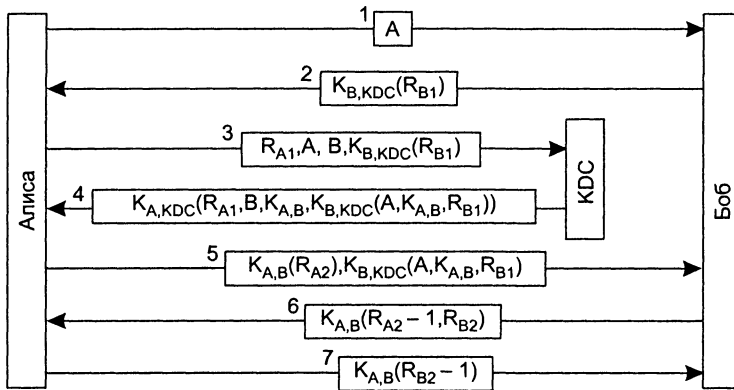


Рис. 8.17. Защита от злонамеренного повторного использования сгенерированного ранее сеансового ключа в протоколе Нидхема—Шредера

Фокус состоит в том, чтобы включить случайное число в ответ, посылаемый Алисе из KDC. Причем случайное число должно прийти от Боба — это уверит Боба, что тот, кто пытается установить с ним защищенный канал связи, получил соответствующую информацию от KDC. Таким образом, Алиса сначала просит Боба послать ей случайное число R_{B1} , зашифрованное ключом, который совместно используется Бобом и KDC. Алиса включает это случайное число в свой запрос к центру KDC, который расшифровывает его и помещает результат в создаваемый талон. Тем самым Боб убеждается в том, что сеансовый ключ связан с исходным требованием Алисы на организацию связи с Бобом.

Аутентификация на основе криптосистем с открытым ключом

Обсудим теперь аутентификацию на основе криптосистем с открытым ключом, которым не нужен центр KDC. Давайте снова рассмотрим тот случай, когда Алиса хочет организовать защищенный канал связи с Бобом, причем оба они владеют

открытыми ключами друг друга. Типичный протокол аутентификации на основе криптосистем с открытым ключом показан на рис. 8.18.

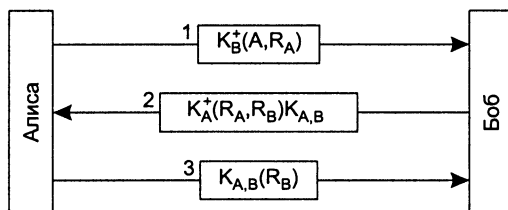


Рис. 8.18. Взаимная аутентификация в криптосистемах с открытым ключом

Алиса начинает с того, что посылает Бобу запрос R_A , зашифрованный его открытым ключом K_B^+ . Дело Боба расшифровать это сообщение и вернуть Алисе свой запрос. Поскольку Боб — единственный, кто в состоянии расшифровать это сообщение (используя закрытый ключ, соответствующий открытому ключу Алисы), Алиса будет знать, что общается с Бобом. Отметим важность того, что Алиса гарантированно использует открытый ключ Боба, а не кого-то другого, кто притворяется Бобом. Как можно получить эту гарантию, мы обсудим чуть позже.

Когда Боб получает от Алисы требование на создание канала, он возвращает расшифрованный запрос вместе со своим собственным запросом R_B для аутентификации Алисы. Кроме того, он создает сеансовый ключ $K_{A,B}$, который можно будет использовать для дальнейшей работы. Ответ Боба на запрос Алисы, его собственный запрос и сеансовый ключ помещаются в сообщение, которое шифруется открытым ключом K_A^+ , принадлежащим Алисе (сообщение 2). Только Алиса в состоянии расшифровать это сообщение, используя закрытый ключ K_A^- , соответствующий открытому ключу K_A^+ .

И, наконец, Алиса возвращает свой ответ на запрос Боба, используя сеансовый ключ $K_{A,B}$, созданный Бобом. Таким образом, она показывает, что в состоянии расшифровать сообщение 2, а значит, та, с кем общается Боб, — действительно Алиса.

8.2.2. Целостность и конфиденциальность сообщений

Кроме аутентификации защищенный канал должен также предоставлять гарантии целостности и конфиденциальности сообщений. Под целостностью сообщений мы подразумеваем защищенность сообщений от изменения, конфиденциальность означает, что сообщения не могут быть перехвачены и прочитаны посторонними лицами. Конфиденциальность легко реализуется путем обычного шифрования сообщений перед их посылкой. Шифрование может производиться как секретным ключом, совместно используемым отправителем и получателем, так и открытым ключом получателя. Однако, как мы увидим далее, защита сообщений от модификации — дело более сложное.

Цифровые подписи

Целостность сообщений часто не ограничивается собственно передачей сообщений по безопасному каналу. Рассмотрим ситуацию, когда Боб продает Алисе какие-то цифровые фотографии в коллекцию за 500 долларов. Вся сделка совершается по электронной почте. В конце Алиса посылает Бобу сообщение, подтверждающее, что она покупает фотографии за 500 долларов. Помимо аутентификации для поддержания целостности сообщения важно соблюдать еще как минимум два момента.

- ♦ Алиса должна быть уверена, что Боб не сможет злонамеренно изменить сумму в 500 долларов, указанную в ее сообщении, на большую и утверждать, что она обещала ему больше 500 долларов.
- ♦ Боб должен быть уверен, что Алиса не сможет отказаться от своего предложения после посылки сообщения, например, передумать.

Эти два момента можно обеспечить, если Алиса поставит свою *цифровую подпись* (*digital signature*) под этим документом, однозначно связав ее с содержимым письма. Уникальная связь между сообщением и подписью под ним приведет к тому, что внесение изменений в сообщение не останется незамеченным. Кроме того, если подлинность подписи Алисы может быть подтверждена, ей не удастся в дальнейшем отрицать тот факт, что сообщение подписала именно она.

Существует несколько способов поставить под документом цифровую подпись. Один из наиболее популярных вариантов — использование криптосистем с открытым ключом, таких как RSA (рис. 8.19). Когда Алиса посылает сообщение Бобу, она шифрует его своим закрытым ключом K_A^- и пересылает. Если она хочет вдобавок сохранить содержимое письма в тайне, она может воспользоваться также и открытым ключом Боба и послать $K_B^+(m, K_A^-(m))$, объединив в письме сообщение m и его экземпляр, зашифрованный Алисой.

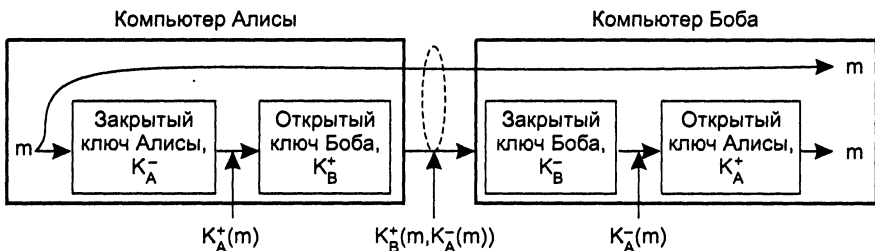


Рис. 8.19. Цифровая подпись сообщения и шифрование с открытым ключом

Когда сообщение дойдет до Боба, он сможет расшифровать его, используя открытый ключ Алисы. Если он будет уверен, что открытый ключ действительно принадлежит Алисе, то расшифровка подписанной версии сообщения m и успешное сравнение его с исходной версией сообщения m будет означать, что оно действительно пришло от Алисы. Алиса защищена от любых умышленных изменений m , вносимых Бобом, поскольку Бобу всегда придется доказывать, что модифицированная версия m также была подписана Алисой. Другими словами, расшифрованное сообщение само по себе не может считаться доказательством. Кроме того,

в интересах Боба хранить подписанную версию m с целью защитить себя от попыток Алисы отказаться от своего предложения.

Этой схеме присущи определенные проблемы, хотя сам протокол абсолютно корректен. Во-первых, подпись Алисы действительна только до тех пор, пока закрытый ключ Алисы содержится в секрете. Если Алиса захочет пересмотреть условия сделки уже после отправки Бобу подтверждения, ей достаточно будет объявить, что до отправки подтверждения у нее был украден закрытый ключ.

Другая проблема возникает в том случае, если Алиса решит изменить свой закрытый ключ. Само по себе это не такая уж и плохая идея, поскольку изменение ключа, выполняемое время от времени, обычно способствует усилению защиты. Однако раз Алиса изменила свой ключ, договор, отправленный Бобу, теряет всякий смысл. В подобных случаях может понадобиться централизованная служба авторизации, которая будет отслеживать изменения ключей и отметки времени подписания сообщений.

Еще одной проблемой подобной схемы является то, что Алиса шифрует все сообщение своим закрытым ключом. Подобное шифрование может потребовать массы вычислительных ресурсов (или даже оказаться невыполнимым, если мы предположим, что сообщение интерпретируется как двоичное число с ограниченной максимальной длиной), а это для нас недопустимо. Напомним, что все, что нам нужно, — это однозначно связать подпись с конкретным сообщением. Гораздо дешевле и значительно элегантнее будет использовать схему с применением дайджестов сообщений.

Как мы уже говорили, дайджест сообщения — это строка битов фиксированной длины h , которая вычисляется из сообщения произвольной длины m посредством криптографической хэш-функции H . Если m изменяется, получается другое сообщение, m' . Его хэш, $H(m')$, будет отличаться от первоначальной строки $h = H(m)$, так что внесенные изменения будут легко обнаружены.

Чтобы подписать сообщение, Алиса должна будет сначала вычислить дайджест сообщения, а затем зашифровать этот дайджест своим закрытым ключом (рис. 8.20). Зашифрованный дайджест пересылается Бобу вместе с сообщением. Отметим, что само сообщение пересылается открытым текстом и его может прочесть кто угодно. Если мы нуждаемся в конфиденциальности, сообщение можно также зашифровать открытым ключом Боба.

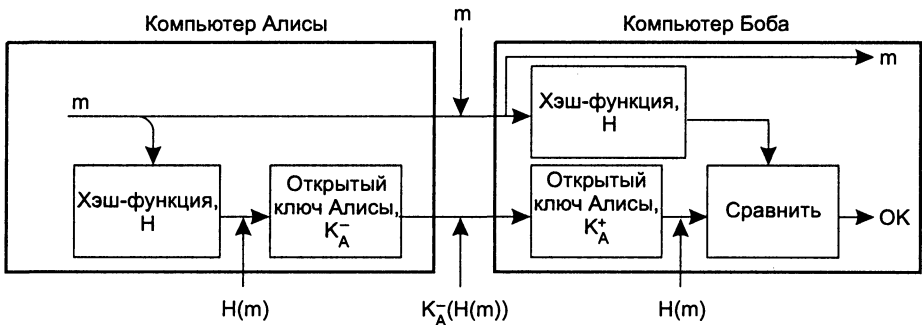


Рис. 8.20. Цифровая подпись сообщения и использование дайджеста сообщения

Получив сообщение и зашифрованный дайджест, Боб должен просто расшифровать дайджест открытым ключом Алисы и независимо вычислить дайджест сообщения. Если вычисленный для полученного сообщения дайджест и расшифрованный дайджест Алисы совпадают, можно предположить, что сообщение было подписано Алисой.

Сеансовые ключи

В ходе организации защищенных каналов после завершения фазы аутентификации с целью конфиденциальности стороны обычно используют уникальный общий сеансовый ключ. По окончании использования канала сеансовый ключ аннулируется. В качестве альтернативы для конфиденциальности можно ограничиться тем же ключом, что и при установлении соединения, однако сеансовые ключи дают множество важных преимуществ [228].

Во-первых, при частом использовании ключа его проще вскрыть. В этом смысле криптографические ключи устаревают ничуть не медленнее обычных. Основная идея здесь состоит в том, что если злоумышленник сможет перехватить большой объем данных, зашифрованных одним и тем же ключом, он сможет построить атаки так, чтобы выявить важные характеристики ключа и, возможно, получить простой текст или даже сам ключ. По этой причине желательно как можно реже задействовать ключ аутентификации. Кроме того, подобные ключи часто меняются при помощи каких-либо медленных внешних механизмов, например регулярных почтовых отправок или телефонограмм. Такие способы обмена ключами должны быть сведены к минимуму.

Другая важная причина генерации отдельного ключа на каждый защищенный канал состоит в гарантированной защите от атак воспроизведения сообщений. Используя уникальные сеансовые ключи при каждом открытии защищенного канала, общающиеся стороны как минимум застрахованы от воспроизведения всего сеанса. Чтобы защититься от воспроизведения отдельных сообщений из предыдущего сеанса, обычно требуются дополнительные меры, такие как добавление к сообщениям отметок времени или последовательных номеров.

Предположим, что целостность и конфиденциальность сообщений достигается использованием того же ключа, что и для организации сеанса. В этом случае всякий раз, когда ключ раскрывают, злоумышленник оказывается в состоянии расшифровать сообщения, передававшиеся в ходе предыдущих сеансов. Понятно, что это нежелательно. Вместо этого гораздо безопаснее использовать сеансовые ключи, поскольку при раскрытии такого ключа можно будет расшифровать сообщения только одного сеанса. Сообщения, переданные в ходе других сеансов, останутся конфиденциальными.

Немного о последнем замечании. Алиса может захотеть обменяться с Бобом некоторыми конфиденциальными данными, не доверяя ему настолько, чтобы передавать информацию, зашифрованную ключами длительного действия. Она может пожелать зарезервировать эти ключи для обмена конфиденциальной информацией с собеседниками, которым она действительно доверяет. В этом случае ей лучше использовать относительно дешевые сеансовые ключи.

Вообще говоря, ключи аутентификации часто организованы таким образом, что их замена — достаточно дорогостоящая операция. Таким образом, комбина-

ция из подобных ключей длительного действия и более дешевых и краткосрочных сеансовых ключей — нередко лучший выбор при реализации защищенных каналов обмена данными.

8.2.3. Защищенное групповое взаимодействие

До сих пор наше внимание было сосредоточено на вопросе организации защищенных каналов связи между двумя сторонами. В распределенных системах, однако, часто необходимо поддерживать защищенную связь между большим количеством сторон. Типичный пример этого — реплицируемый сервер, для которого все связи между репликами следует защищать от изменения, подделки и перехвата так же, как защищенный канал между двумя пользователями. В этом подразделе мы поближе познакомимся с вопросами защищенного группового взаимодействия.

Конфиденциальное групповое взаимодействие

Для начала обсудим проблему защиты взаимодействия в группе из N пользователей от подслушивания. Простейшей схемой обеспечения конфиденциальности будет совместное использование всей группой одного секретного ключа для шифрования и расшифровки сообщений, передаваемых членами группы друг другу. Поскольку секретный ключ в этой схеме совместно используется всеми членами группы, необходимо, чтобы все участники верили, что ключ действительно секретный. Это требование само по себе делает использование одного общего секретного ключа при групповом взаимодействии значительно более уязвимым для атак по сравнению с двухсторонним защищенным каналом.

Альтернативное решение — каждая пара членов группы совместно использует отдельные ключи. Как только один из членов начнет допускать потерю информации, другие члены группы просто прекращают посылать ему сообщения, но продолжают применять для связи друг с другом старые ключи. Однако в отличие от предыдущего случая с поддержкой одного ключа теперь нужно поддерживать $N(N-1)/2$ ключей, что уже само по себе может быть непросто.

Облегчить положение может помочь криптосистема с открытым ключом. Тогда каждый член группы будет иметь собственную пару (открытый ключ, закрытый ключ), в которой открытый ключ будет использоваться всеми членами группы для отправки конфиденциальных сообщений. В этом случае требуется всего N пар ключей. Если один из членов группы перестает внушать доверие, он просто удаляется из группы, при этом другие ключи не затрагиваются.

Защищенная репликация серверов

Теперь обсудим совсем другую проблему: клиента, посылающего запрос группе реплицируемых серверов. Серверы могут реплицироваться по разным причинам: для защиты от сбоев, для повышения производительности. Но в любом случае клиент ожидает, что ответ будет заслуживать доверия. Другими словами, независимо от того, подвержена ли группа серверов византийским ошибкам, которые мы обсуждали в предыдущей главе, клиент ожидает, что защите возвращенного

ему результата ничто не будет угрожать. Подобные атаки могут произойти в случае успешного взлома злоумышленником одного или нескольких серверов.

Способом защиты клиента от подобных атак является сбор ответов всех серверов с идентификацией каждого из них. Если ответы невзломанных (то есть аутентифицированных) серверов составляют большинство, клиент может считать, что ответ корректен. К сожалению, подобный подход нарушает прозрачность репликации серверов.

В [376] было предложено решение для защищенного реплицируемого сервера, позволяющее сохранить прозрачность репликации. Достоинство такого подхода состоит в том, что клиенты остаются в неведении о реальных репликах и поэтому значительно проще скрытно добавлять и удалять реплики. Мы вернемся к вопросу управления защищенными группами ниже, при обсуждении вопросов управления ключами.

Смысл защищенных и прозрачно реплицируемых серверов кроется в том, что называется *разделением секрета* (*secret sharing*). При разделении секрета ни один из нескольких пользователей (или процессов) не знает секрета целиком, секрет может быть открыт только в случае их совместной работы. Такие схемы могут быть очень удобными. Рассмотрим, например, запуск ракеты с ядерной боеголовкой. Это действие требует подтверждения как минимум двумя людьми. Каждый из них имеет собственный закрытый ключ, причем для пуска он должен использоваться в паре с другим. Попытка пуска с помощью единственного ключа ни к чему не приведет.

В случае защищенных реплицируемых серверов при поиске ответа максимум k из N серверов могут дать неверный ответ, а из этих k максимум $s \leq k$ серверов могут быть действительно взломаны злоумышленником. Отметим, что это требование предполагает, что служба устойчива к k ошибкам. Этот вопрос мы обсуждали в предыдущей главе. Разница состоит в том, что сейчас серверы, которые работают ошибочно, взломаны намеренно.

Теперь рассмотрим ситуацию, когда серверы активно реплицируются. Другими словами, запрос рассылается всем серверам одновременно, после чего обрабатывается каждым из них. Каждый сервер генерирует ответ, который возвращается клиенту. В случае защищенной реплицируемой группы серверов предполагается, что каждый сервер сопровождает свой ответ цифровой подписью. Если r_i — ответ сервера S_i , то пусть $md(r_i)$ обозначает дайджест сообщения, вычисленный сервером S_i . Этот дайджест подписан закрытым ключом K_i^- сервера S_i .

Представим себе, что мы хотим защитить клиента от максимум s взломанных серверов. Другими словами, группа серверов должна быть в состоянии скрыть последствия взлома максимум s серверов, оставаясь при этом в состоянии сгенерировать ответ, которому клиент сможет доверять. Если подписи отдельных серверов можно скомбинировать таким образом, чтобы построение правильной подписи для результата требовало минимум $s + 1$ подписей, это решает проблему. Другими словами, мы хотим, чтобы реплицируемые серверы создавали секретную правильную подпись так, чтобы s взломанных серверов были не в состоянии собрать эту подпись без помощи минимум одного «честного» сервера.

В качестве примера рассмотрим группу из пяти реплицируемых серверов, которые должны быть в состоянии выдержать взлом двух из них и давать при этом

результат, которому сможет доверять клиент. Каждый сервер S_i посылает ответ r_i вместе со своей подписью $sig(S_i, r_i) = K_i^-(md(r_i))$ клиенту. Соответственно, клиент последовательно получает пять триплетов $\langle r_i, md(r_i), sig(S_i, r_i) \rangle$, из которых он должен выделить правильные. Эту ситуацию иллюстрирует рис. 8.21.

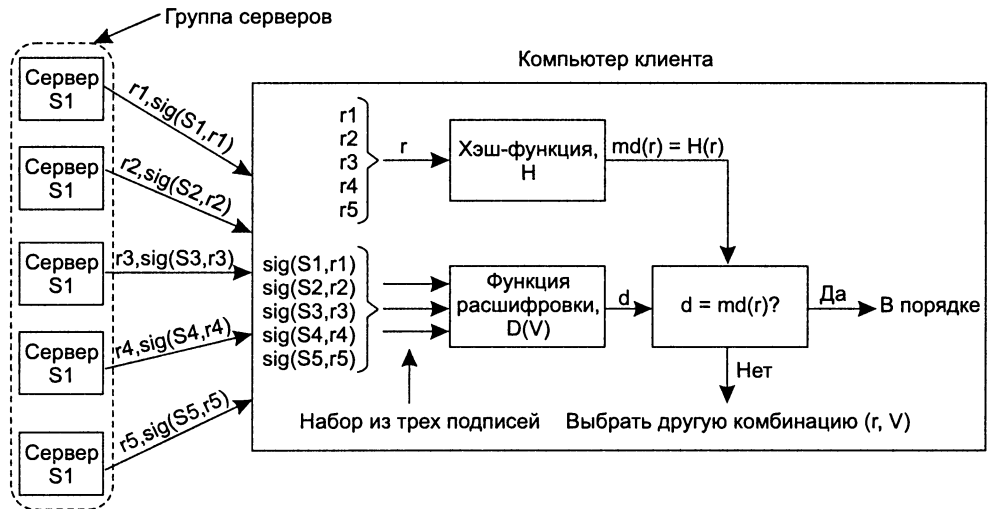


Рис. 8.21. Разделяемая секретная подпись группы реплицируемых серверов

Каждый дайджест $md(r_i)$ вычисляется также и на клиенте. Если ответ r_i неверен, это обычно можно определить, вычислив $K_i^+(K_i^-(md(r_i)))$, однако, поскольку мы не можем доверять ни одному серверу в отдельности, этот метод применять нельзя. Вместо него клиент использует специальную открытую функцию расшифровки D , которая в качестве исходных данных получает набор из трех сигнатур $V = \{ sig(S, r), sig(S', r'), sig(S'', r'') \}$, а в качестве результата генерирует единственный дайджест:

$$dout = D(V) = D(sig(S, r), sig(S', r'), sig(S'', r'')).$$

Детали о функции расшифровки D можно найти в [376]. Существует $5!/(3!2!) = 10$ комбинаций трех подписей, которые клиент может получить в качестве результата D . Если одна из этих комбинаций даст правильный дайджест $md(r_i)$ для некоторого результата r_i , клиент может считать ответ r_i правильным. В частности, он может считать, что этот результат дан как минимум тремя правильно работающими серверами.

Для повышения прозрачности репликации в [376] предлагается, чтобы каждый сервер S_i рассылал свой результат r_i остальным серверам вместе с соответствующей подписью $sig(S_i, r_i)$. Когда сервер получит как минимум $c + 1$ подобных сообщений, включая его собственное, он сможет вычислить правильную подпись для одного из ответов. Если, например, это вычисление для ответа r и набора V из $c + 1$ подписей будет успешным, он пересылает r и V клиенту одним сообщением. Клиент после этого проверяет корректность r путем проверки подписи, то есть того, что $md(r) = D(V)$.

Описанный нами алгоритм известен как *m,n-пороговая схема* (*m,n-threshold scheme*), где, в случае нашего примера, $m = c + 1$, а $n = N$ (число серверов). В *m,n-пороговой* схеме сообщение делится на n частей, известных под названием *теней* (*shadows*). Для воссоздания исходного сообщения могут быть использованы любые m теней, но любых $m - 1$ теней для этого будет недостаточно. Существует несколько способов построения *m,n-пороговых* схем. Подробности можно найти в [404].

8.3. Контроль доступа

В модели клиент-сервер, которую мы рассматривали все это время, после создания между клиентом и сервером защищенного канала клиент создавал запросы, которые передавались серверу. Запросы включали операции над ресурсами, контролируемые сервером. Стандартная ситуация предполагала, что сервер объектов управляет несколькими объектами. Запрос клиента обычно содержал обращение к методу конкретного объекта. Такой запрос мог быть выполнен только в том случае, если клиент обладал достаточными для такого обращения *правами доступа* (*access rights*).

Формально подтверждение прав доступа называется *контролем доступа* (*access control*), в то время как для выдачи прав доступа применяется термин *авторизация* (*authorization*). Эти два понятия тесно связаны друг с другом, и нередко одно из них используется вместо другого. Существует множество методов контроля доступа. Мы начнем с обсуждения некоторых общих вопросов, а потом займемся различными моделями контроля доступа. Один из особенно значимых методов контроля доступа к ресурсам — это создание брандмауэра, защищающего приложение или даже всю сеть целиком. Брандмауэры мы обсудим отдельно. В связи с повышением мобильности кода контроль доступа становится невозможно осуществлять только традиционными методами. Для этого были разработаны новые технологии, которые мы также обсудим в этом разделе.

8.3.1. Общие вопросы контроля доступа

Чтобы разобраться в различных аспектах контроля доступа, рассмотрим простую модель, представленную на рис. 8.22. Она состоит из *субъектов* (*subjects*), которые посылают запросы на доступ к *объекту* (*object*). Этот объект очень похож на те объекты, которые мы уже обсуждали. Мы можем думать об инкапсуляции его внутреннего состояния и реализации операций внутри этого состояния. Операции объекта, запросы на выполнение которых посылаются субъектами, доступны через интерфейсы. Лучше всего представлять, что субъекты осуществляют по заданию пользователей процессы, но могут также быть и объектами, которые для выполнения своей работы нуждаются в помощи других объектов.

Управление доступом к объектам — это все то, что защищает объекты от обращений субъектов, не имеющих прав на вызов соответствующих методов, а возможно, и вовсе не имеющих прав на вызов каких бы то ни было методов. Кроме того, защита может относиться и к некоторым аспектам управления объектами,

таким как создание, переименование или удаление объектов. Защита часто реализуется при помощи программы под названием *монитор ссылок* (*reference monitor*). Монитор ссылок записывает, что может делать тот или иной субъект, и решает, допустимо ли для данного субъекта выполнение конкретной операции. Этот монитор вызывается (например, базовой доверенной операционной системой) при любых обращениях к объекту. Таким образом, очень важно, чтобы монитор ссылок сам по себе имел повышенную защиту от атак: злоумышленник должен быть не в состоянии его обмануть.

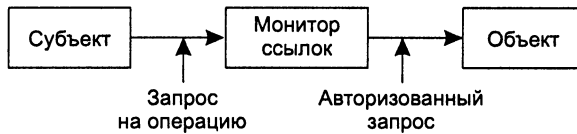


Рис. 8.22. Обобщенная модель контроля доступа к объектам

Матрица контроля доступа

Общепринятый подход к моделированию прав доступа субъекта по отношению к объектам состоит в построении *матрицы контроля доступа* (*access control matrix*). Каждый субъект представлен строкой этой матрицы, каждый объект — столбцом. Если ввести для матрицы обозначение M , то элемент $M[s,o]$ точно определяет, выполнения каких операций субъект s может требовать от объекта o . Другими словами, если субъект s пытается обратиться к методу m объекта o , монитор ссылок должен проверить, указан ли метод m в $M[s,o]$. Если m в $M[s,o]$ отсутствует, вызова не произойдет.

Учитывая, что системе может потребоваться поддерживать тысячи пользователей и миллионы требующих защиты объектов, реализация матрицы контроля доступа в виде реальной матрицы представляется не лучшей из идей. Большинство элементов матрицы будут пустыми, поскольку один субъект обычно имеет доступ к относительно небольшому числу объектов. Поэтому для реализации матрицы контроля доступа используются более эффективные способы.

Один из широко применяемых способов состоит в том, что каждый из объектов поддерживает список прав доступа субъектов, которые желают иметь доступ к этому объекту. Это означает, что матрица разбивается на столбцы, которые распределяются по объектам, при этом пустые элементы отбрасываются. Такой способ реализации приводит нас к так называемому *списку контроля доступа* (*Access Control List, ACL*). Предполагается, что каждый объект имеет собственный, ассоциированный только с ним список ACL.

Другой подход состоит в том, что матрица разбивается по строкам и каждый субъект получает список *мандатов* (*capabilities*) для каждого из объектов. Другими словами, мандат соответствует элементу в матрице контроля доступа. Отсутствие для конкретного объекта мандата означает, что субъект не имеет прав на доступ к этому объекту.

Мандат можно сравнить с талоном — тот, кто им владеет, имеет определенные права, записанные в этом талоне. Понятно также, что талон должен быть за-

щищен от попыток владельца модифицировать его. Один из способов, подходящий для распределенных систем и активно используемый в системе Amoeba, состоит в защите мандатов (списка мандатов) с помощью подписи [449]. Мы вернемся к этому и другим вопросам позже при рассмотрении вопросов управления защитой.

Разницу между использованием для защиты объектов списков контроля доступа и списков мандатов иллюстрирует рис. 8.23. В случае списков контроля доступа клиент посылает запрос на сервер, монитор ссылок выясняет, что ему известно про этого клиента, и если клиенту разрешается выполнять запрошенную операцию, она будет выполнена, что и показано на рис. 8.23, а.

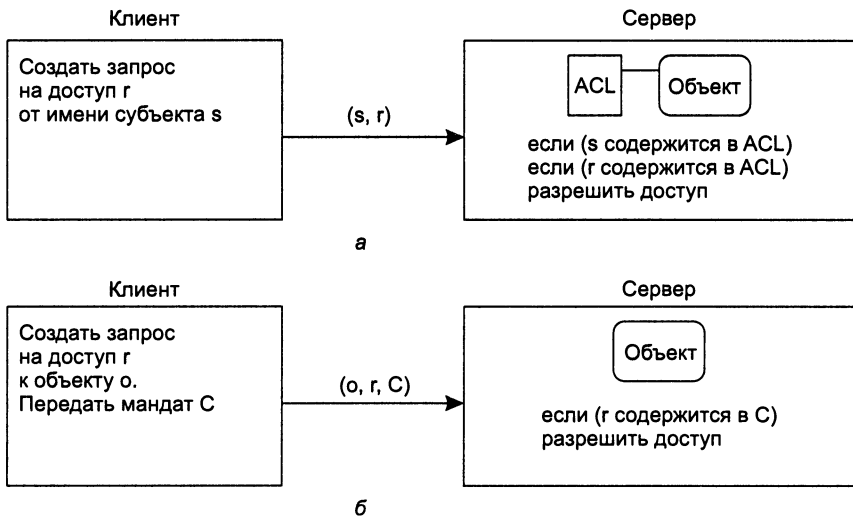


Рис. 8.23. Сравнение вариантов защиты объектов с помощью ACL и мандатов. Использование ACL (а). Использование мандатов (б)

При использовании списка мандатов клиент просто передает свой запрос вместе со списком мандатов на сервер. Сервер не хранит сведений о клиенте, список мандатов и так скажет ему все необходимое. Таким образом, сервер должен просто проверить, имеется ли в запросе корректный список мандатов и указана ли запрашиваемая операция в этом списке. Такой способ защиты объектов — на основании мандатов — иллюстрирует рис. 8.23, б.

Защищенный домен

Списки ACL и мандаты помогают успешно реализовать матрицу контроля доступа с исключенными пустыми элементами. Тем не менее если не принимать никаких дополнительных мер, и списки контроля доступа, и списки мандатов в конце концов могут стать слишком большими.

Один из основных способов уменьшить размер списка ACL — использование защищенных доменов. *Защищенный домен (protection domain)* — это набор пар (объект, права доступа). Каждая пара идентифицирует операции, которые для

данного объекта разрешается выполнять [391]. Запросы на выполнение операции всегда разрешаются внутри домена. Таким образом, когда субъект запрашивает у объекта выполнение операции, монитор ссылок сначала находит защищенный домен, связанный с этим запросом. Затем, выбрав домен, он проверяет, может ли быть выполнен указанный запрос. Существуют различные методы использования защищенных доменов.

Один из подходов состоит в построении *групп (groups)* пользователей. Рассмотрим, например, web-страницу внутренней сети компании. К этой странице должны иметь доступ все сотрудники компании, и никто кроме них. Вместо того чтобы добавлять в ACL по элементу на каждого сотрудника, можно создать отдельную группу, содержащую список всех текущих сотрудников. Каждый раз при попытке обращения пользователя к web-странице монитор ссылок должен будет только проверить, является ли пользователь сотрудником. Пользователи, включенные в группу сотрудников, должны быть перечислены в отдельном списке (разумеется, защищенном от неавторизованного доступа).

Схема станет более гибкой, если сделать структуру групп иерархической. Так, например, если организация имеет три различных филиала, скажем, в Амстердаме, Нью-Йорке и Сан-Франциско, она может пожелать разбить группу сотрудников на три подгруппы, по одной для каждого города, что приведет нас к структуре, представленной на рис. 8.24.

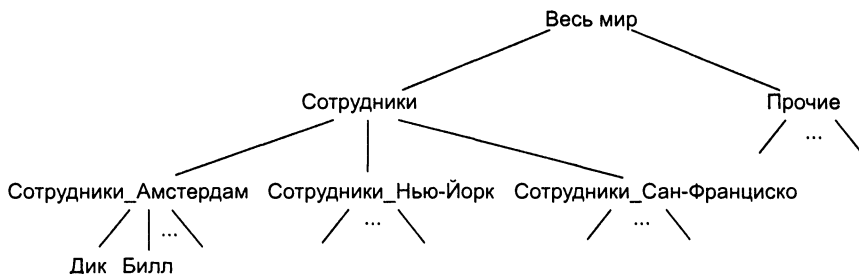


Рис. 8.24. Иерархическая организация защищенных доменов в виде групп пользователей

Доступ к web-странице внутренней сети компании может быть разрешен всем ее работникам. Однако изменять, например, web-страницы, относящиеся к амстердамскому филиалу, может быть позволено только подмножеству сотрудников из Амстердама. Если пользователь Дик из Амстердама захочет почитать web-страницу внутренней сети, монитор ссылок должен будет сначала найти подгруппы *Сотрудники_Амстердам*, *Сотрудники_Нью-Йорк* и *Сотрудники_Сан-Франциско*, которые вместе образуют группу *Сотрудники*. После этого надо будет проверить, входит ли Дик в одну из этих подгрупп. Преимущество иерархической организации групп состоит в том, что управлять членством в группе становится значительно легче. Кроме того, появляется возможность эффективного построения очень больших групп. Основной минус такой структуры в том, что поиск элемента группы изрядно затрудняется, особенно если база данных о членстве в группах является распределенной.

Вместо того чтобы заставлять монитор ссылок делать всю работу в одиночку, мы можем заставить каждого субъекта хранить *сертификат* (*certificate*) со списком групп, в которые он входит. Таким образом, когда Дик захочет почитать web-страницу внутренней сети компании, он предъявит монитору ссылок свой сертификат, показывающий, что он является членом подгруппы *Сотрудники_Амстердам*. Чтобы гарантировать, что этот сертификат настоящий, не поддельный, его можно защитить, например, цифровой подписью. Сертификаты немного похожи на списки мандатов. Мы вернемся к этой теме позже.

Кроме использования групп, защищенные домены можно также реализовать в виде *ролей* (*roles*). В случае контроля доступа на основе ролей пользователь всегда входит в систему под определенной ролью, которая обычно связана с его полномочиями в организации [396]. Пользователь может иметь несколько ролей. Так, например, Дик может одновременно быть начальником отдела, менеджером проекта и членом группы по поиску сотрудников. В зависимости от роли, выбранной им при подключении, он может получить определенные права. Другими словами, его роль определяет защищенный домен (то есть группу), в котором он будет состоять.

Присваивая роли пользователям и требуя, чтобы во время подключения пользователь выступал под одной из них, мы позволяем пользователю при необходимости изменить свою роль. Так, например, можно потребовать, чтобы Дик, войдя в систему начальником отдела, перешел затем к своей роли менеджера проекта. Отметим, что эти переходы нелегко осуществить, если защищенные домены реализованы только в виде групп.

Кроме использования защищенных доменов производительность можно повысить путем иерархической или какой-либо другой группировки объектов на основании операций, которые они осуществляют. Так, например, независимо от самих объектов, объекты можно группировать по предоставляемым ими интерфейсам, возможно, используя подтипы (также называемые унаследованные интерфейсы, [157]) для построения иерархии. В этом случае, если субъект запрашивает операцию, которую должен выполнить объект, монитор ссылок ищет, в каком интерфейсе находится операция над этим объектом. Затем, вместо того чтобы проверять право субъекта вызывать операции данного объекта, он проверяет, имеет ли субъект право вызывать операции, принадлежащие к данному интерфейсу.

Также допускается и комбинирование защищенных доменов и групп объектов. В [170] описано, как, используя оба этих приема вместе со специальными структурами данных и защищенными операциями объектов, реализовать списки ACL в очень больших наборах объектов цифровых библиотек.

8.3.2. Брандмауэры

Ранее мы рассматривали, как организовать защиту при помощи криптографических методов в комбинации с реализацией матриц контроля доступа. Эти способы с успехом работают до тех пор, пока связывающиеся стороны играют по одним правилам. Однако заставить соблюдать эти правила можно только при

разработке распределенной системы, изолированной от всего остального мира. Дело значительно усложняется, когда доступ к ресурсам, которыми управляет распределенная система, открывается сторонним пользователям. Примеры подобных случаев включают в себя, например, посылку почты, загрузку файлов или выгрузку форм.

Чтобы в подобных условиях защитить ресурсы, необходим другой подход. На практике доступ извне к любой из частей распределенной системы отслеживается монитором ссылок особого типа ([101, 504]), известным под названием *брандмауэра (firewall)*. Грубо говоря, брандмауэр отсекает часть распределенной системы от окружающего мира, как показано на рис. 8.25. Все выходящие и особенно все входящие пакеты проходят через специальный компьютер и перед тем, как будут переданы дальше, проверяются. Неавторизованный трафик дальше не передается. Следует отметить, что сам брандмауэр должен быть надежно защищен от угроз защиты любого рода: он никогда не должен выходить из строя.

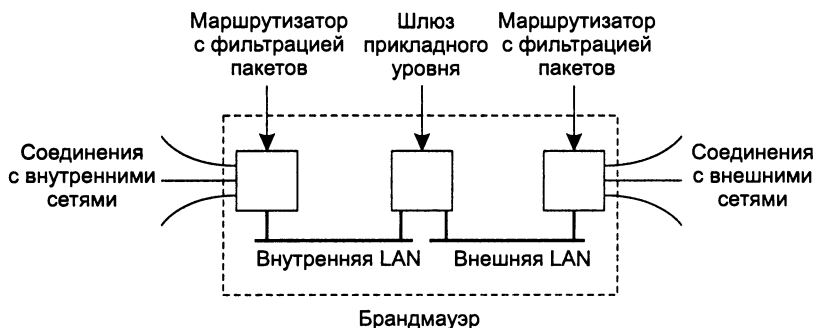


Рис. 8.25. Обобщенная реализация брандмауэра

Различают в основном брандмауэры двух типов, но часто реальные брандмауэры представляют собой комбинацию этих типов. Важнейший тип брандмауэра — это *шлюз фильтрации пакетов (packet-filtering gateway)*. Брандмауэр такого типа работает как маршрутизатор. Он принимает решение о том, пропускать или нет сетевой пакет на основании адресов отправителя и получателя в заголовке пакета. Обычно (см. рис. 8.25) шлюз фильтрации пакетов, находящийся вне локальной сети (LAN), защищает систему от входящих пакетов, а такой же шлюз, находящийся во внутренней сети, фильтрует исходящие пакеты.

Так, например, для защиты внутреннего web-сервера от запросов хостов, не входящих во внутреннюю сеть, шлюз фильтрации пакетов должен отбрасывать все входящие пакеты, адресованные web-серверу.

Более тонкой является ситуация, когда сеть компании состоит из нескольких локальных сетей, соединенных между собой, например, сетью SDMS. Мы уже обсуждали такой вариант. Каждая внутренняя сеть может быть защищена шлюзом фильтрации пакетов, который сконфигурирован на пропуск входящего трафика только от хостов других внутренних сетей. Таким образом, строятся частные виртуальные сети.

Другой тип брандмауэров — *шлюз прикладного уровня (application-level gateway)*. В противоположность шлюзу фильтрации пакетов, который проверяет только заголовки пакетов, этот тип брандмауэра просматривает содержимое входящих и исходящих сообщений. Типичным примером может быть шлюз почты, который отбрасывает входящие или исходящие письма, превышающие установленный размер. Существуют и более сложные почтовые шлюзы, способные, например, отфильтровывать электронную почту рекламного характера (спам).

Другим примером шлюза прикладного уровня может быть шлюз внешнего доступа к серверу цифровой библиотеки, позволяющий ознакомиться только с рефератами документов. Если внешний пользователь хочет получить что-то еще, начинает работать протокол электронных платежей. Внутренний пользователь имеет прямой доступ к библиотечным службам.

Существует особый тип шлюза прикладного уровня, известный под названием *прокси-шлюза (proxy gateway)*. Этот тип брандмауэра скрывает работу некоторых типов приложений. Он гарантирует прохождение через него только сообщений, удовлетворяющих заданным критериям. Рассмотрим, например, путешествия в Web. Многие web-страницы содержат апплеты или сценарии, которые должны выполняться браузером пользователя. Чтобы предотвратить попадание этого кода в локальную сеть, весь web-трафик пропускается через прокси-шлюз. Этот шлюз получает обычные HTTP-запросы как изнутри, так и снаружи брандмауэра. Другими словами, для пользователей прокси-шлюз выглядит как обычный web-сервер, однако он фильтрует весь входящий и исходящий трафик, отбрасывая определенные запросы и страницы или модифицируя страницы, содержащие исполняемый код.

8.3.3. Защита мобильного кода

Как мы говорили в главе 3, важной особенностью современных распределенных систем по сравнению с прежними является возможность переноса с хоста на хост кода, а не только пассивных данных. Однако мобильный код приводит к появлению серьезных угроз защиты. Так, например, отправляя в путешествие по Интернету агента, владелец захочет защитить его от злоумышленников, желающих похитить или изменить информацию, собранную агентом.

Другой вопрос состоит в том, что и сами хосты должны быть защищены от агентов-злоумышленников. Большинство пользователей распределенных систем не являются экспертами в системотехнике и не в состоянии сказать, повреждают или нет их компьютер программы, пришедшие с других хостов. Во многих случаях для эксперта может быть затруднительно обнаружить даже сам факт загрузки программы.

Если не предпринимать специальных мер по поддержанию защиты, однажды вредоносная программа может попасть в ваш компьютер и с легкостью поразить хост. Перед нами стоит проблема контроля доступа: программы не должны иметь неавторизованного доступа к ресурсам хоста. Как мы увидим, защита хоста от загрузки вредоносных программ иногда оказывается непростым делом. Проблема настолько серьезна, что может заставить нас вообще отказаться от за-

грузки программ. Вместо этого следует рассмотреть возможность использования мобильного кода, разрешая ему доступ, гибкий к локальным ресурсам при условии полного контроля за ситуацией с нашей стороны.

Защита агента

Прежде чем мы перейдем к вопросу защиты компьютерных систем от загрузки вредоносного кода, рассмотрим противоположную ситуацию. Представим себе мобильного агента, перемещающегося по распределенной системе по заданию пользователя. Покупая по поручению владельца билет на самолет или резервируя место, агент может использовать электронную кредитную карту.

Ясно, что в этом случае нам необходима защита. По мере того как агент перемещается с хоста на хост, эти хосты не должны иметь возможности узнать информацию о кредитной карте агента. Кроме того, агент должен быть защищен от модификации, которая могла бы заставить его заплатить больше, чем нужно. Так, например, хост *Самых дешевых чартеров Чака* может заметить, что агент еще не посетил хост его конкурента, *Авиалиний Алисы*, с более низкими ценами. Чак может попытаться изменить код агента таким образом, чтобы тот не добрался до хоста *Авиалиний Алисы*. Другие примеры необходимости защиты агента от атак враждебных хостов подразумевают злонамеренное разрушение агента или фальсификацию агента таким образом, чтобы он по возвращении атаковал своего владельца.

К сожалению, полная защита агента от всех типов атак невозможна [141]. Эта невозможность вызвана в первую очередь фактом отсутствия четких гарантий того, что хост будет работать, как обещано. Поэтому альтернативой является такая организация агентов, чтобы их изменение можно было, по крайней мере, заметить. Такой подход используется в системе Ajanta [225]. Ajanta предоставляет собой три механизма, позволяющих владельцу агента заметить, что агент был фальсифицирован — это состояние только для чтения, журналы только для записи и выборочный показ состояния для заданных серверов.

Состояние только для чтения (read-only state) агента системы Ajanta содержит набор элементов данных, которые подписаны владельцем агента. Подписание происходит в момент создания и инициализации агента, до того как он будет отправлен на какой-либо другой хост. Владелец сначала создает дайджест сообщения, а затем шифрует его своим закрытым ключом. Когда агент появляется на хосте, тот легко может заметить, что состояние «только для чтения» было фальсифицировано, просто проверив это состояние по подписанному дайджесту исходного состояния.

Чтобы позволить агенту при перемещении от хоста к хосту собирать информацию, Ajanta использует защищенные *журналы только для записи (append-only logs)*. Эти журналы отличаются тем, что данные в них можно только добавлять, удалить или изменить данные незаметно для владельца агента невозможно. Журналы только для записи работают следующим образом. Изначально журнал пуст и содержит только ассоциированную с ним контрольную сумму C_{init} , вычисляемую как $C_{init} = K_{owner}^+(N)$, где K_{owner}^+ — открытый ключ владельца агента, а N — секретный одноразовый ключ, известный только владельцу.

Когда агент перемещается на сервер S , который желает передать ему некоторые данные X , S добавляет X в журнал, пометив X своей подписью $\text{sig}(S, X)$, и вычисляет контрольную сумму:

$$C_{\text{new}} = K_{\text{owner}}^+(C_{\text{old}}, \text{sig}(S, X), S).$$

Здесь C_{old} — предыдущая контрольная сумма.

Когда агент возвращается к своему владельцу, последний может легко проверить, не фальсифицирован ли журнал. Владелец начинает считывать журнал с конца, последовательно вычисляя значения $K_{\text{owner}}^-(C)$ для контрольной суммы C . Каждая итерация возвращает контрольную сумму C_{next} для следующей итерации, а также $\text{sig}(S, X)$ для каждого сервера S . Владелец в состоянии проверить, действительно ли очередной элемент журнала соответствует $\text{sig}(S, X)$. Если это так, элемент удаляется и обрабатывается, после чего производится очередной шаг итерации. Итерации заканчиваются по достижении исходной контрольной суммы или когда владелец обнаруживает по несовпадению подписей, что журнал был фальсифицирован.

И, наконец, Ajanta поддерживает *выборочный показ* (*selective revealing*) состояния, предоставляя массив элементов данных, в котором каждый элемент соответствует определенному серверу. Каждый элемент для сохранения конфиденциальности зашифрован открытым ключом соответствующего сервера. Массив подписан владельцем агента для сохранения целостности всего массива. Иными словами, если какой-либо элемент модифицируется хостом-злоумышленником, любой из серверов может обнаружить это и принять соответствующие меры.

Кроме защиты агента от хостов-злоумышленников Ajanta предоставляет разнообразные механизмы защиты хостов от агентов-злоумышленников. Как мы в дальнейшем увидим, многие из этих механизмов поддерживаются и другими системами, рассчитанными на перенос кода. Дополнительную информацию по системе Ajanta можно найти в [464].

Защита получателя

Несмотря на то что защита мобильного кода от злого умысла со стороны хостов, несомненно, важное дело, большее внимание следует все же уделять защите хостов от злого умысла со стороны мобильного кода. Поскольку посылка агента во внешний мир кажется делом опасным, пользователь обычно имеет альтернативные способы выполнения работы, которую он собирался возложить на мобильного агента. Что касается агента, попавшего в вашу систему, нередко все, что вы можете, — это либо разрешить ему действовать, либо полностью запретить всякую работу. Если мы допускаем возможность попадания агента в вашу систему, то пользователю необходимо контролировать абсолютно все действия агента.

Как мы только что говорили, защита агента от изменения его содержимого может быть невозможна, но всегда можно хотя бы дать возможность владельцу агента обнаружить внесенные изменения. В худшем случае владелец может уничтожить агента по его возвращении, но и в этом случае не произойдет ничего плохого. Однако при работе с приходящими вредоносными агентами частенько то, что ваши ресурсы уже испытали на себе их воздействие, обнаруживается

слишком поздно. В этом случае важно защитить все ресурсы от неавторизованного доступа со стороны полученного по сети кода.

Один из способов защиты — создать сито. *Cumo (sandbox)* — это механизм, с помощью которого загруженная программа выполняется таким образом, что каждая ее инструкция полностью находится под контролем. Если делается попытка вызвать запрещенную хостом инструкцию, выполнение программы останавливается. Также выполнение может быть прервано при попытке получить доступ к определенным регистрам или областям памяти, обращения к которым запрещены хостом.

Реализовать сито не так-то просто. Один из методов предполагает проверку исполняемого кода до загрузки и вставку дополнительных инструкций в тех местах, которые можно проверить только во время исполнения [480]. К счастью, при работе с интерпретируемым кодом задача сильно упрощается. Давайте кратко рассмотрим подход, используемый в языке Java [456]. Каждая программа на языке Java состоит из нескольких классов, объекты которых создаются при выполнении программы. Глобальных переменных или функций не существует, все они объявляются частями классов. Выполнение программы начинается с метода *main*. Программа на языке Java компилируется в набор инструкций, интерпретируемый так называемой *виртуальной Java-машиной (Java Virtual Machine, JVM)*. Поэтому клиент, загружающий и выполняющий скомпилированную программу на языке Java, должен запустить JVM как клиентский процесс. После этого JVM начнет собственно выполнение загруженной программы путем интерпретации каждой ее инструкции, начиная с инструкций, содержащихся в методе *main*.

В Java-сите работа начинается с подтверждения того, что компоненту, который управлял переносом программы на машину клиента, можно доверять. Загрузка в Java производится посредством набора *загрузчиков классов (class loaders)*. Загрузчик классов предназначен для доставки определенного класса с сервера и загрузки его в адресное пространство клиента так, чтобы Java-машина могла создать объект этого класса. Загрузчик классов — это просто еще один класс Java, поэтому возможно, что загружаемая программа содержит собственные загрузчики классов. Использование только доверенных загрузчиков классов — это первое, что обеспечивает сито. В частности, Java-программам не разрешается создавать собственные загрузчики классов, которые могли бы обойти стандартный порядок загрузки классов.

Второй компонент Java-сита — это *верификатор байтов кода (byte code verifier)*. Верификатор проверяет, соблюдает ли загруженный класс правила защиты, предлагаемые ситом, в частности, не содержит ли класс запрещенных инструкций, которые могут каким-то образом повредить стек или память. Проверяются не все классы, как показано на рис. 8.26, а только те, которые загружаются на клиента с внешних серверов. Классы, находящиеся на машине клиента, обычно являются доверенными, хотя их целостность также легко можно проверить.

Наконец, после того как класс правильно загружен и проверен, JVM может создавать объекты этого класса и вызывать методы этих объектов. Чтобы предотвратить неавторизованный доступ этих объектов к ресурсам клиента, исполь-

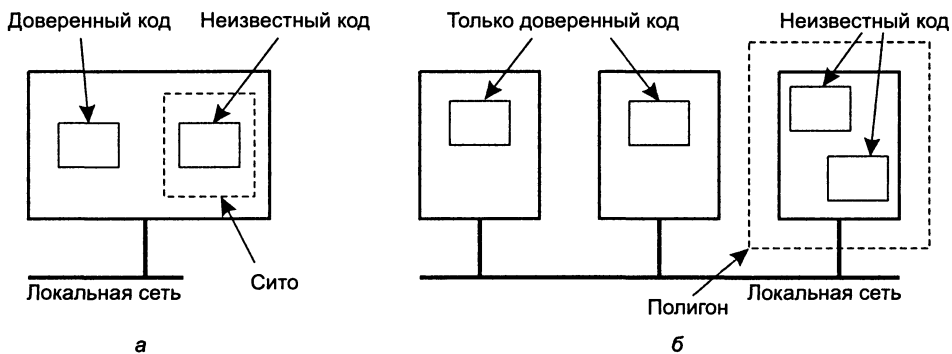


Рис. 8.27. Сито (а). Полигон (б)

Следующий шаг по пути ослабления ограничений — аутентификация каждой загруженной программы, с последующим соблюдением специфических правил защиты в зависимости от того, откуда была получена эта программа. Требовать, чтобы программы аутентифицировали себя, относительно просто: мобильный код может быть подписан так же, как и любой другой документ. Метод *подписания кода* (*code-signing*) часто используется как альтернатива ситу. Допускается использовать лишь код, поступивший с доверенных серверов.

Однако соблюдать правила защиты в этом случае становится затруднительно. В [484] для Java-программ предложено три механизма. Первый из них состоит в использовании объектов, называемых мандатами. Для доступа к локальным ресурсам, таким как файлы, программа должна получить ссылку на специальный объект, осуществляющий при его загрузке файловые операции. Если ссылка не получена, способа получить доступ к файлам нет. Этот принцип иллюстрирует рис. 8.28.

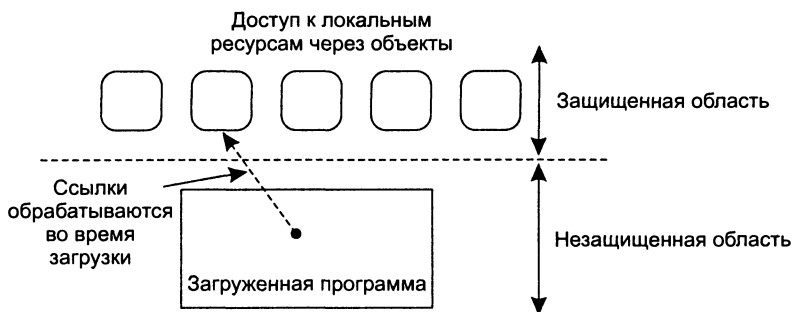


Рис. 8.28. Использование в качестве мандатов ссылок на Java-объекты

Все интерфейсы объектов, реализующих файловую систему, изначально скрыты от программы просто потому, что ей не предоставлены ссылки на эти интерфейсы. Строгий контроль типов в Java гарантирует невозможность построения ссылок на один из интерфейсов во время исполнения. Кроме того, мы можем использовать свойство Java хранить все переменные и методы внутри классов. В частности, программе можно запретить создавать свои собственные объекты для работы с файлами, скрыв операции создания новых объектов данного класса

(по терминологии Java, конструктор может быть объявлен закрытым для классов-наследников).

Второй механизм соблюдения правил защиты — это *расширенный самоанализ стека* (*extended stack introspection*). По существу, это означает, что любой вызов метода *m* локального ресурса предваряется вызовом специальной процедуры `enable_privilege`, которая проверяет, имеет ли вызывающий код право на вызов этого метода данного ресурса. Если обращение авторизовано, вызвавший код на время вызова получает соответствующие права. Перед возвращением управления вызвавшему коду по окончании выполнения метода *m* вызывается специальная процедура `disable_privilege`, которая отбирает эти права.

Чтобы иметь возможность вызова процедур `enable_privilege` и `disable_privilege`, разработчик интерфейса с локальными ресурсами должен вставить эти вызовы в соответствующие места программы. Однако будет лучше, если интерпретатор Java станет осуществлять эти вызовы автоматически. Это стандартный подход для работы с Java-апплетами, принятый, например, в web-браузерах. Очень хорошо выглядит следующее решение: когда происходит обращение к локальным ресурсам, интерпретатор Java автоматически вызывает процедуру `enable_privilege`, которая проверяет, допустимо ли такое обращение. Если да, то в стек добавляется вызов `disable_privilege`, чтобы гарантировать, что после возврата из метода привилегии будут сняты. Подобный способ предотвращает обход правил программистами с дурными намерениями.

Другое важное преимущество использования стека состоит в том, что он предоставляет очень хороший способ проверки привилегий. Предположим, что программа обращается к локальному объекту *O1*, который, в свою очередь, обращается к объекту *O2*. При этом даже если *O1* имеет разрешение на обращение к *O2*, некоторая процедура *O1* может не иметь такого разрешения на обращение к некоторому методу объекта *O2* — тогда эта цепочка вызовов будет запрещена. Самоанализ стека делает задачу выявления подобных цепочек несложной, поскольку интерпретатор должен лишь просмотреть каждую ячейку стека, начиная с вершины, проверяя, имеет элемент соответствующую привилегию (тогда вызов разрешен) или нет (тогда вызов немедленно прерывается). Этот подход иллюстрирует рис. 8.29.

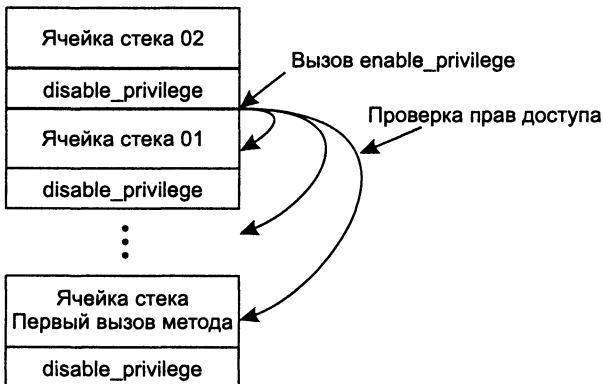


Рис. 8.29. Принцип самоанализа стека

В сущности, самоанализ стека применяется для связывания привилегий с классами или методами и отдельной проверки этих привилегий для каждого из вызывающих объектов. Таким образом, становится возможным реализовать защищенные домены на основе классов. Детально это рассматривается в [173].

Третий метод соблюдения правил защиты — *управление пространством имен (name space management)*. Идея следующая. Чтобы программа получила доступ к локальным ресурсам, ей сперва необходимо получить доступ к файлам, содержащим классы, в которых эти ресурсы реализованы. Для этого нужно, чтобы интерпретатору стали известны имена, которые будут разрешены в классе, загружаемом во время выполнения. Для соблюдения правил защиты конкретной загруженной программы одинаковые имена разрешаются в разных классах, в зависимости от того, откуда к нам попала эта программа. Обычно разрешение имен производится загрузчиками классов, которые для реализации подобного подхода должны быть специально адаптированы. Детали можно найти в [484].

Этот подход связывает привилегии классов и методов с тем местом, откуда была загружена программа. Возможность поддерживать соблюдение правил защиты описанными выше методами существует только благодаря интерпретатору Java. Понятно, что архитектура защиты в значительной степени зависит от используемого языка и для других языков должна разрабатываться «с нуля». Не зависящие от языка решения, описанные, например, в [211], требуют более обобщенных методов защиты и реализовать их значительно труднее. В этих подходах предполагается поддержка со стороны защищенной операционной системы, на которую возлагается обязанность анализа кода загружаемых модулей и пропуска всех обращений к локальным ресурсам через ядро с последующей их проверкой.

8.4. Управление защитой

Ранее мы рассмотрели защищенные каналы и контроль доступа, но лишь слегка затронули такие вопросы, как создание ключей. В этом разделе мы поближе познакомимся с управлением защитой. В частности, мы обсудим три вопроса. Во-первых, нам нужно рассмотреть общие вопросы управления криптографическими ключами и особенно вопросы их распространения. Мы обнаружим, что в этом важную роль играют сертификаты.

Во-вторых, мы рассмотрим проблему безопасного управления группой серверов, сосредоточившись при этом на проблеме добавления в группу нового члена, за которого ручаются присутствующие в группе серверы.

В-третьих, мы уделим внимание вопросу управления авторизацией, рассмотрев мандаты и то, что мы понимаем под сертификатами атрибутов. Для процесса авторизации в распределенных системах важную роль играет возможность делегирования одним процессом другому части или всех своих права доступа. Безопасное делегирование прав само по себе имеет определенные тонкости, которые мы обсудим в этом разделе.

8.4.1. Управление ключами

Ранее мы уже описывали различные криптографические протоколы, считая при этом, что ключи у нас уже есть. Так, например, в случае криптосистем с открытыми ключами мы считали, что отправитель сообщения имеет в своем распоряжении открытый ключ получателя и для поддержания конфиденциальности может зашифровать им свое сообщение. Точно так же, рассматривая аутентификацию с использованием центра распространения ключей (KDC), мы полагали, что каждая из сторон уже предоставила свой секретный ключ в распоряжение KDC.

Однако создание и распространение ключей — это не такая простая задача. Например, нам, вне всякого сомнения, придется найти нестандартные решения для распространения секретных ключей по небезопасным каналам и еще во многих других случаях. Кроме того, необходим механизм отмены ключей, то есть предотвращения использования ключа после того, как он скомпрометирован или испорчен. Так, например, если ключ попадет в чужие руки, его нужно будет объявить недействительным.

Создание ключей

Начнем с вопросов создания сеансовых ключей. Когда Алиса хочет организовать защищенный канал связи с Бобом, она может использовать для инициирования связи открытый ключ Боба, как это показано на рис. 8.18. Если Боб согласится начать взаимодействие, он должен создать сеансовый ключ и вернуть его Алисе, зашифровав открытым ключом Алисы. После шифрования общего сеансового ключа его можно смело передавать по сети.

Подобная схема может использоваться для генерации и распространения сеансового ключа в том случае, если Алиса и Боб уже обладают общим секретным ключом. Однако оба метода требуют, чтобы связывающиеся стороны могли организовать защищенный канал. Другими словами, у них уже должен быть некий способ создания и распространения ключей. Тот же самый аргумент применим и в том случае, когда общий секретный ключ создается доверенной третьей стороной, например KDC.

Красивый и широко используемый способ создания общего ключа и передачи его по небезопасным каналам [126] — это *обмен ключами по Диффи–Хеллману* (*Diffie–Hellman key exchange*). Этот протокол работает следующим образом. Допустим, что Алиса и Боб желают создать общий секретный ключ. Первое требование состоит в том, что они должны договориться об использовании двух больших чисел, n и g , которые удовлетворяют некоторым математическим ограничениям (о них мы поговорим позже). Оба этих числа нет необходимости скрывать от посторонних. Далее Алиса выбирает большое случайное число, скажем, x , которое она держит в секрете. Боб также выбирает свое секретное число, назовем его y . В этот момент, как мы видим из рис. 8.30, у нас имеется вся информация, необходимая для создания секретного ключа.

Алиса начинает с того, что посылает Бобу число $g^x \bmod n$, вместе с n и g . Важно отметить, что эту информацию можно послать открытым текстом, так как, имея значение $g^x \bmod n$, вычислить x практически невозможно. Получив сообще-

ние, Боб вычисляет $(g^x \bmod n)^y$, что равно $g^{xy} \bmod n$. Кроме того, он посылает Алисе число $g^y \bmod n$, а она, в свою очередь, вычисляет $(g^y \bmod n)^x = g^{xy} \bmod n$. Соответственно, Алиса и Боб, и только они двое, имеют теперь общий секретный ключ $g^{xy} \bmod n$. Отметим, что никому из них не понадобилось сообщать другому свое секретное число (соответственно, x или y).

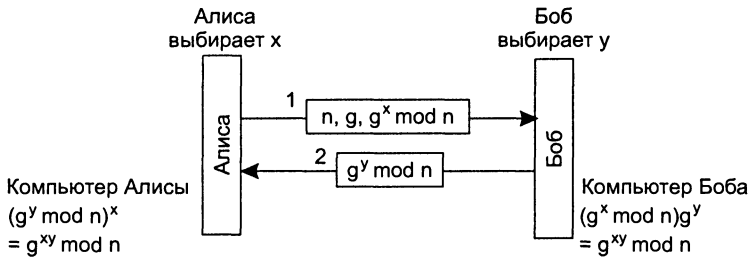


Рис. 8.30. Обмен ключами по Диффи—Хеллману

Протокол Диффи—Хеллмана можно рассматривать как криптосистему с открытым ключом. В случае Алисы x — это ее закрытый ключ, а $g^{xy} \bmod n$ — открытый. Как мы далее увидим, процедура защищенного распространения открытых ключей очень напоминает протокол Диффи—Хеллмана.

Распространение ключей

Одна из наиболее сложных частей управления ключами — это распространение исходных ключей. В симметричных криптосистемах исходный общий секретный ключ должен распространяться по безопасным каналам с аутентификацией и конфиденциально, как показано на рис. 8.31, а [290]. Если у Алисы и Боба нет ключа для организации защищенного канала, необходимо обменяться ключами другим способом. Иными словами, Алиса и Боб должны войти в контакт при помощи других средств связи, а не через сеть. Так, например, они могут связаться по телефону или послать ключ на дискете обычной бандеролью.

В случае криптосистемы с открытым ключом нам необходимо распространить открытый ключ так, чтобы получатели были уверены, что ключ действительно является парой заявленному закрытому ключу. Другими словами, как показано на рис. 8.31, б, хотя открытый ключ сам по себе может быть послан даже простым текстом, необходимо, чтобы канал, по которому он пересылается, обеспечивал аутентификацию. Закрытый ключ, разумеется, необходимо пересылать только по безопасному каналу, с идентификацией и конфиденциально.

Аутентифицированное распространение открытого ключа выглядит более любопытно. На практике распространение открытых ключей осуществляется при помощи *сертификатов открытого ключа* (*public key certificates*). Эти сертификаты состоят из открытого ключа и строки, определяющей сущность, с которой ассоциирован этот ключ. Сущностью может быть пользователь, хост или некое специальное устройство. Открытый ключ и идентификатор вместе подписываются *сертифицирующей организацией* (*certification authority*), и эта подпись также добавляется к сертификату (наименование сертифицирующей организации

является естественной частью сертификата). Подпись производится закрытым ключом K_{CA} , который принадлежит сертифицирующей организации. Соответствующий открытый ключ K_{CA}^+ не считается секретным. Так, открытые ключи различных сертифицирующих организаций встроены в большинство web-браузеров и распространяются вместе с файлами программ.

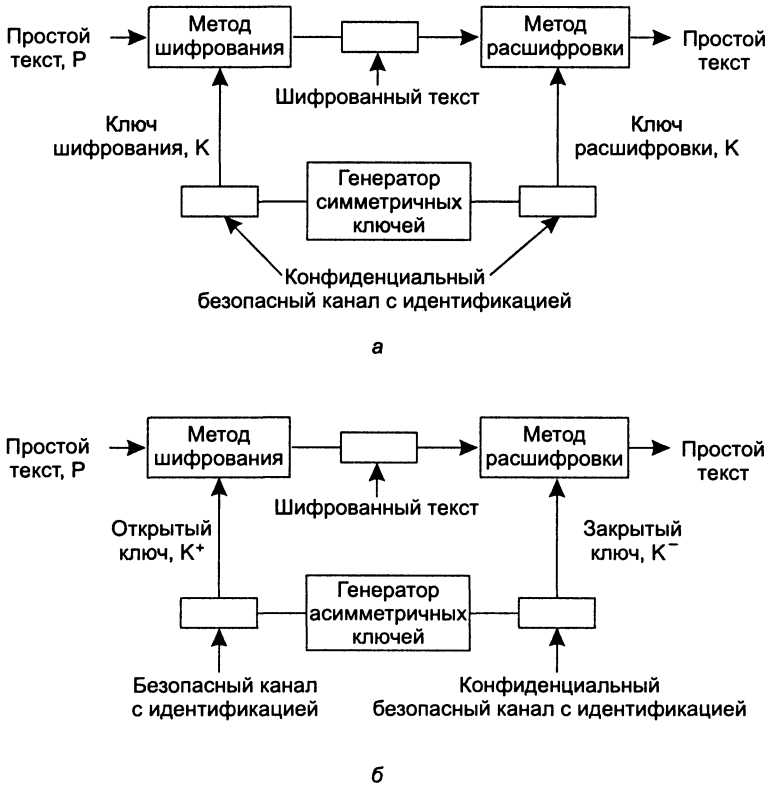


Рис. 8.31. Распространение секретного ключа (а). Распространение открытого ключа (б)

Сертификаты открытого ключа используются следующим образом. Предположим, что клиент желает установить, принадлежит ли на самом деле открытый ключ, содержащийся в сертификате, указанной сущности. Он с помощью открытого ключа соответствующей сертифицирующей организации проверяет подпись сертификата. Если подпись сертификата соответствует паре (*открытый ключ, идентификатор*), то клиент считает, что открытый ключ действительно принадлежит указанной сущности.

Важно отметить, что, считая сертификат годным, клиент на самом деле верит, что сертификат не подделан. В частности, клиент должен предполагать, что открытый ключ K_{CA}^+ действительно принадлежит соответствующей сертифицирующей организации. Если на этот счет имеются сомнения, можно проверить правильность K_{CA}^+ через другой сертификат, полученный от другой сертифицирующей организации, которой, возможно, пользователь доверяет больше.

Такие иерархические *модели доверия (trust models)*, в которых сертифицирующей организации самого верхнего уровня доверяют все — не редкость. Так, например, *конфиденциальная почта (Privacy Enhanced Mail, PEM)* использует трехуровневую модель доверия, в которой сертифицирующие организации нижнего уровня идентифицируются *организациями сертификации правил (Policy Certification Authorities, PCA)*, которые, в свою очередь, идентифицируются *организацией регистрации правил в Интернете (Internet Policy Registration Authority, IPRA)*. Если пользователь не доверяет IPRA или не считает, что может безопасно общаться с IPRA, нет надежды, что он когда-либо поверит письмам, передаваемым почтой PEM. Дополнительную информацию по этой модели можно почерпнуть из [232]. Другие модели доверия обсуждаются в [290].

Срок жизни сертификатов

Важный вопрос, связанный с сертификатами, — это их *срок жизни (lifetime)*. Рассмотрим сначала ситуацию, когда сертифицирующая организация выдает постоянные сертификаты. По существу, такой сертификат означает, что открытый ключ для сущности, им идентифицируемой, всегда является истинным. Ясно, что это нереально. Если закрытый ключ идентифицируемой сущности однажды будет скомпрометирован, ничего не подозревающие клиенты не смогут использовать открытые ключи (в отличие от клиентов с дурными намерениями). Для подобных случаев нужен механизм *отзыва (revoke)* сертификата, который позволил бы сделать известным тот факт, что сертификат более недействителен.

Существует несколько способов отзыва сертификата. Один из традиционных методов — воспользоваться *списком отозванных сертификатов (Certificate Revocation List, CRL)*, который регулярно публикуется сертифицирующей организацией. Когда клиент проверяет сертификат, он просматривает CRL, чтобы определить, не был ли данный сертификат отозван. Это означает, что клиент должен связываться с сертифицирующей организацией как минимум каждый раз по выходе нового списка CRL. Отметим, что если CRL публикуется ежедневно, то на отзыв сертификатов также дается всего один день. Таким образом, скомпрометированный сертификат может использоваться злоумышленником до тех пор, пока он не будет включен в очередной список CRL. Соответственно, время между публикациями CRL не должно быть слишком большим. Кроме того, получение CRL вызывает дополнительные затраты.

Альтернативный подход предполагает ограничение срока жизни сертификата. В сущности, этот подход напоминает аренду, о которой мы говорили в главе 6. Действие сертификатов по истечении определенного времени автоматически прекращается. Если по каким-то причинам сертификат был отозван до прекращения времени его действия, сертифицирующая организация может опубликовать сведения об этом в CRL. Однако подобный подход побуждает клиентов просматривать последний список CRL вне зависимости от времени получения сертификата. Другими словами, они вынуждены связываться с сертифицирующей организацией или доверенной базой данных, содержащей последний список CRL.

Последний крайний вариант — сократить срок жизни сертификатов почти до нуля. В действительности это означает полный отказ от использования сертифи-

катов. Вместо этого клиент всегда должен связываться с сертифицирующей организацией и проверять годность открытого ключа. Соответственно, сертифицирующая организация должна постоянно быть на связи.

На практике сертификаты обладают ограниченным сроком жизни. В случае Интернет-приложений время жизни часто составляет один год [432]. Такой подход требует регулярной публикации CRL и проверки срока действия сертификата. Практика показывает, что приложения клиентов редко заглядывают в CRL, автоматически считая сертификаты годными до окончания их срока действия. В этом смысле практика обеспечения защиты в Интернете имеет массу возможностей для совершенствования.

8.4.2. Управление защищенными группами

Множество систем защиты пользуются услугами специальных служб, таких как центры распространения ключей (KDC) или сертифицирующие организации. Работа этих служб в распределенных системах вызывает различные проблемы. Во-первых, им необходимо доверие. Чтобы доверие к службам защиты повышалась, им необходимо демонстрировать высокую степень защищенности от различных типов атак. Так, например, как только сертифицирующие организации оказываются скомпрометированными, становится невозможно проверить правильность открытого ключа, что лишает смысла всю систему защиты.

С другой стороны, необходимо также, чтобы весь этот набор служб защиты имел высокую доступность. Так, если рассматривать KDC, всякий раз, когда два процесса хотят организовать защищенный канал связи, минимум один из них должен связаться с KDC, чтобы получить общий секретный ключ. Если центр KDC в этот момент недоступен, без альтернативных методов создания ключа, таких как обмен ключами Диффи–Хеллмана, установить безопасное соединение невозможно.

Решить проблему высокой доступности помогает репликация. Однако репликация повышает уязвимость сервера перед лицом атак на защиту. Мы уже обсуждали, как путем разделения секрета членами группы организуется защищенное групповое взаимодействие. При таком подходе ни один из членов группы не способен самостоятельно скомпрометировать сертификат, что значительно повышает защиту группы в целом. Остается обсудить только вопрос реального управления группой реплицируемых серверов. В [377] предлагается для этого следующее решение.

Итак, требуется подтвердить, что процесс, который просит предоставить ему членство в группе G , не в состоянии нарушить целостность группы. Предполагается, что у группы G имеется закрытый ключ SK_G , который используется совместно всеми ее членами для шифрования сообщений внутри группы. Кроме того, они используют также пару открытый/закрытый ключ (K_G^+ , K_G^-) для связи с процессами, не входящими в число членов группы.

Когда процесс P хочет войти в число членов группы G , он посылает запрос JR на прием в члены, в котором определяет G и P , локальное время отправки запроса T по часам P , общую заготовку ответа RP и сгенерированный закрытый ключ $K_{P,G}$. RP и $K_{P,G}$ зашифрованы с использованием открытого ключа группы K_G^+ , как

показано в сообщении 1 на рис. 8.32. Запрос JR на прием в члены группы подписывается P и посылается вместе с сертификатом, содержащим открытый ключ P . Для обозначения того, что сообщение M подписано субъектом A , мы используем традиционную запись $[M]_A$.

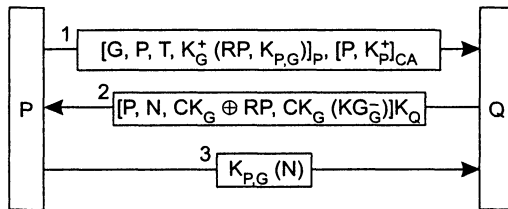


Рис. 8.32. Безопасное включение в группу нового члена

Когда член группы Q получает запрос на прием в группу нового члена, он первым делом идентифицирует P , после чего связывается с другими членами группы, чтобы обсудить, следует ли принять P в группу. Аутентификация P происходит обычным путем посредством сертификата. Чтобы гарантировать, что сертификат в момент отправки был действителен, используется отметка времени T (отметим, что нам необходима уверенность, что время не фальсифицировано). Член группы Q проверяет подпись сертифицирующей организации, после чего извлекает открытый ключ P из сертификата для проверки подлинности JR . Для проверки того, все ли члены группы готовы принять в нее P , используется внутренний протокол группы.

Если члены группы согласны принять P в группу, Q возвращает сообщение с согласием группы GA , показанное на рис. 8.32 как сообщение 2, которое идентифицирует P и содержит одноразовый запрос N . Для шифрования коммуникационного ключа группы CK_G используется заготовка ответа RP . Кроме того, P нужен также закрытый ключ группы K_G^- , который зашифрован при помощи CK_G . Сообщение GA подписывается Q при помощи ключа $K_{P,G}$.

Процесс P может теперь идентифицировать Q , поскольку только истинный член группы может знать секретный ключ $K_{P,G}$. Случайное число N в этом протоколе для обеспечения защиты не используется, просто когда P посылает назад число N , зашифрованное ключом $K_{P,G}$ (сообщение 3), Q узнает, что процесс P получил все необходимые ключи и действительно вошел в группу.

Отметим, что вместо использования заготовки ответа RP процессы P и Q также могут зашифровать CK_G открытым ключом P . Однако поскольку заготовка RP требуется только один раз, а именно для шифрования коммуникационного ключа группы в сообщении GA , вариант с заготовкой более безопасен. Если закрытый ключ P будет перехвачен, это может привести и к расшифровке CK_G , что нарушит секретность всего группового взаимодействия.

8.4.3. Управление авторизацией

Управление защитой в распределенных системах также включает в себя и управление правами доступа. Ранее мы старались избежать разговора о том, каким об-

разом пользователям или группам пользователей назначаются права доступа и каким образом затем осуществляется поддержка прав доступа, исключая их фальсификацию. Настало время заполнить эти пробелы.

В нераспределенных системах управлять правами доступа относительно несложно. Когда к системе добавляется новый пользователь, он получает базовый набор прав, например, на создание файлов и вложенных каталогов в определенном каталоге, создание процессов, использование процессорного времени и т. д. Другими словами, на одной конкретной машине создается учетная запись пользователя, все права для которой заранее расписаны системным администратором.

В распределенных системах задача усложняется тем фактом, что ресурсы разбросаны по нескольким машинам. Если следовать подходу, принятому в нераспределенных системах, необходимо создавать учетные записи всех пользователей на всех машинах. В сущности, именно такой подход используется в сетевых операционных системах. Задачу можно немного упростить, создав одну учетную запись на центральном сервере. С этим сервером можно будет консультироваться всякий раз, когда пользователь запросит доступ к определенным ресурсам или машинам.

Мандаты и сертификаты атрибутов

Наилучший подход, чаще всего применяемый в распределенных системах, — это использование мандатов. Как мы уже говорили раньше, *мандат* (*capability*) — это нефальсифицируемая структура данных, относящаяся к некоторому ресурсу и точно определяющая права доступа владельца мандата к этому ресурсу. Существуют различные реализации мандатов. Здесь мы кратко обсудим реализацию, используемую в операционной системе Amoeba [448].

Система Amoeba — это одна из первых объектно-ориентированных распределенных систем. Распределенные объекты в ней являются удаленными. Другими словами, объект располагается на сервере, а клиентам разрешен прозрачный доступ к этому объекту через заместителя. Для обращения к операциям объекта клиент посылает мандат локальной операционной системе, которая находит сервер, содержащий объект, и посылает этому серверу вызов RPC.

Мандат — это 128-битный идентификатор, внутренняя структура которого показана на рис. 8.33. Первые 48 бит инициализируются сервером объекта, когда этот объект создается, и содержат машинно-независимый идентификатор сервера, известный под названием *порта сервера* (*server port*). Для поиска машины, на которой в настоящее время размещен сервер, Amoeba использует широковебательную рассылку.

Порт сервера (48 бит)	Объект (24 бит)	Права (8 бит)	Проверка (48 бит)
-----------------------	-----------------	---------------	-------------------

Рис. 8.33. Мандат системы Amoeba

Следующие 24 бита используются для идентификации объекта на данном сервере. Отметим, что порт сервера вместе с идентификатором объекта образуют

72-битный уникальный глобальный идентификатор каждого объекта Атмоеба. Следующие 8 бит идентифицируют права доступа владельца мандата. И наконец, 48-битное поле проверки, как мы покажем далее, требуется для того, чтобы сделать подделку мандата невозможной.

Когда создается объект, его сервер выбирает случайное поле проверки и записывает его как в мандат, так и в собственную внутреннюю таблицу. Все биты прав доступа в мандате исходно выставлены в единицу, и это тот *мандат владельца (owner capability)*, который возвращается клиенту. Когда мандат пересылается обратно на сервер в составе запроса на совершение операции, поле проверки проверяется на соответствие его значению, хранящемуся на сервере.

Для создания ограниченного мандата клиент должен передать мандат серверу с указанием новой битовой маски в соответствии с новыми правами доступа. Сервер берет исходное поле проверки из своих таблиц, производит операцию исключающего ИЛИ (XOR) над мандатом и новыми правами (которые представляют собой подмножество прав мандата владельца) и пропускает результат через одностороннюю функцию.

Затем сервер создает новый мандат с тем же значением поля объекта, новыми битами прав в поле прав и результатом односторонней функции в поле проверки. Новый мандат возвращается тому клиенту, который захотел его получить. Клиент, если пожелает, может переслать новый мандат другому процессу.

Метод создания ограниченных мандатов иллюстрирует рис. 8.34. В этом примере владелец отключает все права, кроме одного. Такой ограниченный мандат, например, может разрешать чтение объекта и ничего более. Значения поля прав для разных типов объектов могут быть разные, поскольку набор разрешенных операций сам может различаться от одного типа объекта к другому.

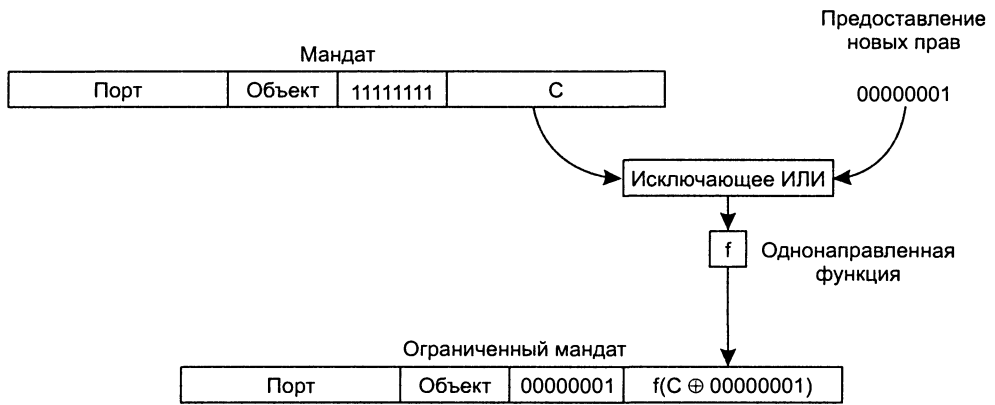


Рис. 8.34. Создание ограниченного мандата из мандата владельца

Когда ограниченный мандат приходит на сервер, сервер по полю прав обнаруживает, что перед ним не мандат владельца, поскольку хотя бы один из битов сброшен. После этого сервер считывает из таблицы исходное случайное число, производит операцию исключающего ИЛИ (XOR) над ним и полем прав ман-

дата, после чего пропускает результат через одностороннюю функцию. Если результат соответствует полю проверки, мандат считается действительным.

Из алгоритма должно быть ясно, что пользователь, попытавшись добавить себе права, которых он не имеет, просто испортит мандат. Инвертировать поле проверки ограниченного мандата для получения исходного аргумента ($C \text{ XOR } 00000001$, как показано на рис. 8.34) невозможно, поскольку функция f односторонняя. Посредством подобной криптографической техники мандаты защищаются от подделки. Отметим, что действие функции f , по существу, аналогично вычислению дайджеста сообщений, о котором мы говорили ранее. Любое изменение исходного сообщения, такое как инверсия битов, будет немедленно замечено.

Более обобщенный вариант мандата, который используется в современных распределенных системах, — это *сертификат атрибута* (*attribute certificate*). В отличие от ранее обсуждавшихся сертификатов, которые требуются для подтверждения открытых ключей, сертификаты атрибутов служат для хранения пар (*атрибут, значение*), относящихся к определенным элементам. В частности, сертификаты атрибутов могут использоваться для хранения прав доступа к определенным ресурсам, которыми обладает владелец сертификата.

Как и другие сертификаты, сертификаты атрибутов выдаются специальными сертифицирующими организациями, которые называются *организациями сертификации атрибутов* (*attribute certification authorities*). Как и в ситуации с мандатами системы Атоева, эти организации соответствуют серверам объектов. Обычно, однако, организация сертификации атрибутов и сервер, содержащий сущность, для которой создается сертификат, не совпадают. Права доступа, перечисленные в сертификате, подписываются организацией сертификации атрибутов.

Делегирование

Обсудим теперь следующую проблему. Пользователь хочет напечатать большой файл, к которому имеет доступ только на чтение. Чтобы не затруднять жизнь окружающим, пользователь посылает запрос на сервер печати, требуя начать печать не раньше двух часов ночи. Вместо того чтобы посылать на принтер сам файл, пользователь передает принтеру имя файла, с тем чтобы он мог сам скопировать этот файл в свой каталог спулинга, когда ему это понадобится.

Хотя эта схема кажется великолепной, в ней имеется одна серьезная проблема: принтер обычно не имеет необходимых привилегий на доступ к указанному файлу. Другими словами, если не предусмотреть никакого специального механизма, то как только сервер печати захочет считать файл, чтобы напечатать его, система откажет серверу в доступе к этому файлу. Проблему можно решить, если пользователь сможет временно *делегировать* (*delegated*) серверу печати свои права доступа к файлу.

Делегирование прав доступа — это один из весьма важных приемов реализации защиты в компьютерных системах, и в частности в распределенных системах. Основная идея делегирования проста: если иметь возможность передавать определенные права доступа от одного процесса к другому, то проще будет распределять работу между несколькими процессами без ослабления защиты ресур-

сов. В случае распределенных систем процессы могут запускаться на разных машинах и даже в разных административных доменах (о них мы говорили, обсуждая систему Globus). Делегирование позволяет избежать излишней затраты ресурсов, поскольку защиту часто удается реализовать локально.

Существует несколько способов реализации делегирования. Традиционный подход, описанный в [315], предполагает наличие заместителя. *Заместитель (proxy)* в контексте защиты компьютерных систем представляет собой маркер, который позволяет своему владельцу работать с теми же самыми ограниченными правами и привилегиями, которые имел субъект, выдавший этот маркер. (Отметим, что данный заместитель отличается от заместителя — синонима заглушки на стороне клиента¹. Хотя мы стараемся избегать переопределения терминов, для «заместителя» мы в данном случае делаем исключение, поскольку этот термин слишком активно используется, чтобы это можно было игнорировать.) Процесс может создать заместителя, в лучшем случае с такими же правами доступа и привилегиями, какие имеет он сам. Если процесс создает нового заместителя на основе одного из тех, которыми он владеет в данный момент, порожденный заместитель получает как минимум такие же ограничения, как оригинальный, а возможно и большие.

До обсуждения общей схемы делегирования рассмотрим два частных случая. Во-первых, делегирование — относительно простое дело, если Алиса всех знает. Если она хочет делегировать права Бобу, ей достаточно создать сертификат о присвоении Бобу прав R , приблизительно такой: $[A, B, R]_A$. Если Боб хочет передать часть этих прав Чарли, он должен предложить Чарли войти в контакт с Алисой и запросить соответствующий сертификат у нее.

Во-вторых, Алиса может просто создать сертификат, гласящий, что его предьявитель обладает правами R . Однако в этом случае придется защитить этот сертификат от несанкционированного копирования, как мы это делали при защищенной передаче мандатов от процесса к процессу. Схема Ньюмана (Neuman) в состоянии справиться с подобными задачами, кроме того, эта схема снимает для Алисы обязательное требование знать каждого, кому она захочет делегировать права.

По схеме Ньюмана заместитель, как показано на рис. 8.35, состоит из двух частей. Пусть A — процесс, который создает заместителя. Первая часть заместителя — это множество $C = \{R, S_{proxy}^+\}$, состоящее из набора прав доступа R , которые делегируются A вместе с общеизвестной частью секрета, необходимого для аутентификации держателя сертификата. Ниже мы рассмотрим, как используется S_{proxy}^+ . Сертификат имеет от процесса A подпись $sig(A, C)$ для защиты от изменений. Вторая часть заместителя содержит вторую часть секрета, обозначенную как S_{proxy}^- . Важно отметить, что часть S_{proxy}^- защищена от вскрытия в случае делегирования прав другому процессу.

Другой взгляд на заместителя состоит в следующем. Если Алиса хочет делегировать часть своих прав Бобу, она создает список прав (R), которыми должен

¹ Отметим также, что в данном случае термин *заместитель (proxy)* не имеет отношения к термину *прокси (proxy)*, который используется для определения *прокси-шлюза (proxy gateway)*, или *прокси-сервера (proxy server)*. — Примеч. ред.

обладать Боб. Подписывая список, она защищает его от взлома Бобом. Однако подписанного списка прав часто бывает недостаточно. Если Боб захочет реализовать свои права, ему может понадобиться доказать, что он действительно получил этот список от Алисы, а не своровал его где-нибудь. Поэтому Алиса задает весьма неприятный вопрос S_{proxy}^+ , ответ на который знает только она (S_{proxy}^-). Каждый может легко проверить правильность ответа, задав вопрос. Вопрос добавляется к списку еще до того, как Алиса поставит на нем свою подпись.

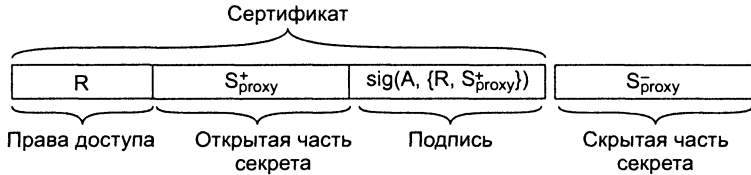


Рис. 8.35. Обобщенная структура заместителя, используемого при делегировании прав

При делегировании части своих прав Алиса предоставляет подписанный список этих прав вместе с вопросом Бобу. Также она отправляет Бобу ответ, уверившись предварительно, что никто не сможет его перехватить. Теперь у Боба есть подписанный Алисой список прав, который он может передать при необходимости, скажем, Чарли. Чарли задаст ему вопрос из конца списка. Если Боб сможет ему ответить, Чарли будет уверен, что Алиса действительно делегировала ему свои права.

Важное свойство этой схемы состоит в том, что Алисе ни с кем не нужно консультироваться. На деле Боб может решить передать права или их часть Дейву. Делая это, он также сообщит Дейву ответ на вопрос, чтобы Дейв мог подтвердить, что список прав был передан ему кем-то из законных владельцев. Алисе про Дейва ничего знать не понадобится.

Протокол делегирования и реализации прав иллюстрирует рис. 8.36. Допустим, что Алиса и Боб совместно используют секретный ключ $K_{A,B}$, которым шифруют отправляемые друг другу сообщения. Так, Алиса первой отправляет Бобу сертификат $C = \{R, S_{proxy}^+\}$, подписанный $sig(A, C)$ и обозначаемый далее как $[R, S_{proxy}^+]_A$. Шифровать это сообщение нет необходимости, его вполне можно отправить и простым текстом. Шифруется только закрытая часть секрета, обозначенная на рисунке как $K_{A,B}(S_{proxy}^-)$ в сообщении 1.

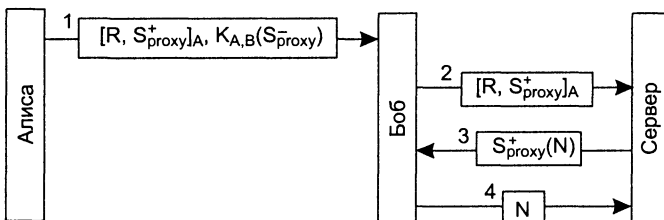


Рис. 8.36. Использование заместителя для делегирования прав и их подтверждения

Теперь допустим, что Боб хочет выполнить операцию над объектом, расположенным на определенном сервере. Считаем, что Алиса имеет право на проведение подобной операции и делегировала это право Бобу. Таким образом, Боб передает серверу свои верительные грамоты в виде подписанного сертификата $[R, S_{proxy}^+]_{A\cdot}$.

В этот момент сервер может проверить, не был ли сертификат S сфальсифицирован: любые изменения списка прав или секретного вопроса будут обнаружены, поскольку и тот и другой подписаны Алисой. Однако сервер пока еще не знает, является ли Боб законным владельцем сертификата. Чтобы проверить это, сервер должен использовать секрет, который попал к Бобу вместе с S . Существует несколько способов реализации S_{proxy}^+ и S_{proxy}^- . Предположим, например, что S_{proxy}^+ — открытый, а S_{proxy}^- — соответствующий закрытый ключ. Сервер может проверить Боба, послав ему случайное число N , зашифрованное ключом S_{proxy}^+ . Расшифровав $S_{proxy}^+(N)$ и вернув N , Боб докажет, что знает секрет, а значит, является законным владельцем сертификата. Существуют и другие способы реализации безопасного делегирования прав, но основная идея всегда одна и та же: нужно показать, что вы знаете секрет.

8.5. Пример — Kerberos

Теперь уже должно быть понятно, что обеспечение защиты в распределенных системах — задача не из простых. Проблемы возникают из-за того, что защита должна быть всеобъемлющей, если в ней встретятся слабые места, то легко преодолимой становится защита всей системы. Чтобы помочь разработчику распределенных систем поддерживать мириады правил защиты, было создано несколько систем поддержки защиты, которые можно использовать в качестве базы для дальнейших разработок. Одной из часто применяемых систем подобного рода является Kerberos [239, 433].

Система Kerberos была разработана в М.И.Т. и основывается на протоколе аутентификации Нидхема—Шредера (Needham—Schroeder), который мы ранее рассматривали. В настоящее время используются две версии Kerberos, версия 4 и версия 5. Обе версии концептуально одинаковы, но версия 4 проще для понимания. Версия 5 по сравнению с версией 4 содержит множество дополнительных возможностей и поэтому на практике обычно предпочтительнее. Мы сосредоточимся исключительно на проблемах аутентификации.

Kerberos можно рассматривать как безопасную систему, которая помогает клиентам создавать защищенный канал связи с сервером. Защита реализуется при помощи общих секретных ключей. В системе имеются два компонента. *Сервер аутентификации (Authentication Server, AS)* отвечает за обработку запросов на организацию соединения, присылаемых пользователями. AS идентифицирует пользователей и предоставляет им ключи, которые можно использовать для организации защищенных каналов связи с сервером. Создание защищенных каналов производится *службой предоставления талонов (Ticket Granting Service, TGS)*. TGS рассылает специальные сообщения, известные под названием *талонов (tickets)*,

которые используются, чтобы сообщить серверу, что клиент — действительно тот, за кого он себя выдает. Ниже мы приведем конкретные примеры талонов.

Давайте рассмотрим, как Алиса входит в распределенную систему, защищенную системой Kerberos, и устанавливает защищенный канал связи с сервером Боба. Для входа в систему Алиса может использовать любую свободную рабочую станцию. Рабочая станция отправляет ее имя серверу AS, который возвращает сеансовый ключ $K_{A,TGS}$ и талон, который она должна будет передать службе TGS.

Талон, возвращенный AS, содержит идентификатор Алисы и созданный секретный ключ, который Алиса и TGS могут использовать для связи между собой. Сам талон будет передан Алисой службе TGS, поэтому важно, чтобы никто, кроме TGS, не мог его прочитать. Для этого талон шифруется секретным ключом $K_{AS,TGS}$, которым совместно пользуются AS и TGS.

Эта часть процедуры входа в систему соответствует сообщениям 1, 2 и 3 на рис. 8.37. Сообщение 1 на самом деле не является сообщением, оно соответствует вводу Алисой на рабочей станции своего *входного имени (login)*. Сообщение 2 содержит это имя и посылается AS. Сообщение 3 содержит сеансовый ключ $K_{A,TGS}$ и талон $K_{AS,TGS}(A, K_{A,TGS})$. Для обеспечения защиты сообщение 3 шифруется секретным ключом $K_{A,AS}$, совместно используемым Алисой и AS.

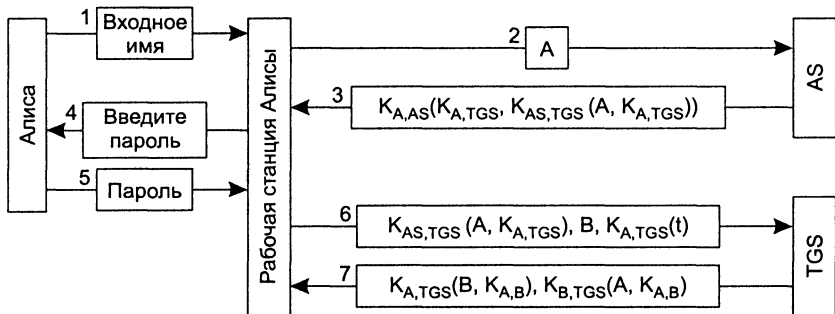


Рис. 8.37. Аутентификация в системе Kerberos

После того как рабочая станция получает ответ от AS, она предлагает Алисе ввести пароль (сообщение 4), который будет использован для последующей генерации секретного ключа $K_{A,AS}$ (относительно несложно взять строку символов, применить к ней криптографическое хэширование и использовать первые 56 бит хэша в качестве секретного ключа). Отметим, что подобный подход имеет не только то преимущество, что пароль Алисы никогда не передается по сети в открытом виде; он даже временно не сохраняется на рабочей станции. Более того, сразу после создания общего ключа $K_{A,AS}$ рабочая станция получает сеансовый ключ $K_{A,TGS}$ и может забыть о пароле Алисы, пользуясь только секретным общим ключом $K_{A,AS}$.

После этой части аутентификации Алиса может считать себя вошедшей в систему. Теперь она может связываться с другими пользователями и серверами. Если она хочет пообщаться с Бобом, она запрашивает у TSG сеансовый ключ для работы с Бобом (сообщение 6 на рис. 8.37). Тот факт, что Алиса обладает та-

лоном $K_{A,TGS}(A, K_{A,TGS})$, подтверждает, что это Алиса. TGS возвращает ей сеансовый ключ $K_{A,B}$, вновь упакованный в талон, который Алиса позже перешлет Бобу.

Сообщение 6 также содержит отметку времени t , зашифрованную секретным ключом, который совместно используется Алисой и TGS. Эта отметка времени требуется для того, чтобы Чак не смог злонамеренно послать сообщение 6 еще раз и попытаться организовать защищенный канал с Бобом. TGS проверяет отметку времени, прежде чем возвращать Алисе талон. Если указанное в отметке время отличается от текущего времени более чем на несколько минут, запрос на талон будет оставлен без ответа.

Как показано на рис. 8.38, организация защищенного канала связи с Бобом теперь проста и не вызывает никаких сложностей. Сначала Алиса вместе с зашифрованной отметкой времени посылает Бобу сообщение, содержащее талон, который она получила от TGS. Расшифровав талон, Боб узнает, что с ним общается Алиса, поскольку создать талон могла только служба TGS. Кроме того, он получает секретный ключ $K_{A,B}$, который позволяет ему проверить отметку времени. После этого Боб понимает, что общается с именно Алисой. Ответив послышкой сообщения $K_{A,B}(t+1)$, Боб убеждает Алису, что это действительно он.

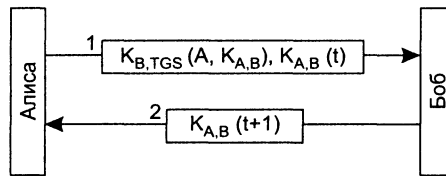


Рис. 8.38. Создание защищенного канала в системе Kerberos

8.6. Пример — SESAME

Давайте теперь рассмотрим другую систему защиты, которая похожа на Kerberos, но в ней используется комбинация криптографии с открытым ключом и общих секретных ключей. Проект *SESAME* был начат по совместной инициативе нескольких крупных европейских компаний с целью разработки стандартов защиты открытых систем. *SESAME* — это акроним слов *Secure European System for Application in a Multi-vendor Environment* (*Европейская система защиты приложений, работающих в разнородной среде*). Первая реализация *SESAME* была создана в 1991 году, четвертая и наиболее известная существует с 1995 года и известна также под названием *SESAME V4*. Мы начнем с обзора основных компонентов, затем обсудим процесс авторизации, выполняемый при помощи сертификатов атрибутов. Вместо того чтобы рассматривать в деталях все алгоритмы, мы сосредоточим наше внимание на общих вопросах взаимодействия между компонентами, которые иллюстрируют архитектуру защищенных систем. Обзор системы *SESAME* можно найти в [342].

8.6.1. Компоненты системы SESAME

SESAME может рассматриваться как система клиент-сервер, в которой клиенты имеют доступ к приложениям, выполняющимся на удаленных серверах. Подобная организация служит причиной разделения компонентов SESAME на три группы, как показано на рис. 8.39. Эти три группы, соответственно — общие компоненты защиты, компоненты защиты клиента и компоненты защиты сервера. Их мы и обсудим далее.

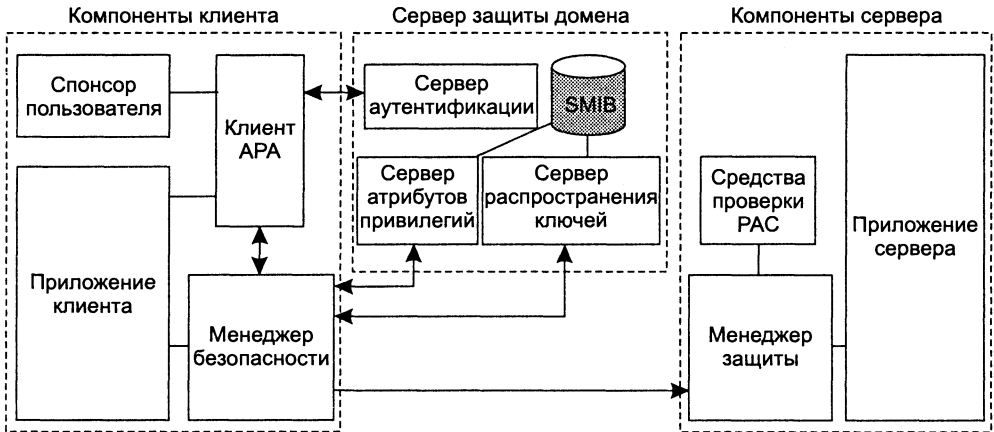


Рис. 8.39. Компоненты SESAME

Общие компоненты защиты

Общие компоненты защиты отвечают за аутентификацию, авторизацию и распространение ключей. Эти компоненты обычно размещаются на отдельной машине, известной под названием *сервера защиты домена* (*Domain Security Server, DSS*).

Сервер аутентификации (*Authentication Server, AS*) отвечает за аутентификацию пользователей (людей) и приложений (программ). В случае приложений аутентификация происходит при помощи секретных ключей, хранимых в отдельной базе данных, называемой *информационной базой управления защитой* (*Security Management Information Base, SMIB*). База SMIB реализована в виде набора отдельных файлов и хранит также информацию, относящуюся к другим компонентам, входящим в сервер защиты домена. Пользователи аутентифицируются при помощи секретного ключа, порождаемого, как и в системе Kerberos, из пароля, введенного пользователем. Однако если пользователь имеет закрытый ключ, то для аутентификации используется этот ключ.

Если Алиса захочет войти в систему, используя свой закрытый ключ, она посылает AS запрос на вход вместе с сертификатом, подписанным сертифицирующей организацией, и ее открытым ключом. Если все в порядке, AS в ответном сообщении посылает сеансовый ключ, который Алиса может использовать для связи с сервером атрибутов привилегий (о котором мы скажем чуть ниже), шиф-

руя его открытым ключом Алисы. Кроме того, ответ содержит талон для сервера атрибутов, так же как и в Kerberos. Вдобавок сервер AS возвращает сертификат, содержащий его собственный открытый ключ.

Основное назначение *сервера атрибутов привилегий (Privilege Attribute Server, PAS)* — выдавать сертификаты атрибутов со списками прав доступа клиента к приложениям. Возвращаясь к Алисе, это означает, что она должна передать серверу PAS полученный при аутентификации талон.

Алиса также посылает PAS запрос на определенные права доступа к одному или более приложениям, с которыми она собирается работать. Этот запрос сопровождается так называемой *криптографической пломбой (cryptographic seal)*, которая представляет собой хэш, рассчитанный, исходя из запрашиваемого списка прав [168]. Пломба позволяет PAS проверить, не подправил ли кто-то этот список. Сам список не шифруется. После получения запроса PAS проверяет, можно ли предоставить Алисе запрашиваемые права, и если это так, собирает их в *сертификате атрибутов привилегий (Privilege Attribute Certificate, PAC)*. Более детально мы поговорим о PAC чуть ниже. Кроме возвращения сертификата PAC (который также запечатан), сервер атрибутов также создает новый сеансовый ключ, который Алиса может использовать для связи с KDS. Эта информация сохраняется в еще одном талоне.

Отметим, что если никакого общего ключа для связи между клиентом и сервером в дальнейшем не требуется, Алиса может не связываться с *сервером распространения ключей (Key Distribution Server, KDS)*. Однако при работе с приложениями, использующими симметричные ключи, Алиса может задействовать свой талон KDS для того, чтобы потребовать от KDS создать сеансовые ключи для организации защищенной связи с серверными приложениями. В SESAME сервер KDS может быть реализован с помощью сервера TGS системы Kerberos.

Компоненты клиента

В системе SESAME имеется три клиентских компонента защиты. Во-первых, для конечного пользователя SESAME предоставляет программу для входа в систему, которая называется *спонсор пользователя (user sponsor)*. Эта программа позволяет пользователю входить в систему, выходить из нее и изменять роли. Она выполняется как отдельное приложение и предполагает, что пользователь уже работает в локальной операционной системе. При входе пользователь должен ввести пароль, по которому будет создан общий секретный ключ, используемый для связи с сервером защиты домена. Эта процедура входа соответствует аналогичной процедуре системы Kerberos. С другой стороны, пользователь может инициировать строгий вход, в этом случае будет взят и использован, как описано выше, локально хранящийся закрытый ключ пользователя.

В зависимости от той роли, с которой пользователь подсоединился к системе, он получает определенные стандартные привилегии, позднее, после аутентификации, передаваемые в форму PAC. Также соответствующие привилегии можно запросить во время соединения.

Клиент доступа к аутентификации и привилегиям (The Authentication and Privilege Access Client, APA Client) — это не что иное, как библиотека, содержащая

простые процедуры, которые предоставляют пользователю интерфейс, а клиентским приложениям — доступ к серверу защиты домена. Отметим, что эта библиотека при необходимости поддерживает также аутентификацию клиентских приложений.

Третий компонент — это система *управления контекстом защиты* (*Secure Association Context Management, SACM*). Этот компонент отвечает за инициализацию и поддержку информации, необходимой для клиента при связи с выполняющимся на сервере приложением. Так, например, SACM, как и PAC, может локально сохранять сеансовые ключи, полученные при входе пользователя в систему и пр. Таким образом, SACM позволяет клиентским приложениям совместно использовать любую информацию по защите, необходимую для связи с серверными приложениями.

Функциональность SACM традиционно поддерживается многими системами защиты, в том числе и SESAME, и может быть реализована в разных системах по-разному. Поэтому вместо того, чтобы поддерживать специальный интерфейс, SACM реализует стандартный *обобщенный интерфейс программ службы защиты* (*Generic Security Service Application Program Interface, GSS-API*), определенный в [267].

Компоненты сервера

В последнюю очередь среди групп компонентов мы рассмотрим компоненты, работающие на стороне сервера. Подобно компонентам клиента, серверные приложения также имеют свою систему SACM, с помощью которой они хранят информацию по защите, необходимую для связи с клиентами и другими приложениями.

SACM сервера отвечает за передачу в *систему проверки PAC* (*PAC Validation Facility, PVF*) входящей информации, связанной с защитой. PVF извлекает из входящих запросов всю необходимую информацию и проверяет ее. Так, например, она может расшифровать сообщение и извлечь из него сеансовый ключ или после проверки подписи сертификата PAC извлечь из него права доступа. Или, например, после того как система PVF идентифицирует пользователя, она может передать информацию обратно в систему SACM, которая, в свою очередь, передаст клиенту сообщение для аутентификации сервера.

Система PVF может быть размещена на одном сервере с другими приложениями или на отдельной доверенной машине, чтобы работать в интересах множества приложений. В последнем случае необходимо, чтобы связь между сервером PVF и приложениями была защищена с обязательной аутентификацией сторон и поддержанием конфиденциальности.

8.6.2. Сертификаты атрибутов привилегий

SESAME осуществляет не только аутентификацию клиентов и серверов, но и обеспечивает авторизацию. Для этих целей используются сертификаты атрибутов, которые, как уже говорилось ранее, выдаются сервером PAC. Чтобы получить

представление о том, как выглядят сертификаты атрибутов, рассмотрим PAC системы SESAME детальнее.

Каждый сертификат PAC содержит несколько полей (табл. 8.2). Первые два поля идентифицируют место, откуда передан сертификат. Имя сервера PAC может определяться именем домена. Далее в PAC включен уникальный серийный номер, который может быть использован для проверки.

Таблица 8.2. Структура сертификата атрибутов привилегий системы SESAME

Поле	Описание
Домен вызова (Issuer domain)	Название защищенного домена, из которого передан вызов
Идентификатор вызова (Issuer identity)	Название PAC домена, из которого передан вызов
Серийный номер (Serial number)	Уникальный номер данного сертификата PAC, созданный сервером PAC
Время создания (Creation time)	Время создания PAC по UTC
Срок годности (Validity)	Интервал времени, в течение которого данный сертификат PAC считается действительным
Периоды времени (Time periods)	Дополнительные периоды времени, вне которых сертификат PAC считается устаревшим
Идентификатор алгоритма (Algorithm ID)	Идентификатор алгоритма, используемого для подписи этого сертификата PAC
Значение подписи (Signature value)	Подпись сертификата PAC
Привилегии (Privileges)	Список пар (атрибут, значение), описывающих привилегии
Информация о сертификате (Certificate information)	Дополнительная информация, используемая системой PVF
Прочее (Miscellaneous)	В настоящее время используется исключительно для аудита
Методы защиты	Поля для контроля использования сертификата PAC

Следующие три поля относятся к сроку жизни сертификата. Значение времени создания PAC по UTC хранится в отдельном поле. Это время, определяемое сервером PAC, создавшим PAC. Кроме того, сохраняется время начала и конца периода годности PAC. Возможно, PAC можно использовать в течение нескольких временных отрезков. Это поле PAC может не использоваться. Временные отрезки определяются в других полях.

Существует несколько способов подписать PAC. Конкретный метод определяется специфическим для SESAME идентификатором алгоритма. Сама подпись сохраняется в другом поле. Вместе с идентификатором алгоритма годность PAC может быть впоследствии проверена получателем.

Поле привилегий содержит список пар (атрибут, значение), определяющих права доступа, полученные владельцем PAC. Каждая привилегия ассоциирована с набором полномочий. По определению эти полномочия относятся к домену защиты, который вызвал PAC, в частности домен, в котором расположен сервер

PAS, создавший этот сертификат PAC. Однако можно также ассоциировать с привилегией другой защищенный домен.

При необходимости можно предоставить дополнительную информацию, которая поможет принимающей системе PVF подтвердить подпись PAC. В частности, в поле информации о сертификате может храниться сертификат, содержащий открытый ключ PAS, которым был подписан сертификат PAC. И наконец, имеется поле для дополнительной информации, используемое в настоящее время только с целью аудита.

Сертификат атрибута может быть, а может и не быть делегируемым. В любом случае можно определить, для какой цели (делегирование или использование на месте) пригоден данный сертификат PAC. Эта информация хранится в отдельном поле методов защиты PAC.

8.7. Пример — электронные платежные системы

Ранее мы в основном рассматривали вопросы защиты традиционных распределенных систем — клиент и сервер создавали защищенный канал связи, после чего клиент обращался к службам сервера. В основном вся защита ограничивалась двухсторонним взаимодействием.

Однако в реальном мире использование распределенных систем постепенно перестало сводиться к взаимодействию клиента с сервером. Мы видим, в частности, что по мере распространения Интернета распределенные системы находят применение в различных типах приложений, связанных с электронной коммерцией. Другими словами, люди покупают или продают через Интернет. В этом разделе мы кратко рассмотрим важнейший аспект электронной коммерции, а именно, вопрос о том, как происходит оплата товаров. Мы покажем, что электронные платежные системы основаны на методиках, разработанных для распределенных систем, особенно в части их надежности и защиты. Описания электронных платежных систем можно найти в [19, 350].

8.7.1. Электронные платежные системы

Оплата всегда предполагает наличие двух участников, продавца и покупателя. В большинстве случаев в процесс вовлекаются также один или два банка. Так или иначе, покупатель должен быть уверен, что продавец получит свои деньги. Это следует из того, что мы не можем рассчитывать на доставку товаров, за которые не заплачено.

Существует несколько способов организации платежных систем. Наиболее часто в реальной жизни используются наличные, чеки и кредитные карты. Соответствующие три структуры представлены на рис. 8.40. В системе на базе наличных денег покупатель должны получить в своем банке наличные деньги, которые затем передаются продавцу в обмен на товары (рис. 8.40, *а*). Продавцы впоследст-

вии могут использовать эти деньги, чтобы заплатить кому-нибудь еще или положить их на счет в своем собственном банке.

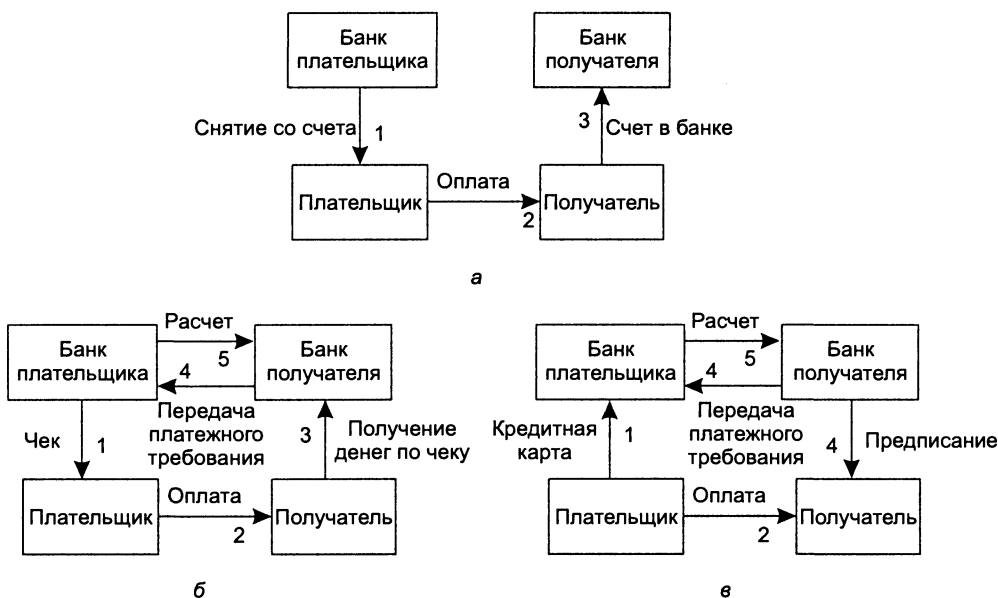


Рис. 8.40. Платежные системы на основе прямых платежей между покупателем и продавцом. Оплата наличными (а). Использование чека (б). Использование кредитных карт (в)

Другой подход применяется в системе на основе чеков, показанной на рис. 8.40, б. В этом случае покупатели имеют подписанные документы своего банка в виде чеков, дающие право каждому, представившему эти чеки в указанный банк, на перевод денег со счета покупателя. На практике банки покупателя и продавца после предъявления продавцом чека в свой банк сами производят взаимные расчеты.

Третий вариант, очень похожий на метод с использованием чека, основан на кредитных картах (рис. 8.40, в). Для оплаты в системе с кредитными картами покупатель требует от банка перевода денег на счет продавца, предоставляя продавцу кредитную карту и удостоверяя считывание кредитной карты. Реальное перечисление денег производится, когда продавец представляет результат считывания в свой собственный банк, который связывается с банком покупателя.

Существуют также и другие формы оплаты. Так, например, часто можно заплатить за товар, перечислив деньги из банка покупателя в банк продавца авансом. Получив информацию о перечислении денег на свой счет, продавец осуществляет доставку товаров (рис. 8.41, а). Эта схема платежа обычно используется в том случае, если компания-продавец не знает покупателя и не хочет поставлять дорогой продукт до получения денег.

При использовании авизо, как показано на рис. 8.41, б, покупатель разрешает продавцу предписать банку покупателя списывать определенные суммы с его сче-

та. Такой подход, например, часто используется при необходимости регулярных платежей — членских взносов или ежемесячных платежей за газ и электричество.

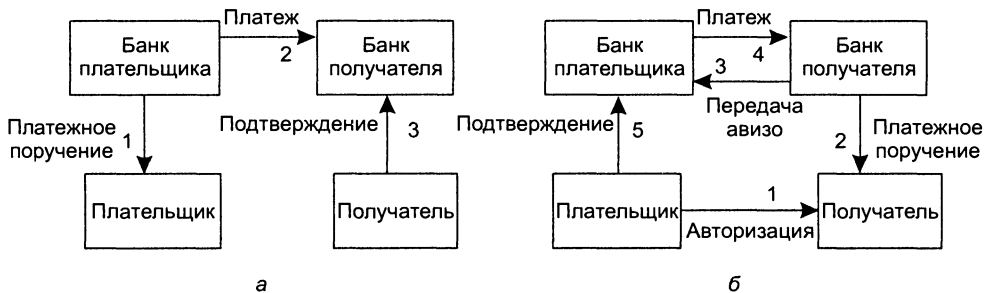


Рис. 8.41. Платежные системы на базе переводов денег из банка в банк. Платеж по платежному поручению (а). Платеж по авизо (б)

Эти модели оплаты используются уже длительное время и большинство из нас хорошо знакомы как минимум с некоторыми из них. В традиционном случае (см. рис. 8.40) модели требуют для произведения оплаты определенного физического взаимодействия покупателя и продавца. Если подобный контакт невозможен или неудобен, могут использоваться другие модели (см. рис. 8.41). Однако по мере того, как люди начали все активнее общаться друг с другом через такие системы, как Интернет, использовать традиционные модели оплаты стало затруднительно или вообще невозможно, поскольку покупателям и продавцам больше не нужно встречаться.

Таким образом, мы начали испытывать необходимость в специальных решениях для систем электронных платежей, которые соответствуют моделям оплаты наличными, чеками или кредитными картами, но не требуют физического взаимодействия покупателя и продавца. Ниже мы рассмотрим два примера подобных систем, обсудив сперва требования по защите к электронным платежным системам. Детальную информацию по защите электронных платежных систем можно найти в [415].

8.7.2. Защита в электронных платежных системах

Должно быть понятно, что защита играет в электронных платежных системах важную роль. Никому не понравится неавторизованный доступ к его банковскому счету, и никто не обрадуется продавцу, обвиняющему вас в неуплате. Это позволяет нам утверждать, что защита в платежных системах играет значительно более важную роль, чем в других типах распределенных систем.

Общие требования

Давайте рассмотрим для начала некоторые из наиболее очевидных требований к защите моделей платежей, представленных на рис. 8.40. В системах на основе

наличных денег наиболее широко распространены *автоматические кассовые машины* (*Automatic Teller Machines, ATM*), или банкоматы, которые работают в контакте с одной из банковских баз данных. Когда Алиса хочет получить деньги в своем банке, она должна сначала аутентифицировать себя. Она может использовать специальный маркер в форме карты с магнитной полосой или чипом. Эта карта, считываемая банкоматом, защищена *персональным идентификационным номером* (*Personal Identification Number, PIN*). Алиса должна набрать свой номер PIN, после чего АТМ связывается с банком. Если аутентификация оказалась успешной, Алисе разрешается получить определенную сумму денег. Дополнительную информацию по банкоматам и PIN можно найти в [119].

Существует и другой подход. Алиса может, не прибегая к помощи банкомата, использовать для перевода денег со своего счета домашний персональный компьютер. В любом случае, если у Алисы есть деньги, она может заплатить тем или иным способом. Один из способов, применяемых в электронных системах, — использование смарт-карты, которая может хранить *цифровые деньги* (*digital money*), существующие в форме битов. С другой стороны, Алиса может указать, что предпочитает хранить свои цифровые деньги локально, на собственном жестком диске.

Использование цифровых денег в схемах, разработанных для наличных, требует надежной защиты от мошенничества. В частности, нужно предотвратить возможность использования цифровых денег более одного раза или возможность изготовления фальшивых денег. Другими словами, мы явно нуждаемся в механизме обеспечения целостности.

Когда Алиса использует свои цифровые деньги, чтобы заплатить Бобу, Боб должен быть в состоянии проверить подлинность денег. Цифровые деньги считаются подлинными, если Боб может пойти в свой банк и положить их на счет. А банк Боба хочет быть уверен, что эти деньги «реальны» и ни Боб, ни Алиса ничего с ними «не накрутили». В некоторых случаях это означает, что банк Боба должен будет связаться с банком, который изначально выпустил эти деньги. Затем последует традиционная процедура аутентификации, в ходе которой банк Боба будет аутентифицировать банк Алисы.

Теперь обсудим систему, основанную на использовании чеков или кредитных карт. Снова нетрудно увидеть, что необходимым условием прохождения платежа являются стандартные требования аутентификации и авторизации чека или кредитной карты, которые Алиса выдала или предъявила. Когда Боб принимает оплату электронным чеком, он хочет, чтобы чек был защищен от подделок. Его банк также хочет быть уверен, что в чек не вносились исправления.

Однако можно указать на необходимость более строгих требований к защите от отказа в оплате, чем в случае систем на основе наличных денег. Так, например, стандартный способ покупки товаров через Интернет — посылка заказа с указанием номера кредитной карты. Хотя номер кредитной карты часто шифруется до пересылки его продавцу, он не является эквивалентом подписи на чеке или данных считывания в случае платежа кредитной картой. Таким образом, относительно легко (теоретически) отрицать сам факт заказа. Защита от отказа в оплате может быть реализована, например, путем требования к покупателю передавать в своей части транзакции цифровую подпись.

В связи с возможностью отказа от оплаты мы особенно жестко требуем, чтобы транзакция происходила атомарно — это гарантирует, что она либо будет проведена полностью, либо не произойдет вообще. Это требование вытекает из того факта, что после того, как Алиса заплатит магазину Боба, следует защищать не только Боба от отказа Алисы признать факт отправки заказа. Алиса также должна быть защищена от отказа Боба признать факт оплаты. Если система подтвердит Алисе, что оплата была произведена, это должно означать, что Боб получил деньги и не сможет впоследствии отказаться от этого факта.

Хотя мы сосредоточили свое внимание на вопросах защиты, следует понимать, что электронные платежные системы предполагают также и требование надежности доступа. В частности, система в целом должна быть легкодоступной и иметь высокую степень надежности.

Закрытость

Хотя запросы к требованиям защиты в электронных платежных системах выше, чем в традиционных распределенных системах, они в принципе могут быть реализованы с использованием описанных выше технологий. Однако существует одна проблема, которая заслуживает особого внимания и нередко выделяет электронные платежи из общего ряда приложений — это закрытость.

Просто защитить платеж от любопытных глаз относительно просто — достаточно воспользоваться стандартными методами шифрования. Трудно обеспечить анонимность. В стандартных системах платежей анонимный платеж несложен: Алиса просто передает Бобу мешок денег в обмен на купленные ей товары. Никаких вопросов. Алисе обычно нет необходимости знать, кто такой Боб, а Боба абсолютно не интересует, кто такая Алиса. Единственное, что имеет значение — товар и деньги.

Дело усложняется, если Алиса и Боб используют электронную платежную систему. Одна из первых вещей, которую следует рассмотреть, — это защита микроданных. *Микроданные (microdata)* — это малые элементы данных, которыми сопровождается каждая транзакция. Если собрать их вместе, они способны открыть одну из сторон, вовлеченных в эту транзакцию. Рассмотрим, например, анонимный платеж, в ходе которого записывается имя хоста, инициировавшего этот платеж, и время оплаты. Сравнение этой информации с записями о входах в систему позволит сразу же определить личность покупателя. Другая информация, доступ к которой обычно можно получить таким образом, — это определенные значения атрибутов, которые постепенно приводят нас к получению полного профиля интересов этого человека.

На практике защита системы от нарушения закрытости состоит в том, чтобы сделать покупателя анонимным. Делать анонимным продавца обычно нет необходимости. Анонимность в данном случае означает невозможность аутентификации покупателя в ходе или по окончании транзакции. Иногда может быть необходимо опознать личность, например, в случае дискуссии. Такая ситуация называется *условной анонимностью (conditional anonymity)*.

Имеется и другой подход, состоящий в сокрытии личности покупателя под псевдонимом. Псевдоним — это альтернативное имя, используемое в ходе тран-

закции или серии транзакций. Псевдоним обладает специфическим свойством: пользователя фактически можно идентифицировать по псевдониmu, но узнать истинные данные пользователя по псевдониmu покупателя в принципе невозможно. Обычно псевдонимы используются для реализации условной анонимности.

Чтобы сообразить, как скрыть от платежной системы лишние данные, рассмотрим сначала систему на основе традиционных методов платежа [81]. Каждый столбец на рис. 8.42 содержит атрибут транзакции, который следовало бы скрыть. Мы выделили пять атрибутов: личность продавца, личность покупателя, дата транзакции, уплаченная сумма и информация об оплачиваемом предмете. В каждой строке указана одна из сторон, вовлеченных в транзакцию. Мы выделяем три стороны: продавец, покупатель и банк. Кроме того, представим себе идеально расположившегося наблюдателя (известного как Чак), который просто отслеживает происходящие транзакции. Так, например, в случае покупки товаров в магазине Чак может быть следующим человеком в очереди.

Информация					
Сторона	Продавец	Покупатель	Дата	Сумма	Товар
	Продавец	Полностью	Частично	Полностью	Полностью
	Покупатель	Полностью	Полностью	Полностью	Полностью
	Банк	Нет	Нет	Нет	Нет
	Наблюдатель	Полностью	Частично	Полностью	Полностью

Рис. 8.42. Соккрытие информации в традиционных платежных системах

Для каждого участника транзакции определяется элемент, показывающий, насколько хорошо он будет знать конкретный атрибут. Так, например, покупателю в платежной системе на основе наличных видны продавец, дата транзакции и сумма, а также купленный товар. Однако продавец, как обычно в случае покупки товара в магазине, не слишком нуждается в знании личности покупателя.

Интересно отметить, что банк на самом деле не знает ничего, хотя у него и могут возникнуть некоторые соображения о сумме платежа в результате простого наблюдения за тем, сколько денег продавец в результате транзакции положит на счет. Другими словами, саму транзакцию записать невозможно, а это делает покупателя полностью анонимным. Продавцу в случае особо крупных транзакций анонимным быть ни к чему. Большинство банков по закону обязаны делать записи о транзакциях, в ходе которых продавец кладет деньги в банк.

Теперь рассмотрим традиционные транзакции с кредитными картами? Что касается сокрытия информации, несложно заметить, что абсолютно все атрибуты, перечисленные на рис. 8.42, будут видны всем участникам. Однако наблюдатель испытает некоторые затруднения при аутентификации покупателя, и вдобавок банк никогда не получит информации о том, что именно приобретено покупателем. Это дает нам таблицу, представленную на рис. 8.43 [81].

	Информация				
	Продавец	Покупатель	Дата	Сумма	Товар
Сторона	Продавец	Полностью	Полностью	Полностью	Полностью
	Покупатель	Полностью	Полностью	Полностью	Полностью
	Банк	Полностью	Полностью	Полностью	Нет
	Наблюдатель	Полностью	Частично	Полностью	Полностью

Рис. 8.43. Соккрытие информации в традиционных системах, использующих кредитные карты

Понятно, что оплата кредитной картой не обеспечивает особой закрытости, оставляя нам лишь анонимность. Электронные версии платежных систем должны предоставлять как минимум такой же, а по возможности и больший уровень защиты. Например, обычно недопустимо, чтобы номер кредитной карты пересылался по открытым сетям в виде простого текста.

8.7.3. Примеры протоколов

Чтобы лучше разобраться в вопросах защиты электронных платежных систем, давайте кратко обсудим две характерные системы. Сначала мы поговорим о системе, позволяющей проводить платежи цифровыми деньгами. Затем рассмотрим стандартизованный протокол для безопасных транзакций с кредитными картами через Интернет.

E-cash

Существует несколько электронных платежных систем, которые базируются на концепции цифровых денег. Одной из наиболее известных является система e-cash, описание которой можно найти в [94, 95]. Основной целью при разработке этой системы было введение анонимности в платежи. Чтобы понять, как работает система e-cash, рассмотрим сначала не анонимную систему на основе наличных денег.

Допустим, Алиса хочет купить некоторые товары у Боба через сеть на общую сумму 12 долларов, используя цифровые деньги. Сначала она связывается со своим банком и просит снять 12 долларов с ее счета. Банк создает цифровые деньги в форме подписанных виртуальных банкнот на определенную сумму, причем каждая банкнота имеет уникальную ассоциированную с ней подпись. Так, например, виртуальная банкнота в 10 долларов может быть подписана специальным, принадлежащим банку закрытым ключом K_{10}^- , а однодолларовая банкнота — другим специальным закрытым ключом K_1^- . Для предотвращения копирования виртуальных банкнот каждая из них имеет уникальный серийный номер, с помощью которого банк может проверить факт повторного использования банкноты.

В нашем примере Алиса получает в своем банке десятидолларовую банкноту и две однодолларовых, которые позднее передает Бобу. По подписи банка Боб может проверить, что банкноты хотя и виртуальные, но не поддельные. Однако для проверки на попытку повторного использования банкнот ему необходимо связаться с банком, который по серийным номерам проверит, не применялись ли эти банкноты для оплаты раньше. Если это не попытка повторного использования виртуальных банкнот, Боб отправляет сообщение ОК как знак того, что готов принять эти деньги. Отметим, что Бобу все равно нужно связаться с банком, поскольку банк должен также проверить целостность банкнот.

Основной минус этой схемы состоит в том, что банк вынужден отслеживать серийные номера всех цифровых банкнот, то есть точно контролировать все денежные потоки. Если Алиса запросит десятидолларовую банкноту, банк должен записать серийный номер банкноты, выданной Алисе, и позже отследить, как Боб использует ее для оплаты. Не существует другого способа понять, что Алиса платит Бобу, если не будет некоей третьей стороны, которая примет платеж от Алисы, поверив ей на слово, что эти деньги еще ни разу не использовались.

Эту систему необходимо дополнить, чтобы не нужно было отслеживать, какие цифровые банкноты кому были переданы, сохраняя при этом защищенность банкнот от попыток подделки или повторного использования. Как показано в [93], такая защита может быть осуществлена при помощи так называемой *слепой подписи* (*blind signature*). Принцип, лежащий в основе слепой подписи, проще всего объяснить на примере конверта, покрытого изнутри копировальной бумагой. [94]. Если положить внутрь конверта обычную бумагу, все, написанное на конверте, отпечатается и на бумаге внутри него.

Для Алисы, получающей свою десятидолларовую банкноту, подписанную ее банком, это выглядит так, будто она помещает банкноту в конверт с копировальной бумагой и просит свой банк поставить подпись на конверте. Банк снимает 10 долларов со счета Алисы и ставит свою подпись на конверте. После этого банк никогда не увидит саму банкноту, но в любой момент может прочесть свою десятидолларовую подпись, стоящую на конверте. После этого Алиса может использовать банкноту как обычные деньги. Эта схема отлично работает, если учесть, что банк ежедневно подписывает множество десятидолларовых виртуальных банкнот. В этом случае, когда Боб передаст в банк десятидолларовую банкноту и потребует перечислить 10 долларов на свой счет, отследить Алису по этой банкноте будет невозможно.

Чтобы понять, как работают эти подписи в алгоритмах RSA, используем ту же нотацию, что и ранее при описании принципов RSA: $n = p \times q$, где p и q — два больших простых (и секретных) числа. Число d вычисляется на основе этих простых чисел как $d = (p - 1) \times (q - 1)$, а e получается следующим образом: $e \times d = 1 \bmod z$.

Предположим, Алиса хочет получить со своего банковского счета десятидолларовую банкноту. Фактически она сама создает себе эту банкноту, но нуждается в банковской подписи для ее подтверждения, причем так, чтобы банк впоследствии был не в состоянии проследить ее движение. Она создает случайное число k между 1 и n и делает m нечитаемым, скрывая его в сообщении

$$t = mk^e \bmod n.$$

Банк подписывает сообщение t своим защищенным ключом K_{10}^- (в нашем примере — d), превращая его в сообщение t^d . Отметим, что мы используем ключи, специально предназначенные для десятидолларовых банкнот. Для изготовления банкнот другого достоинства требуются другие ключи. Когда Алиса получит сообщение t^d , она раскроет его, делая следующие вычисления:

$$s = t^d / k \bmod n = (mk^e)^d / k \bmod n = m^d k^{ed-1} \bmod n = m^d \bmod n.$$

С этого момента она имеет в своем распоряжении десятидолларовую банкноту, которая подписана банком так, что банк никогда ее не видел. Теперь Алиса может использовать m^d в качестве десятидолларовой банкноты для того, чтобы заплатить Бобу, как это показано на рис. 8.44.

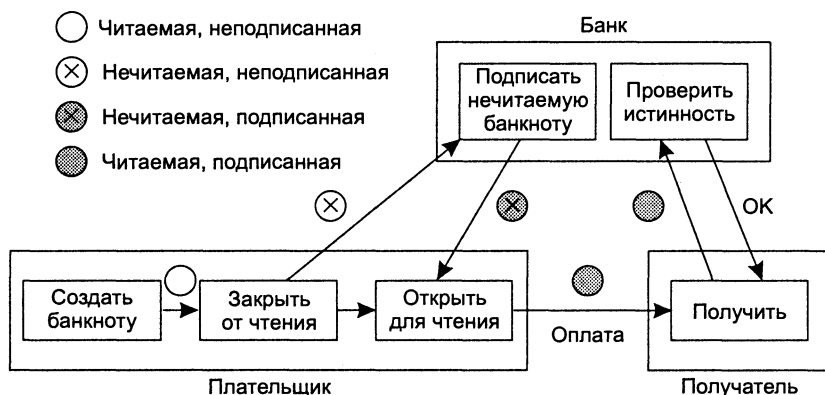


Рис. 8.44. Функционирование анонимных электронных денег со слепой подписью

Хотя этот протокол в общем хорош, в нем все же имеются некоторые проблемы. Основная состоит в том, чтобы защититься от повторного использования денег. Хотя банк может легко проверить, не применялась ли банкнота m в качестве средства платежа ранее, нет сомнения, что было бы полезно наказывать людей, пытающихся делать это снова. Другими словами, хорошо было бы разработать такую схему, при которой попытка повторного использования предоставляла бы информацию, позволяющую идентифицировать человека, пытающегося это сделать. Детали подобной схемы можно найти в [94, 404].

Защищенные электронные транзакции

Рассмотрим теперь электронную платежную систему для транзакций с кредитными картами. Протокол *защищенных электронных транзакций* (*Secure Electronic Transactions, SET*) — это совместная попытка Visa и Mastercard в кооперации с несколькими другими организациями, такими как Netscape и Microsoft, разработать стандартный способ покупки товаров через Сеть с использованием кредитных карт. SET — открытый стандарт. Это означает, что вся информация о протоколе опубликована (информацию по SET можно найти по адресу <http://www.setco.org>).

Далее мы представим схему реального протокола, опуская при этом множество второстепенных деталей, таких как конкретный формат сообщений, включе-

ние изменений, отметки времени и пр. Основная задача этого описания состоит в том, чтобы дать представление о SET как многостороннем протоколе для электронных платежей при помощи кредитных карт.

В SET использована важная новая концепция двойной подписи. Прежде чем мы перейдем к обсуждению различных этапов протокола, нам следует в ней разобраться. Рассмотрим следующую проблему: Алиса хочет купить у Боба некоторые товары, пользуясь кредитной картой. Все, что она должна сделать при этом, — отправить Бобу информацию о заказанных товарах вместе с данными о своей кредитной карте. Боб должен быть в состоянии передать эти данные в банк для завершения процесса оплаты за товары. Нам необходимо связать заказ и платежную информацию.

Один из способов — поместить всю необходимую информацию в одно сообщение и потребовать от Алисы, чтобы она подписала его. Однако в этом случае и Боб, и банк для проверки подписи Алисы будут вынуждены получать как заказ, так и платежную информацию. Соответственно, банк узнает, что заказывала Алиса. Обычно эти данные должны быть известны только Алисе и Бобу. Точно так же Алиса не хочет, чтобы Бобу стали известны детали информации о произведенной оплате. Данные о кредитоспособности Алисы — дело банка, а не Боба.

Решение будет следующим. Пусть m_1 и m_2 обозначают соответственно заказ и платежную информацию. *Двойная подпись (dual signature)* состоит в построении дайджеста сообщения $md_1 = H(m_1)$ для m_1 и $md_2 = H(m_2)$ для m_2 с последующим построением третьего дайджеста путем слияния md_1 и md_2 : $md_{dual} = H(\text{concat}(md_1, md_2))$. Этот дайджест md_{dual} затем подписывается Алисой с использованием ее закрытого ключа. Теперь Алиса посылает Бобу сообщение $[m_1, md_{dual}, m_2, K_A^-(md_{dual})]$, которое позволяет ему идентифицировать Алису и убедиться, что именно она поставила подпись под всем сообщением. Однако Боб не в состоянии прочесть m_2 . Для обозначения двойной подписи сообщений m_1 и m_2 , из которых открыто только m_1 , мы используем запись $[m_1|m_2]_A$:

$$[m_1|m_2]_A = [m_1, H(m_2), H(\text{concat}(m_1, m_2)), K_A^-(H(\text{concat}(m_1, m_2)))].$$

Давайте обсудим теперь основы протокола SET. Этапы протокола иллюстрирует рис. 8.45.

Мы предполагаем, что Алиса хочет заплатить Бобу. Первое, что ей необходимо сделать, — послать Бобу заказ и платежную информацию. На рисунке это действие показано в виде сообщения 1. Информация о заказе обозначена как *order*, информация о платеже — как *pay_info*. Алиса создает дважды подписанное сообщение $[\text{order}|\text{pay_info}]_A$. Платежная информация, которая должна быть прочитана только в банке, шифруется сеансовым ключом $K_{A,bank}$, который банк создал для Алисы. Этот ключ также посылается Бобу, но зашифрованный открытым ключом банка K_{bank}^+ .

После того как Боб проверит сообщение Алисы, он пересылает платежную информацию в банк и запрашивает авторизацию оплаты. При этом он создает сеансовый ключ $K_{B1,bank}$, который использует для шифрования своего запроса *auth*. Этот запрос пересылается в банк вместе с информацией Алисы об оплате, а ключ $K_{B1,bank}$ шифруется открытым ключом банка. Все это показано на рисунке как сообщение 2.

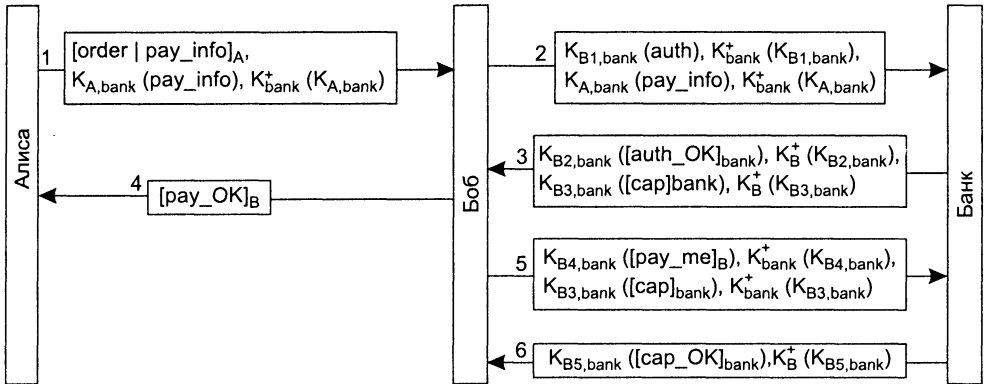


Рис. 8.45. Этапы протокола SET

Допустим, что банк признал действительность оплаты. В этом случае он возвращает Бобу подписанное сообщение о прохождении авторизации *auth_OK*, зашифрованное новообразованным сеансовым ключом $K_{B2,bank}$, который, в свою очередь, зашифрован открытым ключом Боба. Этот ответ, обозначенный как сообщение 3, содержит также и так называемое *сообщение о получении (capture message)*, которое потребуется позже, когда Боб действительно будет получать деньги из банка. Сообщение о получении однозначно ассоциируется с конкретной транзакцией посредством идентификатора транзакции. Этот идентификатор также используется и в сообщениях *order* и *pay_info*. Сообщение о получении подписывается банком и шифруется при помощи третьего сеансового ключа, $K_{B3,bank}$.

Когда Боб получает от банка «добро», он посылает Алисе подписанное подтверждение, показанное на рисунке как сообщение 4.

Теперь Боб хочет получить свои деньги. Он посылает банку сообщение *pay_me* вместе с ранее полученным сообщением о получении. И вновь для шифрования сообщения *pay_me* создается и используется новый сеансовый ключ. Это сообщение 5.

И, наконец, после того как банк проверит сообщение о получении, он возвращает Бобу подтверждение *cap_OK*, зашифрованное сеансовым ключом $K_{B5,bank}$.

Хотя на рис. 8.45 это и не указано, в протоколе SET на каждом из шагов выполняется аутентификация путем отправки сертификатов всякий раз, когда возникает необходимость в использовании открытого ключа. С этой целью в протоколе задействуется иерархия сертифицирующих организаций, сходная с ранее обсуждавшейся иерархией моделей доверия в почте PEM. Детали этой модели, равно как и протокола, можно найти в [407].

8.8. Итоги

Защита играет в распределенных системах чрезвычайно важную роль. Распределенные системы должны предоставлять пользователям и разработчикам меха-

низмы, обеспечивающие реализацию разнообразных правил защиты. Разработка и правильное применение подобных механизмов обычно делают обеспечение защиты в распределенных системах сложной инженерной задачей.

Следует выделить три важных момента. Первый из них состоит в том, что распределенные системы должны иметь средства для организации защищенных каналов связи между процессами. Защищенный канал в принципе предоставляет средства взаимной аутентификации сторон и защищает сообщения во время пересылки от фальсификации. Защищенный канал обычно предоставляет также и средства поддержания конфиденциальности. Это означает, что никто, кроме связавшихся друг с другом сторон, не в состоянии читать передаваемые по каналу сообщения.

Одним из серьезных вопросов, который следует решить при проектировании, является выбор между исключительно симметричной криптосистемой (основанной на совместном использовании секретных ключей) или сочетанием ее с системой с открытым ключом. В текущей практике криптография с открытым ключом используется для рассылки общих секретных ключей. Последние известны также как сеансовые ключи.

Второй вопрос защиты распределенных систем — это контроль доступа, или авторизация. Авторизация касается защиты ресурсов, чтобы только процессы, имеющие соответствующие права доступа, могли получать доступ к соответствующим ресурсам и использовать их. После аутентификации процесса всегда производится контроль доступа.

Существует два способа реализации контроля доступа. Во-первых, каждый из ресурсов может поддерживать собственный список доступа, в котором перечисляются права доступа всех пользователей или процессов. Кроме того, процесс может иметь сертификат, точно устанавливающий его права на определенный набор ресурсов. Основное достоинство сертификатов состоит в том, что процесс может с легкостью передать свой талон другому процессу, делегировав свои права доступа. Однако сертификаты имеют и недостаток — их обычно не просто отзывать.

Особое внимание следует уделить вопросам управления доступом в случае мобильного кода. Помимо необходимости защиты мобильного кода от вредоносных хостов, обычно важнее защитить хосты от вредоносного мобильного кода. Для этого существуют различные способы, из которых наиболее часто применяется так называемое сито. Однако сито чрезмерно ограничивает возможности программ, поэтому для решения проблемы были разработаны более гибкие методы на основе реально защищенных доменов.

Третий ключевой вопрос защиты распределенных систем — это управление защитой. Здесь есть два важных аспекта — управление ключами и управление авторизацией. Управление ключами включает распространение криптографических ключей, в котором значительную роль играют сертификаты, выдаваемые доверенным третьим лицом. В деле управления авторизацией важны сертификаты атрибутов и делегирование.

Kerberos — это широко распространенная система защиты, основанная на шифровании общих секретных ключей. Ее основное назначение — аутентифика-

ция, хотя она также поддерживает протоколы управления доступом и делегирование прав доступа.

SESAME — это типичный пример системы защиты, которая встраивается в распределенные системы. Она основана на комбинированном использовании шифрования открытых ключей и общих секретных ключей. Она многое взяла от системы Kerberos и совместима с Kerberos.

И, наконец, важной областью использования распределенных систем являются электронные платежные системы. Интерес они представляют с точки зрения тех сторон, которые могут связываться через глобальные сети. Особое внимание часто уделяется обеспечению определенного уровня анонимности покупателя, который возможен в традиционных расчетных системах на базе наличных денег и в их электронных двойниках.

Вопросы и задания

1. Какие механизмы могут предоставляться в распределенных системах в качестве служб защиты разработчикам приложений, которые, как говорилось в главе 5, в проектировании систем доверяют только сквозным аргументам?
2. Можно ли в случае подхода RIISC сосредоточить все службы защиты на защищенных серверах?
3. Предположим, что вас попросили разработать распределенное приложение, которое должно помочь преподавателям принимать экзамены. Укажите как минимум три условия, которые должны стать частью правил защиты такого приложения.
4. Будет ли безопасно объединить сообщения 3 и 4 протокола аутентификации, приведенного на рис. 8.11, в $K_{A,B}(R_B, R_A)$?
5. Почему (см. рис. 8.14) для центра KDC очевидно, что он общался с Алисой, при получении запроса на передачу секретного ключа, который Алиса может использовать совместно с Бобом?
6. Почему вместо случайного числа *nonce* нельзя использовать отметку времени?
7. В сообщении 2 протокола аутентификации Нидхема—Шредера талон шифруется секретным ключом, который совместно используют Алиса и KDC. Является ли это шифрование необходимым?
8. Можем ли мы безопасно изменить протокол, приведенный на рис. 8.19, так, чтобы сообщение 3 содержало только запрос R_B ?
9. Придумайте простой алгоритм аутентификации с использованием подписей для криптосистемы с открытым ключом.
10. Пусть Алиса хочет передать Бобу сообщение. Вместо того чтобы шифровать m открытым ключом Боба K_B^+ , она создает сеансовый ключ $K_{A,B}$ и посылает $[K_{A,B}(m), K_B^+(K_{A,B})]$. Почему эта схема обычно оказывается лучше? (Подсказка: примите во внимание вопросы производительности.)
11. Как может быть реализована смена ролей в матрице контроля доступа?

12. Как реализованы списки контроля доступа в файловой системе UNIX?
13. Как организация может заставить использовать для доступа в Web прокси-шлюз и предотвратить прямой доступ своих пользователей к внешним web-серверам?
14. В какой степени (см. рис. 8.28) использование в качестве мандатов ссылок на объекты Java действительно зависит от языка Java?
15. Назовите три проблемы, с которыми сталкиваются разработчики интерфейсов, когда для защиты от неавторизованного доступа к локальным ресурсам со стороны мобильных программ они вынуждены, как описано в соответствующем разделе, вставлять инструкции включения и отключения привилегий.
16. Назовите несколько достоинств и недостатков использования централизованного сервера управления ключами.
17. Протокол обмена ключами Диффи—Хеллмана можно использовать также и при создании общего секретного ключа для трех сторон. Объясните, как.
18. В протоколе обмена ключами Диффи—Хеллмана отсутствует аутентификация. Из-за этого третья сторона, Чак, может легко вмешаться в обмен ключами между Алисой и Бобом, после чего проникнуть через механизмы защиты. Объясните, как.
19. Приведите простой способ отзыва мандатов в системе Атоева.
20. Имеет ли смысл ограничение срока жизни сеансового ключа? Если да, приведите пример.
21. Какова роль отметки времени в сообщении 6 на рис. 8.37? Почему она должна быть зашифрована?
22. Придайте законченный вид рис. 8.37, добавив связь для аутентификации Алисой и Бобом друг друга.
23. Рассмотрим связь между Алисой и Бобом и службу аутентификации AS, такую как в системе SESAME. Какая разница (если она есть) между сообщениями $m_1 = K_{AS}^+(K_{A,AS}(data))$ и $m_2 = K_{A,AS}(K_{AS}^+(data))$?
24. Определите как минимум два различных уровня атомарности транзакций в электронных платежных системах.
25. Покупатель в системе e-cash до использования денег, полученных в банке, обычно делает паузу случайной длительности. Почему?
26. Анонимность продавца в платежных системах нередко запрещена. Почему?
27. Рассмотрим электронную платежную систему, в которой покупатель посылает деньги продавцу (удаленному). Постройте таблицу (см. рис. 8.42 и 8.43), описывающую сокрытие информации.

Глава 9

Распределенные системы объектов

9.1. CORBA

9.2. DCOM

9.3. Globe

9.4. Сравнение систем CORBA, DCOM и Globe

9.5. Итоги

В этой главе мы переходим от обсуждения принципов построения распределенных систем к изучению различных парадигм, используемых при их построении. Первая парадигма — это распределенные объекты. В распределенных системах объектов понятие объекта играет ключевую роль для реализации прозрачности распределения. В принципе считать объектами можно все; клиенты получают службы и ресурсы в форме объектов, к которым они могут обращаться.

Распределенные объекты образуют важную парадигму, поскольку все аспекты распределения относительно просто скрыть за интерфейсом объектов. Кроме того, объекты могут присутствовать фактически повсюду, и это также усиливает значение этой парадигмы для построения систем. Распределенные объекты лежат в основе двух важных, широко распространенных распределенных систем, которые мы обсудим в этой главе.

В качестве первого примера мы рассмотрим CORBA, промышленный стандарт распределенных систем. В настоящее время используется множество систем CORBA, и их разработка и стандартизация развиваются по мере того, как растет число установленных систем.

Нашим вторым примером будет DCOM корпорации Microsoft. DCOM можно рассматривать как систему промежуточного уровня, реализованную поверх различных операционных систем Windows, начиная с Windows 95. Фактически все приложения Windows опираются на предоставляемую DCOM функциональность. Таким образом, DCOM можно считать наиболее широко распространенной распределенной системой промежуточного уровня.

Кроме этих двух коммерческих систем известно также множество исследовательских работ по построению различных распределенных систем объектов. В этой главе мы рассмотрим экспериментальную систему Globe, которая разрабатывается в настоящее время в рамках исследовательского проекта по глобальным распределенным сетям. Интересной особенностью Globe, которая отличает ее от систем типа CORBA и DCOM, является тот факт, что состояние распределенных объектов Globe может отдельно храниться и реплицироваться на нескольких машинах.

Важно так структурировать наше обсуждение, чтобы иметь возможность сравнивать различные системы. По этой причине каждая из систем рассматривается в отдельном разделе, который начинается с обзора наиболее важных концепций системы, а именно ее объектной модели и основных служб, которые она поддерживает. Затем идет обсуждение семи принципов реализации системы: связи, процессов, именования, синхронизации, непротиворечивости и репликации, отказоустойчивости, защиты.

Преимущество подобной структуры в том, что мы можем провести детальное сравнение различных систем, рассматриваемых в разных разделах. С другой стороны, поскольку основные аспекты этих принципов мы уже обсудили в первой части книги, нам осталось описать лишь некоторые дополнительные моменты, а скорее даже детализацию этих основ. Однако эти детали помогут нам понять, как на самом деле работает каждая из систем и каким образом сравнивать их между собой.

Существует множество книг, посвященных распределенным системам объектов. Несложный обзор CORBA и DCOM приводится в [333]. В [138] системы CORBA и DCOM вместе с Java RMI также используются в качестве примеров, причем основное внимание уделяется нескольким принципам, лежащим в основе распределенных объектов. Как благодаря технологии распределенных объектов можно гибко строить системное программное обеспечение, рассказывается в [209].

9.1. CORBA

Мы начнем изучение распределенных систем объектов с рассмотрения *обобщенной архитектуры брокера объектных запросов (Common Object Request Broker Architecture, CORBA)*. Как следует из названия, CORBA — это не столько распределенная система, сколько ее спецификация. Подобные спецификации разработаны *группой управления объектами (Object Management Group, OMG)*, некоммерческой организацией, в которую входят более 800 членов, в основном промышленных компаний. Основной целью OMG при разработке CORBA было создание распределенной системы, способной преодолеть большинство проблем межоперационной совместимости при интеграции сетевых приложений. Первые спецификации CORBA появились в начале 90-х годов. В настоящее время широко распространена реализация CORBA версии 2.4, начинают появляться первые образцы систем CORBA версии 3.

Подобно многим другим системам, родившимся в недрах комитетов, CORBA перенасыщена описаниями разнообразных свойств и средств. Базовая спецификация состоит из более чем 700 страниц, и еще 1200 страниц описывают различные службы, построенные на этой основе. Естественно, каждая реализация CORBA имеет собственные расширения, поскольку всегда в спецификацию не включается что-нибудь такое, что хочет иметь конкретный производитель. Пример CORBA вновь иллюстрирует тот факт, что создание распределенной системы — дело безумно сложное.

Мы не станем обсуждать все, что включено в CORBA, взамен сосредоточившись только на тех аспектах, которые имеют значение для распределенных систем и отличают CORBA от других распределенных систем объектов. Спецификации CORBA можно найти в [331], а также по адресу <http://www.omg.org>. Отличный обзор CORBA приведен в [475]. А в [359] предлагается описание даже более детальное, чем исходная спецификация. Информацию по созданию приложений на C++ с использованием CORBA можно найти в [25, 198].

9.1.1. Обзор

Глобальная архитектура CORBA восходит к модели OMG, описанной в [324]. Эта модель, приведенная на рис. 9.1, содержит четыре группы архитектурных элементов, связанных с так называемым *брокером объектных запросов* (*Object Request Broker, ORB*). ORB образует ядро любой распределенной системы CORBA; он отвечает за поддержание связи между объектами и их клиентами, скрывая проблемы, связанные с распределением и разнородностью системы. Во многих системах ORB реализуется в виде множества библиотек, которые komponуются с приложениями клиента и сервера, предоставляя им базовые службы связи. Позже, когда мы будем обсуждать объектную модель CORBA, мы еще вернемся к ORB.



Рис. 9.1. Глобальная архитектура CORBA

Кроме объектов, которые являются частями конкретных приложений, в модель также входят так называемые *средства CORBA* (*CORBA facilities*), которые представляют собой сочетания внутренних служб CORBA (которые мы обсудим позже) и делятся на две различные группы. *Горизонтальные средства* (*horizontal facilities*) представляют собой высокоуровневые службы общего назначения, которые не зависят от прикладной области использующих их программ. Подобные службы в настоящее время обслуживают пользовательский интерфейс, управле-

ние информацией, управление системой и управление задачами (которое требуется для определения систем рабочих потоков). *Вертикальные средства (vertical facilities)* — это высокоуровневые службы, предназначенные для конкретных предметных областей, таких как электронная коммерция, банковское дело, производство и т. д.

Мы не будем детально рассматривать прикладные объекты и средства CORBA, а сосредоточимся на базовых службах и ORB.

Объектная модель

В CORBA используется модель удаленных объектов, которую мы рассматривали в главе 2. В этой модели реализация объектов происходит в адресном пространстве сервера. Следует отметить, что спецификации CORBA никогда не указывали, что объекты должны быть реализованы исключительно как удаленные. Однако фактически все системы на базе CORBA поддерживают только эту модель. Кроме того, спецификации часто рекомендуют, чтобы распределенные объекты в CORBA были реализованы в форме удаленных объектов. Позднее, при обсуждении объектной модели Globe, мы покажем, каким образом CORBA может в принципе существовать, используя совершенно иную объектная модель.

Объекты и службы описываются при помощи *языка определения интерфейсов (Interface Definition Language, IDL)* CORBA. Язык IDL в CORBA похож на другие языки определения интерфейсов, имеет традиционный для этих языков синтаксис описания методов и их параметров. Описать семантику IDL для CORBA невозможно. Интерфейс — это набор методов, и объект сам определяет, какие интерфейсы он реализует.

Спецификации интерфейса можно задать только при помощи IDL. Как мы увидим в дальнейшем, в таких системах, как Distributed COM и Globe, интерфейсы определяются на более низких уровнях, в виде таблиц. Эти так называемые *бинарные интерфейсы (binary interfaces)* по самой своей природе не зависят от языков программирования. В CORBA, однако, необходимо полностью задать правила отображения спецификации на языке IDL на существующий язык программирования. В настоящее время такие правила существуют для множества языков, включая C, C++, Java, Smalltalk, Ada и COBOL.

Считая, что система CORBA организована в виде набора клиентов и серверов объектов, ее общую организацию можно понять из рис. 9.2.

Основой каждого процесса в CORBA на клиенте или на сервере является ORB. ORB лучше всего рассматривать как систему времени исполнения, которая отвечает за базовые функции связи между клиентом и объектом. Эти базовые функции связи гарантируют обращение к серверу объектов и возвращение клиенту ответов сервера.

С точки зрения процесса, брокер ORB сам по себе предоставляет лишь некоторые услуги. Одна из этих услуг — обработка ссылок на объекты. Вид этих ссылок обычно зависит от конкретного брокера ORB. Поэтому любой брокер ORB предоставляет операции для маршрутизации и демаршрутизации ссылок на объекты в ходе обмена между процессами, а также операции для сравнения ссылок. Ссылки на объекты мы обсудим чуть позже.

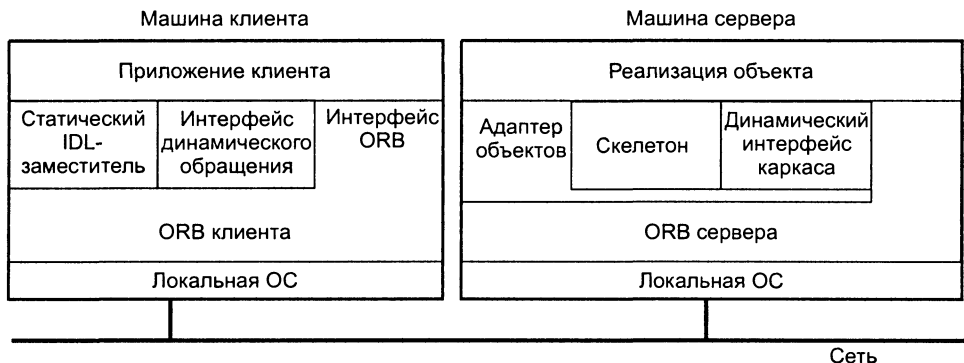


Рис. 9.2. Общая организация системы CORBA

Другие операции, предоставляемые брокером ORB, — начальный поиск доступных для процесса служб. Обычно он предоставляет данные для получения начальной ссылки на объект, реализующий конкретную службу CORBA. Так, например, для использования службы именования необходимо, чтобы процесс знал, как на нее сослаться. Подобную задачу инициализации равно необходимо решать и для других служб.

Кроме интерфейса ORB, клиенты и серверы вряд ли могут различить в ORB хоть что-нибудь еще. Обычно они видят только заглушки, обрабатывающие обращения к соответствующим объектам. Клиентское приложение имеет заместитель, предназначенный для реализации того же интерфейса, который использует объект. Как мы показали в главе 2, заместитель — это клиентская заглушка, единственное назначение которой — выполнить маршалинг (то есть упаковку) обращения в запрос и переслать этот запрос серверу. После получения ответа с сервера выполняется его демаршалинг (распаковка) и передача клиенту.

Отметим, что интерфейс между заместителем и ORB не должен быть стандартизован. Поскольку CORBA предполагает, что все интерфейсы определены на языке IDL, реализации CORBA предоставляют разработчикам компилятор IDL, который генерирует код, необходимый для обеспечения связи между клиентским и серверным брокерами ORB.

Однако бывают случаи, когда статически определенные интерфейсы просто недоступны клиенту. В этом случае во время исполнения необходимо выяснить, как выглядит интерфейс для конкретного объекта, чтобы иметь возможность последовательно реализовать запрос с обращением к этому объекту. Для этой цели CORBA предоставляет клиентам *интерфейс динамического обращения (Dynamic Invocation Interface, DII)*, который позволяет создавать запросы в ходе выполнения приложений. Так, в частности, DII предлагает базовую операцию *invoke*, которая получает в качестве параметров ссылку на объект, идентификатор метода и список входных значений, а возвращает в результате список выходных переменных, предоставленный вызывающей стороной.

Серверы объектов организованы в соответствии с тем описанием, которое мы давали в главе 3. Как видно из рис. 9.2, система CORBA предоставляет адаптер

объектов, который отвечает за передачу входящих запросов нужному объекту. Реальный демаршалинг (распаковка данных запроса) на стороне сервера выполняется заглушками, которые в CORBA называют скелетонами, однако возможно также, что реализация объекта берет демаршалинг на себя. Как и в случае клиента, серверные заглушки также могут компилироваться статически из описания на языке IDL или иметь вид базового динамического скелетона. При использовании динамического скелетона объект должен содержать правильную реализацию обращения к доступным клиенту функциям. Чуть ниже мы еще вернемся к серверам объектов.

Хранилища интерфейсов и реализаций

Чтобы обеспечить динамическое создание запросов с обращениями, необходимо, чтобы во время исполнения процесс мог определить, как выглядит интерфейс. CORBA предоставляет *хранилище интерфейсов (interface repository)*, в котором хранятся все определения интерфейсов. Во многих системах хранилище интерфейсов реализуется в виде отдельного процесса, имеющего стандартный интерфейс сохранения и выдачи определений интерфейсов. Хранилище интерфейсов также может рассматриваться как часть CORBA, содействующая работе механизмов проверки типов во время исполнения.

Когда компилируется определение интерфейса, компилятор IDL назначает *идентификатор хранения (repository identifier)* этого интерфейса. Этот идентификатор — основное средство извлечения определения интерфейса из хранилища. Идентификатор по умолчанию порождается из имени интерфейса и его методов, что подразумевает отсутствие каких-либо гарантий его уникальности. Если необходима уникальность, стандартный идентификатор можно изменить.

Учитывая, что все определения интерфейсов хранятся в хранилище интерфейсов в соответствии с их синтаксисом на языке IDL, становится возможным организовать каждое определение стандартным образом. (В терминологии баз данных это означает, что концептуальная схема, ассоциированная с хранилищем интерфейсов, одинакова для всех хранилищ.) В результате хранилища интерфейсов в системах CORBA имеют одинаковые операции навигации среди определенных интерфейсов.

Кроме хранилища интерфейсов система CORBA обычно предлагает также *хранилище реализаций (implementation repository)*. Концептуально хранилище реализаций содержит все необходимое для реализации и активизации объектов. Поскольку подобная функциональность изначально связана с самим брокером ORB и базовой операционной системой, создать стандартную реализацию было бы затруднительно.

Хранилище реализаций, кроме того, тесно связано с организацией и реализацией серверов объектов. Как мы объясняли в главе 3, задача адаптера объектов — активизировать объекты путем их запуска в адресном пространстве сервера таким образом, чтобы к их методам можно было обращаться. Получив ссылку на объект, адаптер объектов связывается с хранилищем реализаций, чтобы найти ту реализацию, которая требуется.

Так, например, хранилище реализаций может поддерживать таблицу, указывающую на возможность запуска нового сервера, и номер порта, который может

быть отведен новому серверу для прослушивания конкретным объектом. Хранилище, кроме того, может иметь информацию об исполняемом файле (то есть программе), который сервер должен загрузить в память и выполнять.

С другой стороны, необходимости в запуске отдельного сервера может и не возникнуть, достаточно будет скомпоновать существующий сервер с некоторой библиотекой, содержащей указанный метод или объект. И снова соответствующая информация может содержаться в хранилище реализаций. Эти два примера иллюстрируют, насколько в действительности тесно связано хранилище с брокером ORB и платформой, на которой он работает.

Службы CORBA

Важной частью структурной модели CORBA является набор служб CORBA. Службы CORBA правильнее всего считать службами общего назначения, не зависящими от приложений, в которых используется CORBA. Таким образом, типы служб CORBA очень похожи на типы служб операционных систем. Полный список служб CORBA приведен в табл. 9.1. К сожалению, не всегда можно провести черту между разными службами, поскольку часто их функциональность пересекается. Давайте кратко опишем каждую из служб, чтобы иметь возможность позже сравнить их со службами DCOM и Globe.

Таблица 9.1. Службы CORBA

Служба	Описание
Служба коллекций	Средства группирования объектов в списки, очереди, множества и т. п.
Служба запросов	Средства для декларирования запросов к наборам объектов
Служба параллельного доступа	Средства обеспечения параллельного доступа к совместно используемым объектам
Служба транзакций	Простые и вложенные транзакции для вызова методов различных объектов
Служба событий	Средства асинхронного взаимодействия на основе механизма событий
Служба уведомлений	Расширенные средства асинхронного взаимодействия на основе механизма событий
Служба внешних связей	Средства маршалинга и демаршалинга объектов
Служба жизненного цикла	Средства создания, удаления, копирования и перемещения объектов
Служба лицензирования	Средства для присоединения к объекту лицензии
Служба именования	Средства именования объектов в пределах системы
Служба свойств	Средства присоединения к объекту пар (атрибут, значение)
Служба обмена	Средства публикации и поиска служб, нужных объекту
Служба сохранности	Средства длительного хранения объектов
Служба отношений	Средства выражения отношений между объектами
Служба защиты	Механизмы создания защищенных каналов, авторизации и аудита
Служба времени	Предоставление текущего времени с заданной ошибкой

Служба коллекций (collection service) предоставляет возможность группировать объекты в списки, очереди, стеки, множества и т. д. В зависимости от природы группы имеются различные механизмы доступа. Так, например, списки можно просматривать поэлементно при помощи механизма, обычно называемого итератором. Имеются также и средства поиска объектов по заданному ключевому значению. В принципе служба коллекций близка к тому, что в объектно-ориентированных языках программирования именуется библиотекой классов.

Имеется также и отдельная *служба запросов (query service)*, предоставляющая данные для создания коллекций объектов, к которым можно обращаться, используя декларативные языки запросов. Запрос может возвращать ссылку на объект или коллекцию объектов. Служба запросов добавляет к службе коллекций возможность создания расширенных запросов. Она отличается от службы коллекций тем, что последняя предполагает возможность создания коллекций различных типов.

Служба параллельного доступа (concurrency service) предоставляет расширенный механизм блокировок, который клиент может применять при доступе к общим объектам. Эта служба может использоваться для реализации транзакций, которые выделяются в отдельную службу. *Служба транзакций (transaction service)* позволяет клиенту определять цепочки обращений к методам нескольких объектов в виде единой транзакции. Эта служба поддерживает простые и вложенные транзакции.

Обычно клиенты обращаются к методам, после чего ожидают получения результата этого обращения. Для поддержания возможности асинхронного взаимодействия CORBA предоставляет *службу событий (event service)*, с помощью которой клиент и объект могут прерывать работу при наступлении определенного события. Дополнительные средства для асинхронного взаимодействия предоставляет отдельная *служба уведомлений (notification service)*. Эту службу мы подробно обсудим чуть позднее.

Служба внешних связей (externalization service) занимается маршалингом объектов таким образом, чтобы их можно было сохранить на диск или передать по сети. Ее можно сравнить со средствами сериализации Java, позволяющими записывать объекты в поток данных в виде последовательности байтов.

Служба жизненного цикла (life cycle service) предоставляет возможности создания, уничтожения, копирования и перемещения объектов. Ключевая концепция здесь — *объект-фабрика (factory object)*, представляющая собой специальный объект, используемый для создания других объектов [157]. Практика показывает, что отдельной службы требует только создание объектов. Уничтожение же, копирование и перемещение объектов часто удобнее определять внутри самих объектов. Причина — на эти операции влияет состояние объекта, то есть способ их осуществления зависит от объекта.

Служба лицензирования (licensing service) позволяет разработчикам присоединять к их объектам лицензии и поддерживать определенные правила лицензирования. Лицензии определяют права клиента на использование объекта. Так, например, лицензия, присоединенная к объекту, может указывать, что объект разрешается использовать одновременно только одному клиенту. Другая лицен-

зия может обеспечивать автоматическую деактивацию объекта по истечении определенного времени.

Для объектов, имеющих осмысленные имена, CORBA поддерживает отдельную *службу именования* (*naming service*), которая занимается отображением этих имен на идентификаторы объектов. Основные средства описания объектов собраны в отдельной *службе свойств* (*property service*). Эта служба позволяет клиентам ассоциировать с объектами пары (*атрибут, значение*). Отметим, что эти атрибуты не являются частями состояния объекта, а используются именно для его описания. Другими словами, они предоставляют информацию об объекте, не являющуюся его частью. С этими двумя службами связана *служба обмена* (*trading service*), которая позволяет объектам рекламировать то, что они могут делать (посредством их интерфейсов), а клиентам производить поиск нужных им служб, используя специальный язык, поддерживающий описание ограничений.

Отдельная *служба сохранности* (*persistence service*) содержит средства сохранения информации на диске в виде объектов хранения. Наиболее важно, что при этом предполагается прозрачность хранения — клиенту не нужно явно передавать данные между диском и оперативной памятью.

Перечисленные службы не предоставляют средств для явного определения отношений между двумя или более процессами. Подобные средства предлагаются *службой отношений* (*relationship service*), которая, в сущности, поддерживает организацию объектов в соответствии с концептуальной схемой, подобно схемам отношений между объектами в базах данных.

Защита обеспечивается *службой защиты* (*security service*). Реализация этой службы напоминает системы защиты SESAME и Kerberos. Служба защиты в CORBA поддерживает средства аутентификации, авторизации, аудита, защищенной связи и администрирования. Ниже мы рассмотрим вопросы защиты более детально.

И, наконец, CORBA поддерживает *службу времени* (*time service*), возвращающую текущее время с некоторой заданной ошибкой.

Как написано в [359], службы CORBA разрабатывались с использованием объектной модели CORBA в качестве базовой. Это означает, что все службы описаны на языке IDL для CORBA и что между описанием и реализацией интерфейсов существует разница. Другой важный принцип построения состоит в том, что службы должны быть просты и минимальны по объему. Далее мы обсудим некоторые из этих служб более детально.

9.1.2. Связь

Исходно CORBA имеет простую модель связи: клиент обращается к методу объекта и ожидает ответа. Эта модель разрабатывалась так, чтобы быть как можно проще, но при необходимости существовала возможность добавлять к ней дополнительные методы взаимодействия. В связи с этим мы особо рассмотрим обращение к средствам в CORBA, обсудив также альтернативы подобного обращения к объектам. Как мы увидим, расширения базовой модели обращения к объекту вызываются необходимостью поддержания асинхронного взаимодействия. Эта необходимость вызывается также фактом существования других моделей передачи сообщений, которые мы обсуждали в главе 2.

Модели обращения к объектам

По умолчанию, когда клиент обращается к объекту, он посылает запрос соответствующему серверу объектов и блокируется до получения ответа. При отсутствии ошибок эта семантика полностью соответствует обычному обращению к методам, когда вызывающая и вызываемая стороны находятся в одном адресном пространстве.

Однако при наличии ошибок все усложняется. В случае описанного выше синхронного обращения клиент в результате получает исключение, указывающее на незаконченность обращения. CORBA определяет, что в этом случае обращение следует семантике «максимум однажды», которая предполагает, что вызываемый метод будет выполнен один раз или не будет выполнен вообще. Отметим, что реализация должна поддерживать эту семантику.

Следовательно, синхронные обращения полезны, когда клиенту нужно дождаться ответа. Если возвращается правильный результат, CORBA гарантирует, что метод был выполнен ровно один раз. Однако в том случае, когда ответ не требуется, для клиента лучше просто вызвать метод и как можно быстрее продолжить выполнение. Этот тип обращения соответствует асинхронному вызову RPC, который мы обсуждали в главе 2.

В CORBA такой тип асинхронных обращений именуется *односторонним запросом* (*one-way request*). Метод может быть определен как односторонний только в том случае, если он не возвращает результата. Однако в отличие от гарантированной доставки асинхронных вызовов RPC, односторонние запросы CORBA предполагают наличие только службы доставки «с максимальными усилиями» (*best effort delivery*). Другими словами, отправитель не получает никаких гарантий того, что запрос будет доставлен серверу объектов.

Кроме односторонних запросов CORBA поддерживает также механизм так называемого *отложенного синхронного запроса* (*deferred synchronous request*). Такой запрос, который фактически является вариантом одностороннего запроса, позволяет серверу возвращать результат клиенту асинхронно. После того как клиент посылает серверу свой запрос, он может, не дожидаясь ответа, сразу же продолжать работу. Другими словами, клиент не знает, был его запрос в действительности доставлен серверу или нет.

Эти три различных модели обращения обобщены в табл. 9.2.

Таблица 9.2. Модели обращений в CORBA

Тип запроса	Семантика при ошибках	Описание
Синхронный	Максимум однажды	Отправитель блокируется до получения ответа или возникновения исключения
Односторонний	Доставка «с максимальными усилиями»	Отправитель продолжает работу немедленно, не ожидая ответа от сервера
Отложенный синхронный	Максимум однажды	Отправитель продолжает работу немедленно и может быть позднее заблокирован до получения ответа

Службы событий и уведомлений

Хотя модели обращений, предлагаемые CORBA, вполне покрывают задачи связи в распределенной системе объектов, чувствуется, что одних обращений к объектам недостаточно. В частности, необходимо наличие службы, которая должна просто сигнализировать о наступлении событий. Клиенты, узнав об этом событии, смогут выполнить соответствующие действия.

Результатом является определение *службы событий (event service)*. Базовая модель событий в CORBA очень проста. Каждое событие ассоциируется с единственным элементом данных, обычно представленным в виде ссылки на объект или же значением, характерным для данного приложения. Событие генерирует *поставщик (supplier)* и получает *потребитель (consumer)*. События доставляются по *каналу событий (event channel)*, логически связывающего поставщика и потребителя, как показано на рис. 9.3.



Рис. 9.3. Логическая организация поставщиков и потребителей событий в соответствии с моделью продвижения

На рисунке показана так называемая *модель продвижения (push model)*. Согласно этой модели при возникновении определенной ситуации поставщик генерирует событие и продвигает его через канал событий, по которому событие доставляется потребителям. Модель продвижения напоминает асинхронный режим работы, который часто ассоциируют с событиями. На практике потребители пассивно ожидают прихода события, полагая, что при возникновении события их работа будет так или иначе прервана.

Альтернативная модель, также поддерживаемая CORBA, — это *модель извлечения (pull model)*, продемонстрированная на рис. 9.4. Согласно этой модели потребители опрашивают канал событий, проверяя, не произошло ли ожидаемое событие. Канал событий, в свою очередь, опрашивает поставщиков событий.



Рис. 9.4. Логическая организация поставщиков и потребителей событий в соответствии с моделью извлечения

Хотя служба событий имеет простой и надежный способ распространения событий, у него есть несколько серьезных недостатков. Для распространения событий необходимо, чтобы поставщики и потребители были соединены с каналом

событий. Это также означает, что если потребитель соединится с каналом после того, как событие произошло, это событие будет потеряно. Другими словами, служба сообщений CORBA не обеспечивает сохранности событий.

Очень важно, что потребители не в состоянии предпринять каких-либо мер для фильтрации событий. Любое событие, в принципе, пересылается всем потребителям. Если необходимо выделять различные типы событий, нужно создавать отдельные каналы событий для каждого из этих типов. Средства фильтрации имеются в расширении [325], известном под названием службы уведомлений (*notification service*). Помимо всего прочего эта служба содержит средства для предотвращения распространения событий, в которых не заинтересован ни один из потребителей.

И, наконец, распространение событий изначально ненадежно. Спецификация CORBA указывает, что по вопросу доставки событий никаких гарантий не дается. Как мы говорили в главе 2, существуют приложения, для которых надежная доставка событий очень важна. Для таких приложений служба событий CORBA не подходит и требуются другие способы взаимодействия.

Передача сообщений

Любое взаимодействие в CORBA, которое мы обсуждали до сих пор, — нерезидентное. Это означает, что сообщения сохраняются в базовых системах связи лишь в процессе выполнения как отправителя, так и получателя. Как мы говорили в главе 2, существует множество приложений, которые требуют сохранной связи, то есть такой, при которой сообщение хранится в системе до тех пор, пока не будет доставлено. При сохранной связи не имеет значения, выполняется или нет после отправки сообщения отправитель или получатель, в любом случае сообщение будет храниться столько, сколько необходимо.

Хорошо известна такая модель сохранной связи, как очереди сообщений. CORBA поддерживает эту модель в виде дополнительной *службы сообщений* (*messaging service*). Обмен сообщениями в CORBA отличается от других систем объектным подходом к взаимодействию. В частности, проектировщики службы сообщений вынуждены были соблюсти условие обязательности обращений к объектам при любом взаимодействии. В случае обмена сообщениями эти ограничения привели к появлению двух дополнительных видов асинхронного обращения к методам.

В *модели обратного вызова* (*callback model*) клиент представляет собой объект, реализующий интерфейс, в котором содержатся методы обратного вызова. Базовая коммуникационная система может вызвать эти методы для передачи результата асинхронного обращения. Важная особенность этой модели состоит в том, что асинхронные обращения к методу не влияют на исходную реализацию объекта. Другими словами, преобразовать исходное синхронное обращение в асинхронное — задача клиента; сервер делает обычный (синхронный) вызов.

Построение асинхронного обращения выполняется в два этапа. Сначала исходный интерфейс, реализуемый объектом, заменяется двумя другими, реализуемыми исключительно программным обеспечением клиента. Один из интерфейсов содержит спецификацию методов, которые может вызывать клиент. Ни один

из этих методов не возвращает значений и не имеет выходных параметров. Второй интерфейс — это интерфейс обратного вызова. Он содержит методы, которые могут быть вызваны брокером ORB клиента для передачи результатов соответствующего метода вызвавшему его клиенту для всех операций исходного интерфейса.

В качестве примера рассмотрим объект, реализующий простой интерфейс с единственным методом:

```
int add(in int i, in int j, out int k);
```

Предположим, что этот метод (который мы описали на IDL для CORBA) получает два неотрицательных целых числа i и j и возвращает $i+j$ в выходном параметре k . Операция должна возвращать -1 в случае неудачного завершения. Преобразование исходного (синхронного) обращения к методу в асинхронное путем обратного вызова происходит так: сначала создается пара описаний методов (для нашей задачи мы предпочли удобство имен строгим правилам именования [331]):

```
void sendcb_add(in int i, in int j);           // Вызывается клиентом
void replycb_add(in int ret_val, in int k);    // Вызывается клиентским ORB
```

В действительности все выходные параметры спецификации исходного метода изымаются из вызываемого клиентом метода и превращаются во входные параметры операции обратного вызова. Точно так же если в исходном методе определялось возвращаемое значение, это значение передается в операцию обратного вызова в качестве входного параметра.

Второй шаг заключается в простой компиляции созданного интерфейса. В качестве результата клиент получает заглушку, которая способна асинхронно вызывать метод `sendcb_add`. Однако клиент должен сам предоставить реализацию интерфейса обратного вызова, содержащего, в нашем примере, метод `replycb_add`. Отметим, что эти изменения не влияют на реализацию объекта на сервере. Описанную модель обратного вызова иллюстрирует рис. 9.5.

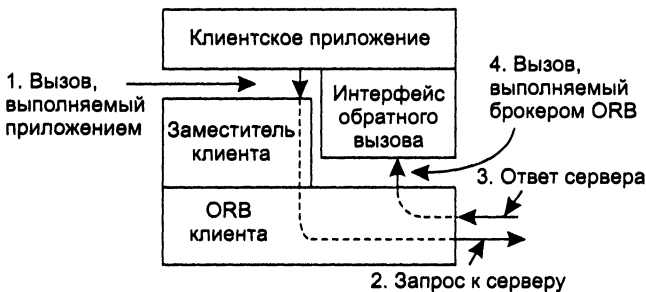


Рис. 9.5. Модель обратного вызова CORBA для асинхронного обращения к методам

В качестве альтернативы обратному вызову CORBA поддерживает *модель опроса (polling model)*. Согласно этой модели клиенту предоставляется набор операций для опроса своего брокера ORB на предмет наличия поступивших результатов. Как и в модели обратного вызова, за преобразование синхронного

обращения к методу в асинхронное отвечает клиент. И снова большую часть работы можно выполнять автоматически, отталкиваясь от спецификации соответствующего метода в исходном интерфейсе, который реализован в объекте.

Рассмотрим вновь наш пример. Метод `add` превратится в следующие два описания методов (и снова для удобства мы используем свои правила именования):

```
void sendpoll_add(in int i, in int j):           // Вызывается клиентом
void replepoll_add(out int ret_val, out int k): // Тоже вызывается клиентом
```

Наиболее важное различие между моделью опроса и моделью обратного вызова состоит в том, что метод `replepoll_add` реализуется клиентским брокером ORB. Эта реализация может быть автоматически сгенерирована компилятором IDL. Модель опроса приведена на рис. 9.6. И вновь мы считаем нужным отметить, что исходная реализация объекта на стороне сервера не претерпевает никаких изменений.

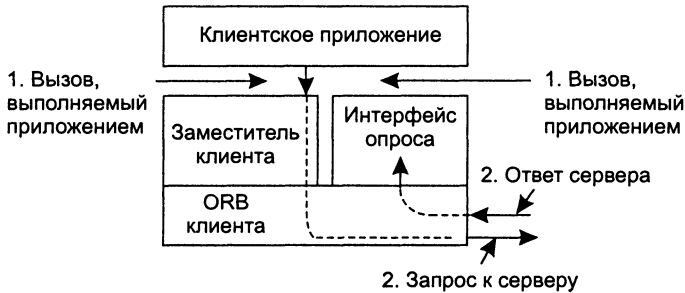


Рис. 9.6. Модель опроса CORBA для асинхронного обращения к методам

В описанных здесь моделях мы не учли тот факт, что сообщения, пересылаемые между клиентом и сервером, в том числе и ответы на асинхронные обращения, в случае неработоспособности клиента или сервера сохраняются базовой системой. К счастью, большинство моментов, связанных с сохранностью связи, не оказывают влияния на эти модели асинхронных обращений. Все, что нам необходимо сделать для создания коллекции серверов сообщений, — это разрешить временное хранение сообщений (будь то обращения или ответы) до момента доставки. Этот подход, описанный в [331], аналогичен системе очередей сообщений IBM, которую мы уже рассматривали в пункте 2.4.4 и не будем повторяться.

Совместимость

Ранние версии CORBA отдавали решение многих вопросов на откуп производителям. В результате системы CORBA разных производителей имеют собственные способы реализации взаимодействия между клиентами и серверами объектов. В частности, клиент может обратиться к объекту на сервере только в том случае, если клиент и сервер используют один и тот же брокер ORB.

Эта проблема совместимости была решена путем введения стандартизованного протокола обмена между различными брокерами ORB в комбинации с универсальным способом создания ссылок на объекты. Если вернуться к рис. 9.2,

это означает, что указанная между клиентом и сервером связь обслуживается по стандартному протоколу, который в CORBA известен как *обобщенный протокол обмена между ORB (General Inter-ORB Protocol, GIOP)*.

GIOP — это на самом деле заготовка (каркас) протокола. Предполагается, что конкретная реализация будет выполняться поверх существующего транспортного протокола. Этот транспортный протокол должен быть надежным, ориентированным на соединение, а также поддерживать понятия потока байтов и некоторые другие. Нас не особенно удивляет, что TCP удовлетворяет всем этим требованиям, однако имеются и другие подходящие транспортные протоколы. Реализация GIOP поверх TCP называется *Интернет-протоколом обмена между ORB (Internet Inter-ORB Protocol, IIOP)*.

Протокол GIOP (включая IIOP и другие реализации GIOP) поддерживает восемь типов сообщений, которые перечислены в табл. 9.3. Два наиболее важных типа сообщений — это Request и Reply, которые вместе образуют часть реализации обращений к удаленным методам.

Таблица 9.3. Типы сообщений GIOP

Тип сообщения	Источник	Описание
Request	Клиент	Содержит обращение
Reply	Сервер	Содержит отклик на обращение
LocateRequest	Клиент	Содержит запрос о точном местоположении объекта
LocateReply	Сервер	Содержит информацию о местоположении объекта
CancelRequest	Клиент	Указывает на то, что клиент более не ожидает ответа
CloseConnection	Оба	Указывают на завершение соединения
MessageError	Оба	Содержат информацию об ошибке
Fragment	Оба	Часть (фрагмент) более длинного сообщения

Сообщение Request содержит запрос, полученный после завершения маршрутизации обращения, включая ссылку на объект, имя вызываемого метода и все необходимые входные параметры. Ссылка на объект и название метода входят в заголовок. Каждое сообщение Request также содержит собственный идентификатор запроса, с помощью которого впоследствии можно найти соответствующий ответ.

Сообщение Reply содержит возвращаемые значения и выходные параметры (также подвергнутые маршрутировке), ассоциируемые с ранее вызванным методом. Нет необходимости полностью указывать объект или метод, достаточно возвратить такой же идентификатор запроса, который был в соответствующем сообщении Request.

Как мы увидим чуть ниже, клиент может потребовать от хранилища реализаций предоставить детальную информацию о том, где размещается некий объект. Такому запросу соответствует посылка сообщения LocateRequest. Хранилище реализаций отвечает на этот запрос посылкой сообщения LocateReply, которое обычно идентифицирует текущий сервер объекта, к которому можно обратиться.

Сообщение CancelRequest может быть послано клиентом серверу в том случае, если клиент хочет отменить действие ранее посланных сообщений Request или

LocateRequest. Отмена запроса означает, что клиент не намерен более ждать ответа сервера. Клиент может отменить посланный запрос по различным причинам, но обычно это связано с окончанием тайм-аута клиентского приложения. Важно отметить, что отмена запроса не означает, что соответствующее обращение вообще не будет выполнено. Справляться с ситуацией должно приложение клиента.

В GIOP клиент всегда устанавливает соединение с сервером сам. Серверы могут лишь принимать или отвергать запросы на установку соединения, но сами устанавливать соединения с клиентами не в силах. Однако и клиент, и сервер имеют право разорвать соединение, для этого они должны послать другой стороне сообщение CloseConnection.

Если при выполнении произошла ошибка, другая сторона уведомляется об этом сообщением MessageError. Это сообщение содержит заголовок сообщения GIOP, вызвавшего ошибку. Такой же подход принят в сообщениях ICMP протокола IP, которые используются для возвращения информации об ошибке, если что-то идет не так. В этом случае заголовок IP-пакета, вызвавшего ошибку, возвращается в форме данных сообщения ICMP.

И, наконец, GIOP позволяет разбивать различные сообщения с запросами и ответами на части. Эта возможность позволяет упростить пересылку между клиентом и сервером больших объемов данных. Отдельные фрагменты пересылаются в виде специальных сообщений Fragment, определяющих исходное сообщение и позволяющих производить сборку этого сообщения на стороне получателя.

9.1.3. Процессы

CORBA выделяет два типа процессов — клиентские и серверные. Одна из основных задач CORBA — сделать клиентские приложения как можно проще. Базовая идея состоит в том, чтобы максимально упростить разработчикам использование существующих служб, реализованных на серверах.

Серверы, со своей стороны, изначально открыты для различных реализаций и предоставляют минимальную поддержку реализации в виде базового адаптера объектов. К сожалению, эта минимальная поддержка также приводит к проблемам переносимости, которые в настоящее время разрешаются при помощи более полной, улучшенной спецификации возможностей, предоставляемых адаптером объектов. Ниже мы подробно рассмотрим клиентское и серверное программное обеспечение CORBA.

Клиенты

Как говорилось ранее, клиентское программное обеспечение CORBA сведено к минимуму. Спецификация объекта на языке IDL просто компилируется в заместитель, который выполняет сначала маршалинг (упаковку) запроса, например, в сообщения Request протокола IIOP, а затем — демаршалинг (распаковку) соответствующих сообщений Reply, получая результаты, которые можно вернуть вызвавшему метод клиенту.

Заместители в CORBA не выполняют других задач, кроме соединения клиентских приложений с соответствующим брокером ORB. Вместо того чтобы генерировать отдельные заместители, учитывающие специфику каждого объекта, клиент может также обратиться к объекту динамически посредством DII.

В результате подобного подхода, если объект нуждается в специфической реализации своих интерфейсов на стороне клиента, он должен либо предоставить свой клиентский заместитель, либо оставить задачу генерации необходимого кода разработчикам, вооруженным компилятором IDL. Так, например, реализация объекта может поставляться в комплекте с набором заместителей, реализующих специфическую для объекта стратегию кэширования на стороне клиента. Разумеется, такой подход полностью противоречит назначению CORBA в части переносимости и прозрачности распространения.

Другой подход состоит в том, чтобы забыть о специфике объекта и положиться на то, что клиентский брокер ORB сам предоставит ему всю необходимую поддержку. Например, вместо того чтобы предлагать объекту кэширование, реализуемое в клиентском заместителе, реализация объекта может надеяться, что кэширование обычным образом выполнит клиентский брокер ORB. И вновь должно быть понятно, что такой подход изначально является ограниченным.

Необходим такой механизм использования заместителей, сгенерированных компилятором IDL вместе с существующим клиентским брокером ORB, который не требовал бы адаптации клиентского программного обеспечения. В CORBA эта проблема решается с помощью перехватчиков. Как следует из названия, *перехватчик* (*interceptor*) — это механизм перехвата обращений на пути от клиента к серверу и при необходимости их изменения «на лету». В сущности, перехватчик — это фрагмент кода, который модифицирует запрос на пути от клиента к серверу и соответствующим образом перерабатывает ответное послание сервера. К ORB могут быть добавлены самые разные перехватчики. Какой из них в реальности будет активизирован, зависит от объекта или сервера, указанного в запросе.

Перехватчик в CORBA может находиться на одном из двух логических уровней, что показывает рис. 9.7. *Перехватчик уровня запросов* (*request-level interceptor*) логически находится между клиентским заместителем и брокером ORB. До того как запрос попадет в ORB, он пройдет через перехватчик, который может модифицировать этот запрос. Со стороны сервера перехватчик уровня запросов располагается между ORB и адаптером объекта.

В противоположность этому *перехватчик уровня сообщений* (*message-level interceptor*) располагается между ORB и базовой сетью. Типичный пример перехватчика уровня сообщений — перехватчики, занимающиеся дроблением сообщений GIOP типа Fragment на стороне отправителя и их сборкой на стороне получателя.

Перехватчики видит только ORB, то есть именно ORB отвечает за активизацию перехватчиков. Клиенты и серверы обычно не замечают перехватчиков, за исключением того момента, когда клиент связывается с сервером. Отметим, что ORB может использовать оба типа перехватчиков одновременно. Обзор перехватчиков CORBA можно найти в [309], а полную их спецификацию — в [331].

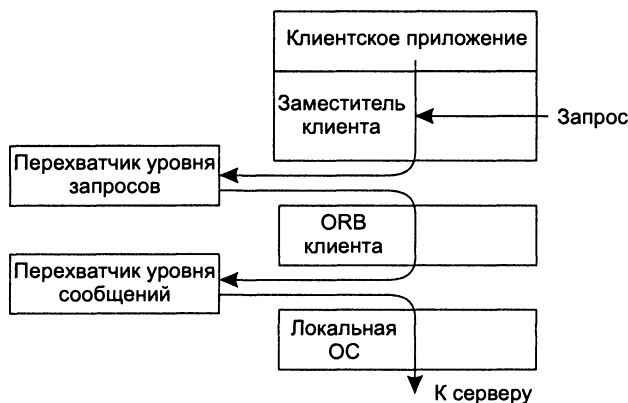


Рис. 9.7. Перехватчики в CORBA

Хотя идея с перехватчиками кажется на первый взгляд весьма привлекательной, следует отметить, что этот механизм был добавлен в реализацию для исправления некоторых предыдущих упущений. Так, например, CORBA содержит средства разработки и использования заместителей конкретных объектов, а при наличии перехватчиков необходимость в этих заместителях отпадает. В реальности единственное, что нам необходимо, — легко расширяемые брокеры ORB. Перехватчики, предлагаемые CORBA, действительно обеспечивают обобщенный механизм расширения, но тот ли это механизм, который нам нужен, еще вопрос. Дополнительную информацию о перехватчиках с точки зрения программной архитектуры можно найти в [402].

Переносимый адаптер объектов

В главе 3 мы детально рассмотрели понятие адаптера объектов — механизма реализации конкретных правил активизации группы объектов. Так, один адаптер может реализовывать обращения к методам, используя для каждого обращения отдельный поток выполнения, в то время как другой ограничится единственным потоком выполнения для всех управляемых им объектов.

В основном адаптеры объектов не просто вызывают методы объектов. Как можно понять из их названия, адаптеры объектов отвечают за создание целостного образа объекта; они адаптируют программы так, чтобы клиенты могли рассматривать программы как объекты. Адаптеры называют еще *оболочками* (*wrappers*).

В CORBA *переносимый адаптер объектов* (*Portable Object Adapter, POA*) — это компонент, отвечающий за то, чтобы серверный код представлялся клиенту объектом CORBA. POA определяется так, чтобы серверный код можно было написать без привязки к конкретному брокеру ORB.

Для поддержания переносимости между различными брокерами ORB CORBA предполагает, что реализация объекта частично выполняется так называемым слугой. *Слуга* (*servant*) — это часть объекта, реализующая методы, к которым могут обращаться клиенты. Реализация слуги, естественно, зависит от используемого языка программирования. Так, например, реализация слуги на C++ или

Java обычно выполняется в виде экземпляра класса. С другой стороны, слуга, написанный на С или другом процедурном языке, обычно состоит из набора функций, работающих со структурами данных, которые представляют состояние объекта.

Каким образом POA использует слугу для построения образа объекта CORBA? В первую очередь, в каждом адаптере POA имеется следующая операция, которую мы описываем на IDL для CORBA:

```
ObjectId activate_object(in Servant p_servant);
```

Эта операция использует в качестве входного параметра указатель на слугу, а в качестве результата возвращает идентификатор объекта CORBA. Универсального определения типа Servant не существует, он отображается на типы данных выбранного языка программирования. Так, например, в С++ Servant отображается на указатель предопределенного класса ServantBase. Этот класс содержит несколько описаний методов, которые каждый слуга, написанный на С++, должен реализовывать самостоятельно.

Идентификатор объекта, создаваемый POA, возвращается операцией `activate_object`. Этот идентификатор указывает на слугу и используется в качестве индекса в *карте активных объектов* (*active object map*) POA, как показано на рис. 9.8, а. В этом случае POA реализует отдельного слугу для каждого поддерживаемого объекта. Чтобы сделать обсуждение более конкретным, предположим, что разработчик написал класс, дочерний для ServantBase, и назвал его My_Servant. Объект С++, созданный в виде экземпляра класса My_Servant, может быть превращен в объект CORBA так, как показано в листинге 9.1.

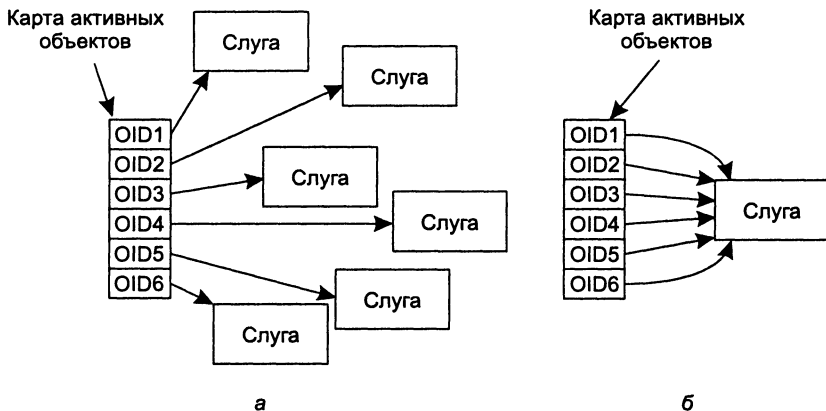


Рис. 9.8. Отображение идентификаторов объектов CORBA на слуг. POA поддерживает несколько слуг (а). POA поддерживает одного слугу (б)

Листинг 9.1. Преобразование объекта С++ в объект CORBA

```
My_Servant *my_object;           // Объявление ссылки на объект С++
CORBA::ObjectId_var oid;         // Объявление идентификатора CORBA
my_object = new MyServant;       // Создание нового объекта С++
// Регистрация объекта С++ в качестве объекта CORBA
oid = poa->activate_object(myobject);
```

В листинге 9.1 мы объявляем ссылку на объект C++. Для создания идентификатора CORBA мы используем инструкцию `CORBA::ObjectId_var`, где CORBA — тип данных C++, предопределенный во всех стандартных реализациях CORBA на языке C++. После этих объявлений создается истинный объект C++ `my_object`. По терминологии CORBA этот объект соответствует слуге. Преобразование объекта C++ в объект CORBA производится путем регистрации его в POA (мы считаем, что доступ к нему осуществляется через переменную `poa`). Результатом регистрации является идентификатор CORBA.

Важно отметить, что при создании второго объекта типа `My_Servant` регистрация этого объекта C++ в POA приведет к созданию такого же слуги, работающего с другим состоянием этого класса. В тех случаях, когда POA нужно поддерживать объекты, порожденные из одного и того же определения класса, разумнее зарегистрировать одного слугу и различать объекты только по их состоянию. Этот принцип продемонстрирован на рис. 9.8, б, на котором каждый идентификатор объекта ссылается на одного и того же слугу. В этом случае при обращении к объекту идентификатор объекта неявно передается слуге. Таким образом, слуга сможет работать только с тем набором данных, который уникально описывает идентифицированный объект.

Этот пример также иллюстрирует и другой важный момент: идентификатор объекта CORBA однозначно ассоциирован с POA. Когда в POA регистрируется слуга, POA возвращает идентификатор объекта для этого слуги. С другой стороны, мы можем не заметить, что разработчик приложения сначала создает идентификатор, а уже потом передает его в POA. Однако в обоих случаях этот идентификатор будет спрятан в большой структуре данных, существующей в виде глобальной ссылки на объект и идентифицирующей, кроме всего прочего, и POA, и сервер, на котором этот адаптер POA расположен.

Независимо от того, поддерживает POA одного слугу на объект или одного слугу на все объекты, существует только один набор правил, который можно ассоциировать с POA. Правила могут быть самыми разными. Так, например, POA может поддерживать как нерезидентные, так и сохраненные объекты. Точно так же разными правилами может регламентироваться использование потоков выполнения. Подробности можно найти в [198].

Агенты

Чтобы способствовать использованию приложений-агентов, в CORBA принята модель, согласно которой агенты систем разных типов могут работать совместно. Вместо определения собственной модели агента, CORBA предоставляет стандартный интерфейс для работы с агентами, который должны реализовывать любые системы. В таком подходе имеется потенциальное преимущество, поскольку в одном распределенном приложении могут быть задействованы разные типы агентов. Так, например, в CORBA Java-апплет может создать агента Tcl на платформе D'Agents (мы описывали систему D'Agents в главе 3).

В CORBA агенты всегда определяются, исходя из системы агентов. *Система агентов (agent system)* определяется как платформа, позволяющая создавать, выполнять, переносить на другую машину или прекращать работу агентов. Каждая

система агентов имеет соответствующий профиль, точно определяющий, как выглядит агент этой системы. Так, например, в профиле агента системы D'Agents, определяется тип системы (D'Agents), поддерживаемый язык (например, Tcl) и способ преобразования агента в последовательность байтов для передачи от системы к системе.

Агент всегда находится в специальном *месте (place)* в системе агентов. Это место соответствует серверу, на котором располагается агент. Таких мест в системе агентов очень много. Другими словами, CORBA предполагает, что одна система агентов может состоять из множества процессов, каждый из которых содержит один или несколько агентов. Такая организация предполагает группировку нескольких держателей агентов в одну административную единицу и возможность обращаться к этой группе как к единому целому, называя это целое системой агентов.

Системы агентов, в свою очередь, могут группироваться в *регион (region)*, представляющий собой административную единицу, внутри которой находятся системы агентов. Так, например, факультет университета может иметь несколько систем агентов различного типа. Каждая система может быть разнесена по нескольким хостам, или местам, и на каждом хосте могут работать несколько агентов. Эта модель демонстрируется на рис. 9.9.

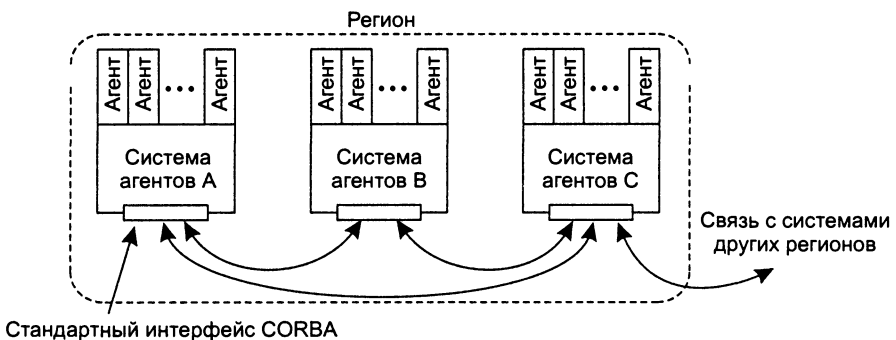


Рис. 9.9. Общая модель агентов, систем агентов и регионов CORBA

Агент в CORBA рассматривается как сущность, собранная из набора классов или, что значительно реже, из файла, содержащего необходимый текст программы. В любом случае можно дать реализации агента имя и передать это имя в систему агентов, которая получит возможность создавать этого агента и пересылать его на другие машины.

Любая система агентов в CORBA должна реализовывать несколько стандартных операций, например, создания агента и прекращения его существования, запуска агента на исполнение, ведения списка существующих агентов, ведения списка мест, где могут располагаться агенты, приостановки и повторного запуска агента. Отметим, что реализация этих операций зависит исключительно от данной системы агентов. Однако CORBA требует, чтобы любая из систем агентов укладывалась в общую модель. Это означает, что если какая-то существующая

система агентов изначально не поддерживала понятия места, в CORBA она вынуждена будет это делать.

Каждый регион должен иметь своего *искателя* (*finder*), который позволит в пределах этого региона определять местоположения агентов, мест для них и систем агентов. Искатель также имеет набор стандартных операций для регистрации и снятия с регистрации агентов, мест для них и систем агентов. В сущности, искатель для данного региона — это не что иное, как просто служба каталогов.

Дополнительную информацию по мобильным агентам в CORBA можно найти в [329].

9.1.4. Именованное

CORBA поддерживает различные типы имен. Имена самого низкого уровня — это ссылки на объекты и символьные имена, поддерживаемые службой именования CORBA. Кроме того, существует несколько расширенных механизмов именования, при помощи которых можно находить объекты по их свойствам. Далее мы рассмотрим базовую службу именования CORBA, особенно внимательно изучая ссылки на объекты. Что касается расширенных механизмов именования, доступных через службу обмена CORBA, мы отсылаем читателя к [326, 330].

Ссылки на объекты

Основа CORBA — это способ организации ссылок на объекты. Когда клиент получает ссылку на объект, он может обращаться к методам, реализованным внутри этого объекта. Важно отличать ссылку на объект, которую использует клиентский процесс для обращения к методу, от реализации ссылки в базовом брокере ORB.

Процесс (будь то клиент или сервер) может использовать только такую реализацию ссылки на объект, которая характерна для текущего языка программирования. В большинстве случаев она принимает вид указателя на локальное представление объекта. Эта ссылка не может быть передана из процесса *A* в процесс *B*, поскольку имеет смысл только в адресном пространстве процесса *A*. Чтобы передать ссылку в другой процесс, процесс *A* должен сначала выполнить маршалинг указателя, чтобы его вид не зависел от процесса. Эта операция осуществляется брокером ORB. После завершения маршалинга ссылка может быть передана в процесс *B*, который путем демаршалинга извлечет ее из сообщения. Отметим, что процессы *A* и *B* могут быть исполняемыми программами, написанными на разных языках программирования.

В противоположность этим процессам обслуживающий их брокер ORB имеет свое, не связанное с языками программирования представление ссылки на объект. Это представление может отличаться и от той версии, которую он путем маршалинга передает процессу при обмене ссылками. Важно отметить, что когда процесс использует ссылку на объект, обслуживающий его брокер ORB неявно передает полный набор информации, необходимой для того, чтобы узнать, к какому объекту на самом деле происходит обращение. Эта информация обычно передается клиентской и серверной заглушками, сгенерированными из спецификаций объекта на языке IDL.

Одна из проблем ранних версий CORBA состояла в том, что каждый брокер ORB сам решал, как представлять ссылку на объект. Таким образом, если процесс *A* хотел, как описывалось выше, послать ссылку процессу *B*, это успешно получалось только в том случае, если оба процесса работали поверх одного и того же брокера ORB. В противном случае версия ссылки после маршалинга, которую посылал процесс *A*, была непонятна брокеру ORB, обслуживающему процесс *B*.

Все современные системы CORBA поддерживают одинаковое, не зависящее от языка программирования представление ссылок на объекты, которое называется *межоперационной ссылкой на объект* (*Interoperable Object Reference, IOR*). Не так уж важно, использует или нет ORB формат IOR для своих внутренних нужд. Однако при передаче ссылки на объект другому брокеру ORB он делает это в формате IOR. Ссылка IOR содержит всю информацию, необходимую для идентификации объекта. Общий вид IOR представлен на рис. 9.10 вместе со специфичной информацией для протокола IIOP, которую мы далее рассмотрим в деталях.

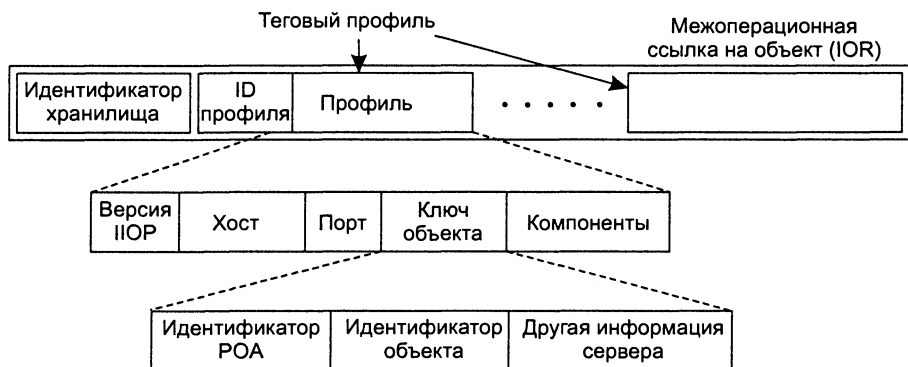


Рис. 9.10. Организация IOR со специфической информацией для IIOP

Каждая ссылка IOR начинается с идентификатора хранилища. Как мы говорили, этот идентификатор назначается интерфейсу в момент его помещения в хранилище интерфейсов, используется для поиска информации об интерфейсе во время исполнения и может применяться, например, для проверки типов или создания обращения к интерфейсу во время исполнения. Отметим, что благодаря этому идентификатору и клиент, и сервер получают доступ в одно и то же хранилище интерфейсов или как минимум идентифицируют интерфейсы по одному и тому же идентификатору.

Наиболее важная часть каждой ссылки IOR — так называемый *теговый профиль* (*tagged profile*). Каждый теговый профиль содержит всю информацию, необходимую для обращения к объекту. Если сервер объектов поддерживает несколько протоколов, информация о каждом из них будет содержаться в отдельном теговом профиле. Детали профиля для IIOP также представлены на рис. 9.10.

Профиль для ИОР идентифицируется соответствующим полем тегового профиля и состоит, в свою очередь, из пяти полей.

- ◆ Поле версии ИОР определяет версию протокола ИОР, использованную в данном профиле.
- ◆ Поле хоста — это строка, идентифицирующая хост, на котором находится объект. Хост может быть задан либо в виде полного доменного DNS-имени (например, `soling.csvu.nl`), либо в виде IP-адреса хоста (например, `130.37.24.11`).
- ◆ Поле порта содержит номер порта, с которого сервер объекта ожидает входящих сообщений.
- ◆ Поле ключа объекта содержит информацию, необходимую серверу для выделения запросов, предназначенных для данного объекта. Так, например, частью этого ключа, как правило, является созданный POA идентификатор объекта. Этот ключ будет также идентифицировать POA.
- ◆ И, наконец, поле компонентов может содержать дополнительную информацию, необходимую для правильного обращения к объекту, к которому относится ссылка. Это может быть, например, информация по защите, показывающая, как должна обрабатываться ссылка или что делать, если указанный в ссылке сервер временно недоступен. Ниже мы еще вернемся к этим вопросам.

Теперь, когда мы детально рассмотрели ссылки на объекты, будет несложно понять, как клиент связывается с объектом для последующего вызова его методов. Вспомним, что в главе 2 мы говорили, что привязка клиента к объекту — это процесс организации соединения с объектом, выполнив который клиент может обращаться к методам объекта. Привязка возможна только в том случае, если у клиента есть ссылка на объект.

Предваряя обсуждение службы именования CORBA, предположим, что клиент запросил службу именования разрешить некое осмысленное (для человека) имя. В ответ служба именования вернет хранимую ею и зависящую от языка программирования реализацию ссылки IOR. Подобная реализация может быть возвращена только в том случае, если клиентский брокер ORB полностью завершил процесс привязки. Это происходит следующим образом.

ORB клиента получает ссылку IOR, возвращенную ему службой именования, и для проверки идентификатора хранилища, содержащегося в IOR, может поместить заместитель на клиента и вернуть указатель *p* на этот заместитель. Внутрь себя ORB сохраняет сведения о том, что *p* соответствует IOR объекта.

Разные брокеры ORB имеют разную реализацию, но одинаковый сценарий работы. До передачи *p* клиенту клиентский брокер ORB проверяет теговые профили, входящие в IOR. Предположим, что объект вызывается по протоколу ИОР. В этом случае клиентский брокер ORB может, используя адрес хоста и номер порта, найденные в IOR, установить с сервером объектов соединение по протоколу TCP. В этот момент он и передает *p* клиенту.

Когда клиент обращается к одному из методов объекта, ORB клиента выполняет маршрутинг запроса в сообщение Request протокола ИОР. Это сообщение содержит ключ объекта для данного сервера, который также хранится в IOR.

Сообщение через соединение TCP передается на сервер, где пересылается адаптеру POA, ассоциированному с ключом объекта. Далее POA передаст запрос нужному слуге, где путем демаршалинга он будет преобразован в реальный вызов метода.

Согласно этому сценарию IOR непосредственно ссылается на объект, что приводит к так называемой *прямой привязке (direct binding)*. Альтернативой прямой привязке является *непрямая привязка (indirect binding)*, при которой построенный запрос сначала пересылается в хранилище реализаций. Хранилище реализаций — это еще один процесс, идентифицируемый в IOR объекта. Он служит реестром, в котором может находиться и активизироваться перед получением запроса указанный объект. На практике непрямая привязка используется в первую очередь для сохраненных объектов, то есть объектов, управляемых POA и соблюдающих правила, обязательные для объектов длительного хранения.

Когда клиентский брокер ORB использует ссылки IOR, основанные на непрямой привязке, он просто начинает процедуру привязки с хранилища реализаций. Хранилище уведомляет, что запрос на самом деле предназначен для другого сервера, и просматривает свои таблицы в поисках работающего сервера и его местоположения. Если соответствующий сервер в данный момент не запущен, хранилище реализаций может запустить его, если он поддерживает возможность автоматического запуска.

Когда клиент впервые обращается к объекту по ссылке, запрос передается хранилищу реализаций, которое отвечает за предоставление информации о том, где на самом деле находится сервер объектов (рис. 9.11). После этого запрос пересылается соответствующему серверу.

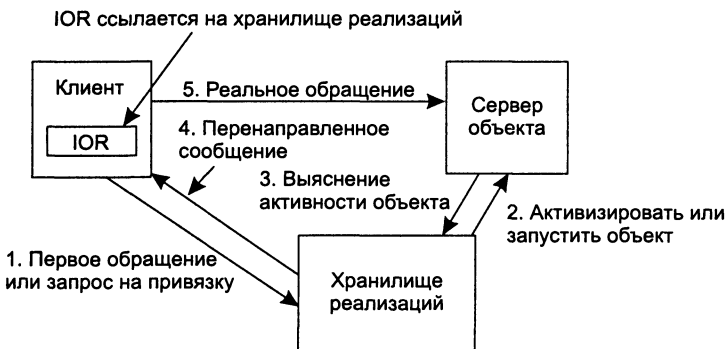


Рис. 9.11. Непрямая привязка в CORBA

Служба именованя CORBA

Как и другие распределенные системы, CORBA имеет службу именованя, которая позволяет клиентам находить ссылки на объекты по символьным именам. Формально имя в CORBA — это последовательность компонентов имени, каждый из которых имеет вид пары (*идентификатор, сорт*), где *идентификатор* и *сорт* — строки. Обычно *идентификатор* используется для именованя объекта строкой символов, например «steen» или «elke». Атрибут *сорт* — это просто ин-

дикатор именованного объекта, похожий на расширение в именах файлов. Так, например, объект, названный «steen», может иметь сорт «dir», указывающий на то, что это объект каталога.

Не существует способа представить путь и имя в виде одной строки. Другими словами, CORBA не определяет разделитель между компонентами имени. Вместо этого имени можно передавать просто как последовательность компонентов имени. Представление последовательности зависит от используемого языка программирования, но для клиента, пользующегося услугами службы именования, оно остается непрозрачным.

На структуру графа именования не существует ограничений. Каждый из узлов графа именования трактуется как объект. *Контекст именования (naming context)* — это объект, хранящий таблицу отображения компонентов имен в ссылки на объекты. Контекст именования, таким образом, это то, что в главе 4 мы называли направляющим узлом. Отметим, что ссылка на объект в контексте именования может ссылаться на другой контекст именования.

Граф именования не имеет корневого контекста. Однако каждый брокер ORB имеет исходный контекст именования, который в данных условиях представляет собой корень графа именования. Имена всегда разрешаются в соответствии с данным контекстом именования. Другими словами, если клиент хочет разрешить имя, он должен вызвать метод *resolve* конкретного контекста именования. Если разрешение имени произошло успешно, он всегда возвращает ссылку либо на контекст именования, либо на именованный объект. В этом смысле разрешение имен происходит именно так, как было описано в главе 4, и здесь мы не будем повторно приводить всю эту информацию.

9.1.5. Синхронизация

Две наиболее важные службы, облегчающие синхронизацию в CORBA, — это служба параллельного доступа и служба транзакций. Эти две службы работают совместно, реализуя распределенные и вложенные транзакции путем двухфазной блокировки.

В основе транзакций в CORBA лежит следующая модель. Транзакция инициируется клиентом и содержит последовательность обращений к объектам. Когда соответствующий объект вызывается в первый раз, он автоматически становится частью транзакции. При этом сервер объекта уведомляется о том, что теперь он участвует в транзакции. Эта информация неявно передается серверу при обращении к объекту.

Частью транзакции могут стать объекты двух основных типов. *Восстановимый объект (recoverable object)* — это объект, выполняемый сервером объектов, который способен включиться в протокол двухфазного подтверждения. В частности, серверы восстановимых объектов в состоянии обработать прерывание транзакции с откатом всех изменений, внесенных в результате обращения к одному из восстановимых объектов. Однако в ходе транзакции может осуществляться и обращение к объектам, которые невозможно вернуть в состояние, предшествующее началу транзакции. Конкретно — это *транзакционные объекты (transactional objects)*, выполняемые серверами, которые не могут включиться в прото-

кол двухфазного подтверждения транзакций. Транзакционные объекты обычно предназначены только для чтения.

Таким образом, понятно, что транзакции в CORBA подобны распределенным транзакциям и их протоколам, которые мы обсуждали в главах 5 и 7.

Точно так же и службы блокировок, предоставляемые службой параллельного доступа, — это именно то, что мы ожидаем увидеть. На практике служба реализуется при помощи центрального менеджера блокировок, приемы распределенной блокировки не применяются. Служба отличает блокировки записи от блокировок чтения и способна поддерживать несколько степеней детализации блокировки, что часто бывает нужно при работе с базами данных. Так, например, она в состоянии отличать блокировку всей таблицы от блокировки одной записи. Информацию о степенях блокировки можно получить в [160, 177]

9.1.6. Кэширование и репликация

CORBA не поддерживает обобщенных кэширования и репликации. В третью версию CORBA, как мы увидим далее, включена только возможность репликации объектов в целях отказоустойчивости. Отсутствие поддержки обобщенной репликации означает, что в случае необходимости разработчики приложений в плане поддержания репликации могут прибегнуть к специальным мерам. В большинстве случаев эти меры основаны на использовании перехватчиков.

Давайте рассмотрим один из примеров того, как для повышения производительности в CORBA можно включить возможность репликации. Эта задача решается в системе CASCADE. В CASCADE цель состоит в предоставлении обобщенной масштабируемой схемы, позволяющей кэшировать любой объект CORBA [102]. CASCADE поддерживает службу кэширования, реализуемую с помощью по возможности большого набора серверов объектов, каждый из которых ссылается на так называемый *доменный сервер кэширования (Domain Caching Server, DCS)*. Каждый сервер DCS — это сервер объектов, работающий в ORB системы CORBA. Коллекция серверов DCS может быть распределена по глобальной сети, такой как Интернет.

Кэшируемые копии одного и того же объекта образуют иерархию. Подразумевается, что одиночный клиент, например владелец объекта, может зарегистрировать свой объект на локальном сервере DCS. Этот DCS становится корнем иерархии. Другие клиенты также могут потребовать у своих локальных серверов DCS кэшировать копию этого объекта. Для этого DCS сначала присоединяется к текущей иерархии тех DCS, которые уже кэшируют этот объект.

CASCADE поддерживает клиентскую модель непротиворечивости данных, которую мы обсуждали в главе 6. Кроме того, он поддерживает тотальное упорядочивание, которое гарантирует, что все изменения повсюду будут производиться в одном и том же порядке. Каждый из объектов может иметь собственную модель непротиворечивости, но общесистемных правил поддержания кэшируемых объектов не существует. Как мы показали в главе 6, при репликации с целью повышения производительности важно, чтобы одновременно поддерживались разные модели непротиворечивости, поскольку применимость моделей в значитель-

ной степени зависит от шаблонов обращения и доступа к объекту. CASCADE удовлетворяет этим требованиям.

В качестве службы CORBA система CASCADE тесно связана с перехватчиками. Со стороны клиента система CASCADE фактически невидима, все вопросы, связанные с непротиворечивостью, скрыты за интерфейсами, которые предоставляют объекты. Единственный случай, когда клиент получает явный доступ к CASCADE, — обращение к своему локальному серверу DCS с запросом о начале кэширования конкретного объекта. Когда в дальнейшем к этому объекту происходят обращения, они перехватываются клиентским брокером ORB и пересылаются DCS.

В зависимости от модели непротиворечивости объекта, указанного в ссылке, к запросу перед его отправкой в DCS добавляется дополнительная информация. Так, например, если клиент нуждается в непротиворечивости чтения собственных записей, необходимо сообщить объекту, какую последнюю операцию записи видел клиент.

Обобщенную организацию DCS иллюстрирует рис. 9.12. DCS управляет несколькими копиями объекта. Копия объекта состоит из состояния и реализаций операций, производимых в этом состоянии. В терминологии CORBA, DCS имеет того же слугу, что и исходная копия объекта. Перехватчик, стоящий за DCS, перехватывает входящие обращения и выделяет из них определенную информацию, например, добавленную перехватчиком клиента. Запрос после этого пересылается управляющему модулю обращений к методу, который однозначно связан с объектом, указанным в ссылке. Как показано на рисунке, каждый кэшированный объект имеет собственный объект правил, который содержит специфическую информацию об управлении обращениями к методу. Дополнительная информация, извлеченная перехватчиками, передается в этот объект правил отдельно.

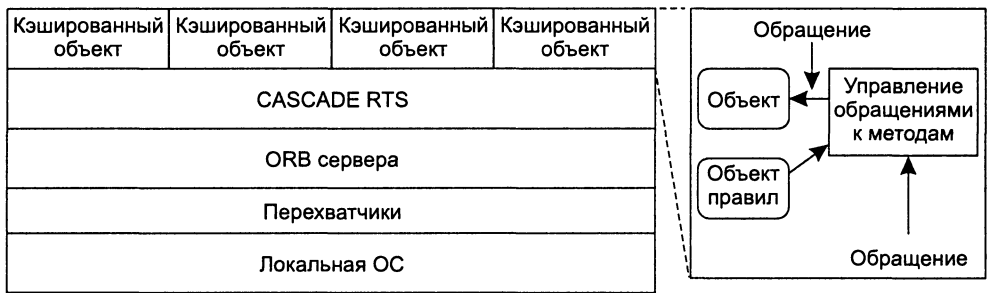


Рис. 9.12. Упрощенная организация DCS

Хотя CASCADE и позволяет более или менее прозрачно кэшировать серверы объектов, однако похоже, что для реализации службы кэширования в CORBA необходимы дополнительные усилия. Хотя при необходимости изменять обращения для реализации подобной услуги можно с помощью перехватчиков, никакой другой поддержки кэширования CORBA не предоставляет.

9.1.7. Отказоустойчивость

Системы CORBA длительное время почти не имели средств поддержания отказоустойчивости. В большинстве случаев о факте ошибки просто сообщалось клиенту, никаких других действий система не предпринимала. Так, например, при невозможности добраться до объекта по ссылке из-за недоступности его сервера (временной) клиент просто ждал. В CORBA версии 3 отказоустойчивостью занялись серьезно. Спецификация отказоустойчивой системы CORBA содержится в [328].

Группы объектов

Базовый подход к обработке ошибок в CORBA состоит в репликации объектов и создании из них *групп объектов (object groups)*. Такая группа содержит одну или несколько абсолютно идентичных копий одного и того же объекта. Адресация группы объектов производится так же, как и одного объекта. Группа имеет тот же самый интерфейс, что и каждая из ее реплик. Другими словами, для клиента репликация прозрачна. Поддерживаются различные стратегии репликации, включая репликацию на основе первичной копии, активную репликацию и репликацию на основе кворума. Все эти стратегии мы обсуждали в главе 6. Существуют также и другие характеристики групп объектов, их детальное рассмотрение можно найти в [328].

Для того чтобы по возможности обеспечить прозрачность репликации и восстановления после сбоев, группы объектов не должны отличаться от обычных объектов CORBA, если приложение не потребует обратного. В связи с этим важно понять, как ссылаться на группы объектов. Применяемая методика состоит в использовании особого типа ссылки IOR, именуемой *межоперационной ссылкой на группу объектов (Interoperable Object Group Reference, IOGR)*. Основное ее отличие от обычной ссылки IOR состоит в том, что каждая ссылка IOGR содержит несколько ссылок на *разные* объекты, а именно на реплики, входящие в одну группу объектов. В противоположность этому ссылка IOR также может содержать несколько ссылок, но все они будут относиться к *одному и тому же* объекту, различаясь, возможно, протоколами доступа.

Когда клиент передает своему брокеру ORB ссылку IOGR, этот брокер пытается выполнить привязку к одной из указанных реплик. В случае протокола IIOP брокер ORB может использовать дополнительную информацию, обнаруженную в одном из профилей IIOP в ссылке IOGR. Эта информация, как мы уже говорили, может храниться в поле компонентов. Например, как показано на рис. 9.13, конкретный профиль IIOP может ссылаться на первичный экземпляр или резервную копию группы объектов посредством разных тегов TAG_PRIMARY и TAG_BACKUP соответственно.

Если привязка к одной из реплик невозможна, ORB клиента может продолжить работу, попытавшись связаться с другой репликой в соответствии с правилом выбора следующей реплики, до получения успешного результата. Для клиента процесс привязки абсолютно прозрачен, для него это выглядит так, как будто он связывается с обычным объектом CORBA.

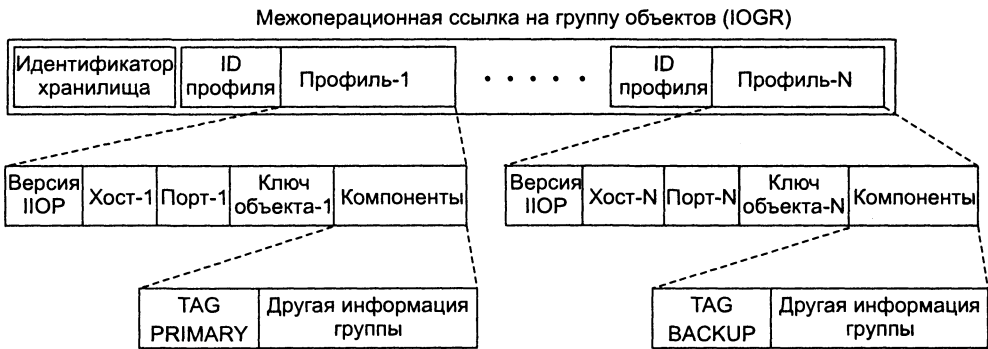


Рис. 9.13. Возможная организация IOGR для группы объектов с первичным экземпляром и резервными копиями

Пример архитектуры

Чтобы поддерживать группы объектов и производить дополнительную обработку отказов, к CORBA необходимо добавить определенные компоненты. Одна из возможных архитектур отказоустойчивой версии CORBA приведена на рис. 9.14. Эта архитектура была применена в системе Eternal [305, 310], которая имеет отказоустойчивую инфраструктуру, построенную поверх системы надежных групповых коммуникаций Totem [304].

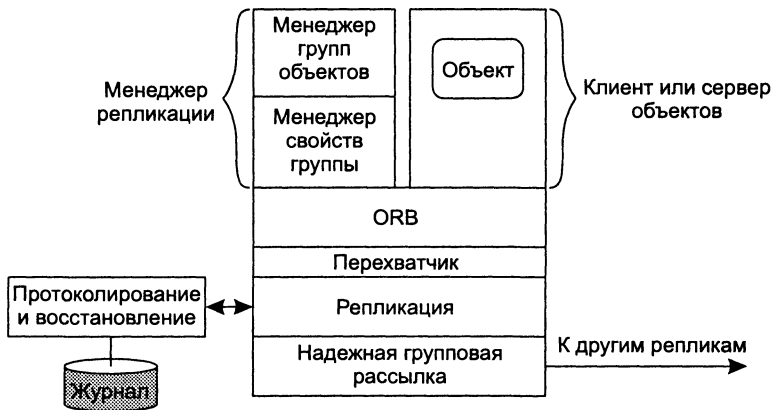


Рис. 9.14. Пример архитектуры отказоустойчивой системы CORBA

В этой архитектуре имеется несколько важных элементов. Наиболее важен *менеджер репликации* (*replication manager*), который отвечает за создание и управление группой реплицированных объектов. В принципе существует только один менеджер репликации, хотя для повышения отказоустойчивости его можно реплицировать.

Как мы установили, для клиента нет разницы между группой объектов и любым другим типом объекта CORBA. Для создания группы объектов клиент просто

обращается к стандартной операции `create_object`, в данном случае менеджера репликации, указывая тип создаваемого объекта. Клиент пребывает в неведении о том, что на самом деле при этом создается группа объектов. Число реплик, создаваемых при запуске новой группы объектов, обычно задается системным значением по умолчанию. Менеджер репликации, кроме того, отвечает за замену реплики в случае ее отказа, гарантируя, таким образом, что число реплик не опустится ниже определенного минимума.

В архитектуре также представлены перехватчики уровня сообщений. В случае системы *Eternal* каждое обращение перехватывается и передается отдельному компоненту репликации, который поддерживает требуемую непротиворечивость групп объектов и гарантирует протоколирование сообщений на случай, если потребуется восстановление системы.

Обращения затем рассылаются остальным членам группы с использованием надежной, полностью упорядоченной групповой рассылки. В случае активной репликации запрос передается каждой из реплик объекта для обработки его каждым брокером ORB объектов. Однако в случае пассивной репликации запрос передается только брокеру ORB исходного экземпляра, в то время как остальные серверы просто протоколируют этот запрос на случай, если потребуется восстановление системы. Когда исходный экземпляр закончит обработку запроса, его состояние рассылается резервным копиям.

9.1.8. Защита

Защита в CORBA имеет долгую историю. Исходные версии спецификаций CORBA вряд ли стоит принимать во внимание, по той простой причине, что отдельные попытки описать службу защиты провалились. В CORBA версии 2.4 описание службы защиты занимает более 400 страниц и четко объясняет, каким образом в системы CORBA вносится защита. Рассмотрим защиту системы CORBA поближе. Обзор по этой теме, написанный одним из авторов исходной спецификации защиты CORBA, можно найти в [66].

Важный момент спецификации защиты CORBA состоит в том, что службы должны предоставлять соответствующий набор механизмов, которые могут использоваться для реализации разного рода правил защиты. Положение осложняется тем, что эти службы могут потребоваться в разных точках и в разное время. Так, например, если клиент желает произвести защищенное обращение к объекту, следует решить, когда использовать механизмы защиты (во время привязки, во время обращения или в обоих этих случаях) и где должны быть использованы эти механизмы (на уровне приложения, внутри ORB или в ходе передачи сообщения).

Ядро защиты в CORBA формируется путем поддержания защищенных обращений к объекту. Базовая идея состоит в том, что объекты прикладного уровня ничего не должны знать об использовании различных служб защиты. Однако если клиент имеет особые требования к защите, у него должен быть способ заявить эти требования, чтобы при обращении к объектам они принимались во внимание. Аналогичная ситуация возникает и с объектом, к которому происходит обращение. Подобный подход ведет к ситуации, представленной на рис. 9.15.

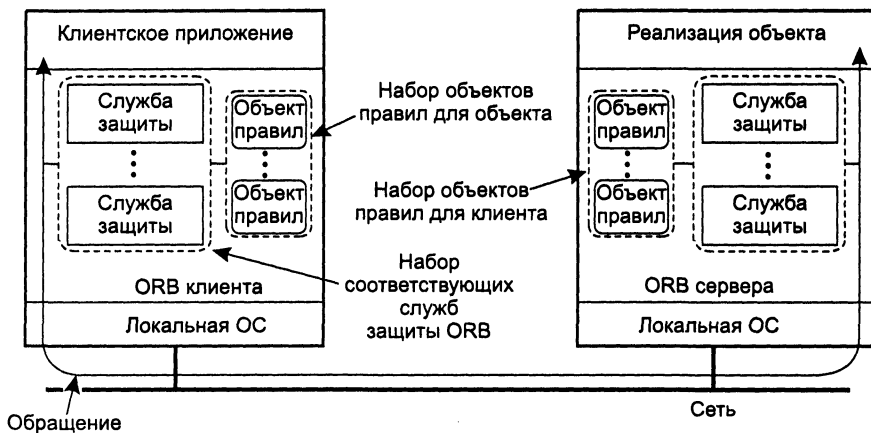


Рис. 9.15. Обобщенная организация защищенного обращения к объекту в CORBA

Когда клиент выполняет привязку к объекту, чтобы потом иметь возможность к нему обращаться, ORB клиента определяет, какие службы защиты со стороны клиента понадобятся для защищенного обращения. Выбор служб определяется правилами защиты, ассоциированными с административным доменом, в котором выполняется клиент, и особо — правилами данного клиента.

Правила защиты задаются *объектами правил* (*policy objects*), ассоциированными с клиентом. На практике домен, в котором исполняется клиент, открывает свои правила защиты для ORB этого клиента посредством указанного набора объектов правил. С клиентом автоматически ассоциируются правила по умолчанию. Примерами объектов правил могут быть те объекты, которые определяют тип применяемой защиты сообщений или хранят списки доверенных сторон. Другие примеры и детали можно найти в [66].

Схожая структура поддерживается и на стороне сервера. Точно так же административный домен, в котором выполняется вызванный объект, задает объекту определенный набор служб защиты. Точно так же вызванный объект имеет дополнительно собственный набор объектов правил, в которых хранится специфическая для данного объекта информация.

Для поддержания защиты в CORBA могут применяться различные подходы. В частности, чтобы сделать брокер ORB по возможности более общим, желательно определить различные службы защиты через стандартные интерфейсы, которые скроют реализацию этих служб. Службы, которые можно определить подобным образом, в CORBA именуются *взаимозаменяемыми* (*replaceable*).

Взаимозаменяемые службы защиты предполагается реализовывать вместе с двумя разными перехватчиками, как показано на рис. 9.16. *Перехватчик контроля доступа* (*access control interceptor*) — это перехватчик уровня запросов, который проверяет права доступа, связанные с обращением. Кроме него имеется *перехватчик защищенных обращений* (*secure invocation interceptor*) уровня сообщений, отвечающий за реализацию защиты сообщений. Другими словами, этот перехватчик занимается шифрованием запросов и ответов, поддерживая их целостность и конфиденциальность.

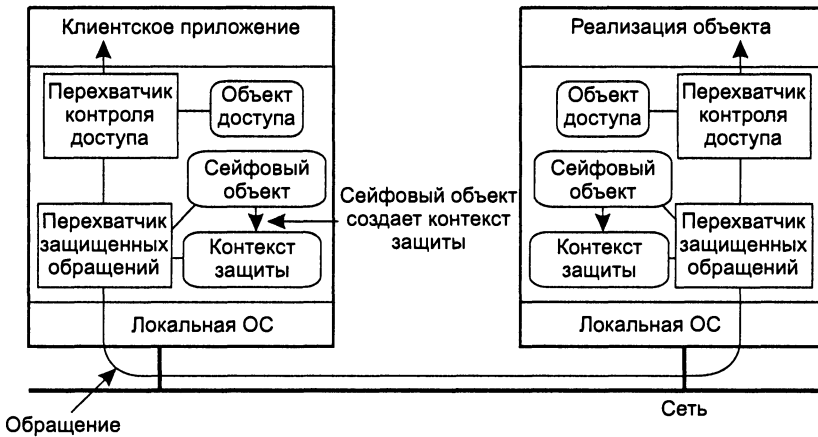


Рис. 9.16. Роль перехватчиков защиты в CORBA

Перехватчик защищенных обращений играет здесь основную роль, поскольку он отвечает за организацию *контекста защиты* (*security context*) клиента, который собирается осуществлять защищенные обращения к нужному ему объекту. Этот контекст защиты, представленный в виде объекта контекста защиты, содержит всю необходимую информацию и методы, необходимые для защищенного обращения к целевому объекту. Так, например, он определяет, какой при этом обращении используется механизм, предоставляет методы шифрования и расшифровки сообщений, хранит ссылки на сертификаты и т. д.

Сервер объекта также создает свои объекты контекста защиты. Перехватчик клиента сначала обычно посылает серверу объекта сообщение, содержащее всю необходимую для аутентификации клиента информацию, таким образом, заставляя сервер создать контекст защиты для последующих обращений. Отметим, что перехватчик защищенных обращений сервера объекта проверяет правила объекта, чтобы проверить, все ли требования соблюдены. Ответ, возвращаемый клиенту, может содержать дополнительную информацию, которая позволит клиенту аутентифицировать сервер.

После этого исходного обмена сообщениями клиент выполняет привязку к нужному объекту, и вместе они создают то, что обычно называется *ассоциацией защиты* (*security association*). После этого можно выполнять защищенные обращения, и перехватчики защищенных обращений будут оберегать запросы и ответы в соответствии с правилами, о которых договорились между собой клиент и сервер объекта.

Критически важную роль в организации ассоциации защиты играет отдельный объект со стандартизованным интерфейсом, именуемый *сейфовым объектом* (*vault object*). Сейфовый объект вызывается перехватчиком защищенных сообщений для создания объекта контекста защиты. Перехватчик читает информацию о правилах из ассоциированных с клиентом объектов правил и передает эту информацию в сейфовый объект. Понятно, что сейфовый объект должен быть реализован в виде части ORB, чтобы его невозможно было подделать, и относится к доверенной вычислительной базе каждой системы CORBA.

9.2. DCOM

Вторая распределенная система объектов, которую мы рассмотрим, — это *распределенная модель COM (Distributed COM, DCOM)* корпорации Microsoft. Как можно понять из ее названия, модель DCOM выросла из *модели компонентных объектов (Component Object Model, COM)*. COM — это технология, лежащая в основе различных версий операционных систем Windows от Microsoft, начиная с Windows 95. В противоположность CORBA, DCOM не является результатом деятельности комитета. Это, в частности, видно из того факта, что существует всего лишь 300-страничное черновое описание COM, датированное октябрём 1995 года [292]. К сожалению, то, что модель DCOM избежала обсуждения в комитете, повлекло за собой отсутствие хорошо спроектированной архитектуры с минимальным набором базовых элементов, из которых строятся компоненты и службы. Наоборот, в настоящее время DCOM — крайне запутанная система, в которой множество сходных действий выполняются множеством разных способов, и подобное сосуществование различных решений временами кажется невозможным.

Критиковать модель DCOM нетрудно. Однако сравнивая ее с CORBA, можно обоснованно утверждать, что DCOM — это технология, которая в значительной степени доказала свое право на существование. Десятки миллионов человек каждый день используют Windows в сетевой среде, а значит, DCOM — широко распространенная система. В этом смысле CORBA или любым другим распределенным системам до нее еще идти и идти.

Далее мы рассмотрим наиболее важные части DCOM, следуя тому же порядку изложения материала, которого мы придерживались при изучении CORBA. Чтобы понять, что такое DCOM, лучше всего обратиться к одной из книг для программистов, такой как [357] или [386]. Хорошее введение в различные технические аспекты DCOM имеется в [132]. Обзор DCOM с точки зрения распределенной модели Windows 2000 можно найти в [92].

9.2.1. Обзор

Как уже упоминалось, модель DCOM базируется на модели COM. Целью создания COM была поддержка разработки компонентов, которые могли бы динамически активизироваться и взаимодействовать друг с другом. Компонент в COM — это исполняемый код, содержащийся в динамически компоуемой библиотеке (DLL) или исполняемой программе.

Сама по себе модель COM существует в виде библиотек, компоуемых с процессом. Изначально она разрабатывалась для поддержки так называемых *составных документов (compound documents)*. Как мы говорили в главе 3, составные документы — это документы, построенные из разнородных частей, таких как текст (форматированный), изображения, электронные таблицы и т. п. Каждая из этих частей может быть отредактирована при помощи ассоциированного с ней приложения.

Для поддержки бесчисленного множества составных документов Microsoft нужен был обобщенный метод для разделения отдельных частей и объединения

их в единую сущность. Вначале это привело к технологии *связывания и внедрения объектов (Object Linking and Embedding, OLE)*. Первая версия OLE использовала для передачи информации между частями документа примитивный и негибкий способ обмена сообщениями. Вскоре она была заменена следующей версией, также называвшейся OLE, но построенной на базе более гибкого механизма COM, что привело к появлению структуры, представленной на рис. 9.17.

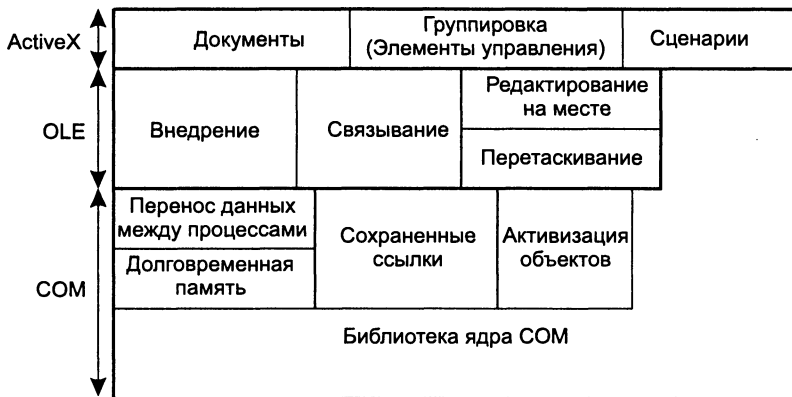


Рис. 9.17. Общая структура ActiveX, OLE и COM

На рисунке также показан элемент *ActiveX* — этим термином в настоящее время называют все, что относится к OLE, плюс некоторые новшества, к которым относят гибкую (как правило) способность компонентов выполняться в разных процессах, поддержку сценариев и более или менее стандартную группировку объектов в так называемые элементы управления ActiveX. Среди экспертов по DCOM (даже внутри Microsoft) не существует согласия по вопросу о точном определении ActiveX, поэтому мы даже не будем пытаться дать подобное определение.

Модель DCOM добавила к этой структуре весьма существенную вещь — способность процесса работать с компонентами, размещенными на другой машине. Однако базовый механизм обмена информацией между компонентами, принятый в DCOM, очень часто совпадает с соответствующими механизмами COM. Другими словами, для программиста разница между COM и DCOM часто скрыта за различными интерфейсами. Как мы увидим, DCOM в первую очередь предоставляет прозрачность доступа. Другие виды прозрачности менее очевидны.

Объектная модель

Как фактически и все остальные распределенные системы объектов, DCOM соответствует модели удаленных объектов. Фактически объекты в DCOM могут размещаться как в одном процессе с клиентом, так и на одной машине с ним, а также в процессе, выполняемом на удаленной машине. Позже мы рассмотрим все эти варианты.

Как и в CORBA, объектная модель в DCOM построена на реализации интерфейсов. Грубо говоря, объект DCOM — это просто реализация интерфейса. Один объект может реализовывать одновременно несколько интерфейсов. Однако в отличие от CORBA в DCOM имеются только *бинарные интерфейсы* (*binary interfaces*). Такой интерфейс, в сущности, представляет собой таблицу с указателями на реализации методов, которые являются частью интерфейса. Разумеется, для определения интерфейса по-прежнему удобно использовать специальный язык определения интерфейса (IDL). В DCOM также имеется такой язык под названием *MIDL* (*Microsoft IDL — язык IDL от Microsoft*), при помощи которого генерируются бинарные интерфейсы стандартного формата.

Достоинство бинарных интерфейсов состоит в том, что они не зависят от языка программирования. В случае CORBA каждый раз для поддержки нового языка программирования необходимо отображать описания IDL на конструкции этого языка. Подобная стандартизация в случае бинарных интерфейсов не нужна. Разницу между этими двумя подходами иллюстрирует рис. 9.18.

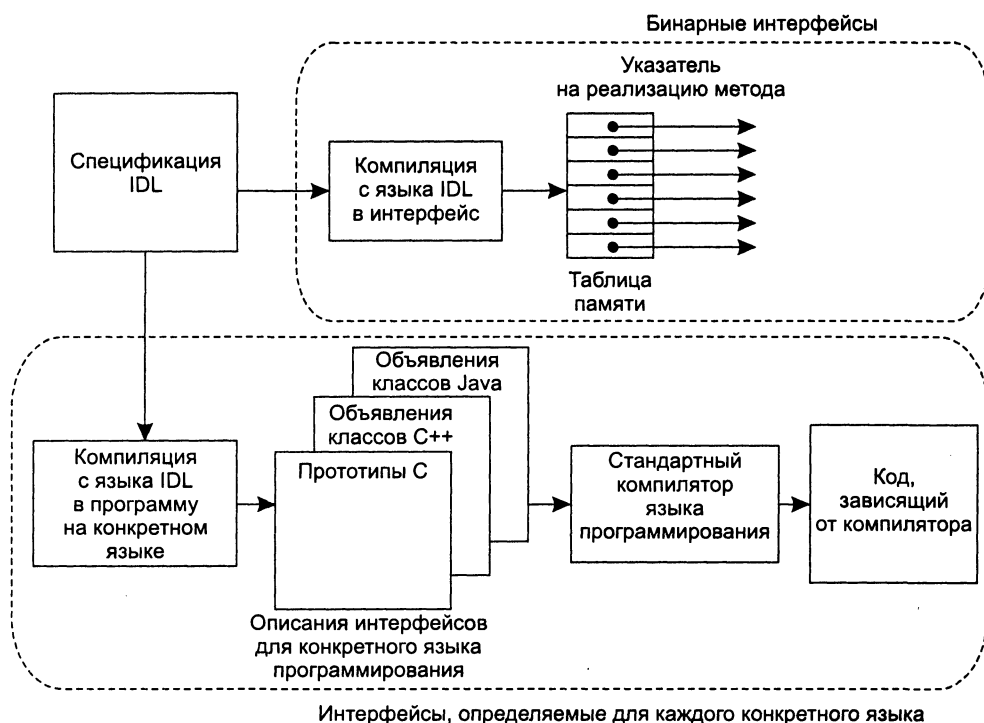


Рис. 9.18. Разница между интерфейсами, определяемыми для каждого конкретного языка, и бинарными интерфейсами

Каждый интерфейс в DCOM имеет уникальный 128-битный идентификатор, который называется *идентификатором интерфейса* (*Interface Identifier, IID*). Каждый IID абсолютно уникален, не существует двух интерфейсов с одинаковым

идентификатором IID. Этот идентификатор создается путем комбинирования большого случайного числа, локального времени и адреса сетевого интерфейса текущего хоста. Вероятность того, что два сгенерированных индикатора совпадут, практически равна нулю.

Объект DCOM создается как экземпляр класса. Чтобы создать такой объект, необходимо иметь доступ к соответствующему классу. Для этой цели DCOM содержит *объекты класса (class objects)*. Формально таким объектом может быть все, что поддерживает интерфейс IClassFactory. Этот интерфейс содержит метод CreateInstance, который похож на оператор new в языках C++ и Java. Вызов метода CreateInstance для объекта класса приводит к созданию объекта DCOM, содержащего реализацию интерфейсов объекта класса.

Таким образом, объект класса представляет собой коллекцию объектов одного типа, то есть реализующих один и тот же набор интерфейсов. Объекты, принадлежащие к одному классу, обычно различаются только в части текущего состояния. Путем создания экземпляров объекта класса становится возможным обращение к методам этих интерфейсов. В DCOM на любой объект класса можно сослаться по его глобальному уникальному *идентификатору класса (Class Identifier, CLSID)*.

Все объекты реализуют стандартный объектный интерфейс IUnknown. Когда путем вызова CreateInstance создается новый объект, объект класса возвращает указатель на этот интерфейс. Самый важный метод, содержащийся в IUnknown, — метод QueryInterface, который на основании IID возвращает указатель на другой интерфейс, реализованный в объекте.

Важное отличие от объектной модели CORBA состоит в том, что все объекты в DCOM нерезидентные. Другими словами, как только у объекта не остается ссылающихся на него клиентов, этот объект удаляется. Подсчет ссылок производится путем вызова методов AddRef и Release, входящих в интерфейс IUnknown. Наличие исключительно нерезидентных объектов делает невозможным хранение каждым из объектов своего глобально уникального идентификатора. По этой причине объекты в DCOM могут вызываться только по указателям на интерфейсы. Соответственно, как мы увидим позднее, для передачи ссылки на объект другому процессу необходимо предпринять специальные меры.

DCOM также требует динамического обращения к объектам. Объекты, запросы к которым могут создаваться динамически, во время выполнения, должны иметь реализацию интерфейса IDispatch. Этот интерфейс подобен интерфейсу динамического обращения (DII) в системе CORBA.

Библиотека типов и системный реестр

Эквивалент хранилища интерфейсов CORBA в DCOM называется *библиотекой типов (type library)*. Библиотека типов обычно ассоциируется с приложением или другим крупным компонентом, содержащим различные объекты классов. Сама по себе библиотека может храниться в отдельном файле или быть частью приложения. В любом случае библиотека типов в первую очередь требуется для точного описания сигнатуры динамически вызываемых методов. Кроме того, библиотеки типов могут использоваться в системах программирования, например, для отображения структуры разрабатываемых интерфейсов на экране.

Для того чтобы действительно активизировать объект, то есть гарантировать, что он создан и помещен в процесс, из которого будет в состоянии принимать обращения к методам, DCOM использует реестр Windows в комбинации со специальным процессом — *менеджером управления службами* (*Service Control Manager, SCM*). Кроме всего прочего, реестр используется для записи отображения идентификаторов классов (CLSID) на локальные имена файлов, содержащих реализации классов. Чтобы создать объект, процесс сначала должен убедиться, что соответствующий объект класса загружен.

Если объект выполняется на удаленном хосте, дело происходит иначе. В этом случае клиент контактирует с менеджером управления службами этого хоста, который представляет собой процесс, ответственный за активизацию объектов, подобно хранилищу реализаций в CORBA. SCM этого удаленного хоста просматривает свой локальный реестр в поисках файла, ассоциированного с CLSID, после чего запускает процесс, содержащий этот объект. Сервер выполняет маршalling указателя на интерфейс и возвращает его клиенту, который выполняет демаршalling указателя и передает его заместителю.

Общая архитектура DCOM, включая объекты классов, реальные объекты и заместители, представлена на рис. 9.19 [105]. Процесс клиента получает доступ к SCM и реестру, что помогает ему найти удаленный объект и выполнить привязку к нему. Клиенту предлагается заместитель, реализующий интерфейсы этого объекта.

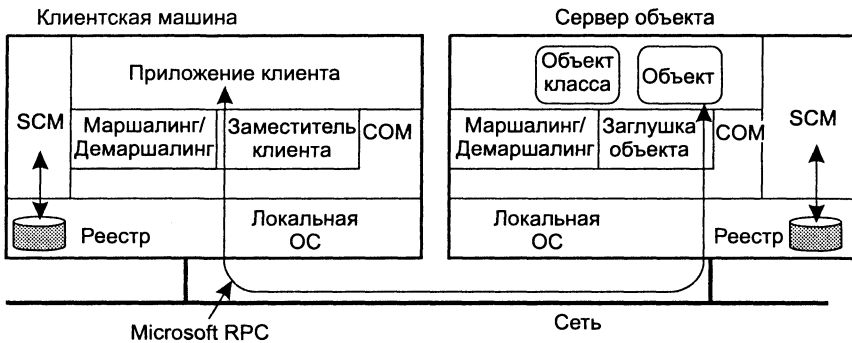


Рис. 9.19. Общая архитектура DCOM

Сервер объектов содержит заглушку для маршallingа и демаршallingа обращений, которые он будет передавать реальному объекту. Связь между клиентом и сервером обычно реализуется путем вызовов RPC, но, как мы увидим чуть ниже, в определенных случаях могут быть использованы и другие способы связи.

Службы DCOM

Модель DCOM можно рассматривать как расширенную за счет добавления средств сообщения с удаленными объектами модель COM. Однако COM и сама по себе стала иной, и новая ее версия называется COM+. Модель COM+ тоже можно рассматривать как расширенный вариант модели COM, содержащий раз-

личные службы, которые ранее были полезными дополнениями к COM. Так, в частности, COM+ включает расширения, которые позволяют серверу успешно поддерживать одновременно несколько объектов. Кроме того, добавлены службы, реализуемые на внешних серверах, например системы очередей сообщений.

Провести черту между COM, DCOM, COM+ и внешними службами часто бывает затруднительно, и подобные попытки лишь способствуют путанице. Далее мы будем просто ссылаться на DCOM как на расширение, включающее и COM, и COM+, и ActiveX. Там, где это возможно и имеет смысл, мы будем разделять службы, просто помогающие DCOM, и компоненты, фактически являющиеся частями DCOM. Чтобы понять, какие службы непосредственно входят в DCOM, рассмотрим табл. 9.4, в которой перечислены службы CORBA и соответствующие им службы DCOM. Также в этой таблице приводятся службы Windows 2000, обычно доступные клиентам модели DCOM, но не являющиеся ее частями.

Таблица 9.4. Обзор служб CORBA, DCOM и Windows 2000

Служба CORBA	Служба DCOM/COM+	Служба Windows 2000
Служба коллекций	Объекты данных ActiveX	—
Служба запросов	Нет	—
Служба параллельного доступа	Параллельные потоки выполнения	—
Служба транзакций	Автоматические транзакции COM+	Координатор распределенных транзакций
Служба событий	События COM+	—
Служба уведомлений	События COM+	—
Служба внешних связей	Утилиты маршалинга	—
Служба жизненного цикла	Фабрики классов, JIT-активизация	—
Служба лицензирования	Специальные фабрики классов	—
Служба именования	Моникеры	Active Directory
Служба свойств	—	Active Directory
Служба обмена	—	Active Directory
Служба сохранности	Специальное хранилище	Доступ к базам данных
Служба отношений	—	Доступ к базам данных
Служба защиты	Авторизация	SSL, Kerberos
Служба времени	—	—

Ниже мы рассмотрим большую часть этих служб. Следует отметить, что сама по себе модель DCOM, в отличие от CORBA, не является законченной распределенной системой, поскольку требует наличия внешних служб. Так, например, очевидно, что служба именования является неотъемлемой частью распределенной системы. Именованное в Windows 2000 поддерживается при помощи службы Active Directory. Эта служба, в сущности, состоит из набора серверов каталогов

LDAP, которые именуются и просматриваются при помощи системы DNS. Мы говорили о DNS и LDAP в главе 4. Однако Active Directory не является частью DCOM. Соответственно, при запуске приложений DCOM в среде UNIX эти приложения не в состоянии использовать ту же самую службу именования, что и в среде Windows 2000. У переносимости всегда есть некоторые границы.

Очевидно, что подобный подход препятствует межоперационной совместимости. С другой стороны, учитывая, что фактически все приложения DCOM рассчитаны на работу в среде Windows, непонятно, насколько этот недостаток совместимости действительно можно считать проблемой.

9.2.2. Связь

Как и в CORBA, в базовом варианте DCOM поддерживает только синхронное взаимодействие: клиент обращается к объекту и блокируется до получения ответа. Синхронные обращения такого рода поддерживаются в системе по умолчанию. Однако, как мы увидим далее, DCOM обслуживает также и другие формы взаимодействия.

Модели обращения к объектам

Как мы говорили, синхронные обращения являются стандартными для DCOM. Если что-нибудь пойдет не так, как ожидалось (а в распределенных системах это обычное дело), клиент получает код ошибки. DCOM не может самостоятельно повторить обращение к объекту, поскольку использует семантику «максимум однажды».

Одно из самых ранних расширений «чистых» синхронных обращений — интерфейсы обратного вызова. Интерфейсы обратного вызова поддерживаются *связующими объектами* (*connectable objects*). Это специальные объекты DCOM, которые могут содержать указатели на интерфейсы обратного вызова, реализуемые клиентами. Связующий объект описывает интерфейс обратного вызова, реализацию которого он ожидает найти на стороне клиента. Когда клиент выполняет привязку к связующему объекту, он должен предоставить ему указатель на интерфейс обратного вызова. Если связующий объект находится на удаленном хосте, для передачи ему указателя на интерфейс сначала нужно выполнить маршалинг. Маршалинг указателей на интерфейсы мы рассмотрим чуть ниже.

После появления модели COM+ и интеграции ее в систему Windows 2000 в DCOM появилась поддержка и других моделей обращения. Одно из таких расширений — возможность отмены затянувшегося синхронного вызова. Когда поток выполнения *A* начинает синхронное обращение, создается *объект отмены* (*cancel object*), реализующий метод `Cancel`. Этот метод может быть вызван из другого потока выполнения *B* с передачей в качестве параметра идентификатора потока выполнения *A*. В результате поток выполнения *A* будет немедленно разблокирован.

Также поддерживаются и асинхронные вызовы, соответствующие модели опроса CORBA для асинхронного обращения к методам. На самом деле, подход, связанный с созданием интерфейсов, поддерживающих асинхронные вызовы,

имеет определенные отличия от модели опроса CORBA. Взяв за отправную точку описание интерфейса на MIDL, компилятор MIDL для каждого метода *m* в этом интерфейсе генерирует два других метода — *Begin_m* и *Finish_m*. Первый из них содержит только входные параметры метода *m*, а второй — только его выходные параметры.

Клиент начинает взаимодействие с вызова метода *Begin_m*, после чего немедленно продолжает свою работу. Для объекта нет никакой разницы между синхронными и асинхронными вызовами, единственное, что он может увидеть, — это обращение к методу *m*. Результаты этого обращения пересылаются обратно клиенту, где буферизуются до момента вызова клиентом метода *Finish_m*.

Асинхронные вызовы в DCOM требуют, чтобы клиенты и объекты были активны и работали. Другими словами, связь является нерезидентной. Сохранная связь поддерживается с помощью очередей сообщений, о которых мы поговорим чуть позже.

События

Изначально для реализации событий в DCOM использовались связующие объекты, однако при этом требовалось, чтобы и клиент, и объект были активны. Однако также имелась возможность соединения нескольких клиентов с одним связующим объектом, причем по их требованию объект мог выполнить обратный вызов всех клиентов одновременно. В этом смысле можно сказать, что связующие объекты предоставляют серверам простой способ раздачи событий нескольким клиентам.

DCOM средствами COM+ поддерживает более сложную (и более практичную) модель событий, известную под названием *систем публикации/подписки* (*publish/subscribe systems*). Мы будем ее обсуждать в главе 12. Основная идея примерно та же, что и в модели продвижения сообщений CORBA. Событие моделируется путем вызова метода, имеющего только входные параметры. События группируются в *класс событий* (*event class*), который представляет собой обычный объект класса DCOM со своим идентификатором CLSID. Нет никакой необходимости реализовывать класс событий самостоятельно: в DCOM для него имеется стандартная реализация. После создания экземпляра класса событий поставщик событий может просто генерировать их вызовом соответствующих методов этого объекта. Отметим, что само событие объектом не является.

Подписка на события также проста. Предположим, что событие представлено методом *m_event*. Подписчик поддерживает реализацию этого метода. Для подписки на событие он передает указатель на интерфейс, реализующий этот метод, в систему событий. Таким образом, когда поставщик вызывает *m_event*, система событий отвечает за то, чтобы при этом произошли также обращения к версии *m_event* каждого подписчика. Рисунок 9.20 иллюстрирует принципы, лежащие в основе механизма событий DCOM. Очевидно, этот подход действительно похож на продвижение событий в системе CORBA.

События в DCOM можно сохранять. Благодаря такой возможности, даже когда подписчик в момент события неактивен, он все равно сможет, если пожелает, получить событие позднее. Для этого система событий создает экземпляр класса

событий поставщика и воспроизводит очередь событий, вызывая соответствующие методы один за другим. В результате подписчик вызывает ассоциированные с этими событиями методы так, как если бы он имел дело с обычными резидентными событиями.

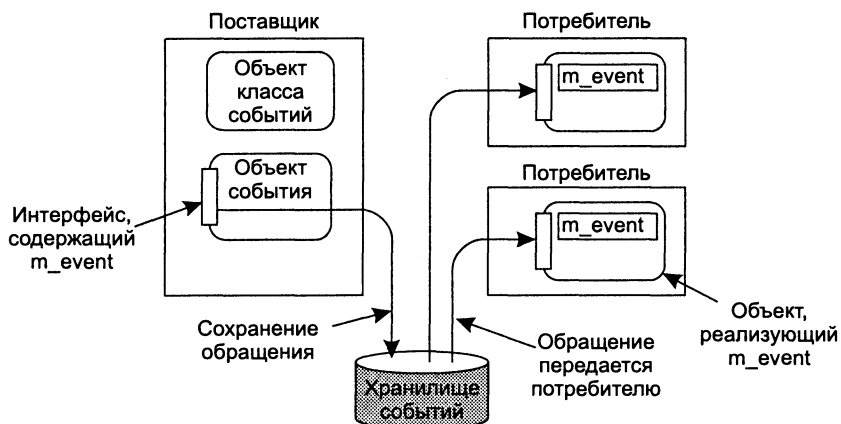


Рис. 9.20. Обработка событий в DCOM

Передача сообщений

Кроме нерезидентной асинхронной связи DCOM также поддерживает и сохраненную асинхронную связь, используя для этого компоненты очередей (*Queued Components, QC*). QC — это на самом деле интерфейс к системе очередей сообщений *Microsoft (Microsoft Message Queuing, MSMQ)*, очень похожей на систему MQSeries компании IBM, которую мы обсуждали в главе 2. Назначение QC — полностью скрыть MSMQ и от клиентов, и от объектов. Взамен им предоставляются точно такие же обращения к методам, как и при асинхронных вызовах, о которых мы уже говорили. Основная деятельность компонентов очередей происходит следующим образом.

Асинхронные вызовы в случае QC урезаются до интерфейсов, содержащих методы, которые имеют только входные параметры. Методы, содержащие возвращаемые значения или выходные параметры, недопустимы. Другими словами, QC допускает только однонаправленное обращение к асинхронным методам, что концептуально совпадает с подходом, принятым в асинхронных вызовах RPC.

Когда клиент выполняет привязку к объекту, поддерживающему QC, объект возвращает указатель на интерфейс, содержащий методы, которые можно вызывать асинхронно. Каждый раз, когда клиент обращается к одному из этих методов, автоматически выполняетсяmarshalingи локальное сохранение обращения в подсистеме QC клиента. После того как клиент завершает обращение к методам и демонстрирует это, освобождая интерфейс вызовом метода *Release*, сохраненные методы передаются объекту через MSMQ.

После прихода сообщений с обращениями к месту назначения подсистема QC выполняет демаршалинг каждого обращения, после чего вызывает объект. Ника-

ких гарантий того, что обращения будут обрабатываться в том же порядке, в котором они выполнялись, нет, если только они не являются частями транзакции. На самом деле не гарантируется даже доставка, если это не было особо оговорено.

MSMQ, как и большинство систем очередей сообщений (например, MQSeries), поддерживает *транзакционные очереди* (*transactional queues*). В своей простейшей форме транзакционная очередь состоит из коллекции сообщений, которая передается в другую очередь (или к месту назначения) либо вся целиком, либо не передается ни одно из сообщений. Кроме того, транзакционная очередь гарантирует восстановление сообщений после сбоя. Существуют также и более совершенные формы транзакций, при которых операции в одной транзакционной очереди представляют собой на самом деле вложенную транзакцию большой распределенной транзакции. В таких случаях применяются выделенные координаторы, о которых мы говорили в главе 7.

9.2.3. Процессы

Модель удаленных объектов DCOM на первый взгляд кажется относительно несложной. Однако углубившись в детали, мы обнаружим, что некоторые вещи не так просты, как кажутся. Далее мы рассмотрим организацию клиентов и серверов DCOM.

Клиенты

Поддержка клиента в модели DCOM — это ее естественное предназначение. Для клиента все выглядит почти так, как если бы работающий объект находился в его собственном адресном пространстве. Другими словами, объекты создаются и вызываются точно так же, как это делалось до появления DCOM, когда существовала только модель COM.

Однако имеются некоторые моменты, требующие особого внимания. Один из наиболее важных вопросов подобного рода — обработка ссылки на объект. Напомним, что в DCOM единственной ссылкой на объект является указатель на интерфейс. Этот указатель используется клиентом для обращения к методам интерфейса. Если сам объект располагается на удаленной машине, интерфейс на стороне клиента реализован в виде заместителя, который выполняет маршрутинг обращения и пересылает запросы объекту.

Как процесс *A* может передать ссылку на объект другому процессу, скажем *B*? Понятно, что пересылка его указателя на интерфейс к успеху не приведет. Процессу *B* нужен, в сущности, указатель на реализацию того же интерфейса, который имеется у процесса *A*. Этот интерфейс имеет уникальный идентификатор — IID. Другими словами, для передачи указателя на интерфейс на самом деле необходимо передать IID этого интерфейса. Кроме того, процесс *A* может передать *B* информацию о привязке и об используемом транспортном протоколе. Предполагая, что процесс *B* имеет локальную реализацию заместителя для переданного ему интерфейса, он может просто загрузить эту реализацию и инициировать вызов нужного метода.

Большинство заместителей реализуют стандартный подход к маршалингу запросов на обращение. Такие заместители автоматически генерируются по спецификации MIDL или могут быть созданы по информации, содержащейся в библиотеках типов. Однако бывают случаи, когда стандартный маршалинг — не лучший способ обработки запросов. Так, например, может возникнуть необходимость кэширования на заместителе результатов предшествующих обращений. Включение функции кэширования в заместитель должно программироваться разработчиком вручную.

Кроме стандартного маршалинга DCOM поддерживает пользовательский маршалинг, при котором разработчик может полностью определять, как заместителю следует обрабатывать соответствующие объекты. Вопрос в том, как организовать передачу ссылок при наличии пользовательских заместителей. При стандартном маршалинге решение состоит в том, что весь код маршалинга содержится в стороне, выполняющей маршалинг, поэтому передача IID интерфейса требует лишь загрузки этого кода и построения заместителя интерфейса.

По существу, передача ссылки на объект в случае пользовательского маршалинга происходит путем маршалинга заместителя, принадлежащего отправителю, передачи этого заместителя получателю и последующего демаршалинга полученного заместителя из сообщения. То есть заместитель отправителя в конечном итоге оказывается у получателя. Отметим, что подобный подход очень похож на передачу ссылок на объекты в Java, о которой мы говорили в главе 2. Поскольку маршалинг выполняется способом, специфичным для данного объекта, необходимо, чтобы процесс-получатель имел соответствующий код для демаршалинга пришедшего потока байтов, представляющего собой, по сути, заместителя после маршалинга (то есть после своего рода упаковки). Предполагается, что этот код доступен получателю и идентифицируется по его идентификатору CLSID. Передачу ссылки на объект в DCOM иллюстрирует рис. 9.21.

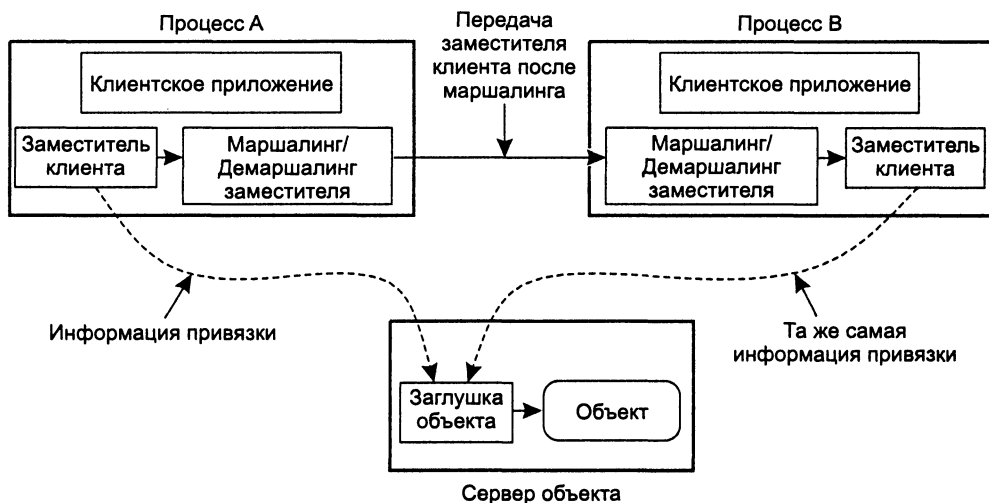


Рис. 9.21. Передача ссылки на объект в DCOM при пользовательском маршалинге

Важно отметить небольшую разницу между кодом, используемым для маршалинга и демаршалинга заместителей, и кодом маршалинга, выполняемого самим заместителем. В последнем случае маршалинг обращений к методам, содержащимся в интерфейсе, выполняет заместитель. То есть заместитель отвечает за преобразование в сообщение обращения к методу (маршалинг) и, соответственно, за извлечение данных из сообщения, содержащего результаты этого обращения (демаршалинг). Маршалинг заместителя производится стандартным образом, то есть код заместителя преобразуется в набор байтов и передается по сети.

Серверы DCOM

Предполагается, что объекты DCOM являются более или менее самодостаточными в том смысле, что различные возможности, которые в CORBA предоставляет переносимый адаптер объектов, жестко закодированы в каждом объекте DCOM. DCOM предлагает стандартный способ активизации объектов, в том числе и размещенных на удаленном хосте.

На хосте, поддерживающем объекты DCOM, должен выполняться уже упоминавшийся ранее *менеджер управления службами (Service Control Manager, SCM)*. SCM отвечает за активизацию объектов, которая производится следующим образом. Предположим, что клиент владеет CLSID объекта класса. Чтобы создать новый объект, он передает этот идентификатор CLSID в свой локальный реестр, где выполняется поиск хоста, на котором должен находиться нужный сервер объектов. Затем CLSID передается менеджеру SCM, ассоциированному с найденным хостом. В поисках файла, позволяющего загрузить объект класса, который может создавать экземпляры нового объекта, SCM ищет этот идентификатор CLSID уже в своем локальном реестре.

SCM обычно начинает выполнение нового процесса с загрузки найденного объекта класса, после чего создает экземпляр объекта. В SCM регистрируется также порт, из которого «свежеиспеченный» сервер будет получать входящие запросы, а также идентификатор нового объекта. Эта информация привязки возвращается клиенту, который после ее получения может работать с сервером объекта напрямую, без вмешательства SCM.

Такой подход оставляет решение всех вопросов, связанных с управлением объектами на сервере, разработчику. Чтобы немного облегчить ему жизнь, DCOM предоставляет средства управления объектами. Одно из таких средств носит название *активизации «на лету» («just-in-time» activation)*, или *JIT-активизации (JIT-activation)*. Под этим названием скрывается механизм, при помощи которого сервер может управлять активизацией и удалением объектов. Он работает следующим образом.

Обычно объект активизируется в результате запроса клиента на создание экземпляра нового объекта данного класса. Сервер объекта хранит объект в памяти до тех пор, пока существуют клиенты, хранящие ссылку на этот объект. Однако при JIT-активизации сервер может удалить объект тогда, когда захочет. Так, например, если сервер обнаруживает, что у него не осталось памяти, он может освободить место для новых объектов, удалив старые.

Когда клиент вызывает метод ранее удаленного объекта, сервер немедленно создает новый объект того же класса и обращается к указанному методу этого объекта. Разумеется, такой подход допустим только при работе с объектами без фиксации состояния, то есть объектами, которые в паузах между обращениями не хранят никакой информации о своем внутреннем состоянии. Все, что остается для работы с таким объектом, — его методы. Подобные объекты использовались для вызова библиотечных функций еще в те дни, когда не придумали объектно-ориентированного программирования. Если же предполагается сохранять информацию о состоянии объекта, после каждого обращения ее придется записывать на диск и при каждом создании объекта вновь загружать.

9.2.4. Именование

Модель DCOM сама по себе предоставляет объектам лишь относительно низкоуровневые средства именования. Ранее мы уже обсуждали, каким образом в качестве примитивных ссылок на объекты можно использовать указатели на интерфейсы. Указатель на интерфейс имеет смысл только в тех процессах, в которых он используется и может быть в любой момент удален процессом. В дополнение к указателям на интерфейсы DCOM предоставляет также ссылки на сохраненные объекты, которые могут совместно использоваться несколькими процессами. Эти ссылки, известные под названием *моникеров (monikers)*, мы и обсудим. DCOM не имеет высокоуровневой службы именования. Однако, являясь частью операционной системы Windows 2000, объекты DCOM могут пользоваться службой Active Directory Windows, которая является настоящей службой каталогов. Здесь мы также рассмотрим и эту службу.

Моникеры

В противоположность CORBA базовая объектная модель DCOM поддерживает только нерезидентные объекты. Другими словами, когда в объекте больше нет необходимости, он удаляется. Чтобы продлить жизнь объекта, когда его перестанет использовать последний клиент, в DCOM применяются сохраненные ссылки. Подобная ссылка, называемая в DCOM *моникером*, может быть сохранена на диске. Моникер содержит всю информацию, необходимую для воссоздания соответствующего объекта и приведения его в то состояние, которое он имел перед тем, как его перестал использовать последний клиент.

DCOM предоставляет различные варианты моникеров. Наиболее важным из них является *файловый моникер (file moniker)*, ссылающийся на объект, создаваемый из файла локальной файловой системы. В этом случае моникер должен содержать полное имя файла, используемого для создания объекта. Кроме того, он должен содержать также CLSID объекта класса, необходимого для создания объекта этого класса.

Как и все моникеры, файловый моникер поддерживает операцию `BindToObject`, вызываемую клиентом для привязки к объекту, с которым связан моникер. Порядок привязки в случае файлового моникера иллюстрирует табл. 9.5.

Таблица 9.5. Привязка к объекту DCOM при помощи файлового моникера

Шаг	Исполнитель	Описание
1	Клиент	Вызов метода BindToObject моникера
2	Моникер	Поиск соответствующего идентификатора CLSID и указание SCM создать объект
3	SCM	Загрузка объекта класса
4	Объект класса	Создание объекта и возвращение моникеру указателя на интерфейс
5	Моникер	Указание объекту загрузить ранее сохраненное состояние
6	Объект	Загрузка из файла своего состояния
7	Моникер	Возвращение клиенту указателя на интерфейс объекта

Первое, что следует понимать, моникер — это сохраненный объект, то есть объект длительного хранения. В терминологии DCOM это означает, что любой моникер содержит методы для сохранения своего содержимого на диске. Во многих случаях приложение может потребовать от моникера, чтобы тот сохранил себя в виде части файла. Позже, прочитав этот файл, можно будет воссоздать по данным моникера настоящий объект. В этот момент клиент получит в свое распоряжение интерфейс IMoniker, содержащий операцию BindToObject.

Предположим, что клиент только что прочитал с диска файловый моникер и хочет отправить запрос некоторому интерфейсу того объекта, ссылкой на который является моникер. При обращении к методу BindToObject этого моникера его код отыскивает объект класса ассоциированного с моникером объекта в локальном реестре. Этот объект класса идентифицируется по идентификатору CLSID, который является частью моникера. В соответствии с той же процедурой, которую мы обсуждали ранее, говоря о создании экземпляра объекта, CLSID хранится в локальном реестре клиента. После того как он найден, CLSID передается соответствующему менеджеру SCM, который отвечает за создание объекта этого класса.

Моникер связывает код, создаваемый при инициализации созданного объекта, с данными, содержащимися в файле, указанном в моникере. В случае файлового моникера это предполагает, что объект получает интерфейс, реализующий операцию загрузки файла. Другими словами, сам объект знает, как восстановить свое предшествующее состояние при помощи загружаемых из файла данных. Отметим, что этот файл не имеет никакого отношения к файлу, в котором хранится сам моникер.

И, наконец, клиенту возвращается указатель на изначально запрошенный им интерфейс, что и завершает операцию привязки.

В качестве побочного результата привязки моникер регистрирует связанный с ним объект в *таблице запущенных объектов (Running Object Table, ROT)* клиентской машины. Когда бы другой клиент ни использовал моникер для привязки к объекту, указанному ссылкой, моникер первым делом ищет этот объект в ROT. Если он обнаруживает, что объект класса уже создан, он выполняет привязку клиента к этому экземпляру. Таким образом, достигается совместное ис-

пользование объектов разными процессами, работающими на одной и той же машине.

Кроме файловых моникеров DCOM поддерживает несколько других типов моникеров, некоторые из них представлены в табл. 9.6. Кроме этих заранее определенных типов моникеров можно также создавать пользовательские моникеры. Детальное описание вопросов, связанных с реализацией моникеров, можно найти в [132].

Таблица 9.6. Типы моникеров, определенные в DCOM

Тип моникера	Описание
Файловый моникер	Ссылка на объект, создаваемый из файла
Моникер URL	Ссылка на объект, создаваемый из URL
Моникер класса	Ссылка на объект класса
Композитный моникер	Ссылка на композицию моникеров
Моникер элемента	Ссылка на моникер, входящий в композицию
Моникер указателя	Ссылка на объект удаленного процесса

Active Directory

С появлением Windows 2000 приложения DCOM получили возможность использовать полноценную глобальную службу каталогов, известную под названием *Active Directory*. Напомним, что служба каталогов позволяет клиентам производить поиск некоторых сущностей на основании заданного набора их свойств. Так, например, служба каталогов может вернуть имена всех web-серверов, работающих под UNIX. В этом смысле службу каталогов можно сравнить с «желтыми страницами». В противоположность ей служба именования, способная выдать информацию только на основе полного имени, сравнима с обычной телефонной книгой.

До появления Active Directory в различных компонентах DCOM и Windows поддерживались разные службы именования и каталогов, такие как реестр, библиотеки типов, моникеры и т. д. Хотя служба Active Directory на самом деле не является частью DCOM, будет полезно кратко описать ее организацию и методы использования для именования и поиска объектов.

По организации служба Active Directory в значительной степени соответствует распределенной системе на базе Windows 2000. Как предполагается, такая система разделена на *домены (domains)*, каждый из которых содержит некоторое количество пользователей и ресурсов. В каждом из доменов имеется один или несколько *контроллеров домена (domain controllers)*, которые представляют собой локальные серверы каталогов, контролирующие пользователей и ресурсы этого домена. Домен имеет соответствующее имя DNS, например *cs.vu.nl*.

Контроллеры домена представляют собой серверы каталогов LDAP. Как мы уже говорили в главе 4, протокол LDAP представляет собой Интернет-версию протокола доступа к каталогам X.500. В случае Active Directory каждый контроллер домена содержит заранее определенную схему организации сущностей

и их атрибутов. Существуют, например, предопределенные записи для регистрации пользователей, хостов, принтеров и т. п. При необходимости эта схема может быть расширена.

Контроллеры доменов регистрируются в DNS как службы LDAP с использованием записей типа SRV системы DNS, описанных в главе 4. Так, сервер LDAP домена cs.vu.nl может быть зарегистрирован при помощи записи SRV под именем ldap.tcp.cs.vu.nl. Это имя может, в свою очередь, ссылаться на имя DNS хоста соответствующего контроллера домена.

Теперь понятно, что найти в заданном домене контроллер домена очень легко. Клиент должен просто запросить у DNS имя хоста сервера LDAP в этом домене, после чего он сможет связаться с этим сервером и получить информацию о пользователях и ресурсах или запросить данные о произошедших изменениях. Эта схема в общем виде приведена на рис. 9.22.



Рис. 9.22. Обобщенная организация Active Directory

Одна из проблем описанного варианта построения службы каталогов состоит в том, что клиент до начала поисков должен знать нужный ему домен. Так, например, если клиент хочет найти все web-серверы, работающие под UNIX, он должен сначала связаться с контроллером некоего конкретного домена и лишь после этого посылать ему поисковый запрос. Если поиск распространяется на несколько доменов, клиент должен связываться с контроллерами каждого домена по отдельности, а затем объединить полученные результаты.

Чтобы помочь справиться с этой проблемой, Active Directory позволяет группировать домены в деревья и леса. *Дерево доменов (domain tree)* представляет собой поддерево пространства имен DNS, а *лес доменов (domain forest)* — это просто группы независимых доменов. Преимущество группировки доменов состоит в том, что служба Active Directory создает для всей группы единый глобальный индекс, именуемый *глобальным каталогом (global catalog)*. Записи каждого домена группы вместе с наиболее важными парами (*атрибут, значение*) копируются в каталог. Клиенту теперь значительно проще сделать поисковый запрос в каталог группы доменов.

Дополнительную информацию о Active Directory, управлении этой системой, настройке конфигурации, репликации и правил защиты, можно найти в [273].

9.2.5. Синхронизация

Базовый механизм синхронизации в DCOM реализован в форме транзакций. Этот механизм мы рассмотрим чуть позже при обсуждении вопросов отказоустойчивости в DCOM.

9.2.6. Репликация

Стандартной поддержки репликации в приложениях DCOM не существует. Репликация может осуществляться только при помощи специальных служб, таких как Active Directory. Что касается пользовательских приложений, разработчику придется разрабатывать собственные решения.

9.2.7. Отказоустойчивость

В основном отказоустойчивость в DCOM обеспечивается при помощи *автоматических транзакций* (*automatic transactions*), которые представляют собой неотъемлемую часть COM+. Автоматические транзакции позволяют разработчику определить набор обращений к методам, в том числе к методам различных объектов, которые нужно объединить в транзакцию. Также можно представить в виде транзакции и одиночное обращение к методу. Этот вариант используется в том случае, когда реализация метода предполагает работу с базой данных.

Транзакции в объектной среде создаются автоматически, в зависимости от того, как сконфигурирован соответствующий объект класса. Каждый объект класса имеет атрибут транзакции, который определяет поведение этого объекта в транзакции. Возможные значения этого атрибута перечислены в табл. 9.7.

Таблица 9.7. Значения атрибута транзакции объектов DCOM

Значение	Описание
REQUIRED_NEW	При каждом обращении всегда начинается новая транзакция
REQUIRED	Новая транзакция начинается, если она еще не начата
SUPPORTED	Включается в транзакцию, если вызвавший процесс уже начал ее
NOT_SUPPORTED	Никогда не включается в транзакции
DISABLED	Никогда не включается в транзакции, даже если клиент пытается это сделать

Если атрибут транзакции имеет значение `REQUIRES_NEW`, то всякий раз при обращении к объекту он автоматически начинает новую транзакцию, полностью независимую от всех прочих. Более того, если вызвавшая сторона уже участвует в транзакции, объект, требующий новой транзакции, начинает другую транзакцию параллельно с существующей.

Если атрибут транзакции имеет значение `REQUIRED`, наблюдается другая ситуация. В этом случае объект принимает участие в транзакции вызвавшей его стороны, если таковая существует. В противном случае он начинает новую тран-

закцию. В любом случае обращение к нему всегда осуществляется как часть транзакции.

Если транзакции лишь поддерживаются (SUPPORTED), объект никогда не начинает новой транзакции, а лишь включается в транзакцию вызвавшей его стороны. Если вызвавший объект или клиент не участвует в транзакции, обращение к объекту происходит обычным образом, безо всяких транзакций.

Если транзакции не поддерживаются, обращение к объекту может происходить вне всяких транзакций, в которых могла бы принимать участие вызывающая сторона. Другими словами, в этом случае нет никаких гарантий сохранения атомарности. Значение атрибута транзакции объекта класса в этом случае устанавливается равно NOT_SUPPORTED.

И, наконец, если объект показывает, что поддержка транзакций отключена (DISABLED), то вне зависимости от пожеланий клиента объект не будет участвовать в транзакциях. Отключение транзакций — это более жесткий вариант, чем просто отсутствие поддержки. В последнем случае клиент, от которого исходит обращение, может попытаться включить в транзакцию объект, к которому это обращение направлено. Если поддержка транзакций отключена, это невозможно.

Вообще говоря, автоматические транзакции обычно реализуются при помощи отдельного менеджера транзакций. Во многих случаях используется менеджер транзакций Windows, известный как *координатор распределенных транзакций* (*Distributed Transaction Coordinator, DTC*) корпорации Microsoft. DTC — это относительно стандартный менеджер транзакций, реализующий протокол двухфазного подтверждения, описанный в главе 7.

9.2.8. Защита

Как и в CORBA, система защиты DCOM в основном обеспечивает защиту обращений к объектам. Однако способ реализации защищенных обращений здесь абсолютно иной. Хотя DCOM чаще всего входит в одну из операционных систем семейства Windows, разработчики DCOM совершенно оправданно решили по возможности отделить защиту DCOM от служб защиты Windows. Таким образом они сохранили возможность реализации DCOM на других операционных системах с сохранением защиты обращений.

В DCOM существует два направления реализации защиты. Важная роль в защите DCOM отводится реестру, при помощи которого осуществляется декларативная защита. Кроме того, защита может быть реализована и программным путем, в том смысле, что приложение может само решать вопросы контроля доступа и аутентификации. Далее мы рассмотрим оба подхода.

Декларативная защита

При декларативной защите обычно предполагается, что в реестре описываются потребности компонента (то есть объекта) в плане активизации, контроля доступа и аутентификации. Модель защиты DCOM основана на механизме ролей. Как и в случае мандатов, для идентификации ролей необходимо, чтобы записи о них присутствовали в реестре.

Механизмы активизации и контроля доступа относительно несложны. Для каждого зарегистрированного класса записи в реестре определяют, какие пользователи или группы пользователей имеют право создавать экземпляры данного класса, то есть объекты. Точно так же в реестре должен содержаться список контроля доступа, определяющий права доступа каждого пользователя или их групп.

DCOM поддерживает несколько *уровней аутентификации (authentication levels)*, представленных в табл. 9.8. По умолчанию большинство систем считают достаточной аутентификацию, происходящую в ходе первого контакта клиента с сервером (уровень CONNECT). Однако можно также требовать идентификации при каждом обращении к серверу (CALL) или даже при каждом получении данных от клиента (PACKET). Уровни PACKET_INTEGRITY и PACKET_PRIVACY — это уже не просто уровни аутентификации, а скорее добавление к уровню PACKET проверки целостности и конфиденциальности соответственно.

Таблица 9.8. Уровни аутентификации в DCOM

Уровень	Определение
NONE	Аутентификация не требуется
CONNECT	Аутентификация клиента при первом соединении с сервером
CALL	Аутентификация клиента при каждом обращении
PACKET	Аутентификация всех пакетов данных
PACKET_INTEGRITY	Аутентификация пакетов данных и проверка их целостности
PACKET_PRIVACY	Аутентификация пакетов данных, проверка их целостности и шифрование пакетов данных

Чтобы защитить клиента от злонамеренных действий сервера, можно определить, в какой степени сервер имеет право присвоить себе роль или мандат клиента, обращающегося к объекту (табл. 9.9). Для этого используются так называемые *уровни обезличивания (impersonation levels)*. Если уровень обезличивания имеет значение ANONYMOUS, сервер не может определить, кто именно к нему обращается, а значит, не может каким-либо образом выступать от имени клиента.

Таблица 9.9. Уровни обезличивания в DCOM

Уровень	Описание
ANONYMOUS	Клиент абсолютно анонимен
IDENTIFY	Сервер знает клиента и может осуществлять контроль доступа
IMPERSONATE	Сервер может по поручению клиента обращаться к локальным объектам
DELEGATE	Сервер может по поручению клиента обращаться к удаленным объектам

Если клиент хочет получить доступ к ресурсам, он должен позволить серверу осуществлять контроль доступа, что требует уровня обезличивания IDENTIFY. Этот уровень позволяет серверу идентифицировать клиента, однако эта идентификация может использоваться только на этом сервере и только для проверки наличия у клиента необходимых прав доступа.

Если разрешить серверу выдавать себя за клиента (уровень IMPERSONATE), появляется возможность обращений к локальным объектам этого сервера, в ходе которых сервер будет использовать мандат клиента. Если необходимо обращение за пределы одной машины, уровень должен быть установлен в DELEGATE, что позволит клиенту делегировать права доступа серверу.

Программная защита

Кроме декларативной защиты можно также позволять приложениям самим выбирать для себя уровень защиты, а также выбирать между различными службами защиты общего назначения. Возможность установки уровня защиты может понадобиться приложениям для временного повышения уровня защиты по сравнению с заявленным в реестре.

Настройка параметров защиты производится при инициализации процесса путем вызова из процесса функции CoInitializeSecurity DCOM. Процесс должен идентифицировать компонент (то есть объект класса или объект), для которого он хотел бы изменить описанные выше уровни аутентификации и обезличивания.

У программной защиты имеется интересная черта: процесс может также указать и службу защиты, которую он хочет использовать для обеспечения защиты. Обычно эта служба относится к той операционной системе, поверх которой работает DCOM. DCOM различает пять служб аутентификации, представленных в табл. 9.10, а также три службы авторизации, перечисленные в табл. 9.11. Число служб в обоих случаях легко увеличить. Наличие или отсутствие конкретной службы зависит не от DCOM, а от операционной системы. Так, например, в Windows 2000 обычно доступны такие службы, как SSL и версия Kerberos с асимметричной криптосистемой.

Таблица 9.10. Стандартные службы аутентификации, поддерживаемые DCOM

Служба	Описание
NONE	Аутентификация отсутствует
DCE_PRIVATE	Аутентификация DCE на основе общих ключей
DCE_PUBLIC	Аутентификация DCE на основе открытых ключей
WINNT	Аутентификация службой защиты Windows NT
GSS_KERBEROS	Аутентификация при помощи Kerberos

Таблица 9.11. Стандартные службы авторизации, поддерживаемые DCOM

Служба	Описание
NONE	Авторизация отсутствует
NAME	Авторизация путем идентификации клиента
DCE	Авторизация с помощью сертификатов атрибутов привилегий (PAC)

Так настроить параметры активизации, чтобы клиент получил возможность создавать экземпляры объекта, невозможно. Эту информацию можно только декларировать, поскольку SCM машины, на которой создается объект, всегда

проверяет реестр, чтобы убедиться, что привилегии клиента соответствуют активизации. Однако можно делегировать право на активизацию другому процессу. Детали можно найти в [132].

9.3. Globe

Последний пример распределенной системы объектов, который мы рассмотрим, — это система *Globe*. *Globe* (*GLOBAL Object-Based Environment* — *глобальная объектная среда*) — это экспериментальная распределенная система, разработанная авторами и их коллегами в университете Vrije (Амстердам). Основная цель разработки системы, которая отличает *Globe* от CORBA и DCOM, — предоставление возможности поддерживать очень большое число пользователей и объектов, разбросанных по всему Интернету, при сохранении полной прозрачности распределения. Большинство других распределенных систем объектов, как известно, создавались в первую очередь для работы в локальных сетях.

Обзор системы можно найти в [471]; детальное описание архитектуры имеется в [202].

9.3.1. Обзор

Globe — это распределенная система объектов, в которой особую роль играет масштабируемость. Структура *Globe* определялась задачами построения крупных глобальных систем, способных поддерживать огромное число пользователей и объектов. Основным при таком подходе является метод просмотра объектов. Как и в случае других систем объектов, объекты в *Globe* рассматриваются как инкапсуляция сущностей и операций над ними.

Важное отличие *Globe* от других систем объектов, особенно от других масштабируемых систем, таких как *Legion* [181, 182] состоит в том, что объекты могут также инкапсулировать реализацию правил распределения состояния объектов по нескольким машинам. Другими словами, каждый объект определяет, каким образом его состояние можно рассредоточить по репликам. Каждый объект управляет также и другими своими правилами.

Вообще говоря, объекты в *Globe* отвечают за все, что только можно. Так, например, объект определяет, как, когда и куда может переместиться его состояние. Кроме того, объект решает, следует ли делать реплики его состояния и, если да, как именно должна происходить репликация. Объект может также определять свои правила защиты и реализацию. Ниже мы опишем, каким образом достигается такая инкапсуляция.

Объектная модель

В отличие от большинства других распределенных систем объектов, *Globe* не поддерживает модель удаленных объектов. Объекты в *Globe* обладают способностью храниться в физически раздробленном состоянии, то есть состояние объектов может быть распределено и реплицировано между несколькими процессами.

Эту организацию иллюстрирует рис. 9.23, на котором показан объект, распределенный между четырьмя процессами, каждый из которых выполняется на отдельной машине. Объекты в Globe являются *распределенными разделяемыми объектами* (*distributed shared objects*). Это название отражает тот факт, что объекты обычно используются несколькими процессами одновременно. Объектная модель Globe восходит своими корнями к распределенным объектам Огса, описанным в [27]. Схожая объектная модель используется также в операционной системе SOS [282, 409].

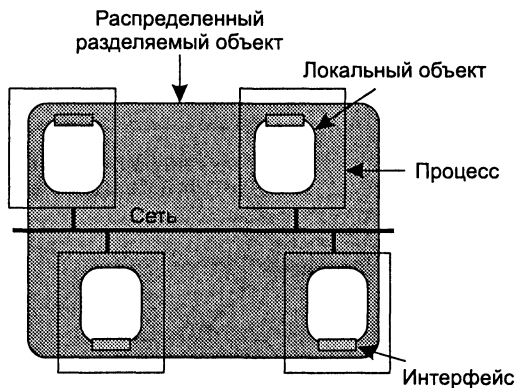


Рис. 9.23. Организация распределенного разделяемого объекта

Процесс, связанный с распределенным разделяемым объектом, получает локальную реализацию интерфейсов этого объекта. Эта локальная реализация называется *локальным представлением* (*local representative*), или просто *локальным объектом* (*local object*). В принципе, так или иначе, локальный объект имеет состояние, полностью прозрачное для содержащего его процесса. Вся реализация объекта скрыта за предлагаемыми процессу интерфейсами.

Всякий локальный объект реализует стандартный интерфейс S0Inf объекта, подобный интерфейсу IUnknown в DCOM. В частности, подобно QueryInterface в DCOM, S0Inf имеет метод getInf, который получает в качестве исходных данных идентификатор интерфейса и возвращает указатель на этот интерфейс, позволяя другим процессам получить доступ к его реализации в объекте. Имеются и другие примеры сходства между локальными объектами Globe и объектами DCOM. Так, например, предполагается, что каждый локальный объект имеет соответствующий объект класса, который может создавать новые локальные объекты.

Локальные объекты реализуют бинарные интерфейсы, которые состоят в основном из таблиц указателей на функции. Интерфейсы описываются на языке определения интерфейсов, который в основном похож на аналогичные языки, используемые в CORBA и DCOM, но имеет некоторые частные отличия.

Существует две разновидности локальных объектов Globe. *Примитивный локальный объект* (*primitive local object*) — это локальный объект, который не содержит других локальных объектов. В противоположность ему, *составной*

локальный объект (composite local object) — это объект, составленный из нескольких (возможно также составных) локальных объектов. Для поддержания составных объектов таблица интерфейсов локальных объектов Globe состоит из пар указателей (*состояние, метод*). Каждый указатель на состояние соответствует данным, принадлежащим одному конкретному локальному объекту. В случае интерфейса примитивного локального объекта все указатели на состояние определяют одни и те же данные, относящиеся к состоянию самого объекта. В случае составных объектов указатели на состояние определяют состояния разных объектов, входящих в этот составной объект.

Составные объекты используются для построения локальных объектов, необходимых для реализации распределенных разделяемых объектов. Подобный локальный объект содержит как минимум четыре подобъекта (рис. 9.24).

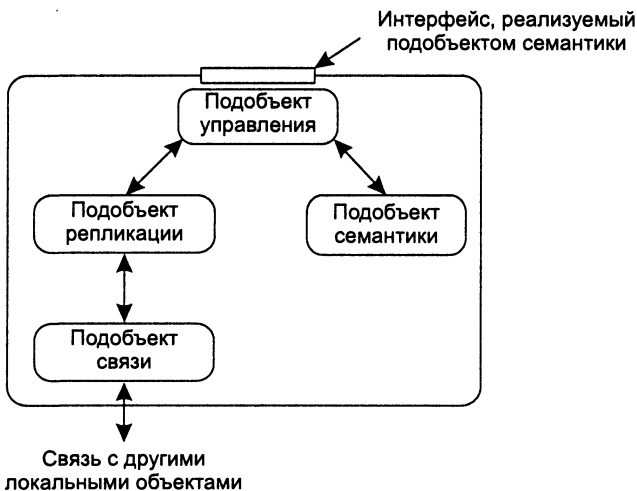


Рис. 9.24. Обобщенная структура локального объекта для распределенных разделяемых объектов Globe

Подобъект семантики (semantics subobject) реализует функциональность распределенного разделяемого объекта. Так, например, с помощью Globe были разработаны распределенные web-сайты, в которых коллекция логически связанных web-страниц, значков, изображений и т. п. была объединена в один документ. Такой документ, называемый *GlobeDoc* [472], реализуется локально при помощи подобъекта семантики, предоставляющего интерфейсы, перечисленные в табл. 9.12. Каждый файл, используемый как часть web-документа, считается элементом объекта *GlobeDoc*. При помощи интерфейса документа можно добавлять или удалять подобные элементы. Этот интерфейс содержит также метод, возвращающий список всех элементов. Предполагается, что web-документы организованы в виде графа с корнем. При помощи отдельных методов можно определить и найти корневой элемент, которым во многих web-приложениях является стандартный файл *index.htm*.

Таблица 9.12. Интерфейсы, реализованные в подобъекте семантики объекта GlobeDoc

Метод	Описание
Интерфейс документа	
AddElement	Добавление элемента к текущему набору элементов
DeleteElement	Удаление элемента из web-документа
AllElements	Возвращение списка входящих в документ элементов
SetRoot	Установка корневого элемента
GetRoot	Возвращение ссылки на корневой элемент
Интерфейс содержимого	
GetContent	Возвращение содержимого элемента в виде массива байтов
PutContent	Замена содержимого элемента заданным массивом байтов
PutAllContent	Замена содержимого всего документа
Интерфейс свойств	
GetProperties	Получение списка пар (<i>атрибут, значение</i>) из элемента
SetProperties	Установка списка пар (<i>атрибут, значение</i>) для элемента
Интерфейс блокировок	
CheckOutElements	Захват группы модифицируемых элементов
CheckInElements	Освобождение группы модифицированных элементов
GetCheckedElements	Получение списка захваченных элементов

Каждый элемент представляется просто массивом байтов. Интерфейс содержимого включает в себя методы получения текущего содержимого документа и его замены заданным массивом байтов. Интерфейс свойств представляет методы для манипулирования с метаданными, связанными с элементом. Метаданные элемента представлены в виде пар (*атрибут, значение*). Так, например, каждый из элементов GlobeDoc имеет свой MIME-тип (MIME-типы мы обсудим в главе 11).

И, наконец, для того чтобы иметь возможность вносить изменения в элементы параллельно, каждый объект GlobeDoc реализует интерфейс блокировок. Этот интерфейс иллюстрирует принятый в Globe подход к управлению параллельным доступом. Поскольку мы предполагаем, что большинство web-документов используется относительно небольшим количеством людей, механизм блокировки можно сделать достаточно простым. Когда элемент необходимо изменить, то первым делом он захватывается. *Захватить (check out)* элемент — это, в сущности, то же самое, что заблокировать его на время модификации. Возможность чтения при этом сохраняется. После проведения модификации элемент *освобождается (check in)*, при этом изменения вступают в силу.

Вернемся к локальному объекту для распределенных разделяемых объектов, показанному на рис. 9.24. Его *подобъект связи (communication subobject)* используется для получения стандартного интерфейса с базовой сетью. Этот подобъект содержит несколько простых методов передачи сообщений для взаимодействия с установлением соединения и без установления соединения. Существуют также

и более совершенные подобъекты связи, реализующие интерфейсы групповой рассылки. Некоторые подобъекты связи могут использоваться для организации надежной связи, в то время как другие в состоянии обеспечить лишь ненадежную связь.

Очень важным практически для всех распределенных разделяемых объектов является *подобъект репликации* (*replication subobject*). Этот подобъект реализует принятую стратегию распределения объекта. Как и в случае подобъекта связи, его интерфейс стандартизован. Подобъект репликации отвечает за то, как именно будет выполняться метод, представленный подобъектом семантики. Так, например, подобъект репликации, реализующий активную репликацию, должен быть уверен, что все обращения к методам выполняются на каждой реплике в одном и том же порядке. В этом случае этот подобъект сможет работать в связке с подобъектом репликации другого локального объекта, представляющего тот же самый распределенный разделяемый объект.

Подобъект управления (*control subobject*) играет роль прослойки между пользовательским интерфейсом подобъекта семантики и стандартизованным интерфейсом подобъекта репликации. Кроме того, он отвечает за экспорт интерфейсов подобъекта семантики в процессы, работающие с распределенным разделяемым объектом. Перед передачей подобъекту репликации подобъект управления выполняет маршрутирование всех обращений к методам, отправляемых такими процессами.

Подобъект репликации в принципе может позволить подобъекту управления самому обработать запрос и вернуть результат процессу, обратившемуся к объекту. Точно так же можно передавать подобъекту управления запросы удаленных процессов. Будет выполнен демаршрутирование такого запроса, после чего подобъект управления выполнит его и передаст результаты обратно подобъекту репликации. Рассмотрение деталей мы отложим до обсуждения вопросов репликации в системе Globe.

Привязка процесса к объекту

В противоположность CORBA и DCOM, система Globe не имеет ни хранилища интерфейсов, ни аналога хранилища реализаций. Отсутствие этих двух служб частично связано с той моделью объектов, которая используется в Globe. Так, в частности, когда происходит привязка процесса к объекту, процесс должен загрузить в свое адресное пространство конкретный локальный объект, соответствующий распределенному разделяемому объекту, к которому выполняется привязка. Рассмотрим, как в Globe происходит привязка процесса к объекту.

Как показано на рис. 9.25, привязка выполняется за пять шагов. Каждый шаг, за исключением последнего, возвращает информацию, необходимую для следующего шага. Таким образом, если мы получили информацию для очередного шага привязки, то этот шаг можно начинать осуществлять. Это повышает эффективность привязки в целом.

Привязка начинается с осмысленного (с позиций человека) имени службе именованию. Это действие обозначено на рисунке как шаг 1. В Globe входит служба именованию на основе DNS, которую мы обсудим чуть ниже. Эта служба возвращает глобально уникальное имя и не зависящий от местоположения деск-

риптор объекта (*object handle*). Это действие обозначено на рисунке как шаг 2. Дескриптор объекта используется в Globe в качестве глобальной ссылки на объект, которая позволяет выполнять поиск объекта при помощи службы локализации Globe. Отметим, что Globe отделяет службу именования от службы локализации (см. главу 4). Достоинство подобного разделения заключается в том, что мы можем изменять имена и адреса, без какого бы то ни было влияния на отображение имен в адреса.

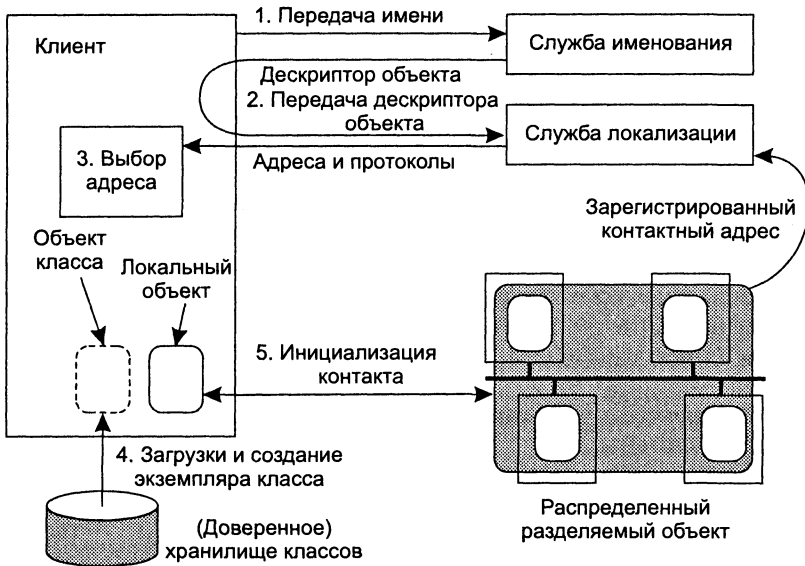


Рис. 9.25. Привязка процесса к объекту в Globe

Служба локализации Globe возвращает набор контактных адресов данного объекта. *Контактный адрес* (*contact address*) точно определяет, где и каким образом можно найти объект. Объект может иметь несколько контактных адресов, например, в связи с репликацией или поддержкой нескольких коммуникационных протоколов. Далее мы увидим, что контактный адрес напоминает межоперационную ссылку на объект (IOR) в CORBA. После получения контактного адреса можно считать, что у процесса есть вся необходимая для привязки к объекту информация.

Поскольку служба локализации может вернуть набор контактных адресов для одного объекта, процесс должен прежде всего выбрать один из них (шаг 3). Выбор может быть произвольным или опираться на те или иные критерии, такие как расстояние до адреса или предполагаемое качество обслуживания в случае привязки к данному адресу.

Любой контактный адрес содержит информацию о том, что необходимо для создания локальной реализации процесса, чтобы он мог работать с объектом. В частности, он точно определяет локальный объект, который процессу следует загрузить. Так, например, контактный адрес может содержать URL-адрес файла,

содержащего объекты класса, которые должен загрузить процесс. Подобный подход напоминает загрузку классов в Java. Загрузку локального объекта из хранилища (доверенного) классов и создание экземпляра иллюстрирует шаг 4. Отметим, что хранилище классов может быть просто файловой системой, возможно, удаленного сайта с доступом по FTP или по другому протоколу обмена файлами.

Отметим, что хранилище классов напоминает хранилище реализаций CORBA в том смысле, что оба они содержат реализации объектов. В Globe, однако, хранилище классов больше соответствует традиционной схеме хранения, в которой выбирается код. В CORBA это процесс, которому можно отправить запрос на создание объекта на сервере объекта.

И, наконец, последний шаг состоит в инициализации локального объекта и организации через этот локальный объект работы с другими локальными объектами, которые представляют собой часть распределенного разделяемого объекта.

Службы Globe

В противоположность CORBA и DCOM, Globe содержит относительно немного служб. На самом деле реализованы только те службы, которые не предоставляются самим объектом. Это проще всего объяснить, рассматривая некоторые службы, имеющиеся в CORBA и DCOM, и их реализацию в Globe (табл. 9.13).

Таблица 9.13. Возможные реализации в Globe обычных служб распределенной системы

Служба	Возможная реализация в Globe	Доступность
Служба коллекций	Выделенный объект, содержащий ссылки на другие объекты	Нет
Служба параллельного доступа	Каждый объект реализует собственную стратегию управления параллельным доступом	Нет
Служба транзакций	Выделенный объект, представляющий собой менеджер транзакций	Нет
Служба событий/уведомлений	Выделенный объект на каждую группу событий (как в DCOM)	Нет
Служба внешних связей	Каждый объект реализует собственные процедуры маршалинга	Да
Служба жизненного цикла	Выделенные объекты класса реализуются внутри объекта	Да
Служба лицензирования	Реализуется каждым объектом в отдельности	Нет
Служба именования	Выделенная служба, реализованная в виде коллекции именующих объектов	Да
Служба свойств/обмена	Выделенная служба, реализованная в виде коллекции направляющих объектов	Нет
Служба сохранности	Реализуется внутри объекта	Да
Служба защиты	Реализуется внутри объекта в комбинации со службами (локальными) защиты	Да

Служба	Возможная реализация в Globe	Доступность
Служба репликации	Реализуется внутри объекта	Да
Служба отказоустойчивости	Реализуется внутри объекта в комбинации с серверами обеспечения отказоустойчивости	Да

Служба коллекций CORBA используется для объединения объектов в списки, очереди и т. д. Обычно такая служба может быть реализована при помощи объекта, состояние которого представляет собой набор ссылок на объекты. В Globe подобная служба реализована именно так.

Служба параллельного доступа в CORBA определена в понятиях коллекции интерфейсов, использующих для доступа различные типы блокировок [327]. В DCOM управление параллельным доступом осуществляется частично службой транзакций, а частично так же, как серверы объектов обслуживают потоки выполнения. В Globe управление параллельным доступом обычно производится каждым объектом самостоятельно. Другими словами, разделяемый объект, которому необходимо предотвратить параллельный доступ к своему состоянию, должен предоставить необходимый специальный интерфейс для блокировки. Никакого стандартного интерфейса для управления параллельным доступом в Globe не существует.

Как и в коллекциях, в транзакциях также происходит объединение объектов. В Globe транзакции обычно реализуются при помощи выделенного объекта, играющего роль менеджера транзакций. Интерфейсы для подобных объектов не стандартизованы.

В Globe нет службы событий. Как мы увидим, объекты в Globe всегда пассивны в том смысле, что у них отсутствует собственный поток выполнения. Таким образом, объект не в состоянии уведомить клиента о произошедшем событии. Для реализации службы событий в Globe подошла бы модель извлечения (pull) CORBA. В этом случае события можно было бы объединить в один объект, как в классе событий DCOM. До настоящего времени события в Globe не использовались.

Служба внешних связей, или служба маршалинга, обычно реализуется для каждого объекта в отдельности. В Globe большинство объектов должны реализовывать собственные процедуры маршалинга, при помощи которых они получают возможность передавать свое состояние на другие машины. Мы вернемся к вопросам маршалинга чуть позже, когда будем обсуждать объектную модель Globe.

Служба жизненного цикла предоставляет средства создания, удаления, копирования и перемещения объектов. Понятно, что в Globe объект создать сам себя не может. Все прочие операции реализуются самим объектом; специальные средства для удаления, копирования или перемещения объектов в Globe отсутствуют. Создание объектов обычно выполняется путем прямого обращения к серверу объекта с требованием создать экземпляр объекта. Подобный подход немного напоминает способ, применяемый в DCOM.

Лицензирование в Globe не поддерживается, но может быть реализовано обычным способом отдельно для каждого объекта, который в этом нуждается.

Служба именования — это хороший пример службы, которую невозможно реализовать при помощи именованного объекта, поскольку она используется для поиска этого объекта. Другими словами, доступ к службе именования по определению требуется раньше, чем доступ к именованным объектам. Разумеется, сама по себе служба именования может быть реализована с использованием объектов, как это сделано, например, в CORBA. В Globe имеется выделенная служба именования, которую мы рассмотрим чуть позже. По схожим причинам службы свойств и торговли также должны быть реализованы отдельно от объектов, ссылками на которые они манипулируют.

Множество распределенных систем имеют выделенную службу сохранности (длительного хранения) в форме файловой системы или базы данных. В Globe такой службы нет. Вместо этого сохранность рассматривается как свойство объекта, а потому должна реализовываться самим объектом. Как добиться сохранности, в основном решает объект. Однако объекту для длительного хранения его состояния необходима определенная поддержка. В Globe эта поддержка обеспечивается в первую очередь серверами объектов, хотя многие реализации объектов используют также локальную файловую систему.

Защита в Globe частично реализуется каждым объектом в отдельности, а частично локальными и глобальными службами защиты. Так, например, хотя объект сам может контролировать большую часть аспектов, связанных с обращением к методам, вопросы получения и сертификации ключей обычно выделяют в отдельную службу. Ниже мы еще вернемся к вопросам защиты в Globe.

Основная разница между Globe и другими распределенными системами объектов становится очевидной при рассмотрении вопросов репликации. Как мы говорили, объекты в Globe сами определяют, как нужно реплицировать их состояние. Другими словами, репликация осуществляется исключительно силами самих объектов. В противоположность этому, в таких системах, как CORBA, обычно применяются специальные серверы репликации, которые управляют репликацией групп объектов. Детали реализации репликации в Globe будут рассмотрены позже.

И, наконец, отказоустойчивость в Globe также поддерживается каждым из объектов, хотя для успешной маскировки ошибок необходима определенная поддержка со стороны серверов отказоустойчивых объектов. И снова мы обещаем вернуться к обсуждению отказоустойчивости чуть позже.

9.3.2. Связь

В отличие от CORBA и DCOM Globe не предоставляет никаких других методов связи кроме синхронных обращений к методам. Когда обращение завершается, то есть когда происходит возвращение результата обращения (обращение при этом выглядит как вызов метода через локальный интерфейс), всякая деятельность внутри распределенного разделяемого объекта прекращается, не считая деятельности, связанной с другими вызовами. В этом смысле объекты Globe считаются *пассивными* (*passive*).

Чтобы понять, что такое пассивный объект, рассмотрим распределенный объект, совместно используемый тремя процессами (рис. 9.26). Процессы *B* и *C* имеют реплики своего состояния, представленные в виде заштрихованных подобъектов семантики (Sem). Процесс *A* не имеет не только состояния, он не имеет даже подобъекта семантики. Локальный объект в адресном пространстве процесса *A* соответствует традиционной клиентской заглушке, или заместителю.

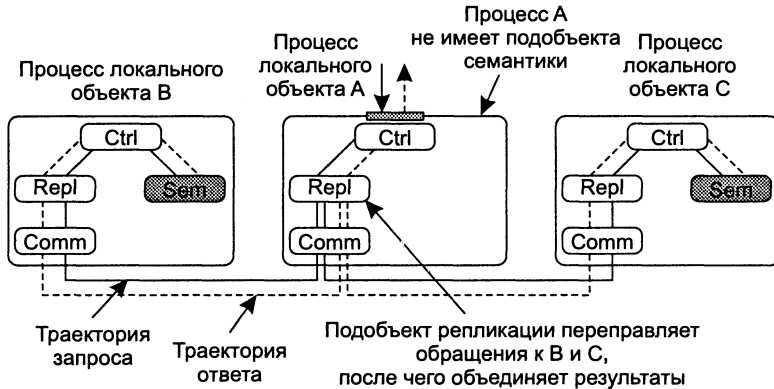


Рис. 9.26. Обращение к объекту в Globe с использованием активной репликации

Предположим, процесс *A* обращается к объекту. В соответствии с простым сценарием репликации этот запрос будет передан соответственно процессам *B* и *C*, на что указывает траектория в виде сплошной линии. Когда запрос достигнет подобъекта связи (Comm), скажем, процесса *B*, будет активизирован поток выполнения для обработки запроса. В конце своей работы этот поток выполнения обратится к подобъекту семантики и вернет процессу *A* результат в отдельном сообщении. Аналогичная деятельность происходит и в процессе *C*.

Пока происходит обращение, поток выполнения процесса *A*, который собственно и выполняет вызов метода, блокируется. После возвращения результатов из других процессов в подобъект репликации (Repl) процесса *A* блокировка с потока снимается, и он возобновляет выполнение. Если мы предположим, что в этот момент другие обращения не выполняются, то как только управление возвратится приложению *A*, от которого исходило обращение, в локальных объектах, соответствующих этому распределенному разделяемому объекту, какая-либо деятельность прекращается.

Globe не поддерживает механизмы отката для клиентских приложений. Подобная поддержка может быть предоставлена только в том случае, если активизация потока выполнения была вызвана запросом, допускающим откат. Однако откаты могут привести к непредсказуемому поведению потока выполнения. При наличии реплик, которые должны сохранять непротиворечивость группы тесно взаимодействующих подобъектов репликации, такое поведение может быть опасно. Это один из примеров того, что разработчики Globe предпочитают богатству средств простоту. Вообще говоря, простота была основной целью проектирования Globe.

9.3.3. Процессы

Хотя в Globe между серверами и клиентами нет принципиальной разницы, часто при организации распределенных разделяемых объектов полезно эту грань провести. Клиентские процессы инициируют обращения к объектам, в то время как серверные в основном реагируют на эти обращения. Далее мы уточним эту разницу.

Клиент Globe

Клиент Globe — это процесс, который относится к распределенному разделяемому объекту и обращается к этому объекту, вызывая доступные методы через интерфейс подобъекта управления (Ctrl), как показано на рис. 9.26. Процесс A в данном случае представляет собой типичный пример клиента Globe, не имеющего состояния, но соединенного с репликами, у которых состояние имеется.

Во многих случаях клиент Globe относительно прост, в том смысле, что он даже не имеет подобъекта семантики. Все его обращения после маршалинга немедленно передаются репликам, выступающим в качестве серверов. Эта модель очень похожа на заместители в CORBA и также может быть использована в качестве основы для интеграции распределенных разделяемых объектов в системах, подобных CORBA [214]. Однако в Globe возможна также настройка клиентов в соответствии с подходом DCOM и RMI языка Java. Такая настройка может потребоваться, например, для того, чтобы позволить клиенту кэшировать результаты обращений.

Стандартный способ придания клиентам CORBA новых свойств — использование перехватчиков. В DCOM подобная настройка, которая производится путем реализации на клиенте выделенного заместителя, возможна благодаря поддержке пользовательского маршалинга. В Globe распределенный разделяемый объект обычно предоставляет несколько реализаций локальных объектов. Эти реализации содержатся в хранилищах классов. Клиент сам решает, какую реализацию ему использовать.

Когда клиент выполняет привязку к объекту, он получает несколько контактных адресов, соответствующих одной или нескольким реализациям локальных объектов, которые этот клиент может загрузить. Возможно также, что контактный адрес определяет только протокол связи, который должен использовать клиент. Если клиент соблюдает правила указанного протокола, он может задействовать любую реализацию, которую захочет. В частности, при желании клиент может ограничиться исключительно теми реализациями, которые имеются в доверенном хранилище классов.

Такая гибкость достигается в Globe немалой ценой. Приходится создавать реализации для разных локальных объектов и иногда для разных операционных систем и машинных архитектур. Некоторые из проблем, связанных с гетерогенностью, можно решить путем использования переносимых реализаций, написанных на интерпретируемых языках, таких как Java.

Сервер Globe

В противоположность клиенту сервер системы Globe — это процесс, который способен только обрабатывать запросы с обращениями, приходящие к нему по

сети. Другими словами (см. рис. 9.26), он не в состоянии вызывать методы через интерфейс, предоставляемый подобъектом управления (Ctrl); процессы *B* и *C* — это примеры серверов Globe.

В Globe имеется также сервер объектов, поддерживающий распределенные разделяемые объекты. Такой сервер предоставляет базовую функциональность, которая требуется локальному объекту распределенного разделяемого объекта. Процессы могут связаться с сервером объектов Globe через интерфейс, методы которого перечислены в табл. 9.14. В сущности, каждый сервер объектов Globe представляется для своих клиентов просто еще одним распределенным разделяемым объектом.

Таблица 9.14. Операции с сервером объектов Globe

Метод	Описание
Bind	Метод позволяет серверу выполнить привязку к указанному объекту, если привязка не была выполнена ранее
AddBinding	Метод позволяет серверу выполнить привязку к указанному объекту, даже если привязка уже была выполнена
CreateLR	Метод позволяет серверу создать локальный объект для нового распределенного объекта
RemoveLR	Метод позволяет серверу удалить локальный объект данного объекта
UnbindDSO	Метод позволяет серверу удалить все локальные объекты данного объекта
ListAll	Метод возвращает список всех локальных объектов
ListDSO	Метод возвращает список всех локальных объектов данного распределенного объекта
StatLR	Метод получает состояние указанного локального объекта

Одна из важных операций сервера — привязка к данному распределенному разделяемому объекту с помощью метода `Bind`. При этом серверу передается дескриптор объекта и критерии выбора контактного адреса (при наличии нескольких контактных адресов). Так, сервер может выбрать произвольный адрес или адрес, обладающий определенными свойствами.

Привязка сервера к объекту является идемпотентной операцией; если привязка уже выполнена, не происходит ничего. Если предполагается загрузка сервером другого локального объекта, относящегося к распределенному разделяемому объекту, привязка к которому уже выполнена, вызывается метод `AddBinding`.

Сервер объектов Globe также содержит базовые средства создания распределенных разделяемых объектов путем создания локальных объектов. Создание производится вызовом метода `CreateLR` с указанием той реализации (то есть объекта класса), которую может загрузить сервер. Когда объект класса загружен, сервер создает по объекту класса новый локальный объект и инициализирует его. Обычно инициализация также предполагает, что контактный адрес нового

объекта становится доступным службе локализации Globe. Таким образом, с этого момента к объекту можно выполнять привязку других процессов.

Метод `RemoveLR` просто удаляет определенный локальный объект с сервера объектов. Обычно эта операция не вызывается никогда. Вместо этого сервер, как правило, получает указание разорвать привязку к распределенному разделяемому объекту методом `UnbindDSO`. В результате разрыва привязки все локальные объекты данного объекта удаляются с сервера, таким образом, объект сам очищает занимаемое им место.

Отметим, что разрыв привязки вовсе не обязательно означает ликвидацию объекта. Во многих случаях к нему могут быть привязаны другие процессы, и пока объект не уничтожен явным образом, он продолжает существовать. Подобно CORBA, но в противоположность DCOM, Globe поддерживает как нерезидентные, так и сохраненные объекты.

И, наконец, существует несколько операций для просмотра привязок серверов объектов и состояния каждого из локальных объектов.

В Globe существует два способа привязки сервера объектов к распределенному разделяемому объекту. Во-первых, привязка может повлечь за собой создание сохраненного локального объекта. *Сохраненный локальный объект* (*persistent local object*) — это локальный объект, для которого сервер объектов Globe будет выполнять маршalling его состояния и записывать это состояние на устройство длительного хранения. Сохранение подобного локального объекта обычно является частью процесса корректного отключения сервера. Когда впоследствии сервер вновь запускается, он предварительно восстанавливает сохраненные ранее локальные объекты. С другой стороны, в результате привязки может образоваться *нерезидентный локальный объект* (*transient local object*), который никогда не записывается на устройства длительного хранения сервера объектов. Таким образом, по окончании работы сервера нерезидентные локальные объекты полностью уничтожаются.

Привязка может потребовать создания сервером объектов контактного адреса, соответствующего установленному локальному объекту. Такой адрес позволяет другим процессам выполнять привязку к удаленному объекту через локальный объект, находящийся на сервере объектов. В подобных случаях сервер объектов будет сам регистрировать контактные адреса в службе локализации Globe и обрабатывать входящие запросы других процессов так, как описано ниже.

9.3.4. Именованние

Именованние играет в Globe важнейшую роль и в значительной степени отличается от методов именования, принятых в других распределенных системах. Как мы говорили в главе 4, в Globe объекты именования отделены от объектов локализации. Другими словами, для поддержания осмысленных имен в виде символьных строк и бессмысленных (с точки зрения человека) имен в виде битовых строк требуются разные службы.

Ссылки на объекты и контактные адреса

Всякий распределенный разделяемый объект в Globe имеет глобальный уникальный идентификатор (OID) — 256-битную строку. OID в Globe — это, согласно определению из главы 4, правильный (true) идентификатор. Другими словами, OID в Globe относится самое большее к одному распределенному разделяемому объекту, не может быть использован повторно для именованного другого объекта и любой объект имеет максимум один идентификатор OID.

OID в Globe используется только для сравнения ссылок на объекты. Так, например, предположим, что процессы *A* и *B* привязаны к распределенному разделяемому объекту. Каждый из процессов может запросить OID объекта, к которому они привязаны. Процессы *A* и *B* будут считаться привязанными к одному и тому же объекту в том и только в том случае, если одинаковы их идентификаторы OID.

Чтобы локализовать объект, необходимо разрешить контактный адрес этого объекта в службе локализации Globe. Объекты ссылаются на эту службу посредством *дескриптора объекта* (*object handle*), который представляет собой битовую строку, содержащую OID и другую информацию, необходимую службе локализации. Дескриптор объекта также является идентификатором, однако нет никаких гарантий того, что один и тот же объект всегда будет иметь один и тот же дескриптор объекта. Важно понимать, что дескриптор объекта, используемый в службе локализации Globe, не несет какой-либо информации о текущем местоположении объекта.

Служба локализации Globe организована по принципу иерархической службы локализации, о которых говорилось в пункте 4.2.4, и мы не станем повторять здесь этот материал. Детальные сведения по этой теме можно найти в [24, 469]. Вопрос проектирования и реализации собственно сервера локализации вместе с соображениями о его производительности разбирается в [33].

Служба локализации Globe возвращает контактный адрес. Этот контактный адрес напоминает ссылки на объект, не зависящие от его местонахождения, которые используются в CORBA и других распределенных системах. Если игнорировать некоторые частные детали, контактный адрес состоит из двух частей. Первая — это *идентификатор адреса* (*address identifier*), по которому служба локализации может определить правильный листовой узел для пересылки на него операций вставки или удаления, предназначенных для соответствующего контактного адреса. Напомним, что поскольку контактный адрес зависит от местоположения, важно, чтобы вставка или удаление для него начинались с нужного листового узла.

Вторая часть содержит реальную информацию об адресе, но эта информация для службы локализации абсолютно недоступна. Для службы локализации адрес — это просто массив байтов, который может в равной степени соответствовать реальному сетевому адресу, указателю на интерфейс (после маршалинга указателя) или даже полностью «упакованному» (путем маршалинга) заместителю.

В Globe в настоящее время поддерживаются два типа адресов. *Упакованный адрес* (*stacked address*) представляет многоуровневый набор протоколов, каждый из уровней которого описывается записью с тремя полями (табл. 9.15).

Таблица 9.15. Представление уровня протоколов в упакованном контактном адресе

Поле	Описание
Идентификатор протокола	Константа, соответствующая протоколу (известному)
Адрес протокола	Адрес, определяемый протоколом
Дескриптор реализации	Ссылка на файл в хранилище классов

Идентификатор протокола (protocol identifier) — это константа, соответствующая одному из известных протоколов. Обычно идентификаторы протоколов описывают такие протоколы, как TCP, UDP и IP. Поле *адрес протокола (protocol address)* содержит определяемый протоколом адрес, такой как номер порта TCP или сетевой адрес IPv4. И наконец, *дескриптор реализации (implementation handle)* — поле (необязательное), идентифицирующее место, в котором может находиться стандартная реализация протокола. Обычно дескриптор реализации представлен в виде URL-адреса.

Второй тип контактного адреса — это *адрес экземпляра (instance address)*, состоящий из двух полей (табл. 9.16). Этот адрес также содержит дескриптор реализации, представляющий собой ссылку на файл в хранилище классов, содержащий реализацию локального объекта. Этот локальный объект может быть загружен привязанным к объекту процессом.

Таблица 9.16. Представление контактного адреса экземпляра

Поле	Описание
Дескриптор реализации	Ссылка на файл в хранилище классов
Строка инициализации	Строка, используемая для инициализации данной реализации локального объекта

Загрузка протекает в соответствии со стандартным протоколом, похожим на протокол загрузки классов в Java. После загрузки реализации и создания локального объекта производится инициализация этого локального объекта, состоящая в передаче ему *строки инициализации (initialization string)*.

Служба именования Globe

Для поддержания осмысленных имен Globe поддерживает относительно простую службу именования на базе DNS [32]. Служба именования позволяет производить поиск и авторизованные изменения имен. Осмысленное имя в Globe [44] представлено в виде *унифицированного идентификатора ресурса (Uniform Resource Identifier, URI)*, например, `hfn://org/globeworld/dns/globesite`. Это имя разрешается в дескриптор именованного объекта.

Для использования DNS имя Globe сначала локально преобразуется в правильное DNS-имя. Так, имя Globe `hfn://org/globeworld/dns/globesite` преобразуется в DNS-имя `globesite.dns.globeworld.org`, которое затем передается локальному преобразователю имен DNS, как показано на рис. 9.27 (для обозначения сервера,

именуемого *org*, используется запись *#org*). Если предполагать, что мы имеем дело с правильным именем, то, в конце концов, сервер имен Globe столкнется с частью имени, которую не сможет разрешить.

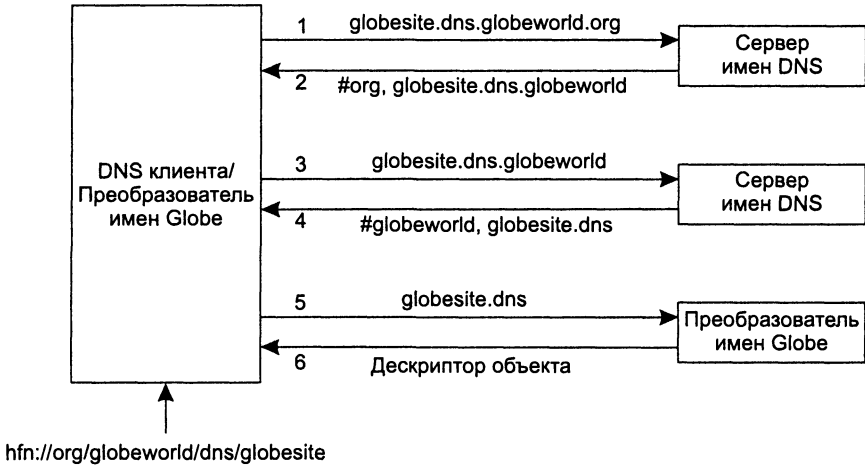


Рис. 9.27. Итеративное разрешение имен в Globe

В нашем примере DNS-имя *globeworld.org* будет разрешено сервером имен Globe, который выделит оставшуюся часть, *globesite.dns*. Сервер имен Globe реализуется как обычный сервер имен DNS, за исключением того, что имена из пространства имен Globe будут ассоциироваться с текстовыми записями о ресурсах, указывающими на вложенный каталог или содержащими дескриптор объекта.

9.3.5. Синхронизация

Globe не имеет каких-либо специальных механизмов синхронизации. При необходимости такие механизмы обычно реализуются в виде части подобъекта репликации. Механизмов синхронизации между объектами не существует. Понятно, что для практической разработки приложений они необходимы.

9.3.6. Репликация

Репликация в Globe имеет важнейшее значение. Всякий распределенный разделяемый объект Globe реализует свою собственную стратегию репликации при помощи подобъектов репликации локальных объектов. Следует отметить, что интерфейсы, связанные с подобъектом репликации, стандартизованы. Таким образом, можно изменять стратегию репликации, на задевая при этом другие компоненты, входящие в распределенный разделяемый объект.

Чтобы понять, как стандартизовать интерфейсы репликации, необходимо разобратся в базовой модели репликации состояния распределенных объектов.

В распределенных разделяемых объектах *Globe* состояние и операции над состоянием реализуются в подобъектах семантики. Репликация состояния производится путем создания нескольких копий подобъектов семантики, распределенных по разным процессам.

Ключевая проблема состоит в том, чтобы сохранить непротиворечивость копий подобъектов семантики. Необходимый тип непротиворечивости определяется распределенным разделяемым объектом, а за поддержание непротиворечивости полностью отвечает набор подобъектов репликации. По этой причине подобъект репликации проверяет, *когда* разрешается обращение к локальному методу подобъекта семантики. При этом принимается во внимание различие между *методами чтения* (*read methods*), которые не изменяют состояние подобъекта семантики, и *методами модификации* (*modify methods*), которые вносят в него изменения.

Когда процесс, привязанный к распределенному разделяемому объекту, обращается к его методам, первое, что делает подобъект управления, — информирует об обращении подобъект репликации путем вызова метода *Start* интерфейса подобъекта репликации (табл. 9.17). Единственная информация, которую передает подобъект управления при вызове *Start*, — указание на то, может ли соответствующее обращение внести изменения в подобъект семантики. Основываясь на этой информации, подобъект репликации сообщает подобъекту управления, что ему делать дальше.

Таблица 9.17. Интерфейс подобъекта репликации, доступный подобъекту управления

Метод	Описание
start	Указывает на появление запроса с новым обращением к методу
send	Выполняет маршалинг обращения и передает запрос с обращением подобъекту репликации
invoked	Указывает на завершение обращения к подобъекту семантики

Существует два возможных результата вызова метода *start*, каждый из которых указывает на следующее действие, которое следует выполнить подобъекту управления. Во-первых, подобъекту управления может быть предложено передать обращение (после маршалинга) подобъекту репликации. Другими словами, подобъект управления может получить команду вызвать метод *send*, после чего ожидать завершения операции.

В качестве второго возможного результата подобъекту управления может быть указано на необходимость реально выполнить запрос, обратившись к подобъекту семантики. После завершения обращения подобъект управления вызывает метод *invoked*, сообщая подобъекту репликации о том, что работа окончена. И в этом случае подобъект управления будет ожидать завершения вызова.

Метод *send* в качестве исходных данных получает обращение, выполняет его маршалинг и обычно пересылает другим репликам. Возможно, подобъект репликации получит ответ, может быть даже от нескольких реплик. В любом случае, если все происходит правильно, подобъект репликации собирает ответы реплик

(если они есть), чтобы передать их процессу, который обращался к объекту, и возвращает эти ответы в исходящем параметре метода `send`.

Когда вызов `send` завершается, подобъект управления получает приказ вернуть управление вызвавшему его процессу или обратиться к подобъекту семантики. В первом случае выполняется демаршалинг результата, возвращаемого подобъектом управления, и передача его вызывающему процессу. Если подобъект управления получает указание обратиться к подобъекту семантики, он по завершении этого обращения вызывает метод `invoked`. После вызова метода `invoked` подобъект управления обычно получает указание вернуть управление в вызывающий процесс.

С точки зрения подобъекта репликации подобъект управления ведет себя как конечный автомат с четырьмя состояниями, как это показано на рис. 9.28. Переходы между состояниями жестко контролируются подобъектом репликации, причем каждое состояние соответствует определенному действию. Эти действия могут осуществляться как путем вызова соответствующих методов подобъекта репликации, так и путем обращения к изначально вызванным методам подобъекта семантики. Таким образом, за решение о том, как и когда осуществлять локальное обращение к методу, отвечает подобъект репликации. Рассмотрим две стратегии репликации, иллюстрирующие этот подход.

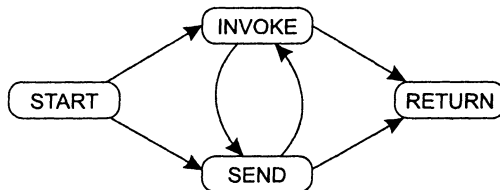


Рис. 9.28. Работа подобъекта управления в качестве конечного автомата

Примеры репликации в Globe

В качестве примера активной репликации рассмотрим процесс, желающий обратиться к методу чтения. Сначала подобъект управления вызывает метод `start`, после чего подобъект репликации может сразу же дать указание подобъекту управления обратиться к методу и вернуть результаты в вызывающий процесс.

При обращении к методу модификации используется другой подход. Предположим, что подобъект репликации задействует подобъект связи, позволяющий производить полностью упорядоченную групповую рассылку. В этом случае при вызове метода `start` подобъект репликации может немедленно дать команду подобъекту управления выполнить маршалинг и передачу обращения при помощи метода `send`. Обращение после маршалинга рассылается всем репликам, включая и ту, которая инициировала рассылку. Когда последняя еще раз получит собственное обращение, она позволит подобъекту управления, в продолжение его работы, обратиться к нужному методу. Когда вызов этого метода завершится, результаты будут возвращены вызывающему процессу (табл. 9.18).

Таблица 9.18. Переходы из состояния в состояние при активной репликации

Состояние	Производимое действие	Вызов метода	Следующее состояние
Метод чтения			
START	Отсутствует	start	INVOKE
INVOKE	Обратиться к локальному методу	invoked	RETURN
RETURN	Вернуть результаты	Нет	START
Метод модификации			
START	Отсутствует	start	SEND
SEND	Выполнить маршалинг и передать обращение	send	INVOKE
INVOKE	Обратиться к локальному методу	invoked	RETURN
RETURN	Вернуть результаты	Нет	START

Рассмотрим теперь репликацию на основе первичной копии. В этом случае методы чтения также будут выполняться локально, а результаты возвращаться вызывающему процессу.

При вызове метода модификации резервной копии ее подобъект репликации должен будет передать запрос первичной копии и ожидать окончания операции. Кроме того, до передачи управления подобъекту управления состояние резервной копии также следует изменить. После этого подобъект управления получает приказ вернуть управление вызывающему процессу.

При вызове метода модификации первичной копии подобъект репликации приказывает подобъекту управления вызвать метод локально и сообщить, когда он закончит вызов. После получения сообщения об окончании вызова подобъект репликации выполняет маршалинг состояния подобъекта семантики и пересылает его в резервную копию. После завершения обновления резервной копии подобъект репликации отдает подобъекту управления указание вернуть управление вызывающему процессу (табл. 9.19).

Таблица 9.19. Переходы из состояния в состояние при репликации на основе первичной копии

Состояние	Производимое действие	Вызов метода	Следующее состояние
Метод чтения			
START	Отсутствует	start	INVOKE
INVOKE	Обратиться к локальному методу	invoked	RETURN
RETURN	Вернуть результаты	Нет	START
Метод модификации резервной копии			
START	Отсутствует	start	SEND
SEND	Выполнить маршалинг и передать обращение	send	RETURN
RETURN	Вернуть результаты	Нет	START

Состояние	Производимое действие	Вызов метода	Следующее состояние
Метод модификации первичной копии			
START	Отсутствует	start	INVOKE
INVOKE	Обратиться к локальному методу	invoked	RETURN
RETURN	Вернуть результаты	Нет	START

9.3.7. Отказоустойчивость

В настоящее время в Globe отказоустойчивость поддерживается в основном за счет репликации. Во многих случаях для обеспечения отказоустойчивости вполне достаточно возможности реплицировать состояние распределенного объекта на нескольких машинах. Что действительно необходимо — так это механизмы восстановления после отказа.

В плане восстановления после сбоев сервер объектов Globe гарантирует, что локальные объекты, которые следует восстанавливать, будут сохранены на диске. Другими словами, как мы говорили ранее, эти объекты превращаются в сохраняемые объекты. Однако помимо сохранности необходимо гарантировать, что восстановление действительно пойдет так, как требуется. Так, например, предотвратить восстановление устаревших данных можно путем немедленной записи изменений локальных объектов на диск или восстановления их из свежей реплики. Подобные средства Globe пока не предоставляет.

9.3.8. Защита

Меры защиты в Globe сосредоточены на предотвращении хакерских атак на одиночный распределенный разделяемый объект. Наиболее важный механизм, используемый для обеспечения подобной защиты, — создание дополнительного *подобъекта защиты* (*security subobject*) как части каждого локального объекта. Место подобъекта защиты в структуре локальных объектов иллюстрирует рис. 9.29.

Подобъект защиты обеспечивает защита на различных этапах обращения к методам. Во-первых, необходимо гарантировать, что обращения удаленных процессов аутентифицированы. По этой причине подобъект связи должен передавать подобъекту защиты информацию о каждом входящем запросе, чтобы он мог аутентифицировать запрашивающий процесс.

После завершения аутентификации информацией с подобъектом защиты обменивается подобъект репликации, чтобы осуществить контроль доступа. В Globe контроль доступа в основном реализован для каждого метода в отдельности. Другими словами, подобъект защиты отдельно для каждого метода решает, допустимо или нет обращение к нему ранее аутентифицированного запрашивающего процесса.

И, наконец, подобъект защиты обменивается информацией с подобъектом управления, чтобы проверить, удовлетворяют ли параметры запроса определенным условиям. Так, например, для объектов, предназначенных для получения

денег, подобъект защиты может запретить операцию выдачи, если сумма превышает определенное значение.

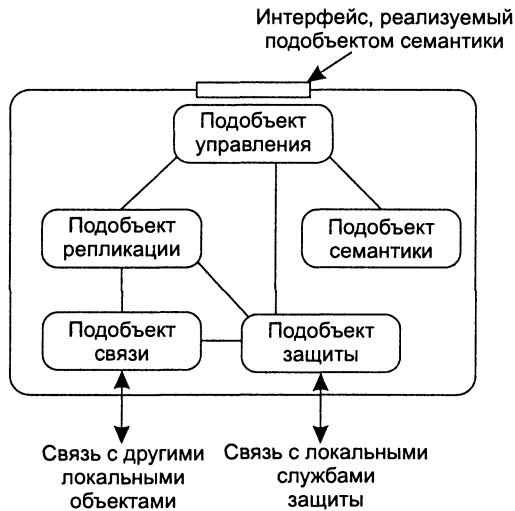


Рис. 9.29. Место подобъекта защиты в локальном объекте Globe

Подобъект защиты связывается с локальными службами защиты. Так, например, для организации защищенного взаимодействия с другими локальными объектами подобъект защиты может требовать получения сеансового ключа. В Globe взаимодействие с локальными службами защиты выполняется через выделенный распределенный разделяемый объект, известный как *объект поддержки* (*principal object*). Этот объект представляет пользователя распределенного разделяемого объекта в конкретном защищенном домене. Объект поддержки отвечает за взаимодействие с доверенными третьими сторонами, например службами аутентификации и сертификации.

В качестве примера рассмотрим, как можно использовать в распределенном разделяемом объекте Globe систему защиты Kerberos для создания сеансовых ключей между двумя локальными объектами A и B , которые привязаны к распределенному разделяемому объекту соответственно процессами P_A и P_B . Рисунок 9.30 иллюстрирует последовательность шагов, которые надо выполнить для организации защищенного взаимодействия между A и B при условии, что процессы P_A и P_B относятся к разным защищенным доменам.

Сначала подобъект защиты вызывает объект поддержки (представляющий процесс P_A), чтобы получить сеансовый ключ (шаг 1). Объект поддержки запрашивает подсистему аутентификации Kerberos и ее службу выдачи талонов (шаг 2), чтобы получить сеансовый ключ и талон для локального объекта B (шаг 3), которые передаются подобъекту защиты (шаг 4).

Начиная с этого момента подобъект защиты может связаться с другим локальным объектом через его подобъект связи и передать ему полученный талон (шаг 5). Далее талон попадает к объекту поддержки, представляющему процесс

P_B (шаг 6), который берет на себя расшифровку талона и возвращает полученный сеансовый ключ подобиъекту защиты процесса B (шаг 7).

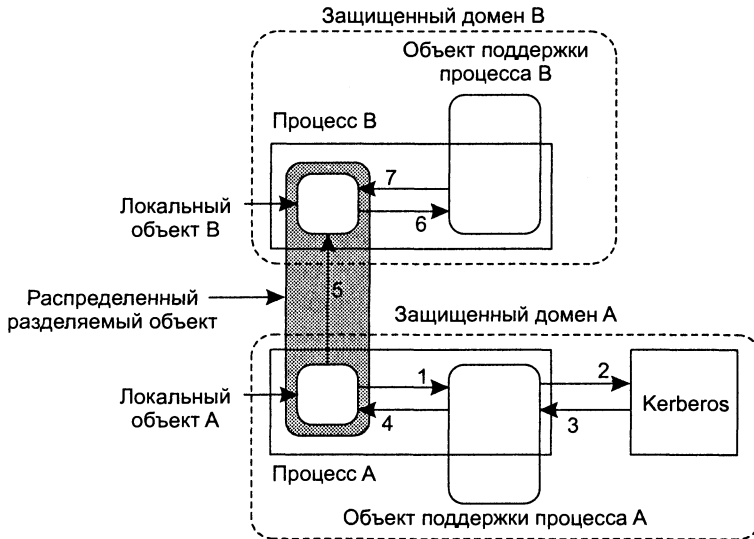


Рис. 9.30. Защита распределенных разделяемых объектов с помощью системы Kerberos

Дополнительную информацию о защите в Globe можно найти в [202, 261].

9.4. Сравнение систем CORBA, DCOM и Globe

CORBA, DCOM и Globe — три разные системы объектов со своими преимуществами и недостатками. После детального обсуждения принципов организации этих трех систем давайте сравним их между собой. Начнем с целей, которые ставились при разработке каждой системы.

9.4.1. Философия

Системы CORBA, DCOM и Globe разрабатывались с разными целями. CORBA представляет собой фактически результат попыток создания стандартной платформы промежуточного уровня, на которой могли бы совместно работать приложения от разных производителей. Сотни участников различных комитетов старались прийти к соглашению о том, как должны выглядеть интерфейсы разнообразных компонентов CORBA, и поразительно уже то, что они вообще о чем-то сумели договориться.

Основной целью разработки DCOM было расширение функциональных возможностей при сохранении совместимости с существующими версиями, вклю-

ченными в ранние системы Windows. Неплохо, что старые приложения по-прежнему работоспособны, гораздо хуже, что когда-то сделанные неудачные решения теперь практически невозможно исправить. Модель DCOM чрезвычайно сложна, и это удивительно, особенно учитывая тот факт, что она в отличие от CORBA *не является* результатом работы многочисленных комитетов и комиссий.

Система Globe представляет собой типичный образец исследовательской работы. Она была разработана небольшим коллективом, предпочитавшим простоту функциональности. Им не нужно было беспокоиться о мнении других людей, не нужно было добиваться совместимости с предыдущими версиями. Как и в большинстве других исследовательских проектов, план Globe, в сущности, прост и понятен. Однако Globe — незавершенная система и многие ее аспекты требуют доработки. Основная цель разработки Globe — обеспечение масштабируемости.

Главные различия объектных моделей, поддерживаемых тремя системами, заключаются в следующем. В CORBA и DCOM используется модель удаленных объектов. CORBA предлагает гибкую и непротиворечивую объектную модель. Сила подхода CORBA состоит в том, что подобная система обеспечивает высокую степень прозрачности операций определения местонахождения и доступа к объектам. Клиенту нелегко заметить разницу между локальным и удаленным объектами. Объекты обладают состоянием, могут быть глобально идентифицированы, а ссылки легко передаются от клиента к клиенту и от машины к машине. Кроме того, объекты могут быть как нерезидентными, так и сохраненными.

Модель DCOM значительно проще модели CORBA, и мы могли бы даже по критиковать ее за излишнюю простоту. Объекты в DCOM нерезидентны, не имеют глобальных идентификаторов и в некоторых случаях, как предполагается, не имеют состояния. Эта модель нарушает многие принципы, лежащие в основе технологии распределенных объектов.

Система Globe поддерживает правильные объекты. Особенность ее объектной модели, однако, состоит в том, что объекты в Globe могут быть реально реплицированы и распределены по нескольким машинам. Более того, объект определяет, как можно выполнять распределение и репликацию его состояния. Другими словами, объект инкапсулирует собственные правила распределения и реализацию. Кроме того, он может также инкапсулировать правила и реализации защиты, обработки ошибок и т. д.

Сравнивая службы этих систем, мы также видим серьезные различия. CORBA содержит обширный набор служб, обширный настолько, что функциональность отдельных служб нередко пересекается. С другой стороны, DCOM предлагает смесь собственных служб со службами окружения, такими как службы именования и каталогов. В Globe реализован только абсолютный минимум служб — простейшая служба именования и расширенная служба локализации. Это соответствует философии построения Globe, однако совершенно ясно, что для превращения Globe в действительно работоспособную распределенную систему общего назначения ей необходимо иметь множество дополнительных служб.

Другое существенное отличие трех систем друг от друга состоит во взгляде на интерфейсы. CORBA предоставляет стандартный язык IDL, на котором составляются определения интерфейсов, превращаемые затем в тексты программ на

выбранном языке программирования. С точки зрения переносимости и межоперационного взаимодействия такое отображение в язык программирования, в сущности, стандартизовано. Основное преимущество такого подхода состоит в том, что CORBA не зависит от кода, генерируемого компиляторами. В сущности, CORBA определяет межоперационное взаимодействие на уровне языка программирования.

В противоположность CORBA системы DCOM и Globe поддерживают бинарные интерфейсы. При подобном подходе интерфейсы объектов определяются независимо от языка программирования. В DCOM достоинства бинарных интерфейсов проявляются на примере многочисленных приложений, написанных на смеси таких языков, как C, Java и Visual Basic. В Globe используется аналогичный подход.

9.4.2. Связь

Рассмотрим теперь, как реализована связь в системах. CORBA изначально использует для связи только простое синхронное обращение к методам. Эта базовая модель поддерживает и асинхронную связь — в форме односторонних и отложенных синхронных запросов. В настоящее время модель взаимодействия CORBA уже не так проста и обладает богатым набором возможностей. Это асинхронные взаимодействия в форме событий и уведомлений с другими средствами для публикации, потребления и фильтрации событий. Кроме того, CORBA поддерживает службу сообщений, в которой возможно сочетание обратных вызовов и опроса сообщений.

В этом аспекте модель DCOM похожа на CORBA. Она обеспечивает синхронное обращение к удаленным методам, но, кроме того, и разнообразные (причем несовместимые) механизмы обратного вызова. Как и CORBA, она также поддерживает работу с сообщениями.

Globe допускает только синхронные обращения к методам. В этой системе нет ни событий, ни обратных вызовов, ни сообщений. Поскольку объекты Globe могут быть реплицированными, объект не может обратиться к другому объекту. Для того чтобы обойти проблемы, связанные с репликацией сообщений, о которых мы говорили в главе 6, необходимы специальные меры [288]. Проблема поддержки цепочек обращений не решена до сих пор. Отметим, что эта проблема не ограничивается Globe; системы CORBA и DCOM также столкнутся с ней, как только попытаются реплицировать объекты.

9.4.3. Процессы

Каждая из трех систем предлагает собственный вариант сервера объектов, совершенно непохожий на другие. Сервер объектов CORBA наиболее гибкий и универсальный. Он поддерживает четкое разделение между реализацией объектов в понятиях методов, управлением объектами и обращениями к методам при помощи адаптера объектов CORBA. Такое разделение помогает разрабатывать распределенные объекты и использовать их в распределенной системе.

Серверы объектов DCOM — это ответ Microsoft на необходимость разработки распределенных объектов. Они не такие гибкие, как серверы объектов CORBA с их адаптером POA, а управление объектами и обращением к методам в основном жестко закодировано в исполняющей системе. Кроме того, когда дело доходит до поддержки потоков выполнения, разработчики объектов до некоторой степени уже не властны над ними. JIT-активизация, которую поддерживают серверы DCOM, может повысить производительность, но работает, только если объект не имеет информации о состоянии. Ценность этого подхода для любой объектной технологии остается под большим вопросом.

Сервер объектов Globe несколько примитивнее, но его функциональность сравнима с адаптером POA системы CORBA. Эта функциональность в основном является следствием того факта, что объекты могут иметь свою собственную реализацию подобъекта управления и подобъекта репликации. Сервер объектов Globe просто выполняет то, что ему командуют хранящиеся на нем объекты. Отсутствует необходимость в конфигурировании сервера, как в случае POA системы CORBA. Однако поскольку объекты в Globe в основном могут определять лишь правила своего распространения, репликации и защиты, оставшиеся виды правил требуют поддержки со стороны сервера объектов. Так, например, в настоящее время имеется лишь ограниченная поддержка сохранности объектов.

9.4.4. Именование

Основная разница между CORBA, DCOM и Globe проявляется при сравнении служб именования этих систем. В CORBA имеется множество механизмов именования и ссылок на объекты. В частности, в число этих механизмов входят выделенная служба именования, службы свойств и обмена. В сущности, каждая из этих служб возвращает одну или несколько ссылок на объекты. Ссылка на объект в CORBA указывает непосредственно на сервер объектов, на котором находится интересующий нас объект. Кроме того, она может указывать на хранилище реализаций, которое отслеживает текущий сервер объекта, или даже непосредственно активизировать объект в случае необходимости.

Другими словами, ссылки на объекты в CORBA зависят от местоположения самих объектов. В результате при перемещении объектов ссылки становятся неверными. Такой подход обычен для распределенных систем небольшого размера, в которых относительно просто создать эффективную службу локализации. Однако решения такого типа невозможно масштабировать, и в крупномасштабных глобальных системах они неприменимы.

Модель DCOM сама по себе не имеет службы именования, а пользуется существующими службами своего окружения, такой как Active Directory. Если подобная служба недоступна, найти объект по его имени не представляется возможным.

Одна из форм ссылок на объекты в DCOM — это указатели на интерфейсы. Маршалинг этих указателей обычно требует маршалинга и доставки в нужное место клиентского заместителя. Если ограничиться стандартным маршалингом, дело немного упрощается, но, вообще говоря, передача клиенту указателя на ин-

терфейс в качестве ссылки на объект — это непростое занятие, особенно когда требуется что-то необычное.

Ссылки на сохраненные объекты имеют форму моникеров (в [333] вместо термина *moniker* используют термин *kludge*). Моникеры — достаточно сложная тема. Они решают многие проблемы ссылок на объекты, однако вопрос о том, насколько это лекарство лучше болезни, остается открытым.

Подход, принятый в Globe, в корне отличается от подходов CORBA и DCOM с жестким разделением между именованием и адресацией объектов. Использование в системе Globe независимых от местоположения долгоживущих идентификаторов объектов позволяет изменить имя объекта, не нарушая отображения имени на адрес, которое хранится в традиционной службе именования. Изменение адреса также не оказывает влияния на отображение имени на адрес. Мы обсуждали достоинства этого подхода в главе 4. Основной его недостаток состоит в необходимости выделенной и хорошо масштабируемой службы локализации. Разработка такой службы — дело далеко не простое.

Именованние объектов в Globe при помощи строк символов относительно несложно. К сожалению, отсутствует служба каталогов, которая позволила бы клиенту искать объекты, исходя из требуемых свойств этих объектов. Подобная служба нужна многим распределенным системам.

9.4.5. Синхронизация

Механизмы синхронизации, имеющие вид традиционных служб блокировки и транзакций, реализованы только в CORBA и DCOM. Globe не имеет механизмов синхронизации, кроме средств обеспечения синхронизации между объектами, которые реализуются подобъектами репликации.

9.4.6. Кэширование и репликация

В механизмах кэширования и репликации трех систем имеются серьезные различия.

Определение репликации в CORBA потребовало длительных усилий, связанных в основном с трудностями интеграции самого понятия репликации в представление об удаленном объекте. Подход к репликации в CORBA в настоящее время состоит в том, что выделенный сервер репликации создает иллюзию существования одного удаленного объекта там, где на самом деле существует группа объектов. Другими словами, с точки зрения клиента нет в сущности никакой разницы, реплицируется объект или нет.

Сервер репликации определяет, как, где и когда происходит репликация. В настоящее время существует несколько наборов правил репликации, которые можно ассоциировать с группами объектов. Все эти правила реализуют последовательную непротиворечивость. Типичная реализация базируется на механизме полностью упорядоченной групповой рассылки.

Не существует каких-то отдельных средств поддержания кэширования, но его легко добавить к существующей реализации, используя механизм перехватчи-

ков. Реализация подсистемы кэширования, однако, полностью возлагается на разработчика приложений. Проблема такого подхода состоит в необходимости проделывать значительную работу всякий раз, когда требуется кэширование.

В DCOM нет поддержки кэширования и репликации. Эти вопросы приложения должны решать для себя сами.

В Globe используется совершенно другой подход, нежели в CORBA. В Globe каждый локальный объект распределенного разделяемого объекта содержит под-объект репликации. Поскольку интерфейс подобъекта репликации стандартизован, объекту могут быть с легкостью приписаны различные реализации разных стратегий репликации. В результате мы получаем гибкость репликации, значительно более высокую, чем та, которую обеспечивает ограниченный список стратегий репликации, принятый в CORBA. Тот же самый механизм можно использовать и для создания механизмов кэширования.

Хотя Globe и предоставляет универсальный механизм поддержки репликации в объектах, в настоящее время для нее не существует большого набора реализаций подобъектов репликации. Другими словами, в настоящее время разработчики часто должны создавать свою собственную реализацию стратегии репликации. В этом смысле подобъекты репликации Globe можно сравнить с перехватчиками CORBA, которые предоставляют только механизм, при отсутствии подсистем, которые можно было бы немедленно использовать для разработки распределенных объектов.

9.4.7. Отказоустойчивость

Отказоустойчивость в CORBA, как мы говорили ранее, в основном обеспечивается механизмами репликации. Кроме того, службы транзакций и параллельного доступа также способствуют разработке отказоустойчивых служб CORBA.

DCOM обеспечивает отказоустойчивость системы сходным образом, при помощи автоматических транзакций, которые поддерживаются выделенным координатором транзакций.

Globe не поддерживает транзакций, а обеспечение отказоустойчивости базируется только на репликации. Существует не слишком много способов восстановления после сбоев, сводящихся в конечном итоге к сохранению состояния в локальном долговременном хранилище.

9.4.8. Защита

И, наконец, сравнивая защиту в CORBA, DCOM и Globe, мы снова обнаруживаем между этими системами серьезные различия.

В CORBA определена полностью защищенная архитектура удаленных объектов. Разбираясь в многочисленных механизмах защиты, следует позаботиться об отделении методов от правил, чтобы иметь возможность определить, где и когда требуются функции защиты. Сама по себе служба защиты представлена в виде привязанных к приложению объектов правил. При обработке брокером ORB запроса или ответа он обращается к объектам правил. Объект правил можно скон-

фигурировать под приложение, обычно он задействует локальные службы защиты. Так, например, объект правил для аутентификации может быть основан на системе Kerberos.

Механизм обращения к объектам правил по-прежнему базируется на перехвате вызовов. Чтобы сделать это безопасным образом, CORBA поддерживает специальные защищенные перехватчики, используемые для обращения к объектам правил и контроля доступа.

Другой подход используется в DCOM. Защита поддерживается на декларативном уровне путем указания в реестре требований объекта по активизации, контролю доступа и аутентификации. Выполнение этих требований совместно обеспечивают операционная система и DCOM. Альтернативный подход состоит в том, чтобы позволить приложениям при инициализации объекта самим выбирать, к каким службам защиты они хотят обращаться для вызова специальных функций.

Защита в Globe частично реализована на базе выделенных подобъектов защиты. Эти подобъекты связываются с локальными службами защиты, такими как Kerberos. Кроме того, они решают вопросы защиты, связанные с взаимодействием, репликацией и обращениями к подобъектам семантики. Однако архитектура защиты Globe остается незавершенной.

Результаты сравнения систем CORBA, DCOM и Globe представлены в табл. 9.20.

Таблица 9.20. Сравнение CORBA, DCOM и Globe

Критерий	CORBA	DCOM	Globe
Цели разработки	Межоперационная совместимость	Функциональность	Масштабируемость
Объектная модель	Удаленные объекты	Удаленные объекты	Распределенные объекты
Службы	В основном собственные	Из окружения	Несколько
Интерфейсы	На основе IDL	Бинарные	Бинарные
Синхронная связь	Да	Да	Да
Асинхронная связь	Да	Да	Нет
Обратные вызовы	Да	Да	Нет
События	Да	Да	Нет
Сообщения	Да	Да	Нет
Сервер объектов	Гибкий (POA)	Жестко кодированный	Определяемый объектами
Служба каталогов	Да	Да	Нет
Служба обмена	Да	Нет	Нет
Служба именования	Да	Да	Да
Служба локализации	Нет	Нет	Да
Ссылки на объекты	По местоположению объекта	Указатель на интерфейс	Правильный идентификатор

продолжение ➤

Таблица 9.20 (продолжение)

Критерий	CORBA	DCOM	Globe
Синхронизация	Транзакции	Транзакции	Только между объектами
Поддержка репликации	Выделенный сервер	Отсутствует	Выделенный подобъект
Транзакции	Да	Да	Нет
Отказоустойчивость	Посредством репликации	Посредством транзакций	Посредством репликации
Поддержка восстановления	Да	Посредством транзакций	Нет
Защита	Различные механизмы	Различные механизмы	Требуется доработки

9.5. Итоги

Распределенные системы объектов почти всегда базируются на модели удаленных объектов. Такой подход часто требует дополнительной настройки, например, для поддержки кэширования или репликации. В CORBA для изменения исходящих и входящих запросов используются перехватчики. В DCOM пользовательский маршalling требует настройки клиентского заместителя. В Globe настройка выполняется с помощью разных локальных объектов, каждый из которых размещается на собственной машине, а все вместе они образуют единый распределенный объект.

Помимо синхронных обращений к методам распределенные системы объектов, такие как CORBA или DCOM, поддерживают альтернативные средства обращения, включая события и асинхронные вызовы методов. Globe не предоставляет таких альтернатив и, в сущности, поддерживает только синхронные обращения.

Организация серверов объектов в различных системах, за исключением некоторых тонкостей, очень похожа. Во всех случаях сервер способен поддерживать более одного объекта. Различия обнаруживаются в гибкой настройке сервера. В CORBA гибкость обеспечивается при помощи адаптеров объектов. DCOM предоставляет стандартный сервер объектов, который может быть перестроен под конкретное приложение. В Globe серверы объектов относительно просты, поскольку те или иные специальные свойства предполагается реализовывать внутри распределенных объектов.

Различия в механизмах именования распределенных систем объектов имеют место на уровне ссылок на объекты. Поддержка осмысленных имен практически одинакова. И в CORBA, и в DCOM внутрисистемные ссылки на объекты зависят от их местоположения. Эти ссылки содержат информацию о местоположении серверов, на которых располагаются объекты. В противоположность им ссылки в Globe не зависят от местоположения, но для этой системы требуется глобальная служба локализации для разрешения ссылок в контактные адреса. Контактный адрес определяет, где и как можно найти указанный объект.

Репликация в CORBA поддерживается только в плане отказоустойчивости. DCOM вообще не поддерживает репликацию, разработчик приложения должен при необходимости использовать специализированный сервер репликации или явную программную репликацию. В Globe репликация поддерживается каждым объектом в отдельности посредством подобъекта репликации, который реализует конкретный протокол репликации объекта, частью которого он является.

С репликацией тесно связаны вопросы отказоустойчивости. CORBA обеспечивает отказоустойчивость при помощи службы репликации, в основании которой лежит базовая служба надежной групповой рассылки. В DCOM отказоустойчивость поддерживается при помощи транзакций (автоматических). Globe поддерживает отказоустойчивость только через репликацию, а механизмов восстановления не имеет.

И, наконец, каждая из распределенных систем, обсуждаемых в этой главе, имеет средства защиты. CORBA предлагает полностью защищенную архитектуру, которая позволяет обеспечить индивидуальную защиту каждого объекта. В DCOM защита тесно связана с организацией доступа к существующим службам защиты, таким как Kerberos. В Globe защита также определяется для каждого объекта в отдельности, при этом объектам предоставляются средства для привязки к существующим службам защиты. Вопросы защиты в Globe продолжают исследоваться и в настоящее время.

Вопросы и задания

1. Почему для определения интерфейсов объектов используется язык определения интерфейсов (IDL)?
2. Когда могут пригодиться механизмы динамического обращения CORBA? Приведите пример.
3. Какой из шести видов связи, обсуждавшихся в разделе 2.4, соответствует модели обращений, принятой в CORBA?
4. Опишите простой протокол, реализующий семантику обращений к объектам «максимум однажды».
5. При асинхронном методе обращения нужно ли, чтобы клиентские и серверные объекты в CORBA были сохраняемыми?
6. В сообщении Request протокола GIOP ссылка на объект и имя метода являются частями заголовка сообщения. Почему их нельзя поместить непосредственно в тело сообщения?
7. Поддерживает ли CORBA правило обращений «один поток выполнения — один объект», которое мы рассматривали в главе 3?
8. Пусть имеется две системы CORBA, каждая из которых имеет свою службу именования. Опишите, как можно объединить две службы именования в одну объединенную службу именования.
9. Мы утверждали, что при привязке к объекту CORBA клиентский брокер ORB может выбрать дополнительные службы защиты на основе ссылок на объекты. Как клиентский брокер ORB может узнать о существовании этих служб?

10. Если ORB системы CORBA использует несколько перехватчиков, не имеющих отношение к защите, насколько важен порядок вызова этих перехватчиков?
11. Проиллюстрируйте, каким образом очередь транзакций может быть частью распределенной транзакции, как было описано в этой главе.
12. Если сравнивать DCOM с CORBA, какой маршалинг обеспечивает CORBA, стандартный или пользовательский? Дайте подробный ответ.
13. Пусть в DCOM имеется метод m , содержащий интерфейс X , который в качестве входного параметра получает указатель на интерфейс Y . Объясните, что произойдет с маршалингом, если клиент, хранящий указатель интерфейса на реализацию заместителя X , обратится к m .
14. Требуется ли для пользовательского маршалинга в DCOM, чтобы процесс, получивший «упакованный» (путем маршалинга) заместитель, работал на машине того же типа, что и процесс-отправитель?
15. Опишите алгоритм переноса объекта DCOM на другой сервер.
16. Нарушает JIT-активизация представление об объекте или соответствует этому представлению?
17. Опишите, что произойдет, если два клиента с разных машин задействуют один и тот же файловый моникер для привязки к одному объекту. Сколько будет создано экземпляров объекта, один или два?
18. Какой существует очевидный способ реализации событий в Globe, не нарушающий принципа пассивности объектов?
19. Приведите пример, когда использование (невнимательное) механизмов обратного вызова легко может привести к нежелательному результату.
20. В CORBA узлы графа имен, то есть каталоги, также считаются объектами. Имеет ли смысл и в Globe считать каталоги распределенными разделяемыми объектами?
21. Предположим, что на сервер объектов Globe установлен локальный сохраняемый объект. Предположим также, что этот объект имеет контактный адрес. Должен ли сохраняться этот адрес при отключении сервера?
22. В этой главе приводился пример использования системы Kerberos для обеспечения защиты в Globe. Имело бы смысл шифрование талона каким-нибудь другим общим ключом (вместо ключа $K_{B,TGT}$) объекта и службы предоставления талонов?

Глава 10

Распределенные файловые системы

10.1. Сетевая файловая система компании Sun

10.2. Файловая система Coda

10.3. Другие распределенные файловые системы

10.4. Сравнение распределенных файловых систем

10.5. Итоги

Совместное использование данных — фундаментальное требование распределенных систем. Поэтому нас не должно удивлять, что в основе множества распределенных приложений лежат распределенные файловые системы. Распределенные файловые системы позволяют нескольким процессам в течение продолжительного времени совместно работать с общими данными, обеспечивая их надежность и защищенность. По этой причине они нередко используются в качестве базового уровня распределенных систем и приложений. В этой главе мы рассмотрим распределенные файловые системы с точки зрения парадигмы распределенных систем общего назначения.

Мы детально изучим две совершенно разные распределенные файловые системы. В качестве первого примера мы рассмотрим сетевую файловую систему NFS компании Sun Microsystems, имеющую гигантское число установок и в настоящее время в форме версий для Интернета постепенно разрастающуюся до глобальных масштабов. Большинство реализаций NFS основаны на спецификации версии 3. Недавно была описана и выпущена пробная редакция версии 4.

Совершенно другой тип распределенных файловых систем представляет Coda. Coda — это потомок файловой системы AFS, крупномасштабной системы, которая разрабатывалась исключительно с целью обеспечить максимальную масштабируемость. От множества других файловых систем Coda отличает поддержка непрерывных операций, для которых границы сегментов сети — не преграда. Так, в частности, подобная поддержка очень удобна для мобильных пользователей (то есть пользователей переносных компьютеров), которые вынуждены часто отключаться от сети.

Также мы кратко рассмотрим еще три системы. Plan 9 — это распределенная система, в которой все ресурсы рассматриваются как файлы. В этом смысле ее можно считать распределенной системой файлов. Следующая система, которую мы рассмотрим, — это xFS, характерная тем, что в ней отсутствуют серверы, а файловую систему реализуют клиенты. И наконец, мы познакомимся с системой SFS, которая выделяется среди других распределенных файловых систем масштабируемой системой защиты.

Все семь принципов, о которых мы рассуждали в начале книги, применимы и к распределенным файловым системам, и в этой главе мы рассмотрим, как реализуется каждый из них на примерах существующих систем. В заключение мы проведем сравнительный анализ различных файловых систем.

10.1. Сетевая файловая система компании Sun

Мы начнем обсуждать распределенные файловые системы с *сетевой файловой системы (Network File System, NFS)* компании Sun Microsystem. Система NFS изначально была разработана Sun для использования в рабочих станциях на базе UNIX, но затем была реализована и на многих других платформах. Основная концепция, нашедшая применение в NFS, состоит в том, что каждый файловый сервер имеет стандартное представление своей собственной локальной файловой системы. Другими словами, неважно, как именно реализована локальная файловая система — каждый сервер NFS поддерживает одну и ту же модель. В эту модель входит протокол связи, который позволяет клиентам получить доступ к хранящимся на сервере файлам. Такой подход позволяет разнородным наборам процессов, которые, возможно, работают под управлением разных операционных систем и на разных машинах, совместно использовать единую файловую систему.

NFS имеет относительно долгую историю. Первая версия никогда не покидала пределов компании Sun. Вторая версия NFS была встроена в операционную систему SunOS 2.0 и описана в [395]. Это произошло за несколько лет до выпуска версии 3 [346]. Обе версии подробно описаны в [80].

NFS версии 3 в настоящее время претерпевает серьезные изменения, которые должны привести к созданию новой версии [413]. Изменения должны привести в первую очередь к повышению производительности при работе через Интернет, что превратит NFS в настоящую глобальную файловую систему. В систему были внесены и другие усовершенствования, связанные с защитой и межоперационной совместимостью. Далее мы детально рассмотрим различные аспекты NFS, уделяя основное внимание версиям 3 и 4.

10.1.1. Обзор

Как мы уже говорили, NFS — это не столько файловая система, сколько набор протоколов, которые создают перед клиентом представление распределенной

файловой системы. В этом смысле NFS можно сравнить с системой CORBA, которая, в сущности, также существует только в виде описаний. Как и CORBA, NFS имеет множество реализаций, работающих под управлением различных операционных систем на разных машинах. Протоколы NFS разрабатывались таким образом, чтобы их разнообразные реализации с легкостью могли работать совместно.

NFS Architecture

Модель, лежащая в основе NFS, — это *удаленная файловая служба (remote file service)*. В такой модели клиенты получают прозрачный доступ к файловой системе, которую поддерживает удаленный сервер. Однако клиенты обычно не осведомлены об истинном местоположении файлов. Вместо этого им предоставляется интерфейс файловой системы, похожий на интерфейс стандартной локальной файловой системы. Обычно в интерфейсе, который имеется у клиента, определены разнообразные операции с файлами, но за реализацию этих операций отвечает сервер. Поэтому такая модель называется также *моделью удаленного доступа (remote access model)*. Она показана на рис. 10.1, а.

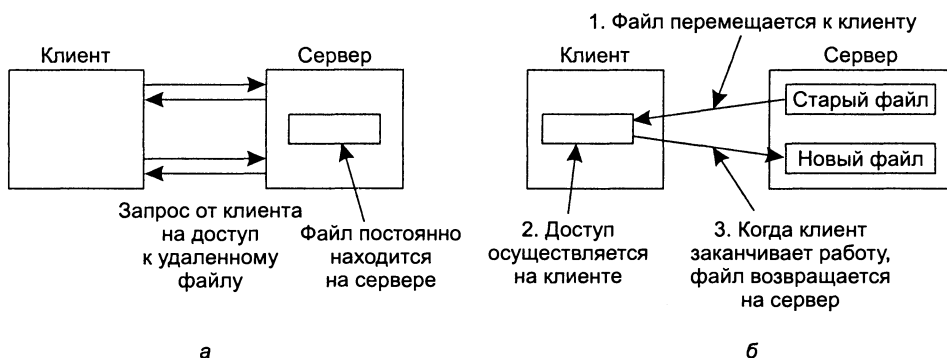


Рис. 10.1. Модель удаленного доступа (а). Модель загрузки-выгрузки (б)

В противоположность ей в *модели загрузки-выгрузки (upload/download model)* после загрузки файла с сервера клиент получает к нему локальный доступ, как показано на рис. 10.1, б. Когда клиент заканчивает работу с файлом, файл выгружается обратно на сервер, после чего с ним может работать другой клиент. Таким образом используется Интернет-служба FTP — клиент загружает файл целиком, вносит в него изменения, а затем возвращает на место.

Система NFS реализована для многих платформ, хотя версии для UNIX преобладают. Фактически для всех современных UNIX-систем NFS реализуется в соответствии с многоуровневой архитектурой, показанной на рис. 10.2.

Клиент, получая доступ к файловой системе, использует системные вызовы локальной операционной системы. Однако интерфейс локальной файловой системы UNIX заменяется интерфейсом *виртуальной файловой системы (Virtual File System, VFS)*, который в настоящее время является де-факто стандартом интерфейса распределенных файловых систем [237]. Операции с интерфейсом VFS передаются локальной файловой системе или специальному выделенному ком-

поненту, известному под названием *клиента NFS (NFS client)* и отвечающему за предоставление доступа к файлам, хранящимся на удаленном сервере. В NFS все взаимодействие между клиентом и сервером осуществляется посредством вызовов RPC. Клиент NFS реализует операции файловой системы NFS в форме запросов RPC, посылаемых серверу. Отметим, что набор операций, который предоставляет интерфейс VFS, может отличаться от набора, предоставляемого клиентом NFS. Основная идея VFS состоит в том, чтобы замаскировать различия между разными файловыми системами.

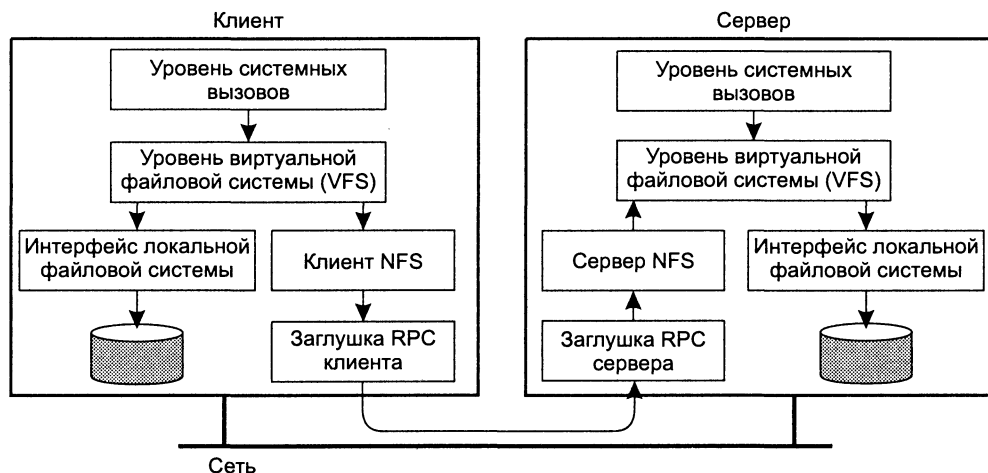


Рис. 10.2. Базовая архитектура NFS для UNIX-систем

Подобным же образом организован и сервер. Сервер NFS отвечает за обработку приходящих запросов клиентов. Заглушка извлекает запросы RPC из сообщений, а сервер NFS преобразует их в корректные операции с файлами интерфейса VFS, которые передаются на уровень VFS. VFS отвечает за реализацию локальной файловой системы, обеспечивающей работу с реальными файлами.

Важное достоинство этой схемы заключается в том, что NFS практически не зависит от локальных файловых систем. В принципе безразлично, какую файловую систему — UNIX, Windows 2000 или даже архаичную MS-DOS — реализует операционная система клиента или сервера. Единственное, что важно, — чтобы эти файловые системы не противоречили модели файловой системы, которую поддерживает NFS. Так, например, система MS-DOS с ее короткими именами файлов не может поддерживать полную прозрачность реализации сервера NFS.

Модель файловой системы

Модель файловой системы, которую поддерживает NFS, аналогична модели UNIX-систем. Файлы трактуются как неинтерпретируемые последовательности байтов. Они организованы в виде иерархического графа имен, узлы которого представляют собой каталоги и файлы. NFS поддерживает жесткие ссылки, а так же символические ссылки, имеющиеся в стандартных файловых системах UNIX.

Файлы имеют имена, но доступ к ним производится при помощи UNIX-подобных *дескрипторов файлов* (*file handle*), о которых мы далее поговорим подробно. Другими словами, для доступа к файлу клиент должен при помощи службы именования разрешить его имя и получить ассоциированный с ним дескриптор файла. Кроме того, каждый файл имеет несколько атрибутов, значения которых можно читать и изменять. Мы вернемся к вопросам именования файлов чуть ниже.

В табл. 10.1 сведены воедино общие операции над файлами, поддерживаемые, соответственно, NFS версий 3 и 4. Операция `create` используется для создания файлов, но имеет разный смысл в версиях 3 и 4. В версии 3 эта операция служит для создания обычного файла. Специальные файлы создаются при помощи другой операции. Операция `link` позволяет создавать жесткие ссылки, а при помощи операции `symlink` создаются символические ссылки. Операция `mkdir` служит для создания вложенных каталогов. Специальные файлы, такие как файлы устройств, сокеты и именованные каналы, создаются операцией `mknod`.

Таблица 10.1. Неполный список файловых операций в NFS

Операция	v. 3	v. 4	Описание
<code>create</code>	Да	Нет	Создать обычный файл
<code>create</code>	Нет	Да	Создать нестандартный файл
<code>link</code>	Да	Да	Создать жесткую ссылку на файл
<code>symlink</code>	Да	Нет	Создать символическую ссылку на файл
<code>mkdir</code>	Да	Нет	Создать вложенный каталог в текущем каталоге
<code>mknod</code>	Да	Нет	Создать специальный файл
<code>rename</code>	Да	Да	Изменить имя файла
<code>remove</code>	Да	Да	Удалить файл из файловой системы
<code>rmdir</code>	Да	Нет	Удалить пустой вложенный каталог из каталога
<code>open</code>	Нет	Да	Открыть файл
<code>close</code>	Нет	Да	Закрыть файл
<code>lookup</code>	Да	Да	Найти файл по имени файла
<code>readdir</code>	Да	Да	Прочитать элементы в каталоге
<code>readlink</code>	Да	Да	Прочитать путь к файлу, записанный в символической ссылке
<code>getattr</code>	Да	Да	Получить значение атрибута файла
<code>setattr</code>	Да	Да	Установить значение одного или нескольких атрибутов файла
<code>read</code>	Да	Да	Прочитать данные, содержащиеся в файле
<code>write</code>	Да	Да	Записать данные в файл

В версии 4 картина совершенно иная. В новой версии операция `create` используется для создания нестандартных файлов, в том числе символических, ссылок, каталогов и специальных файлов. Жесткие ссылки по-прежнему создаются операцией `link`, а для создания обычных файлов используется операция

open, которая является новой для NFS и радикально меняет методы работы с файлами, принятые в прежних версиях. Вплоть до версии 4 система NFS строилась так, что ее файловые серверы не сохраняли информацию о состоянии. По причинам, которые мы обсудим чуть позже в этой главе, в версии 4 от этого принципа пришлось отказаться. В результате можно считать, что при работе с одним и тем же файлом серверы обычно сохраняют состояние файла между операциями.

Операция rename используется для изменения имени существующего файла.

Файлы удаляются при помощи операции remove. В версии 4 эта операция используется для удаления файлов любых типов. В предыдущей версии для удаления вложенных каталогов существовала отдельная операция rmdir. Файлы удаляются по имени, и в результате число жестких ссылок на них уменьшается на единицу. Если число ссылок падает до нуля, файл можно уничтожить.

Версия 4 позволяет клиентам открывать и закрывать обычные файлы. Открытие несуществующего файла приводит к его созданию. Для открытия файла клиент указывает его имя, а также те или иные значения атрибутов. Так, например, клиент может указать, что файл должен быть открыт для чтения. Если файл успешно открыт, клиент может работать с этим файлом при помощи дескриптора файла. Этот дескриптор используется также и для закрытия файла (операция close). При помощи операции close клиент уведомляет сервер, что доступ к файлу ему больше не нужен. Сервер, в свою очередь, может сбросить информацию о состоянии файла, которую он поддерживал для организации доступа клиента к этому файлу.

Операция lookup используется для получения дескриптора файла по заданному полному пути к нему. В NFS версии 3 операция lookup не разрешала имена за монтажной точкой (напомним, что, как мы говорили в главе 4, монтажная точка — это каталог, который, в сущности, представляет собой ссылку на вложенный каталог в чужом пространстве имен). Так, предположим, что имя /remote/vu соответствует монтажной точке в графе имен. При разрешении имени /remote/vu/mbox операция lookup в NFS версии 3 возвращает дескриптор файла монтажной точки /remote/vu вместе с остатком полного имени (то есть mbox). После этого клиент должен был явно смонтировать файловую систему, необходимую для завершения поиска имени. Файловая система в таком контексте — это набор файлов, атрибутов, каталогов и блоков данных, которые совместно реализованы в виде логического устройства с блочной структурой данных [117, 450].

В версии 4 все стало значительно проще. Теперь lookup в состоянии разрешить имя целиком даже в случае перехода через монтажные точки. Отметим, что такой подход возможен только в том случае, если чужая файловая система уже смонтирована в соответствующую монтажную точку. Клиент в состоянии обнаружить переход через монтажную точку, отслеживая идентификатор файловой системы, который будет возвращен ему по окончании разрешения имени.

Для чтения элементов каталога существует отдельная операция readdir, которая возвращает список пар (*имя, дескриптор файла*) вместе со значениями атрибутов, запрошенных клиентом. Клиент может также определить, сколько элементов ему следует вернуть. Операция возвращает смещение, которое может быть использовано при следующем вызове readdir для чтения следующей порции элементов.

Операция `readlink` используется для чтения данных, содержащихся в символической ссылке. Обычно эти данные соответствуют полному имени, которое затем можно разрешить обычным образом. Отметим, что операция поиска не может работать с символической ссылкой. Вместо этого при обнаружении символической ссылки разрешение имени прекращается, и клиент должен сначала вызвать `readlink`, чтобы получить имя, с которого он сможет продолжить процедуру разрешения имен.

Файлы имеют различные атрибуты. В способах работы с атрибутами файлов состоит еще одно серьезное различие между NFS версий 3 и 4. Подробнее о нем мы поговорим позже. В число стандартных атрибутов входят тип файла (указывающий, что мы имеем дело с каталогом, символической ссылкой, специальным файлом и т. п.), длина файла, идентификатор файловой системы, в которой находится файл, и время последней модификации файла. Атрибуты файла можно считывать и записывать при помощи операций `getattr` и `setattr` соответственно.

И, наконец, операции чтения данных из файла и записи данных в файл. Чтение данных производится при помощи операции `read`. Клиент определяет смещение и число байтов, которые он хочет прочитать. Ему возвращается реальное число прочитанных байтов и дополнительная информация о состоянии (например, о достижении конца файла).

Запись данных в файл производится при помощи операции `write`. Клиент вновь определяет позицию в файле, с которой должна начаться запись, число записываемых байтов и данные. Кроме того, он может дать команду серверу обеспечить гарантированную запись всех данных в устойчивое хранилище (мы обсуждали устойчивые хранилища в главе 7). Серверы NFS должны поддерживать устройства хранения данных, способные пережить исчезновение питания, а также отказы операционной системы и аппаратного обеспечения.

10.1.2. Связь

Важной особенностью структуры NFS является независимость операционных систем, сетевых архитектур и транспортных протоколов. Эта независимость обеспечивает, например, возможность связи клиента, работающего под управлением операционной системы Windows, с файловым сервером UNIX. Эта независимость в значительной степени определяется тем фактом, что сам по себе протокол NFS размещается поверх уровня RPC, скрывающего разницу между различными платформами и сетями.

В NFS все взаимодействие между клиентом и сервером происходит в соответствии с протоколом *RPC для открытых сетевых вычислений (Open Network Computing RPC, ONC RPC)*, формально определенным в [430] вместе со стандартами представления данных после маршалинга [431]. Протокол ONC RPC подобен другим системам RPC, обсуждавшимся в главе 2. Его программные интерфейсы и практическое использование описаны в [69].

Все операции NFS могут быть реализованы в виде вызова одиночной удаленной процедуры на файловом сервере. На самом деле до появления NFS версии 4 ответственность за то, чтобы максимально снизить нагрузку на сервер, нес кли-

ент, в обязанности которого входило делать запросы по возможности простыми. Так, например, для первого чтения данных из файла клиент обычно сначала получал дескриптор файла, используя операцию `lookup`, после чего посылал запрос на чтение, как это показано на рис. 10.3, а.

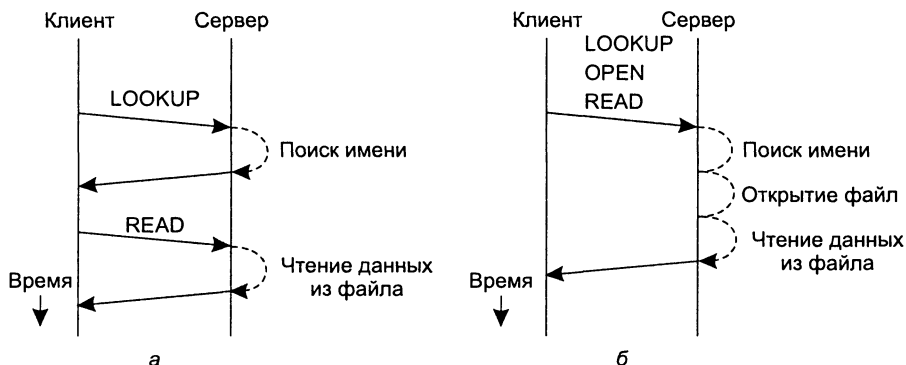


Рис. 10.3. Чтение данных из файла в NFS версии 3 (а). Чтение данных с использованием составной процедуры в версии 4 (б)

Такой подход требовал успешного выполнения двух вызовов RPC. Его недостаток становится очевидным при работе NFS в глобальных системах. В этом случае ожидание второго вызова RPC может привести к резкому падению производительности. Чтобы избежать подобных проблем, NFS версии 4 поддерживает *составные процедуры* (*compound procedures*), в которых несколько вызовов RPC могут быть объединены в один запрос, как показано на рис. 10.3, б.

В нашем примере клиент объединяет запросы на поиск и чтение в один вызов RPC. В случае версии 4 необходимо также открыть файл перед тем, как читать из него данные. После получения дескриптора файла он передается в операцию `open`, после чего сервер продолжает работу, совершая операцию `read`. В результате такого построения вызовов клиент и сервер обмениваются только двумя сообщениями.

Составные процедуры не имеют какой-либо семантики транзакций. Операции, группируемые в составную процедуру, просто выполняются в заданном порядке. Если одновременно выполняются операции других клиентов, то никаких мер для разрешения возможных конфликтов не предпринимается. Если операцию по каким-то причинам выполнить невозможно, последующие операции также не производятся, а полученные промежуточные результаты возвращаются клиенту. Так, например, если поиск имени оказывается неудачным, попыток открытия файла не делается.

10.1.3. Процессы

NFS — это традиционная система клиент-сервер, в которой клиент для выполнения операций над файлами обращается к файловому серверу. Одна из традиционных специфических особенностей NFS по сравнению с другими распре-

деленными файловыми системами состояла в том, что серверы не сохраняли информацию о состоянии. Другими словами, протокол NFS не требовал, чтобы серверы как-то поддерживали состояние клиента. Этот подход соблюдался в версиях 2 и 3, но был пересмотрен в версии 4.

Основное преимущество серверов без фиксации состояния — это их простота. Так, например, если на сервере без фиксации состояния происходит сбой, то, в сущности, у нас нет необходимости проходить фазу восстановления, чтобы перевести сервер в предшествовавшее сбою состояние. Однако, как мы видели в разделе 7.3, следует учитывать, что клиент в этом случае вообще не может претендовать на какую-либо гарантию обработки запроса. Позже мы еще вернемся к вопросу отказоустойчивости NFS.

Подход без фиксации состояния в практических реализациях протокола NFS полностью соблюдался не всегда. Так, например, блокировку файла достаточно сложно осуществить на сервере без фиксации состояния. В случае NFS для решения этой проблемы использовался специальный менеджер блокировок. Кроме того, протоколы идентификации также нуждаются в том, чтобы сервер поддерживал состояние своих клиентов. Тем не менее серверы NFS обычно разрабатывались так, чтобы хранить минимальный объем информации о своих клиентах. В большинстве случаев эта схема работала достаточно успешно.

Начиная с версии 4, разработчики NFS отказались от подхода без фиксации состояния, хотя протокол по-прежнему разрабатывался так, чтобы серверу не приходилось хранить слишком большой объем информации о клиентах. Кроме того, о чем мы уже упоминали, для перехода к серверам с фиксацией состояния были и другие причины. Наиболее важная из них состояла в том, что NFS версии 4 предполагалось использовать в глобальных сетях. При этом необходимо было сделать так, чтобы клиенты могли активно использовать кэширование, что, в свою очередь, требовало эффективных протоколов поддержания непротиворечивости кэша. Подобные протоколы часто хорошо работают в комбинации с сервером, который обеспечивает хранение определенной информации о файлах, используемых его клиентами. Так, например, сервер может назначить аренду каждому файлу, доступ к которому запрашивает клиент, предлагая клиенту исключительное право на чтение или запись в файл на весь срок аренды с возможностью ее продления. Мы еще вернемся к этому вопросу.

Наиболее очевидное отличие от предыдущей версии NFS состоит в поддержке операции *open*, которая изначально требует наличия у сервера информации о состоянии. Кроме того, NFS поддерживает процедуры обратного вызова, при помощи которых сервер может выполнить вызов RPC на клиенте. Понятно, что обратный вызов также требует, чтобы сервер отслеживал состояние своих клиентов.

10.1.4. Именованное

Как и во многих других распределенных файловых системах, именованное играет в NFS важную роль. Основная идея, лежащая в основе модели именования NFS, состоит в том, чтобы предоставить клиентам абсолютно прозрачный доступ к уда-

ленной файловой системе, поддерживаемой сервером. Такая прозрачность достигается в результате того, что клиент может смонтировать удаленную файловую систему в локальную файловую систему клиента, как показано на рис. 10.4.

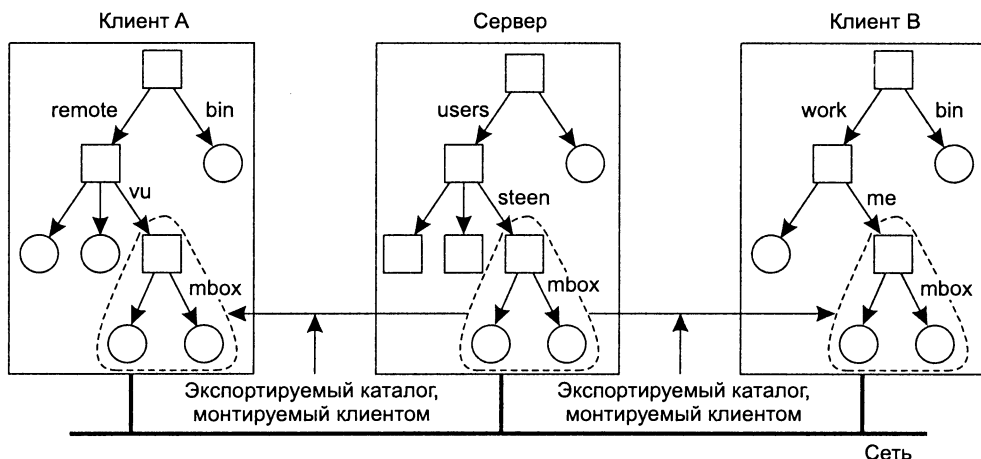


Рис. 10.4. Монтирование части удаленной файловой системы в NFS

Вместо того чтобы монтировать файловую систему целиком, NFS позволяет клиентам монтировать только часть файловой системы. Если сервер открывает клиентам доступ к каталогу и его элементам, то говорят, что сервер *экспортирует* (export) каталог. Экспортированный каталог может быть смонтирован в локальное пространство имен клиента.

Такой подход к построению системы имеет серьезные последствия, поскольку, в принципе, пользователи не могут разделять пространства имен. Как показано на рис. 10.4, файл с именем `/remote/vu/mbox` у клиента *A* называется `/work/me/mbox` у клиента *B*. Таким образом, имя файла зависит от того, как клиент организует свое пространство имен и куда монтирует рассматриваемый каталог. Недостаток подобного подхода в распределенных файловых системах состоит в том, что использовать файлы совместно становится значительно труднее. Так, например, Алиса не может сообщить Бобу о некотором файле, используя то имя, которое дала ему она, поскольку это имя в пространстве имен файлов Боба может иметь совершенно иной смысл.

Существует несколько способов решения этой проблемы, наиболее общим из них является предоставление каждому из клиентов частично стандартизованного внутреннего пространства. Так, например, для монтирования файловой системы, содержащей стандартный набор общедоступных программ, каждый клиент может использовать локальный каталог `/usr/bin`. Точно так же каталог `/local` может по умолчанию использоваться для монтирования локальной файловой системы, размещенной на хосте клиента.

Сам сервер NFS может монтировать каталоги, экспортируемые другими серверами. Однако он не может позволить экспортировать эти каталоги своим кли-

ентам. Клиент должен монтировать каталоги с того сервера, на котором они размещены, как показано на рис. 10.5. Это ограничение введено, в частности, для сохранения простоты конструкции. Если сервер экспортирует монтируемый каталог другого сервера, он должен возвращать специальные дескрипторы файлов, включающие в себя идентификатор сервера. NFS такие усложненные дескрипторы файлов не поддерживает.

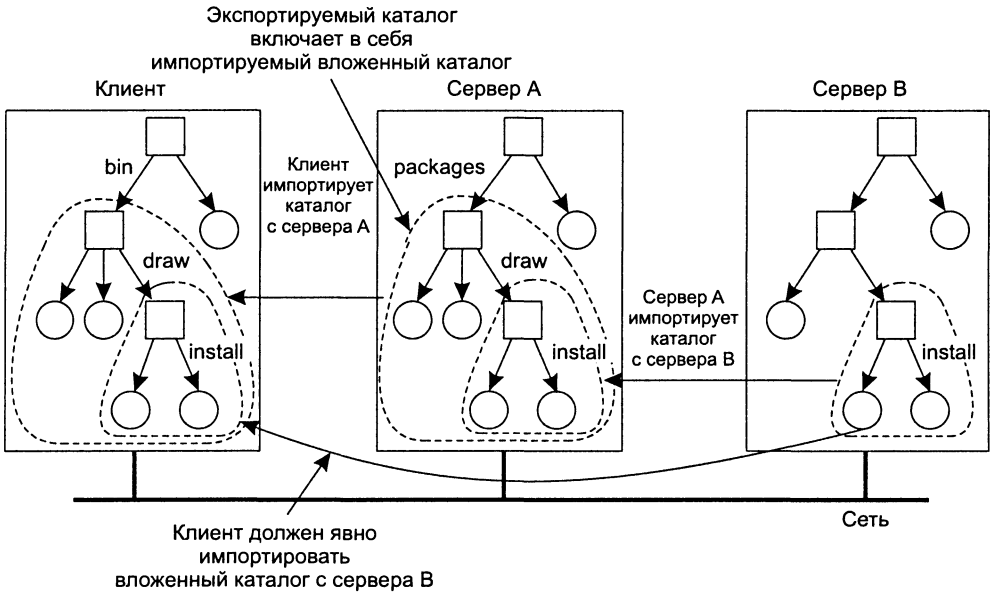


Рис. 10.5. Монтирование вложенных каталогов с нескольких серверов в NFS

Чтобы понять, как это происходит, предположим, что сервер *A* поддерживает файловую систему FS_A , из которой экспортирует каталог `/packages`. Этот каталог содержит вложенный каталог `/draw`, являющийся монтажной точкой файловой системы FS_B , которая экспортируется сервером *B* и монтируется сервером *A*. Пусть *A* экспортирует каталог `/packages/draw` своему клиенту, который монтирует `/packages` в свой локальный каталог `/bin`, как это показано на рис. 10.5.

Если разрешение имен итеративное (как в случае NFS версии 3), то для разрешения имени `/bin/draw/install` клиент, локально разрешив `/bin`, связывается с сервером *A* и запрашивает у него дескриптор файла каталога `/draw`. Сервер *A* в этом случае возвращает дескриптор файла, который включает в себя идентификатор сервера *B*, поскольку только *B* в состоянии разрешить остаток полного имени, в данном случае `/install`. Но как мы уже говорили, такой способ разрешения имен в NFS не поддерживается.

Разрешение имен в NFS версии 3 (и более ранних версиях) строго итеративно в том смысле, что в один момент времени может быть разрешено только одно имя файла. Другими словами, разрешение такого имени, как `/bin/draw/install`, требует трех отдельных обращений к серверу NFS. Более того, за разрешение пол-

ного имени полностью отвечает клиент. NFS версии 4 поддерживает также и рекурсивное разрешение имен. В этом случае клиент может передать серверу полное имя и потребовать, чтобы сервер его разрешил.

В схеме разрешения имен NFS версии 4 имеется и еще одна специфическая особенность. Рассмотрим файловый сервер, поддерживающий несколько файловых систем. При строго итеративном разрешении имен, которое было характерно для версии 3, разрешение завершалось на каталоге, к которому монтируется другая файловая система. При этом возвращался дескриптор каталога. Последующее чтение этого каталога позволяло нам получить только его собственное содержимое, а не содержимое корневого каталога смонтированной в нем файловой системы.

Чтобы понять смысл этого действия, предположим, что в нашем предыдущем примере обе файловые системы, FS_A и FS_B , находятся на одном и том же сервере. Если клиент монтирует `/packages` в свой локальный каталог `/bin`, то поиск имени файла `draw` на сервере приведет к возвращению дескриптора файла `draw`. Следующее обращение к серверу для получения списка элементов каталога `draw` при помощи операции `readdir` приведет к получению списка элементов каталога, изначально хранившихся в FS_A во вложенном каталоге `/packages/draw`. Только если клиент сам смонтирует файловую систему FS_B , он сможет правильно разрешить полное имя `draw/install` относительно `/bin`.

NFS версии 4 решает эту проблему, позволяя операциям поиска файлов пересекать границы монтирования файловых систем на серверах. Так, в частности, операция `lookup` возвратит дескриптор файла смонтированного каталога, а не файла, изначально хранившегося в этом каталоге. Клиент может заметить, что операция `lookup` прошла монтажную точку, если проверит идентификатор файловой системы найденного файла. При необходимости клиент может смонтировать эту файловую систему локально.

Дескрипторы файлов

Дескриптор файла — это ссылка на файл в файловой системе. Он не связан с именем файла, на который ссылается. Дескриптор файла создается сервером, на котором расположена файловая система, и является уникальным для всех файловых систем, экспортируемых сервером. Он создается в момент создания файла. Клиент остается в неведении о содержимом дескриптора файла, он абсолютно непрозрачен. Дескрипторы файлов занимают 32 байта в NFS версии 2, имеют переменную длину до 64 байт в версии 3 и до 128 байт в версии 4. Разумеется, длина дескриптора файла ни для кого не является секретом.

В идеале дескриптор файла реализован как истинный идентификатор файла в данной файловой системе. Это означает, что в течение всего времени существования файла он имеет один и тот же дескриптор. Такое постоянство позволяет клиенту локально сохранить дескриптор файла после того, как соответствующий файл найден по имени. Одно из достоинств такого подхода — высокая производительность. Поскольку для большинства операций с файлами нужен дескриптор файла, а не его имя, клиент может избавиться от повторяющихся операций поиска файла по имени, которые обычно предшествуют операциям с файлами.

Другое преимущество такого подхода состоит в том, что клиент может работать с файлами независимо от их текущих имен.

Поскольку дескриптор файла можно хранить на стороне клиента, важно не позволить серверу повторно использовать дескриптор после удаления файла. В противном случае клиент, используя локально сохраненный дескриптор, может по ошибке получить доступ к другому файлу.

Отметим, что комбинация операции итеративного разрешения имени файла и запрета на пересечение этой операцией монтажной точки создает проблемы с получением исходного дескриптора файла. Для доступа к файлам в удаленной файловой системе клиенту необходимо вместе с именем разрешаемого файла или каталога получить имя сервера с дескриптором каталога, в котором должен производиться поиск. NFS версии 3 решает эту проблему, используя особый протокол монтирования, при помощи которого клиент в действительности монтирует удаленную файловую систему. После монтирования клиент получает *корневой дескриптор файла* (*root file handle*) смонтированной файловой системы, который может использоваться в качестве отправной точки для дальнейшего поиска файла по имени.

В NFS версии 4 эта проблема решается путем предоставления специальной операции `putrootfh`, которая предлагает серверу разрешать все имена файлов относительно корневого дескриптора файлов для файловой системы, которую он поддерживает. Корневой дескриптор может быть использован для поиска любых других дескрипторов файла в файловой системе сервера. Такой подход имеет дополнительное преимущество, состоящее в том, что теперь необходимость в отдельном протоколе монтирования отсутствует. Вместо этого монтирование интегрируется в стандартный протокол поиска файлов. Клиент может смонтировать удаленную файловую систему, просто требуя от сервера при помощи операции `putrootfh` разрешить имена относительно корневого дескриптора файлов.

Автоматическое монтирование

Как мы уже говорили, модель именования NFS, в сущности, предоставляет пользователям их собственные пространства имен. Если пользователи именуют один и тот же файл по-разному, его совместное использование в этой модели затрудняется. Одно из решений этой проблемы состоит в том, чтобы предоставить каждому пользователю частично стандартизованное локальное пространство имен, после чего смонтировать удаленные файловые системы у разных пользователей одинаково.

Другая проблема модели именования NFS связана с принятием решения о том, в какой момент монтировать удаленную файловую систему. Рассмотрим большую систему с тысячами пользователей. Предположим, что каждый из пользователей имеет локальный каталог `/home`, предназначенный для монтирования домашних каталогов других пользователей. Так, например, домашний каталог Алисы может быть доступен для нее локально как `/home/alice`, хотя на самом деле файлы лежат на удаленном сервере. Этот каталог может автоматически монтироваться в тот момент, когда Алиса начнет работать на своей рабочей станции. Кроме того, она может иметь доступ к открытым файлам Боба через каталог Боба `/home/bob`.

Вопрос, однако, состоит в том, должен ли каталог Боба автоматически монтироваться при входе Алисы в систему. Преимущество такого подхода состояло бы в том, что вся работа по монтированию файловых систем оставалась бы абсолютно прозрачной для Алисы. Однако, если такая политика поддерживалась бы для каждого пользователя, любой вход в систему ложился бы большой дополнительной нагрузкой на механизмы связи и администрирования. Кроме того, требовалось бы знать всех пользователей системы заранее. Вероятно, правильнее прозрачно монтировать домашние каталоги других пользователей по мере надобности, то есть тогда, когда в них впервые возникнет необходимость.

Монтирование удаленных файловых систем (или собственно экспортируемого каталога) по запросу осуществляется в NFS *автоматическим монтировщиком* (*automounter*) — программой, которая выполняется в отдельном процессе на машине клиента. Принципы, на которых основана работа автоматического монтировщика, относительно просты. Рассмотрим простой автоматический монтировщик, реализованный на пользовательском уровне сервера NFS в операционной системе UNIX (другие реализации этой программы можно найти в [80]).

Допустим, для каждого пользователя домашние каталоги всех прочих пользователей доступны через каталог /home, как было описано выше. При включении клиентской машины автоматический монтировщик начинает свою работу, монтируя этот каталог. В результате локального монтирования при любой попытке доступа программ к каталогу /home ядро UNIX будет пересылать операцию lookup клиенту NFS, а тот, в свою очередь, перешлет запрос автоматическому монтировщику, который будет играть роль сервера NFS, как показано на рис. 10.6.

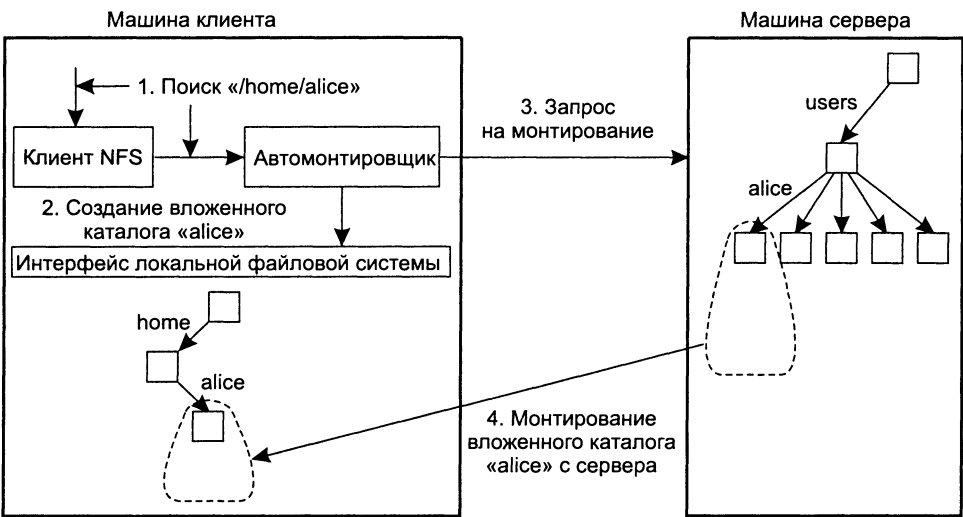


Рис. 10.6. Простой автоматический монтировщик для NFS

Так, например, представим, что Алиса входит в систему. Программа входа в систему пытается прочитать каталог /home/alice, чтобы найти определенную информацию, например сценарии подключения. Автоматический монтировщик,

получив запрос на поиск вложенного каталога `/home/alice`, должен сначала создать вложенный каталог `/alice` в каталоге `/home`. После этого он просматривает сервер NFS, который экспортирует домашний каталог Алисы, и монтирует этот каталог в `/home/alice`. После этого алгоритм входа в систему выполняется далее.

Проблема такого подхода состоит в том, что для обеспечения прозрачности автоматический монтировщик должен участвовать во всех операциях с файлами. Если файл, к которому происходит обращение, недоступен локально по причине того, что соответствующая файловая система еще не смонтирована, автоматический монтировщик будет знать, что делать. То есть он должен обрабатывать все запросы на чтение и запись, даже для файловых систем, которые уже смонтированы. Такой подход порождает серьезные проблемы производительности. Значительно лучше было бы, если бы автоматический монтировщик занимался только монтированием и размонтированием каталогов, все остальное время оставаясь незадействованным.

Простейшее решение — позволить, чтобы автоматический монтировщик монтировал каталоги в специальный вложенный каталог и добавлял символическую ссылку на каждый смонтированный каталог. Этот способ решения проблемы иллюстрирует рис. 10.7.

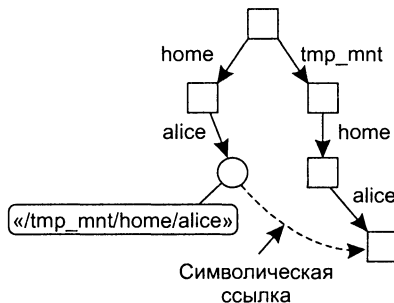


Рис. 10.7. Использование символических ссылок и автоматического монтирования

В нашем примере домашний каталог пользователя монтируется в качестве вложенного в каталог `/tmpmnt`. Когда Алиса входит в систему, автоматический монтировщик монтирует ее домашний каталог в `/tmp_mnt/home/alice` и создает символическую ссылку `/home/alice`, которая указывает на этот вложенный каталог. В этом случае каждый раз, когда Алиса исполняет показанную ниже команду, связь с сервером NFS, экспортирующим домашний каталог Алисы, осуществляется напрямую, без участия автоматического монтировщика:

```
ls -l /home/alice
```

Атрибуты файла

Файл в NFS имеет несколько ассоциированных с ним атрибутов. В версии 3 набор атрибутов был фиксирован, и любая реализация NFS должна была поддерживать эти атрибуты. Однако, поскольку система NFS традиционно строго следовала идеологии, принятой в файловой системе UNIX, полная реализация NFS

на других файловых системах (например, Windows) часто бывала затруднительной или даже невозможной.

В NFS версии 4 набор атрибутов файлов был разделен на набор обязательных атрибутов, которые должна поддерживать любая реализация, набор рекомендованных атрибутов, поддерживать которые желательно, и дополнительный набор именованных атрибутов.

Именованные атрибуты не являются частью протокола NFS, они хранятся в виде массива пар (*атрибут, значение*), в которых атрибуты представлены в виде строк, а их значения представляет собой не интерпретируемую последовательность байтов. Атрибуты хранятся вместе с файлом или каталогом, и NFS поддерживает операции для чтения и записи значений атрибутов. Однако всякая интерпретация атрибутов и их значений полностью отдается приложениям.

В табл. 10.2 перечислено некоторое количество обязательных атрибутов (всего их 12). Эти атрибуты присутствовали и в предыдущих версиях NFS. Следует особо отметить атрибут `CHANGE`, который определяется сервером. По значению этого атрибута клиент может узнать, когда содержимое файла изменялось в последний раз.

Таблица 10.2. Некоторые из обязательных атрибутов файла в NFS

Атрибут	Описание
TYPE	Тип файла (обычный, каталог, символическая ссылка и т. д.)
SIZE	Длина файла в байтах
CHANGE	Индикатор, который позволяет клиенту обнаружить, изменялся ли файл и/или когда он изменялся
FSID	Уникальный идентификатор файловой системы сервера, на котором хранится файл

В табл. 10.3 перечислены некоторые из рекомендованных атрибутов. Текущая версия NFS (версия 4) содержит 43 рекомендованных атрибута файлов. Одним из наиболее важных атрибутов является список контроля доступа, заменивший UNIX-подобные биты полномочий. Также интересен атрибут со списком мест расположения реплик файловой системы. Этот список может быть использован клиентом для связи с другими серверами, например, если текущий сервер окажется недоступным. Полный список рекомендованных атрибутов можно найти в [413].

Таблица 10.3. Некоторые из рекомендуемых атрибутов файла в NFS

Атрибут	Описание
ACE	Список контроля доступа, ассоциированный с файлом
FILEHANDLE	Дескриптор данного файла, установленный сервером
FILEID	Уникальный идентификатор файла в файловой системе
FST_LOCATIONS	Место расположения файловой системы в сети
OWNER	Имя владельца файла в виде символьной строки
TIME_ACCESS	Время последнего доступа к содержимому файла

Атрибут	Описание
TIME_MODIFY	Время последнего изменения содержимого файла
TIME_CREATE	Время создания файла

10.1.5. Синхронизация

Файлы в распределенных файловых системах обычно используются несколькими клиентами совместно. Если разделение не требуется, исчезает главная причина создания распределенных файловых систем. К сожалению, разделение требует издержек: чтобы гарантировать непротиворечивость совместно используемых файлов, необходима синхронизация. Синхронизация — дело относительно несложное, если хранить файлы на центральном сервере. Такое решение, однако, способно вызвать проблемы производительности. Поэтому клиентам часто разрешается иметь локальную копию файла, из которой они читают и в которую они пишут данные. Отметим, что подобная реализация соответствует модели загрузки-выгрузки (см. рис. 10.1, б). Перед тем как мы углубимся в детали синхронизации, давайте попытаемся разобраться, что означает совместно использовать файл.

Семантика совместного использования файлов

Когда два или более пользователей совместно используют (разделяют) один и тот же файл, для предотвращения возникновения определенных проблем необходимо точно определить семантику чтения и записи. Чтобы понять семантику разделения файлов в NFS, рассмотрим сначала некоторые общие понятия.

В однопроцессорных системах, которые допускают разделение файлов процессами, например UNIX, семантика подобных операций обычно состоит в том, что операции чтения следуют за операциями записи, то есть чтение возвращает записанное ранее значение, как показано на рис. 10.8, а. Таким образом, если одна за другой происходят две операции записи, последующее чтение возвращает данные, записанные на диск в ходе последней операции записи. В действительности система поддерживает упорядоченность всех операций по абсолютному времени и всегда возвращает последнее по времени значение. Мы будем называть эту модель *семантикой UNIX (UNIX semantics)*. Эту модель легко понять и просто реализовать.

В распределенных системах семантика UNIX может успешно применяться до тех пор, пока мы обсуждаем систему с единственным файловым сервером и клиентами, не поддерживающими кэширование файлов. Все запросы на чтение и запись передаются файловому серверу, который выполняет их строго последовательно. Такой подход позволяет использовать семантику UNIX (если не принимать во внимание некоторые проблемы, создаваемые задержками в сети, из-за которых операция чтения, произошедшая на микросекунду позже операции записи, может достичь сервера первой и вернуть устаревшее значение).

На практике, однако, производительность распределенных систем, в которых все запросы к файлам передаются на единственный сервер, часто слишком низка. Эта проблема обычно решается путем создания клиентами в их внутреннем локальном кэше копий часто используемых файлов. Хотя мы будем дополни-

тельно обсуждать вопросы, связанные с кэшированием файлов, в данный момент нам важно отметить, что если клиент локально модифицирует кэшируемый файл, а через некоторое время другой клиент считывает этот файл с сервера, второй клиент получает устаревший вариант файла (рис. 10.8, б).

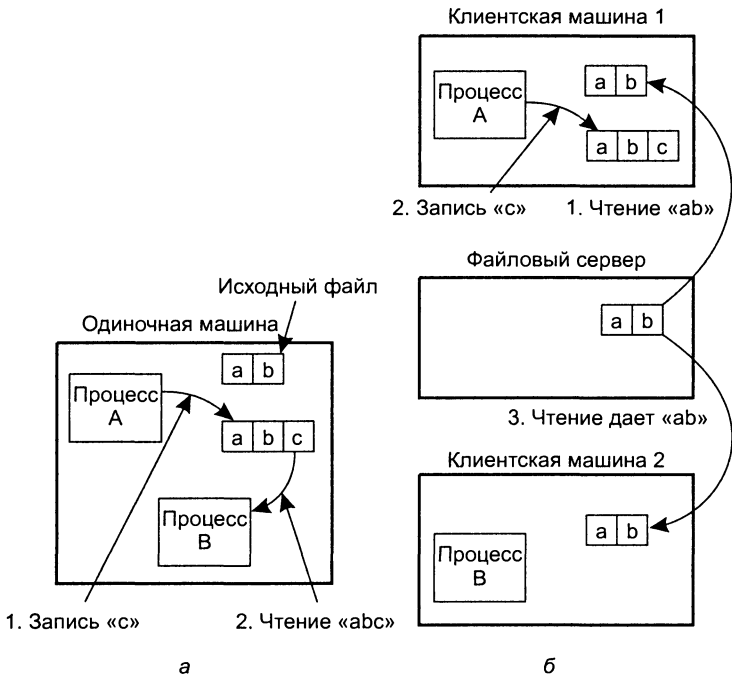


Рис. 10.8. В случае единственного процессора, когда чтение следует за записью, прочитанное значение соответствует ранее записанному (а). В распределенной системе с кэшированием клиент может получить устаревшее значение (б)

Один из способов справиться с этой проблемой — немедленная передача всех изменений в кэшированных файлах на сервер. Несмотря на концептуальную простоту, этот подход неэффективен. В качестве альтернативного решения можно рассмотреть ослабление семантики совместного использования файлов. Вместо того чтобы непременно требовать от операции чтения визуализации эффекта, произведенного всеми предыдущими операциями записи, мы можем ввести новое правило, согласно которому все изменения в открытом файле изначально должны быть видны только процессу (или, возможно, машине), который вносит их в файл; только после закрытия файла доступ к этим изменениям должны получить другие процессы (или машины). Введение этого правила не изменит картину, представленную на рис. 10.8, б, но делает правильным реальный режим работы (В просто получает исходное значение файла). Когда процесс А закроет файл, он отошлет его копию на сервер, и последующее чтение, разумеется, принесет новое значение. Это правило широко применяется и известно под названием *семантики сеансов (session semantics)*. Как и многие другие распределенные

файловые системы, NFS реализует семантику сеансов. Это означает, что хотя система NFS теоретически следует модели удаленного доступа, представленной на рис. 10.1, *а*, большинство ее реализаций использует локальное кэширование, поддерживая тем самым модель загрузки-выгрузки, представленную на рис. 10.1, *б*.

В случае семантики сеансов встает вопрос о том, что произойдет, если два или более клиентов станут одновременно кэшировать и изменять один и тот же файл. Одно из решений состоит в том, чтобы содержимое файла передавалось обратно на сервер после закрытия файла. Итоговый результат будет зависеть от того, какой запрос на закрытие файла будет обработан сервером последним. Менее приятной, но более простой в осуществлении альтернативой является выбор финального результата из нескольких кандидатов с отбрасыванием всех изменений, внесенных в файл другими процессами.

Абсолютно другой подход к семантике совместного использования файлов в распределенных системах состоит в том, чтобы сделать все файлы неизменяемыми. Это означает, что способа открыть файл на запись не существует. В результате единственными файловыми операциями остаются создание и чтение.

Все, что мы можем сделать в этом случае, — открыть абсолютно новый файл и поместить его в систему каталогов под именем ранее существовавшего, который с этого момента становится недоступным (по крайней мере, под прежним именем). Таким образом, хотя изменить файл *x* нельзя, остается возможность автоматически заменить *x* новым файлом. Другими словами, файлы изменять нельзя, но каталоги изменять можно! Как только мы приходим к решению запретить изменять файлы вообще, проблема двух процессов, один из которых записывает в файл, а другой читает из него, просто исчезает.

Остается проблема двух процессов, пытающихся одновременно заменить один и тот же файл. Как и в семантике сеансов, наилучшим решением здесь кажется выбор одного из этих новых файлов для замены старого — либо последнего из пришедших, либо по какому-либо другому критерию.

Довольно типичная проблема — что делать, если файл заменяется в тот момент, когда другой процесс занимается его чтением. Одно из решений состоит в том, чтобы как-нибудь обеспечить использование читающим процессом старого файла, даже если его больше нет в каталогах, аналогично тому, как UNIX позволяет процессам, открывшим файл, продолжать использовать файл даже после того, как он удален из всех каталогов. Другое решение состоит в том, чтобы обнаруживать факт изменения файла и делать все последующие попытки чтения из него невозможными.

Четвертый способ работы с разделяемыми файлами в распределенных системах — использование транзакций, о которых мы подробно говорили в главе 5. Обобщим эту информацию. Для доступа к файлу или группе файлов процесс должен сначала выполнить один из вариантов примитива `BEGIN_TRANSACTION`, указывая, что дальнейшие действия выполняются единым блоком. После этого следует несколько системных вызовов чтения и записи в один или несколько файлов. Когда работа заканчивается, выполняется примитив `END_TRANSACTION`. Система гарантирует, что все вызовы в рамках транзакции будут выполнены в указанном порядке без каких-либо пересечений с другими параллельными транзакциями.

Если две или более транзакции начинаются одновременно, система гарантирует, что итоговый результат будет точно таким же, как если бы они все выполнялись в некоторой (неопределенной) последовательности.

В табл. 10.4 обобщены данные по рассмотренным вариантам работы с совместно используемыми файлами в распределенных системах.

Таблица 10.4. Четыре варианта работы с совместно используемыми файлами в распределенной системе

Метод	Комментарий
Семантика UNIX	Каждая операция с файлом немедленно становится видна всем процессам
Семантика сеансов	Изменения невидимы для других процессов до закрытия файла
Неизменяемые файлы	Изменения невозможны, упрощены разделение и репликация
Транзакции	Все изменения происходят атомарно

Блокировка файлов в NFS

Для распределенной файловой системы, серверы которой могут не сохранять информацию о состоянии, блокировка файлов представляет собой проблему. В NFS блокировка файлов традиционно осуществлялась при помощи специального протокола, реализованного в менеджере блокировок, который, разумеется, состояние сохранял. Однако по различным причинам блокировка файлов с использованием протокола блокировок NFS никогда не была слишком популярной из-за сложности протокола и, как следствие, малопроизводительных или вообще неверных реализаций [412]. В версии 4 блокировка файлов интегрирована в протокол доступа к файлам NFS. Ожидалось, что в подобном подходе будет удобнее применять системную блокировку, а не те частные решения, которые использовались вместо системной блокировки в NFS версии 3.

Блокировка файлов в распределенных файловых системах осложняется тем, что клиенты и серверы в период блокировки могут отказать. Важной становится задача правильного восстановления после сбоев, что гарантирует непротиворечивость общих файлов. Мы вернемся к этому вопросу чуть позже, когда будем обсуждать отказоустойчивость в NFS.

Концепция блокировки файлов в NFS версии 4 проста. В сущности, имеется всего четыре операции, относящиеся к блокировке файлов, они перечислены в табл. 10.5. NFS отделяет блокировку на чтение от блокировки на запись. Несколько клиентов могут одновременно работать с одной и той же частью файла, если они занимаются только чтением данных. Для получения исключительного доступа к файлу для его модификации необходима блокировка на запись.

Таблица 10.5. Операции, относящиеся к блокировке файлов в NFS версии 4

Операция	Описание
lock	Блокировка набора байтов
lockt	Проверка, не назначена ли конфликтующая блокировка

Операция	Описание
locku	Снятие блокировки с набора байтов
renew	Продление аренды указанной блокировки

Операция `lock` используется для запроса на установку блокировки на чтение или запись последовательного набора байтов в файле. Это неблокирующая операция; если блокировку невозможно назначить из-за того, что она будет конфликтовать с другой блокировкой, клиент получает сообщение об ошибке и позже может снова обратиться к серверу с этим запросом. Клиент может также потребовать постановки запрошенной блокировки в список FIFO, который поддерживает сервер. Как только мешающая блокировка будет снята, сервер сможет установить следующую блокировку, взяв ее с вершины списка и на определенное время предоставив соответствующему клиенту право на работу с сервером. Такой подход избавляет сервер от необходимости уведомлять клиентов, запросы которых на установку блокировки были отклонены, поскольку блокировки устанавливаются в порядке FIFO.

Операция `lockt` позволяет проверить факт существования конфликтующей блокировки. Так, например, клиент может проверять, существуют ли блокировки на чтение, установленные на определенную последовательность байтов в файле, прежде чем потребует установки на эту последовательность байтов блокировки на запись. В случае конфликта отправивший запрос клиент информируется о том, кто и на какую последовательность байтов назначил вызвавшую конфликт блокировку. Это позволяет более эффективно реализовать проверку, чем при применении операции `lock`, поскольку в случае `lockt` нет необходимости в открытии файла.

Снятие блокировки с файла производится при помощи операции `locku`.

Блокировки устанавливаются на конкретное определяемое сервером время. Другими словами, они связаны с соответствующей арендой. Если клиент не продлит аренду своей блокировки, сервер автоматически снимет ее. Как мы увидим, такой же подход соблюдается и в отношении других ресурсов, предоставляемых сервером, и способствует его восстановлению в случае отказов. Используя операцию `renew`, клиент просит сервер продлить аренду его блокировки (а также других ресурсов).

Кроме этих операций в распоряжении пользователя имеется еще один особый способ блокировки файлов, известный под названием *совместного резервирования* (*share reservation*). Совместное резервирование никак не связано с обычными блокировками и может использоваться для реализации NFS на платформе Windows. Когда клиент открывает файл, он определяет тип доступа, которого требует для себя (`READ`, `WRITE` или `BOTH`), и тип доступа, в котором сервер должен отказать всем прочим клиентам (`NONE`, `READ`, `WRITE` или `BOTH`). Если сервер не в состоянии удовлетворить требования клиента, операция открытия файла завершается с ошибкой. С помощью табл. 10.6 показано, что происходит при попытке открытия новым клиентом файла, уже успешно открытого другим клиентом. Для открытого файла мы выделяем два состояния. Состояние доступа определяет тип доступа, заявленный текущим клиентом. Состояние запрещения определяет, ка-

кие виды доступа к файлу запрещены подключающимся к нему дополнительным клиентам.

Таблица 10.6. Типы совместного доступа для заданных типов запрещения

Тип запраши- ваемого доступа	Текущий тип запрещения				
		NONE	READ	WRITE	BOTH
	READ	Успешно	Ошибка	Успешно	Ошибка
	WRITE	Успешно	Успешно	Ошибка	Ошибка
	BOTH	Успешно	Ошибка	Ошибка	Ошибка

По табл. 10.6 мы можем судить о том, что происходит, когда клиент пытается открыть файл, запрашивая определенный тип доступа с учетом текущего состояния запрета для этого файла. Соответственно, в табл. 10.7 представлены результаты открытия файла, который уже открыт другим клиентом, с одновременной попыткой запретить определенные типы доступа.

Таблица 10.7. Типы запрещения заданных типов совместного доступа

Текущий тип доступа	Тип запрашиваемого запрещения				
		NONE	READ	WRITE	BOTH
	READ	Успешно	Ошибка	Успешно	Ошибка
	WRITE	Успешно	Успешно	Ошибка	Ошибка
	BOTH	Успешно	Ошибка	Ошибка	Ошибка

10.1.6. Кэширование и репликация

Как и многие другие распределенные файловые системы, NFS для повышения производительности интенсивно использует кэширование на клиенте. Кроме того, она предлагает некоторые средства репликации файлов.

Кэширование на клиенте

Кэширование в NFS версии 3 в основном не было определено в протоколах. Подобный подход приводил к созданию различных методик кэширования, большинство из которых никогда не гарантировали непротиворечивости. В лучшем случае кэшированные данные пребывали в устаревшем состоянии в течение нескольких секунд, необходимых для сравнения их с данными, хранящимися на сервере. Однако существовали и реализации, для которых вполне естественно было оставлять кэшированные данные устаревшими в течение 30 секунд без уведомления клиента. Такое состояние дел было абсолютно нежелательно.

В NFS версии 4 некоторые из этих проблем непротиворечивости были решены, но зависимость непротиворечивости кэша от способа его реализации, в сущности, осталась. Общую модель кэширования, принятую в NFS, иллюстрирует рис. 10.9. Каждый клиент может иметь кэш-память для хранения считанных

с сервера данных. Кроме того, может существовать также и дисковый кэш, являющийся дополнением к кэш-памяти и использующий те же параметры непротиворечивости.

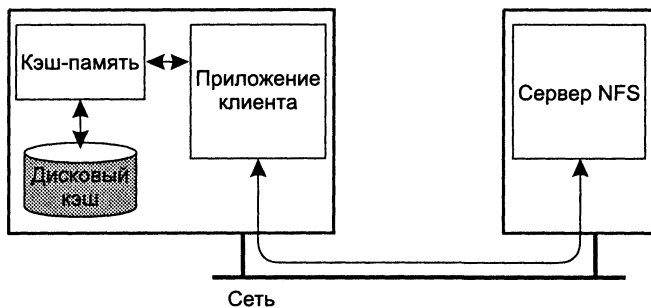


Рис. 10.9. Кэширование на стороне клиента в NFS

Обычно клиенты кэшируют данные из файлов, атрибуты, дескрипторы файлов и каталогов. Для поддержания непротиворечивости кэшированных данных и атрибутов были разработаны различные стратегии. Рассмотрим сначала кэширование данных, которые содержатся в файлах.

NFS версии 4 поддерживает два способа кэширования данных. Наиболее простой из них состоит в том, что клиент открывает файл и кэширует содержащиеся в нем данные, получаемые с сервера в результате различных операций чтения. Кроме того, над кэшированными данными можно осуществлять и операцию записи. После того как клиент закрывает файл, NFS требует, чтобы в том случае, если в кэшированные данные были внесены изменения, они были переданы обратно на сервер. Подобный подход представляет собой не что иное, как реализацию семантики сеансов, которую мы обсуждали выше.

Как только хотя бы часть файла оказывается кэшированной, клиент может хранить соответствующие данные в кэше даже после закрытия файла. Кроме того, несколько клиентов на одной машине могут использовать один и тот же кэш совместно. NFS требует, чтобы когда бы клиент ни открыл ранее закрытый файл с даже частично кэшированными данными, он должен немедленно перепроверить эти данные. Подобная проверка включает в себя проверку времени внесения в файл последнего изменения и обновление кэша, если он содержит устаревшие данные.

Новым для спецификации NFS является то, что сервер после открытия файла может делегировать часть своих прав клиенту. В том случае, если машине клиента разрешается локально обрабатывать операции открытия и закрытия файлов других клиентов той же машины, имеет место *делегирование открытия (open delegation)*. Обычно сервер может определить, было ли успешным открытие файла, например, когда необходимо учитывать совместное резервирование. В случае делегирования открытия машина клиента время от времени может принимать решение сама, что позволяет исключить контакт с сервером.

Так, например, в том случае, если сервер делегировал открытие файлов клиенту, который запросил право на запись в файл, блокировка файла, которую запросил другой клиент той же машины, может быть обработана локально. Сервер продолжает обрабатывать запросы на блокировку от клиентов с других машин, просто запрещая этим клиентам доступ к файлу. Отметим, что подобная схема не работает в случае делегирования файла клиенту, имеющему право только на чтение. В этом случае, когда другой клиент захочет получить право на запись, ему придется связаться с сервером; локально обработать такой запрос невозможно.

Одно из важных следствий делегирования файла клиенту состоит в том, что сервер должен быть в состоянии отменить это делегирование, например, когда права на доступ к файлу захочет получить другой клиент с другой машины. Отмена делегирования требует, чтобы сервер мог обратиться к клиенту, как показано на рис. 10.10.

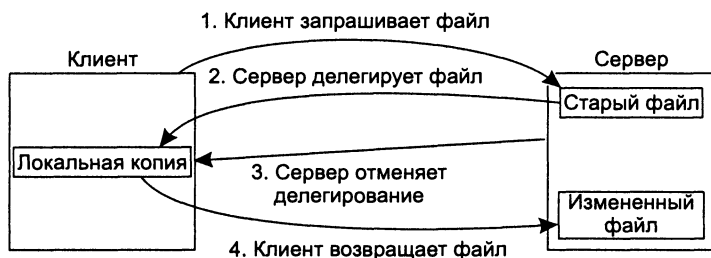


Рис. 10.10. Использование механизма обратного вызова в NFS версии 4 для отмены делегирования файла

Обратный вызов реализуется в NFS при помощи базовых механизмов RPC. Отметим, однако, что для обратного вызова необходимо, чтобы сервер отслеживал тех клиентов, которым он делегировал файл. Вот и еще одна причина, по которой больше не может быть сервера NFS без фиксации состояния. Как мы увидим ниже, совместное использование делегирования и серверов с фиксацией состояния в случае отказа сервера или клиента может стать источником различных проблем.

Клиенты могут также кэшировать значения атрибутов, но обычно они оставляют их теми, которые были, чтобы значения в кэше оставались непротиворечивыми. Однако значения атрибутов одного и того же файла, кэшированные двумя разными клиентами, могут быть разными, если, конечно, клиенты не поддерживают эти атрибуты взаимно непротиворечивыми. Модификации значений атрибутов немедленно передаются на сервер в соответствии с правилами согласованности кэша сквозной записи.

Таким же образом выполняется кэширование дескрипторов файлов (или, скорее, отображений имен файлов в дескрипторы) и каталогов. Чтобы снизить отрицательный эффект от возможной в этом случае противоречивости, NFS использует аренду кэшируемых атрибутов, дескрипторов файлов и каталогов. После истечения определенного периода времени хранимая в кэше информация

автоматически объявляется неверной. В результате для того, чтобы вновь использовать эти данные, их необходимо обновить с сервера.

Серверы реплик

NFS версии 4 предоставляет минимальную поддержку репликации файлов. Реплицирована может быть только файловая система целиком (то есть все логическое устройство, включая файлы, атрибуты, каталоги и блоки данных). Поддержка предоставляется в виде атрибута `FS_LOCATIONS`, который для всех файлов является рекомендуемым. Этот атрибут поддерживает список мест, в которых может находиться файловая система, содержащая данный файл. Каждое из этих мест указывается в форме DNS-имени или IP-адреса. Отметим, что для действительной реализации реплицируемых серверов необходима специальная реализация NFS. NFS версии 4 не определяет, каким образом должна происходить репликация.

10.1.7. Отказоустойчивость

Вплоть до появления последней версии NFS отказоустойчивость в этой системе вряд ли могла бы являться предметом обсуждения. Причиной этому было то, что протокол NFS не предусматривал хранения серверами информации о состоянии. В результате восстановление сервера после сбоя было идеально простым — ведь никакого состояния не терялось. Разумеется, состояние поддерживалось отдельными менеджерами блокировок, и в этих случаях было необходимо предпринимать определенные меры.

После отказа от архитектуры без фиксации состояния в NFS версии 4 возникла необходимость заниматься отказоустойчивостью и восстановлением все-речь. В частности, хранение информации о состоянии проявляется в двух случаях — блокировке файлов и делегировании. Кроме того, необходимо принимать особые меры в связи с ненадежностью механизма RPC, который лежит в основе протокола NFS. Давайте рассмотрим эти вопросы по отдельности и начнем с отказов RPC.

Отказы RPC

Проблема механизма RPC, используемого в NFS, состоит в том, что он не гарантирует надежности. На практике клиентские и серверные заглушки RPC могут создаваться как на базе ориентированного на соединение надежного транспортного протокола, такого как TCP, так и на базе ненадежного транспортного протокола без установления соединения, например UDP.

Основная проблема, которую система NFS унаследовала от базовой семантики RPC, — это отсутствие средств определения дублированных запросов. Как следствие, когда RPC *воспроизводит* потерянный запрос, а клиент заново посылает исходный запрос, сервер в конечном итоге обслужит запрос более одного раза. При работе с не идемпотентными операциями подобная ситуация происходить не должна.

Эти проблемы можно решить при помощи *кэша дублированных запросов* (*duplicate-request cache*), реализуемого сервером [221]. Каждый запрос RPC, по-

сылаемый клиентом, содержит в заголовке уникальный *идентификатор транзакции* (*Transaction Identifier, XID*), который кэшируется сервером, когда запрос до него доходит. До тех пор пока сервер не пошлет ответ, он будет показывать, что запрос RPC находится в процессе обработки. Когда обработка запроса завершается, ассоциированный с ним ответ также кэшируется, после чего возвращается клиенту.

Существует три ситуации, которые нам следует рассмотреть. В первом случае, продемонстрированном на рис. 10.11, а, клиент посылает запрос и запускает таймер. Если время ожидания истекает до прихода ответа, клиент воспроизводит исходный запрос с тем же самым идентификатором XID, что и оригинальный. Если сервер еще не выполнил исходного запроса, повторный запрос он просто игнорирует.

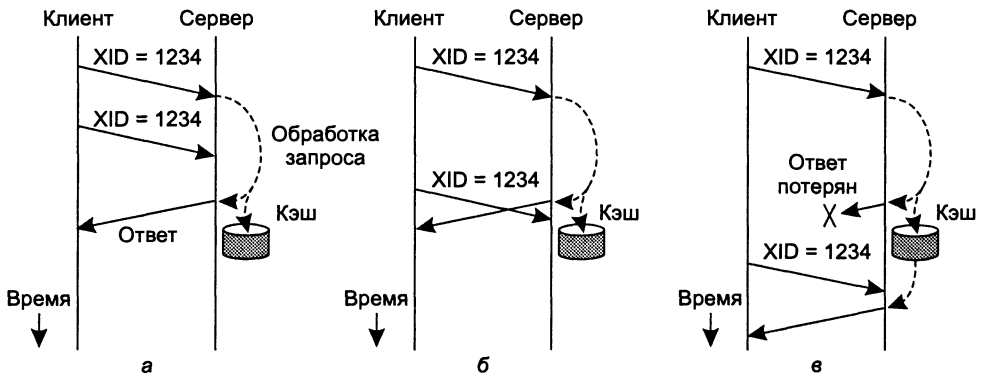


Рис. 10.11. Три ситуации повторной передачи. Запрос еще выполняется (а). Ответ только что отправлен (б). Ответ давно отправлен, но затерялся (в)

Во втором случае сервер может принять повторно посланный запрос сразу после того, как он отошлет клиенту ответ. Если время доставки повторного запроса близко ко времени отправки сервером ответа, сервер делает вывод, что повторный запрос и ответ пересеклись, а потому игнорирует повторный запрос. Эту ситуацию иллюстрирует рис. 10.11, б.

В третьем случае ответ действительно теряется, и результатом повторного запроса должна стать отправка сохраненных в кэше результатов операции клиенту, что и показано на рис. 10.11, в. Отметим, что результат файловой операции должен быть кэширован, поскольку отсутствуют всякие гарантии того, что повторная посылка на этот раз увенчается успехом.

Блокировка файлов при отказах

Блокировка файлов в NFS версии 3 обрабатывалась выделенным сервером. В результате можно было использовать протокол NFS, не предполагающий сохранение информации о состоянии, а все проблемы, связанные с отказоустойчивостью, решались сервером блокировок. Однако с появлением в NFS версии 4 средств блокировки файлов включение в протокол NFS базового механизма обработки

отказов клиента и сервера стало неминуемым. Решаемая этим механизмом проблема относительно проста. Чтобы заблокировать файл, клиент посылает на сервер соответствующий запрос. Далее рассмотрим проблемы, возникающие при отказах клиента или сервера в предположении, что блокировка уже предоставлена.

Чтобы разобраться с проблемами отказа клиента, сервер на каждую предоставленную блокировку устанавливает аренду. Когда срок аренды истекает, сервер удаляет блокировку, освобождая, таким образом, связанные с ней ресурсы (то есть файлы). Чтобы предотвратить удаление сервером блокировки, клиент должен продлить свою аренду до того, как она истечет. Именно этой цели служит в NFS описанная ранее операция `renew`.

Существуют ситуации, когда блокировки удаляются, даже если клиент находится во вполне работоспособном состоянии. Собственно, ложные удаления могут произойти, если клиент вовремя не сообщит серверу о необходимости продления аренды, например, из-за того, что сеть оказалась (временно) разделенной на части. Для разрешения такой ситуации не применяется никаких специальных мер.

Если сервер сначала отказывает, а затем он восстанавливается, информация по блокировкам, предоставленным клиентам, будет предположительно потеряна. Подход, принятый в NFS версии 4, состоит в том, чтобы предоставить клиентам *период амнистии* (*grace period*), в течение которого клиент может повторно предъявить права на ранее предоставленные ему блокировки. Таким образом, сервер восстанавливает свое предыдущее состояние в части, касающейся блокировок. Отметим, что подобное восстановление блокировок никак не связано с восстановлением прочих данных. В период амнистии обычно принимаются только запросы на восстановление блокировок, обычные запросы на предоставление блокировок до окончания этого периода не рассматриваются.

Здесь следует отметить, что аренда вносит многочисленные проблемы, которые решаются в NFS лишь частично. Так, например, аренда требует, чтобы клиент и сервер синхронизировали свои часы. Если сервер считает, что аренда истекает в момент времени T , часы клиента должны показывать то же время, что и часы сервера. С другой стороны, если сервер считает, что аренда истекает через некий период времени D , то ему требуется знать время, которое занимает отправка аренды клиенту. В глобальных системах вопросы точного времени решить не так-то легко.

Другая проблема состоит в надежности сообщения о продлении аренды, отправленного серверу. Если доставка сообщений не работает или производится с опозданием, аренда может быть ненамеренно просрочена. В этом случае клиент, перед тем как продолжить использование файла, должен явно потребовать у сервера новой аренды.

Делегирование открытия при отказах

В случае отказа клиента или сервера делегирование открытия создает дополнительные проблемы. Рассмотрим для начала клиента, которому делегировано право открытия файла. Если клиент отказывает, предполагается, что он, поработав со своей локальной копией файла, не успел передать изменения на сервер. Други-

ми словами, клиент следует правилам согласованности кэша обратной записи. Пока клиент не восстановит локально сохраненный файл в устойчивое хранилище, полное восстановление файла невозможно. В любом случае за восстановление файла отчасти отвечает клиент.

Когда сервер сначала отказывает и затем восстанавливается, он следует процедуре, похожей на установку блокировки. Клиент, которому было делегировано открытие файла, требует восстановления прав на это делегирование, когда сервер вновь становится работоспособным. Однако в противоположность блокировкам сервер заставляет клиента передать все изменения назад на сервер, отменяя таким образом делегирование.

Подобный подход имеет два эффекта. Во-первых, на сервере оказываются последние результаты изменений каждого из файлов, права на которые он делегировал клиенту. Во-вторых, поскольку сервер снова обладает полным контролем над файлом, он может делегировать его другому клиенту.

10.1.8. Защита

Как мы ранее говорили, основная идея, лежащая в основе NFS, состоит в том, что удаленная файловая система может быть представлена пользователям как часть их собственной локальной файловой системы. В свете этого для нас не будет неожиданностью тот факт, что система защиты в NFS сконцентрирована в основном на вопросах взаимодействия между клиентом и сервером. Защищенное взаимодействие, как мы выяснили в главе 8, предполагает наличие защищенного канала между взаимодействующими сторонами.

Вдобавок к защищенным вызовам RPC необходимо контролировать доступ к файлам, который в NFS обрабатывается при помощи атрибутов контроля доступа файлов. За подтверждение прав доступа клиентов, как будет показано далее, отвечает файловый сервер. В сочетании с защищенными вызовами RPC архитектура защиты NFS выглядит так, как показано на рис. 10.12.

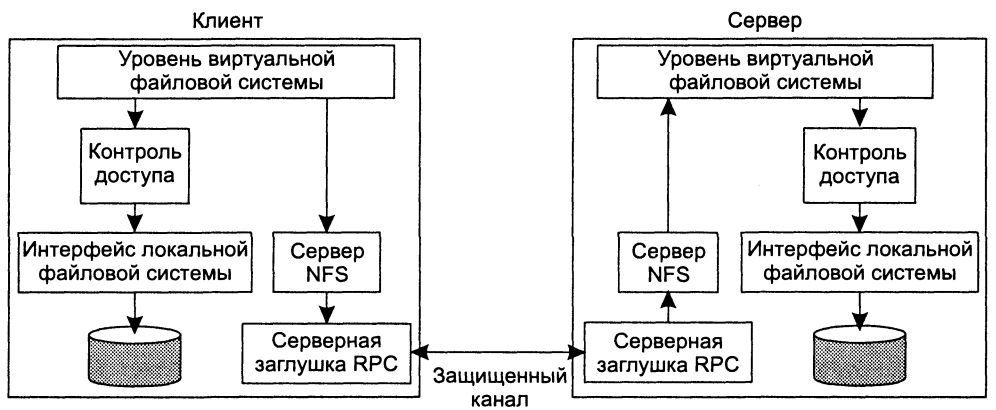


Рис. 10.12. Архитектура защиты NFS

Защищенные вызовы RPC

Поскольку NFS работает поверх системы RPC, создание защищенного канала в NFS выполняется с помощью защищенных вызовов RPC. До NFS версии 4 защищенный вызов RPC означал лишь проведение аутентификации. Существовало три способа аутентификации. Рассмотрим каждый из них по отдельности.

Наиболее широко используемый метод, который на самом деле сложно вообще назвать аутентификацией, известен под названием системной аутентификации. В этом методе, основанном на технологиях системы UNIX, клиент просто передает на сервер в виде простого текста свои текущие идентификаторы пользователя и группы вместе со списком групп, в которых он состоит. Другими словами, сервер не имеет никакой возможности проверить, действительно ли указанные пользователь и группа имеют какое-то отношение к отправителю. В сущности, сервер предполагает, что клиент корректно выполнил процедуру входа в систему и может по праву распоряжаться клиентской машиной.

Второй способ аутентификации, характерный для предыдущих версий NFS, предполагает метод обмена ключами Диффи—Хеллмана для создания сеансового ключа и организации так называемой защищенной системы NFS. Как работает обмен ключами Диффи—Хеллмана, мы рассматривали в главе 8. Такой подход значительно лучше, чем системная аутентификация, но сложнее, а потому реже применяется. Обмен ключами Диффи—Хеллмана можно считать крипто-системой с открытым ключом. Изначально способа защищенного распространения открытого ключа сервера не существовало, но впоследствии этот недостаток был исправлен путем организации защищенной службы именования. Нарекания всегда вызывал относительно короткий открытый ключ, который в NFS имеет длину всего 192 бита. В [248] было продемонстрировано, что взломать систему Диффи—Хеллмана с таким коротким ключом довольно легко.

Третий протокол аутентификации — это система Kerberos (версии 4), которую мы также рассматривали в главе 8.

С появлением NFS версии 4 защита системы была усилена благодаря *RPCSEC_GSS*. *RPCSEC_GSS* — это обобщенная схема защиты, которая при организации защищенных каналов может поддерживать множество механизмов защиты [134]. В частности, она предоставляет не только средства подключения различных систем аутентификации, но и поддерживает целостность и конфиденциальность сообщений, две характеристики, которые в прежних версиях NFS не поддерживались.

RPCSEC_GSS базируется на стандартном для служб защиты интерфейсе *GSS-API*, о котором мы упоминали в главе 8 и который подробно описан в [267]. Схема *RPCSEC_GSS* работает поверх этого интерфейса, как показано на рис. 10.13.

В NFS версии 4 схема *RPCSEC_GSS* должна быть сконфигурирована для поддержки Kerberos версии 5. Кроме того, система в состоянии поддерживать метод под названием *LIPKEY*, описанный в [133]. *LIPKEY* — это система с открытым ключом, позволяющая клиентам проходить аутентификацию по паролю, в то время как сервер аутентифицируется с помощью открытого ключа.

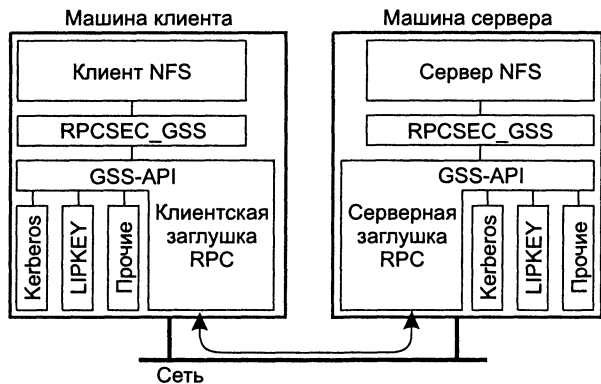


Рис. 10.13. Защищенный вызов RPC в NFS версии 4

Один из важных аспектов защищенных вызовов RPC в NFS состоит в том, что разработчикам не нужно создавать собственных механизмов защиты, достаточно организовать стандартный способ обеспечения защиты. Механизмы защиты, такие как Kerberos, можно встраивать непосредственно в реализацию NFS, при этом другие части системы никак не затрагиваются. Кроме того, если существующие механизмы защиты оказываются ненадежными (как это произошло с алгоритмом Диффи–Хеллмана с короткими ключами), их легко можно заменить.

Следует отметить, что поскольку схема RPCSEC_GSS реализована в виде части уровня RPC, поверх которого работают протоколы NFS, она может быть использована и в предыдущих версиях NFS. Однако такая адаптация уровня RPC стала возможной только с появлением NFS версии 4.

Контроль доступа

Авторизация в NFS аналогична авторизации защищенных вызовов RPC: она предоставляет механизм, но не определяет каких-либо правил. Управление доступом поддерживается при помощи атрибута ACL файлов. Этот атрибут представляет собой список записей для контроля доступа, где каждая запись определяет права доступа определенного пользователя или группы пользователей. Операции над файлами, которые подпадают под контроль доступа в NFS, вполне естественны. Они перечислены в табл. 10.8.

Таблица 10.8. Классификация операций контроля доступа в NFS

Операция	Описание
read_data	Право на чтение данных файла
write_data	Право на изменение данных файла
append_data	Право на добавление данных в файл
execute	Право на выполнение файла
list_directory	Право на получение содержимого каталога
add_file	Право на добавление в каталог нового файла

Операция	Описание
add_subdirectory	Право на создание вложенного каталога внутри каталога
delete	Право на удаление файла
delete_child	Право на удаление файла или вложенного каталога из каталога
read_acl	Право на чтение списка ACL
write_acl	Право на запись в список ACL
read_attributes	Право на чтение других основных атрибутов файла
write_attributes	Право на изменение других основных атрибутов файла
read_named_attr	Право на чтение именованных атрибутов файла
write_named_attr	Право на запись именованных атрибутов файла
write_owner	Право на смену владельца
synchronize	Право на локальный доступ к файлу на сервере с синхронным чтением и записью

По сравнению с простыми механизмами контроля доступа, используемыми, например, в системах UNIX, в NFS значительно больше операций. Следует особо отметить, что операция `synchronize` действует даже в том случае, если процесс, располагающийся на сервере, имеет прямой (в обход протокола NFS) доступ к файлам для повышения производительности. Модель контроля доступа NFS имеет более богатую семантику, чем большинство моделей контроля доступа UNIX. Эта разница вызвана требованием совместной работы NFS и операционной системы Windows 2000. Основная идея состояла в том, что значительно проще нарастить модель управления доступом UNIX до аналогичной модели Windows 2000, чем провести обратную операцию.

Другой аспект, отличающий контроль доступа от механизмов контроля большинства файловых систем, включая UNIX, состоит в том, что доступ можно определять для разных пользователей и разных групп пользователей. Традиционно доступ к файлам задавался для одного пользователя (владельца файла), одной группы пользователей (например, членов коллектива разработчиков, работающего над проектом) и для всех остальных. NFS, как показано в табл. 10.9, поддерживает множество разнообразных пользователей и процессов.

Таблица 10.9. Разные типы пользователей и процессов при контроле доступа в NFS

Тип пользователя	Описание
Owner	Владелец файла
Group	Группа пользователей, ассоциированных с файлом
Everyone	Любой пользователь или процесс
Interactive	Любой процесс, имеющий доступ к файлу через интерактивный терминал
Network	Любой процесс, имеющий доступ к файлу через сеть
Dialup	Любой процесс, имеющий доступ к файлу через коммутируемое соединение с сервером

Таблица 10.9 (продолжение)

Тип пользователя	Описание
Batch	Любой процесс, имеющий доступ к файлу в составе пакетного задания
Anonymous	Всякий, кто получает доступ к файлу без аутентификации
Authenticated	Всякий аутентифицированный пользователь или процесс
Service	Всякий служебный системный процесс

10.2. Файловая система Coda

Coda — следующий пример распределенной файловой системы. Система Coda была разработана в университете Карнеги—Меллона (Carnegie Mellon University, CMU) в 1990-х годах и в настоящее время интегрирована в несколько популярных операционных систем на базе UNIX, например в Linux. Coda во многих отношениях отличается от NFS, особенно в отношении высокой доступности. Требование обеспечить высокую доступность заставила разработчиков применять усовершенствованные схемы кэширования, которые позволяют клиенту продолжать операцию даже в случае отключения от сервера. Обзор Coda приведен в [236, 397]. Детальное описание системы можно найти в [235].

10.2.1. Обзор

Coda разрабатывалась как масштабируемая защищенная распределенная файловая система высокой доступности. Основная задача состояла в обеспечении прозрачности именования и локализации, так чтобы система выглядела для пользователей как можно более похожей на локальную файловую систему. Помимо высокой доступности разработчики Coda постарались также обеспечить высокую степень прозрачности отказов системы.

Coda родилась из второй версии *файловой системы Andrew* (*Andrew File System, AFS*), которая также была разработана в CMU [205, 399] и позаимствовала у AFS множество архитектурных решений. Система AFS, собственно, и создавалась для поддержания инфраструктуры университета CMU. В эту инфраструктуру входит приблизительно 10 000 рабочих станций, которым нужно обеспечить доступ к системе. Чтобы удовлетворить эти требования, узлы AFS были разделены на две группы. Одна группа состоит из относительно небольшого числа выделенных файловых серверов *Vice* с централизованным администрированием. Другая группа — это значительно большее число рабочих станций *Virtue*, которые предоставляют пользователям и процессам доступ к файловой системе, как показано на рис. 10.14.

Coda имеет такую же организацию, как и AFS. Все рабочие станции *Virtue* выполняют прикладной процесс под названием *Venus*, роль которого аналогична роли клиента NFS. Процесс *Venus* отвечает за предоставление доступа к файлам, которые находятся на файловых серверах *Vice*. *Venus* в Coda отвечает также и за то, чтобы клиенты имели возможность продолжать операции в том случае, если доступ

к файловым серверам невозможен (временно). Эта дополнительная роль значительно отличает методы, используемые в Coda, от подхода, принятого в NFS.

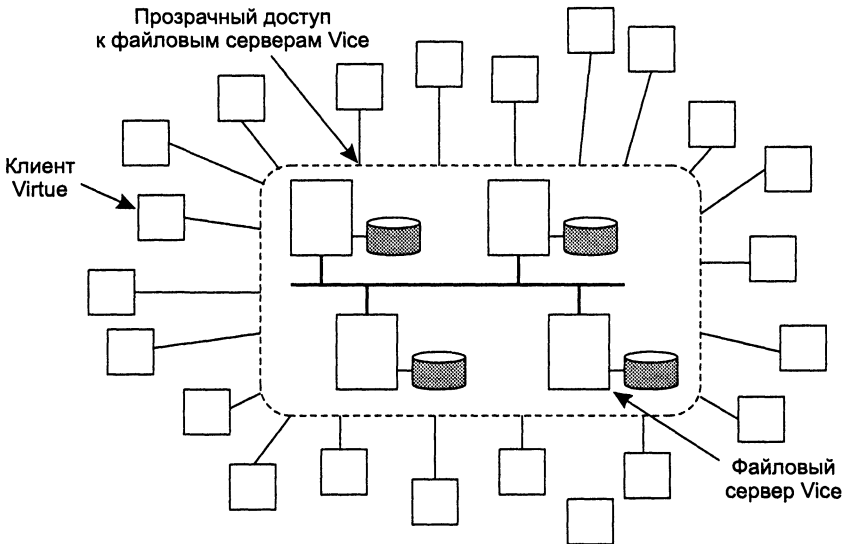


Рис. 10.14. Общая организация AFS

Внутренняя архитектура рабочей станции Virtue показано на рис. 10.15. Наиболее важный момент здесь состоит в том, что Venus является прикладным процессом. На системном уровне клиента запущен отдельный уровень виртуальной файловой системы (VFS), который перехватывает все вызовы из клиентских приложений и, как показано на рисунке, передает вызовы в локальную файловую систему или процессу Venus. Эта структура, включая VFS, аналогична соответствующей структуре NFS. Venus, в свою очередь, общается с файловыми серверами Vice, используя систему RPC пользовательского уровня. Система RPC построена поверх дейтаграмм UDP и поддерживает семантику «максимум однажды».

На стороне сервера имеется три разных процесса. Большая часть работы выполняется собственно файловыми серверами Vice, которые отвечают за поддержку локального набора файлов. Как и Venus, файловый сервер работает на пользовательском уровне. Кроме того, доверенные машины Vice имеют право запустить сервер аутентификации, который мы детально обсудим чуть позже. И наконец, для сохранения непротиворечивости метаданных на каждом из серверов Vice используются процессы обновления.

Coda предоставляет своим пользователям традиционную UNIX-подобную файловую систему. Она поддерживает большинство операций, входящих в ту часть спецификации VFS, которая приводилась в табл. 10.1 [237]. Повторять эту таблицу здесь мы не будем. В отличие от NFS Coda имеет глобальное разделяемое пространство имен, поддерживаемое серверами Vice. Клиенты получают доступ к этому пространству имен через специальный вложенный каталог в их локаль-

ном адресном пространстве, например /afs. Когда клиент ищет в этом вложенном каталоге имя, Venus гарантирует, что соответствующая часть общего пространства имен монтируется локально. Детали мы рассмотрим позже.

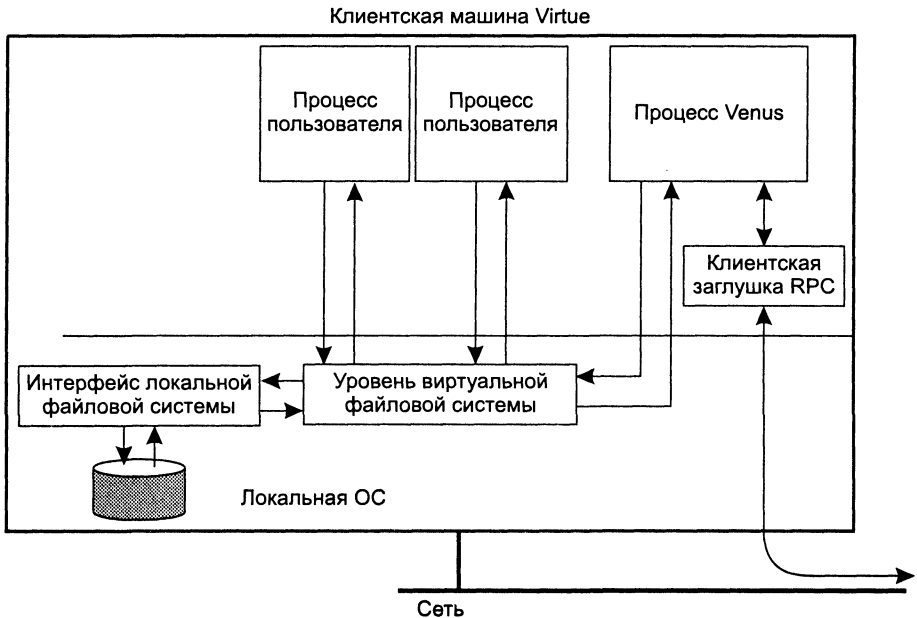


Рис. 10.15. Внутренняя структура рабочей станции Virtue

10.2.2. Связь

Взаимодействие между процессами в Coda реализуется при помощи вызовов RPC. Однако система RPC2, используемая в Coda, значительно сложнее традиционных систем RPC, таких как система ONC RPC, применяемая в NFS.

RPC2 реализует надежные вызовы RPC на базе протокола UDP (ненадежного). При каждом вызове удаленной процедуры код клиента RPC2 запускает новый поток выполнения, который передает запрос на сервер, после чего блокируется до получения ответа. Поскольку обработка запроса может занять произвольное время, сервер регулярно отправляет клиенту сообщения, чтобы уведомить его о том, что он продолжает обработку запроса. Если сервер перестает работать, то раньше или позже поток обработки запроса понимает, что сообщения перестали поступать, и возвращает уведомление об ошибке в приложение, инициировавшее обмен.

Интересным аспектом RPC2 является поддержка *побочных эффектов* (*side effects*). Побочным эффектом может считаться взаимодействие между клиентом и сервером с использованием специфического протокола приложения. Рассмотрим, например, как клиент открывает файл на сервере видеоданных. В этом случае необходимо, чтобы клиент и сервер установили постоянный поток данных в режиме изохронной передачи. Другими словами, задержка в передаче данных

от сервера клиенту гарантированно должна находиться в диапазоне между определенными минимальным и максимальным значениями, как это было описано в главе 2.

RPC2 позволяет клиенту и серверу установить особое соединение для передачи клиенту видеоданных в режиме реального времени. Установка такого соединения является побочным эффектом вызова RPC на сервере. Для этой цели исполняющая система RPC2 поддерживает интерфейс побочных процедур, которые должны реализовываться разработчиком приложений. Так, например, существуют процедуры для установки соединения и процедуры для передачи данных. Эти процедуры автоматически вызываются исполняющими системами RPC2 клиента и сервера соответственно, но их реализация абсолютно независима от RPC2. Этот принцип побочных эффектов иллюстрирует рис. 10.16.

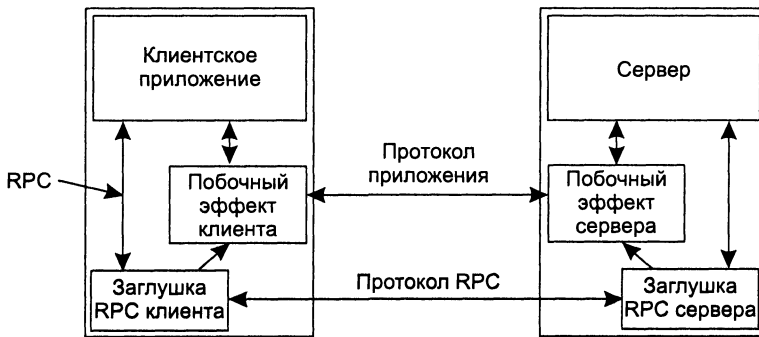


Рис. 10.16. Побочные эффекты системы RPC2 в Coda

Другое свойство системы RPC2, отличающее ее от прочих систем RPC, — поддержка групповой рассылки. Как мы увидим ниже, важная особенность Coda состоит в том, что сервер отслеживает, какие клиенты имеют локальную копию файла. При модификации файла сервер объявляет его локальные копии неправильными и уведомляет об этом при помощи RPC соответствующих клиентов. Понятно, что если сервер будет уведомлять только одного клиента за раз, оповещение всех клиентов может занять изрядное время, что иллюстрирует рис. 10.17, а.

Проблему порождает тот факт, что вызов RPC может потерпеть неудачу. Объявление файлов неправильными в строго последовательном порядке может существенно задержаться, поскольку сервер не в состоянии связаться с отказавшим (возможно) клиентом, но при этом прекратит попытки связаться с этим клиентом только через достаточно длительное время. Тем временем остальные клиенты будут продолжать пользоваться своими локальными копиями.

Лучшее решение продемонстрировано на рис. 10.17, б. Вместо того чтобы объявлять копию неправильной последовательно, сервер посылает сообщение о неправильности копии сразу всем клиентам. В результате все работающие клиенты уведомляются одновременно. Кроме того, сервер после требуемой паузы узнает, что определенные клиенты не в состоянии принять RPC, и может объявить этих клиентов отказавшими.

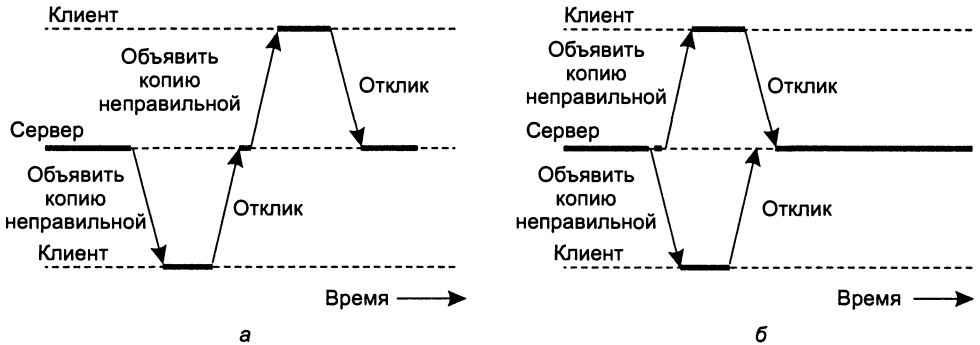


Рис. 10.17. Последовательная отправка сообщения о неправильности копии (а).
Параллельная отправка сообщения о неправильности копии (б)

Параллельные вызовы RPC реализуются при помощи системы MultiRPC [401], которая является частью пакета RPC2. Важная особенность MultiRPC состоит в том, что параллельная рассылка RPC полностью прозрачна для получателя. Другими словами, получатель вызова MultiRPC не может отличить этот вызов от обычного вызова RPC. Со стороны отправителя параллельное выполнение также вполне прозрачно. Так, например, семантика MultiRPC при наличии отказов очень похожа на аналогичную семантику обычного вызова RPC. Механизм побочных эффектов тоже работает схожим образом.

Вызов MultiRPC реализован, в сущности, как параллельное выполнение нескольких вызовов RPC. Это означает, что вызывающая сторона явно отправляет запросы RPC каждому из получателей. Однако вместо того, чтобы немедленно начать ожидать ответ, она откладывает ожидание до того момента, когда будут отосланы все запросы. Другими словами, отправитель рассылает множество односторонних вызовов RPC, после чего блокируется до получения ответов от всех работающих получателей. Альтернативный подход к параллельному выполнению вызовов RPC в MultiRPC обеспечивается путем организации групп рассылки и отправления вызова RPC всем членам группы средствами групповой рассылки по IP.

10.2.3. Процессы

Coda поддерживает четкое разделение процессов на клиентские и серверные. Клиенты представлены процессами Venus, серверы — процессами Vice. Оба типа процессов внутренне организованы в виде коллекции параллельных потоков выполнения. Потоки выполнения в Coda являются невытесняемыми и работают исключительно в пространстве пользователя. Для проведения непрерывной операции в условиях блокирующих запросов ввода-вывода используется отдельный поток выполнения. Этот поток берет на себя все операции ввода-вывода, которые реализуются им при помощи низкоуровневых асинхронных операций базовой операционной системы. Такой поток выполнения эффективно эмулирует синхронный ввод-вывод без блокирования процесса в целом.

10.2.4. Именованние

Как мы говорили, Coda поддерживает систему именования, аналогичную имеющейся в UNIX. Файлы группируются в модули, именуемые *томами* (*volumes*). Том соответствует области диска в UNIX (то есть в реальной файловой системе), но обычно имеет более мелкое дробление. Он соответствует частичному поддереву в общем пространстве имен, которое образуют серверы Vice. Обычно том ассоциируется с коллекцией файлов одного пользователя. Примерами томов являются коллекции разделяемых двоичных, или исходных, файлов и т. п. Как и области дисков, тома могут монтироваться.

Тома важны для нас по двум причинам. Во-первых, они образуют базовую единицу, из которых складывается все пространство имен. Эта сборка общего пространства имен производится путем монтирования томов в монтажных точках. Монтажная точка в Coda — это листовой узел тома, который связан с корневым узлом другого тома. Используя терминологию, введенную в главе 4, можно сказать, что только корневой узел может быть точкой монтирования (то есть клиент может смонтировать только корень тома). Вторая причина важности томов состоит в том, что они представляют собой модули для репликации серверов. К этому аспекту томов мы позже вернемся.

Рассматривая дробление томов, можно заметить, что поиск по имени может пересекать несколько монтажных точек. Другими словами, полное имя часто содержит несколько монтажных точек. Для поддержания высокой прозрачности именования файловый сервер Vice возвращает в процесс Venus информацию о монтировании, полученную в процессе разрешения имени. Эта информация позволяет Venus при необходимости автоматически монтировать тома в пространство имен клиента. Этот механизм схож с пересечением монтажных точек, которое поддерживается в NFS версии 4.

Важно отметить, что когда том общего пространства адресов монтируется в пространство имен клиента, Venus следует структуре общего пространства имен. Для того чтобы понять, что мы имеем в виду, предположим, что каждый клиент имеет доступ к общему пространству имен через вложенный каталог `/afs`. При монтировании тома каждый процесс Venus гарантирует, что граф именования, присоединенный корнем к `/afs`, является подграфом полного пространства имен, поддерживаемого совместными усилиями серверов Vice, как показано на рис. 10.18. Если это так, клиентам гарантируется, что совместно используемые файлы действительно имеют одинаковые имена, хотя разрешение имен и основано на локальной реализации пространства имен. Отметим, что этот подход в корне отличается от принятого в NFS.

Идентификаторы файлов

Если считать, что коллекция совместно используемых файлов может быть реплицирована и разбросана по нескольким серверам Vice, то резко повышается важность уникальной идентификации каждого файла так, чтобы имела возможность отследить его истинное местоположение с одновременной поддержкой репликации и прозрачности локализации.

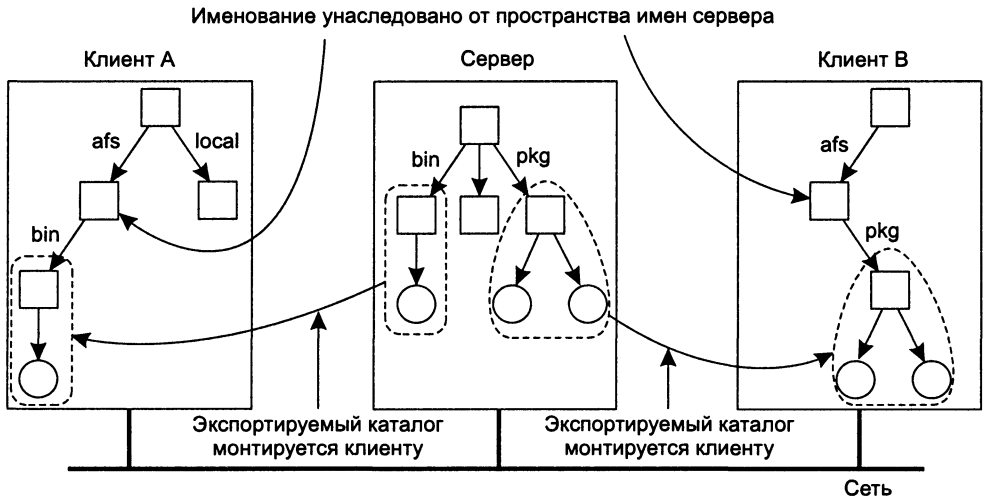


Рис. 10.18. Клиенты в Coda имеют доступ к единому разделяемому пространству имен

Каждый файл в Coda хранится точно в одном томе. Как мы говорили ранее, том может быть реплицирован на нескольких серверах. По этой причине в Coda делается различие между логическими и физическими томами. Логический том представляет собой возможно реплицированный физический том и имеет соответствующий *идентификатор реплицированного тома (Replicated Volume Identifier, RVID)*. RVID — это не зависящий от местоположения и репликации идентификатор тома. С одним и тем же RVID может ассоциироваться множество реplik. Каждый физический том имеет *идентификатор тома (Volume Identifier, VID)*, который идентифицирует конкретную реплику способом, не зависящим от ее местоположения.

Подход, принятый в Coda, состоит в том, чтобы присваивать каждому файлу 96-битный идентификатор файла. Идентификатор файла состоит из двух частей (рис. 10.19).

Первая часть представляет собой 32-битный идентификатор RVID логического тома, частью которого является файл. Для того чтобы найти файл, клиент сначала передает RVID из идентификатора файла в *базу данных репликации томов (volume replication database)*, которая возвращает список идентификаторов VID, соответствующих указанному идентификатору RVID. Получив VID, клиент ищет сервер, на котором находится рабочая реплика логического тома. Этот поиск производится путем послыки VID в *базу данных размещения томов (volume location database)*, которая возвращает текущее местоположение конкретного физического тома.

Вторая часть идентификатора файла представляет собой 64-битный дескриптор файла, который уникально идентифицирует файл внутри тома. В действительности он соответствует идентификатору узла индекса в VFS. Этот *виртуальный узел (vnode)*, как его называют, соответствует понятию *inode* в UNIX-системах.

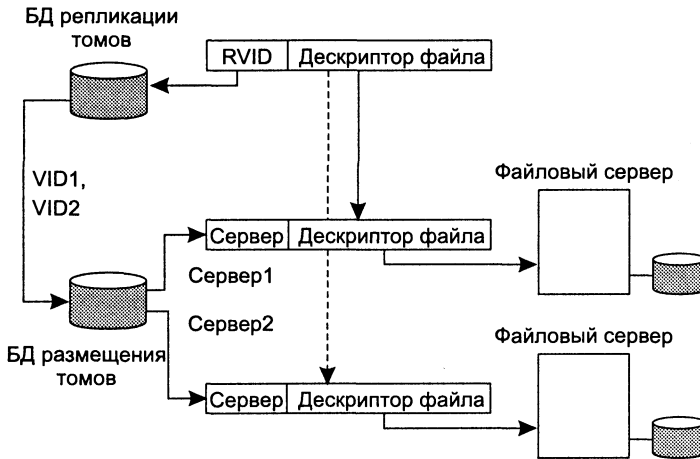


Рис. 10.19. Реализация и разрешение идентификатора файла в Coda

10.2.5. Синхронизация

Многие распределенные файловые системы, включая предка Coda, систему AFS, не поддерживают семантику разделения файлов UNIX, предоставляя вместо этого лишь более слабую семантику сеансов. Поставив своей задачей высокую доступность, Coda использует другой подход, пытаясь поддерживать семантику транзакций, хотя и в более слабом виде, чем обычно принято при транзакциях.

Проблема, которую при этом необходимо решить, заключается в том, что в больших распределенных файловых системах легко может оказаться, что некоторые или даже все серверы временно недоступны. Подобная недоступность может быть вызвана отказами сетей или оборудования, но может быть и результатом преднамеренного отсоединения мобильного агента от файловой службы. Если бы у отсоединившегося клиента все нужные ему файлы были в локальном кэше, он мог бы их использовать в автономном режиме, а при очередном соединении с системой согласовал бы их с хранящимися в системе копиями.

Разделение файлов в Coda

Чтобы поддерживать разделение файлов, Coda использует специальную схему распределения, которая имеет определенное сходство с совместным резервированием в NFS. Чтобы понять, как работает эта схема, следует учитывать следующее. Когда клиент успешно открывает файл f , на машину клиента пересылается точная копия этого файла f . Сервер записывает, что данный клиент имеет копию f . Таким образом, этот подход схож с делегированием открытия в NFS.

Теперь предположим, что клиент A открыл файл f на запись. Когда другой клиент, B , также хочет открыть файл f , у него ничего не выходит, поскольку сервер уже зафиксировал тот факт, что клиент A имел возможность к этому моменту изменить файл f . С другой стороны, если бы клиент A открыл файл f на чтение,

ние, то попытка клиента B получить с сервера копию файла f также для чтения была бы успешной, как и попытка B получить этот файл для записи.

Рассмотрим теперь, что произойдет, если у клиентов будут локально храниться несколько копий файла. Произойдет именно то, о чем мы сейчас говорили — только один клиент сможет модифицировать файл f . Если этот клиент изменит f и закроет файл, файл будет передан обратно на сервер. Однако все остальные клиенты будут продолжать читать свои локальные копии, несмотря на то, что эти копии устарели.

Причина такого поведения, явно нарушающего непротиворечивость данных, состоит в том, что сеанс рассматривается в Coda как транзакция. На рис. 10.20 приведена временная диаграмма двух процессов, A и B . Предполагается, что клиент A в ходе сеанса S_A открыл f на чтение, а клиент B в ходе сеанса S_B — на запись.

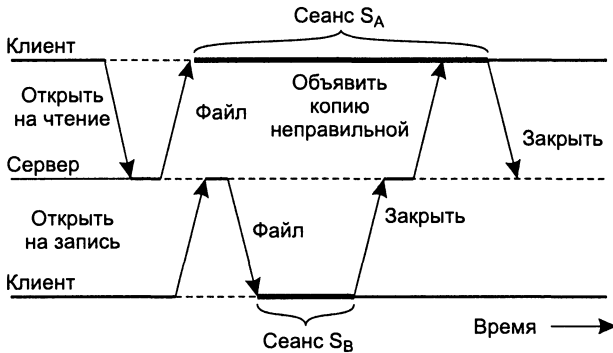


Рис. 10.20. Транзакционное поведение при совместном использовании файлов в Coda

Когда клиент B закрывает сеанс S_B , он пересылает измененный вариант f на сервер, который посылает клиенту A сообщение о неправильности его копии. A теперь знает, что производит чтение устаревшей версии файла f . Однако с точки зрения транзакции это не имеет никакого значения, поскольку сеанс S_A был начат раньше сеанса S_B .

Транзакционная семантика

В Coda ключевую роль для определения транзакционной семантики играет понятие раздела сети. *Раздел (partition)* — это часть сети, изолированная от остальной сети и содержащая коллекцию клиентов, серверов или тех и других. Основная идея состоит в том, что последовательность операций с файлами должна продолжаться выполняться при наличии конфликтующих операций в разных разделах сети. Напомним, что две операции считаются конфликтующими, если обе они работают с одними и теми же данными и как минимум одна из них выполняет запись.

Давайте сначала рассмотрим, как может возникнуть конфликт при наличии разделов сети. Предположим, два процесса, A и B , до момента их отделения друг от друга в результате дробления сети на разделы имеют идентичные реплики

различных разделяемых элементов данных. В идеале файловая система, поддерживающая транзакционную семантику, реализует *поочередную сериализацию копий* (*one-copy serializability*), то есть выполнение операций процессами *A* и *B*, эквивалентно связанному последовательному выполнению этих операций над не-реплицированными элементами данных, совместно используемыми двумя процессами. Эта концепция подробнее обсуждается в [118].

Мы уже рассматривали примеры сериализации в главе 5. Основная проблема, возникающая при разбиении сети на разделы, — это распознавание факта сериализуемого исполнения, имевшего место внутри раздела. Другими словами, при восстановлении из раздела файловая система проверяет множество транзакций, произошедших в каждом разделе (возможно, в теневых копиях, то есть в копиях файлов, которые поддерживались клиентами для проведения пробных модификаций, аналогично теневым блокам в транзакциях). При этом необходимо проверить, можно ли сериализовать выполнение связанных операций в соответствии с последовательностью их поступления. Обычно это нелегкая задача.

Способ, который применяется в Coda для решения этой проблемы, — трактовка сеанса как транзакции. Типичный сеанс начинается с явного открытия файла вызовом `open`. Далее обычно следует последовательность операций записи и чтения, после чего сеанс заканчивается путем закрытия файла операцией `close`. Большинство системных вызовов UNIX образуют один сеанс, состоящий из самих вызовов, который рассматривается в Coda как независимая транзакция.

Coda распознает разные типы сеансов. Так, например, каждый системный вызов UNIX соответствует определенному типу сеанса. Более сложными типами сеансов считаются те, которые начинаются вызовом `open`. Тип сеанса автоматически определяется по системному вызову, сделанному приложением. Важное преимущество подобного подхода состоит в том, что приложение, использующее стандартизованный интерфейс VFS, не требует модификации. Для каждого типа сеанса заранее известно, какие данные будут считаны или модифицированы.

В качестве примера рассмотрим сеанс *сохранения* (*store*), который начинается с открытия файла *f* на запись от имени определенного пользователя *u*, как описывалось выше. В табл. 10.10 перечислены метаданные, ассоциированные с файлом *f* и пользователем *u*. Тип сеанса определяет, будут эти данные только считываться или вдобавок еще и модифицироваться. Так, например, может возникнуть необходимость считать права доступа пользователя *u* к файлу *f*. При явном определении метаданных, которые считываются и модифицируются в конкретном сеансе, определить конфликтующие операции гораздо проще.

Таблица 10.10. Чтение и модификация метаданных для сеанса сохранения в Coda

Ассоциированные с файлом данные	Чтение	Модификация
Идентификатор файла	Да	Нет
Права доступа	Да	Нет
Время последней модификации	Да	Да
Длина файла	Да	Да
Содержимое файла	Да	Да

После явного определения метаданных выяснить ход сеанса достаточно просто. Начнем с рассмотрения серий одновременных сеансов, происходящих в одном разделе сети. Для простоты предположим, что этот раздел расположен на одном сервере. Когда клиент начинает сеанс, процесс Venus на клиентской машине получает от сервера все данные, содержащиеся в наборах операций чтения и записи, в предположении, что правила совместного использования файлов, описанные ранее, не нарушены. Для этого он устанавливает необходимые блокировки чтения и записи.

В этом месте необходимо сделать два важных замечания. Во-первых, поскольку Coda может автоматически определять тип сеанса по системному вызову (первому), который делает приложение, и знает, какие метаданные можно читать и модифицировать в каждом типе сеанса, процесс Venus догадывается, какие данные будут получены с сервера в начале сеанса. В результате он может установить необходимые блокировки в самом начале сеанса.

Второе замечание состоит в том, что раз уж семантика разделения файлов на практике выглядит как предварительная установка всех необходимых блокировок, то используемый при этом подход практически аналогичен технологии двухфазной блокировки (2PL), которую мы разбирали в главе 5. Важным свойством 2PL является возможность упорядочения всех запланированных операций чтения и записи в параллельных сеансах.

Обсудим теперь выполнение сеансов при наличии разделов. Основная проблема состоит в том, что теперь необходимо разрешать конфликты между разделами. С этой целью Coda использует простую схему версий. Каждый файл имеет соответствующий ему номер версии, который показывает, сколько раз в него вносились изменения с тех пор, как он был создан. Как и в предыдущем случае, когда клиент начинает сеанс, на машину клиента копируются все данные, связанные с этим сеансом, включая номер версии, соответствующий каждому из элементов данных.

Предположим, пока клиент выполняет один или более сеансов, происходит разделение сети, при котором клиент и сервер отсоединяются друг от друга. Venus позволяет клиенту продолжить и закончить выполнение данного сеанса, как будто ничего не случилось, в соответствии с описанной ранее транзакционной семантикой для одного раздела. Позднее, после того как соединение с сервером будет восстановлено, изменения будут переданы на сервер в том же порядке, в котором они происходили на клиенте.

Когда изменения, сделанные в файле f , передаются на сервер, мы априори считаем, что другие процессы за то время, пока клиент был отсоединен от сервера, не изменяли информацию в файле f . Конфликт подобного рода может быть легко обнаружен путем сравнения номеров версий. Пусть V_{client} — номер версии f , полученной с сервера при передаче файла клиенту. Пусть $N_{updates}$ — число изменений за прошедший сеанс, переданных клиентом и принятых сервером после повторного подключения. И наконец, пусть V_{now} — текущий номер версии f , находящейся на сервере. В этом случае следующее обновление f в ходе сеанса с клиентом будет принято только в том случае, если выполняется условие:

$$V_{now} + 1 = V_{client} + N_{updates} .$$

Другими словами, изменение, сделанное клиентом, будет принято, только если это изменение приведет к появлению следующей версии файла f . На практике это означает, что в конфликте может победить только один клиент, а значит, конфликт будет так или иначе разрешен.

Конфликты возникнут, если файл f будет изменяться в параллельно происходящих сеансах. При возникновении конфликта обновления, сделанные в ходе сеанса с клиентом, отменяются, но клиент в принципе может сохранить свою версию f для того, чтобы впоследствии можно было согласовать версии вручную. В понятиях транзакций это означает, что сеанс не может быть подтвержден и разрешение конфликта оставляется пользователю. Ниже мы еще вернемся к этому вопросу.

10.2.6. Кэширование и репликация

Как теперь должно быть понятно, кэширование и репликация играют в Coda важную роль. На самом деле они являются основой для достижения заявленной разработчиками Coda цели — высокой доступности. Далее мы сначала рассмотрим кэширование на стороне клиента, которое требуется при выполнении операций в отрыве от сервера. После этого мы рассмотрим репликацию томов на сервере.

Кэширование на клиенте

Кэширование на клиенте чрезвычайно важно для системы Coda по двум причинам. Во-первых, в русле подхода, принятого в AFS [398], кэширование выполняется для улучшения масштабируемости. Во-вторых, кэширование повышает отказоустойчивость, поскольку клиент меньше зависит от доступности сервера. По этим двум причинам клиенты в Coda всегда кэшируют файлы целиком. Другими словами, когда файл открывается — на чтение или на запись, — клиенту передается полная копия файла, которая затем попадает в кэш.

В отличие от многих других распределенных файловых систем, согласованность кэша в Coda обеспечивается при помощи обратных вызовов. Для каждого файла сервер хранит информацию обо всех клиентах, которым была выслана (для последующего кэширования) копия этого файла. Говорят, что сервер записывает для клиента *обещание обратного вызова* (*callback promise*). После того как клиент изменит свою локальную копию файла в первый раз, он уведомляет об этом сервер, который, в свою очередь, рассылает остальным клиентам сообщения о некорректности их копий. Это сообщение о некорректности копий называется *отменой обратного вызова* (*callback break*), поскольку сервер освобождает клиентов от обещания обратного вызова, которое он хранил, ожидая получения измененной версии.

Интересный аспект этой схемы состоит в том, что до тех пор, пока клиент знает, что он оставил серверу обещание обратного вызова, он благополучно работает с файлом локально. В частности, представим, что клиент открывает файл из своего кэша. Он может использовать этот предоставленный сервером файл до тех пор, пока на сервере лежит оставленное для него обещание обратного вызова. Клиент может проверить на сервере, действительно ли еще это обещание. Если

оно действительно, значит, у клиента нет необходимости снова получать файл у сервера.

Этот подход иллюстрирует рис. 10.21, который представляет собой дальнейшую доработку рис. 10.20. Когда клиент *A* начинает сеанс S_A , сервер записывает его обещание обратного вызова. То же самое происходит, когда клиент *B* начинает сеанс S_B . Однако когда *B* закрывает сеанс S_B , сервер отказывается от обещания обратного вызова клиента *A*, посылая ему отмену обратного вызова. Отметим, что в соответствии с транзакционной семантикой Coda, когда клиент *A* закрывает сеанс S_A , не происходит ничего особенного; закрытие просто принимается как ожидаемое событие.

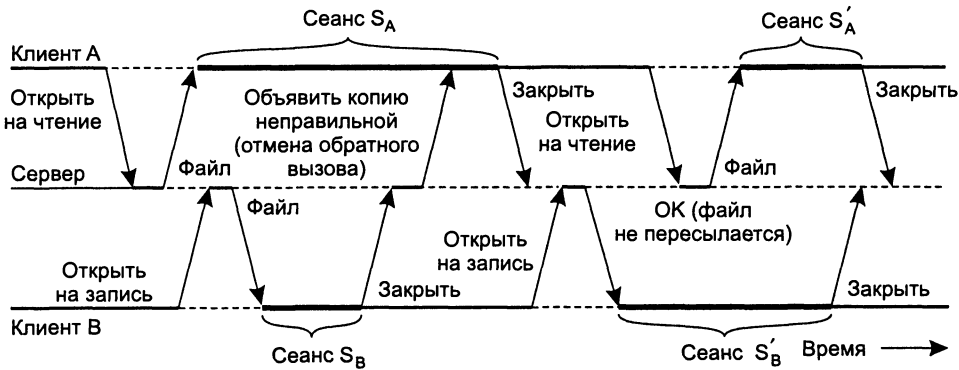


Рис. 10.21. Использование локальных копий при открытии сеанса в Coda

Когда клиент *A* позже захочет открыть сеанс S'_A , он обнаружит, что его локальная копия файла неправильная и нужно получать новую версию с сервера. С другой стороны, когда *B* откроет сеанс S'_B , он обнаружит, что сервер по-прежнему поддерживает оставленное им обещание обратного вызова, а значит, *B* может просто использовать локальную копию файла, сохранившуюся у него в кэше после сеанса S_B .

Репликация серверов

Coda допускает репликацию файловых серверов. Как мы уже говорили, единицей репликации является том. Коллекция серверов, имеющих копию тома, известна под названием *группы хранилищ тома* (*Volume Storage Group, VSG*). В случае отказов клиенту не нужно иметь доступа ко всем серверам из группы хранилищ тома. *Группа доступных хранилищ тома* (*Accessible Volume Storage Group, AVSG*) клиента содержит те серверы из группы хранилищ тома, с которыми клиент может связаться в данный момент. Если группа AVSG пуста, клиент считается *отсоединенным* (*disconnected*).

Для поддержания непротиворечивости реплицированного тома Coda использует протокол реплицированной записи. Так, в частности, используется вариант «читаем раз, пишем все» (*Read-One — Write-All, ROWA*), который мы рассматривали в главе 6. Когда клиенту нужно прочесть файл, он связывается с одним

из членов группы AVSG тома, на котором находится этот файл. Однако при закрытии сеанса с изменением файла клиент пересылает его параллельно всем членам AVSG. Эта параллельная пересылка выполняется при помощи вызовов multiRPC, как было описано ранее.

Эта схема работает правильно до тех пор, пока отсутствуют отказы, то есть для каждого клиента, у которого группа AVSG тома совпадает с его группой VSG. Однако при наличии отказов положение может осложниться. Рассмотрим том, реплицированный на три сервера — S_1 , S_2 и S_3 . Предположим, что для клиента A AVSG состоит из серверов S_1 и S_2 , а клиент B имеет доступ только к серверу S_3 , как показано на рис. 10.22.

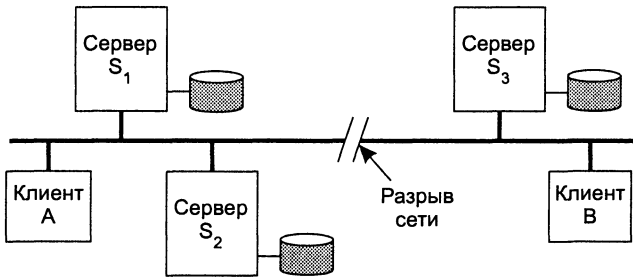


Рис. 10.22. Два клиента с разными группами AVSG для одного реплицированного файла

Coda использует оптимистическую стратегию репликации файлов. В частности, обоим клиентам, A и B , будет разрешено открывать файл f на запись, изменять соответствующие копии и передавать эти копии назад, членам AVSG. Понятно, что в результате в VSG окажутся разные версии файла. Вопрос состоит в том, как обнаружить и устранить это противоречие.

В Coda эта задача решается при помощи расширения схемы версий, которую мы обсуждали в предыдущем разделе. Так, сервер S_i , входящий в VSG, поддерживает *вектор версий Coda* (*Coda version vector*) $CVV_i(f)$ для каждого файла f , содержащегося в данной группе VSG. Если $CVV_i(f)[j] = k$, значит, серверу S_i известно, что сервер S_j содержит как минимум версию k файла f . $CVV_i[i]$ — это номер текущей версии f , хранящейся на сервере S_i . Изменение f на сервере S_i приведет к увеличению $CVV_i[i]$. Отметим, что механизм векторов версий полностью аналогичен механизму векторных отметок времени, который мы обсуждали в главе 5.

Вернемся к нашему примеру с тремя серверами. Вектор $CVV_i(f)$ изначально равен $[1,1,1]$ для каждого из серверов S_i . Когда клиент A считывает f с одного из серверов своей группы AVSG, скажем, S_1 , он получает также и $CVV_1(f)$. После изменения f клиент A производит рассылку f всем серверам своей группы AVSG, то есть серверам S_1 и S_2 . Оба сервера после этого записывают тот факт, что их копии были изменены, а копия S_3 не изменялась. Другими словами:

$$CVV_1(f) = CVV_2(f) = [2,2,1].$$

Тем временем клиент B открывает сеанс, получая при этом копию f с сервера S_3 , после чего изменяет ее. После закрытия сеанса он передает изменения на сервер S_3 , который изменяет свой вектор версий на $CVV_3(f) = [1,1,2]$.

После ликвидации разрыва сети и воссоединения ее сегментов все три сервера должны объединить свои копии *f*. Сравнив векторы версий, они обнаруживают, что между ними существует конфликт, который необходимо устранить. Во многих случаях разрешение конфликта можно автоматизировать, задав способ разрешения в приложении, как утверждается в [243]. Однако остается еще немало случаев, когда пользователи должны разрешать конфликт вручную, особенно если разные пользователи внесли разные изменения в одну и ту же часть одного и того же файла.

10.2.7. Отказоустойчивость

Система Coda была спроектирована с целью поддержания высокой доступности, что в основном отразилось в улучшенной поддержке кэширования на клиентах и репликации серверов. Мы обсуждали то и другое в предыдущем подразделе. Интересная особенность Coda, требующая дополнительного обсуждения: как клиент продолжает работать после отключения от системы, даже если длительность отключения составляет несколько часов или дней.

Операции в автономном режиме

Как мы говорили ранее, клиент считается отсоединенным от тома, если его группа AVSG для этого тома пуста. Другими словами, клиент не может связаться ни с одним из серверов, содержащих копию тома. В большинстве файловых систем (например, в NFS) клиент не может выполнять работу, если он не в состоянии связаться хотя бы с одним сервером. В Coda принят иной подход. Здесь клиент просто переключается на свою локальную копию файла, которую он получил, когда открывал файл на сервере.

Закрытие файла (вернее, сеанса доступа к файлу) в автономном режиме всегда проходит успешно. Однако впоследствии, после восстановления связи, при передаче изменений на сервер могут возникнуть конфликты. В том случае, если автоматическое разрешение конфликтов невозможно, требуется вмешательство человека. Практический опыт использования Coda показывает, что операции в автономном режиме обычно происходят нормально, хотя неразрешимые конфликты и могут привести к невозможности объединения версий.

Успешность подхода, используемого в Coda, в основном объясняется тем фактом, что на практике совместная запись в файл производится редко. Другими словами, на практике редко бывает так, чтобы два процесса открыли один и тот же файл на запись. Разумеется, совместное чтение из файлов встречается гораздо чаще, но оно не вызывает каких-либо конфликтов. Эти замечания относятся также и к другим файловым системам (о разрешении конфликтов в сильно распределенных файловых системах см., например, [338]). Кроме того, транзакционная семантика, лежащая в основе модели совместного использования файлов Coda, также облегчает обработку таких ситуаций, когда несколько процессов одновременно читают один и тот же файл, в то время как один процесс параллельно вносит в него изменения.

Чтобы успешно выполнять операции в автономном режиме, необходимо решить серьезную проблему — как гарантировать, что в кэше клиента окажутся именно те файлы, доступ к которым ему потребуется в автономном режиме. Если использовать простые стратегии кэширования, может оказаться, что клиент не сможет продолжать работу просто по причине отсутствия необходимых файлов. Заблаговременное заполнение кэша нужными файлами называется *накоплением* (*hoarding*). Отношения клиента и тома (а значит, и файлов в этом томе) можно теперь продемонстрировать на диаграмме в виде конечного автомата, которая приведена на рис. 10.23.

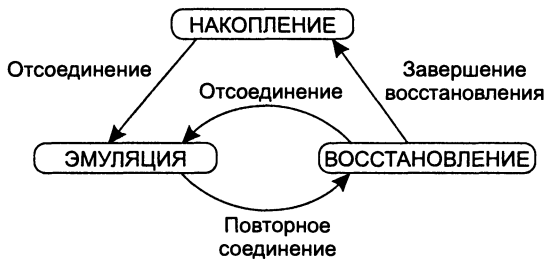


Рис. 10.23. Диаграмма отношений клиента Coda и тома в виде конечного автомата

Обычно клиент находится в состоянии *накопления*. В этом состоянии клиент соединен как минимум с одним сервером, содержащим копию тома. Находясь в этом состоянии, клиент может связываться с сервером и посылать ему запросы на файлы, необходимые для работы. Одновременно он также старается сохранять в своем кэше необходимые данные (например, файлы, атрибуты файлов и каталоги).

В определенный момент число серверов в AVSG клиента падает до нуля, что заставляет его перейти в состояние *эмуляции*, в котором поведение сервера тома эмулируется на машине клиента. На практике это означает, что все запросы к файлам обслуживаются на месте с использованием локально кэшированных копий файла. Отметим, что когда клиент находится в состоянии эмуляции, он по-прежнему может связываться с серверами, поддерживающими другие тома. В этих случаях отсоединение обычно бывает вызвано сбоем конкретного сервера, а не физическим отключением клиента от сети.

И, наконец, когда происходит восстановление связи, сервер переходит в состояние *восстановления*, в котором он пересылает изменения на сервер, чтобы сделать их постоянными. В ходе восстановления обнаруживаются и по возможности автоматически разрешаются конфликты. Как показано на рис. 10.23, существует возможность того, что при восстановлении связь с сервером будет вновь потеряна и клиент вернется обратно в состояние эмуляции.

Для успешного продолжения операций после отсоединения решающим фактором будет наличие в кэше всех необходимых для работы данных. Для того чтобы гарантированно это обеспечить, Coda использует сложный механизм приоритетов. Во-первых, пользователь может явно указать, какие файлы или каталоги он считает важными для себя, перечислив их имена в *базе данных накопления*

(*hoard database*). Coda поддерживает подобные базы на своих рабочих станциях. Сочетание информации из базы данных накопления с информацией о последних запросах к файлам позволяет Coda вычислить текущий приоритет каждого файла, после чего файлы загружаются в кэш в соответствии с их приоритетом так, чтобы выполнялись следующие три условия:

- ◆ не должно существовать некешированных файлов с приоритетом выше, чем у любого кэшированного файла;
- ◆ кэш должен быть заполнен целиком, или все файлы с ненулевым приоритетом должны быть кэшированы;
- ◆ каждый из кэшированных файлов должен быть копией файла, поддерживаемого в группе AVSG клиента.

Детали вычисления текущего приоритета файла описаны в [235]. Если все три условия выполнены, кэш считается находящимся в *равновесии* (*equilibrium*). Поскольку текущий приоритет файлов может со временем меняться, может возникнуть необходимость убрать из кэша файлы, кэшированные ранее, чтобы освободить место для других файлов. То есть равновесие кэша время от времени необходимо пересчитывать. Реорганизация кэша с целью достижения равновесия называется *инвентаризацией накопления* (*hoard walk*) и выполняется каждые 10 минут.

Совместное использование базы данных накопления и функций приоритета вместе с поддержанием равновесия кэша дают значительное улучшение по сравнению с традиционными технологиями управления кэшем, которые обычно основаны на подсчетах и синхронизации. Однако даже эта методика не гарантирует, что клиентский кэш всегда будет содержать все данные, которые понадобятся пользователю в ближайшем будущем. Таким образом, возможна ситуация, когда осуществление операций в автономном режиме будет невозможно по причине недоступности данных.

Восстанавливаемая виртуальная память

Помимо высокой доступности разработчики AFS и Coda предусмотрели также несложный механизм, который помогает строить отказоустойчивые процессы. Простой и эффективный механизм, значительно упрощающий восстановление данных, носит название *восстановимой виртуальной памяти* (*Recoverable Virtual Memory, RVM*). RVM — это технология пользовательского уровня, позволяющая поддерживать важные структуры данных в основной памяти и гарантировать достаточно несложное их восстановление после сбоев. Детально технология RVM описана в [400].

Идея, лежащая в основе RVM, относительно проста: данные, которые должны пережить отказ, обычным образом сохраняются в файле и этот файл при необходимости явно отображается на память. Операции над данными протоколируются, как при использовании журнала с упреждающей записью в транзакциях. На самом деле модель, на которой построена технология RVM, близка к модели плоских транзакций, за тем исключением, что в ней отсутствует поддержка параллельного доступа.

В том случае, когда файл отображается на основную память, приложения могут осуществлять операции с данными этого файла в режиме транзакций. RVM ничего не знает о структурах данных. Поэтому данные в транзакции передаются приложению явно, в виде набора последовательных байтов отображаемого файла. Все операции, которые производятся над этими данными в памяти машины, записываются в отдельный журнал с упреждающей записью. Этот журнал следует держать в устойчивом хранилище. Отметим, что благодаря относительно малым размерам журнала в данном случае можно использовать память с питанием от батареек, которая сочетает долговечность и высокую производительность.

10.2.8. Защита

Система Coda унаследовала архитектуру защиты от AFS. В AFS система защиты состоит из двух частей. Первая часть связана с организацией защищенного канала между клиентом и сервером с использованием защищенных вызовов RPC и системной аутентификации. Вторая часть относится к управлению доступом к файлам. Мы последовательно рассмотрим эти части.

Защищенные каналы

Для организации защищенного канала между клиентом и сервером Coda использует криптосистему с закрытым ключом. Протокол основан на протоколе аутентификации Нидхема—Шредера, который мы рассматривали в главе 8. Говоря вкратце, сначала пользователь должен получить от сервера аутентификации (AS) специальный маркер, или токен. Этот маркер несколько похож на талон, выдаваемый в протоколе Нидхема—Шредера, в том смысле, что он тоже используется для организации защищенного канала с сервером.

Все взаимодействие между клиентом и сервером основано на механизме защищенных вызовов RPC (рис. 10.24). Если Алиса (клиент) хочет пообщаться с Бобом (сервером), она посылает Бобу свои данные вместе с вызовом R_A , который зашифрован общим для Алисы и Боба секретным ключом $K_{A,B}$. Это сообщение обозначено номером 1.

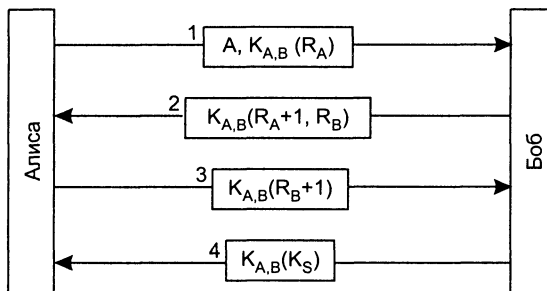


Рис. 10.24. Взаимная аутентификация в RPC2

Боб, расшифровав сообщение 1, отвечает на него и возвращает $R_A + 1$, показывая таким образом, что ему известен секретный ключ, а значит, он — действи-

тельно Боб. Еще он возвращает вызов R_B (как часть сообщения 2), который Алиса расшифровывает и возвращает ему обратно (сообщение 3). После взаимной аутентификации Боб генерирует сеансовый ключ K_S , который может использоваться при последующем взаимодействии между Алисой и Бобом.

Защищенные вызовы RPC в Coda требуются только для организации защищенного взаимодействия между клиентом и сервером, а этого недостаточно для начала защищенного сеанса, в который может быть вовлечено несколько серверов и который может продлиться достаточно долго. Поэтому поверх защищенных вызовов RPC используется второй протокол, в котором клиент, как уже упоминалось, получает от сервера аутентификации маркеры аутентификации.

Маркер аутентификации (authentication token) аналогичен талону Kerberos. Он содержит идентификатор получающего его процесса, идентификатор маркера, сеансовый ключ и отметки времени, указывающие срок начала и срок окончания действия маркера. Для поддержания целостности подлинность маркера может быть подтверждена средствами криптографии. Так, например, если T — это маркер, K_{vice} — секретный ключ, совместно используемый всеми серверами Vice, а H — хэш-функция, то $[T, H(K_{vice}, T)]$ — криптографически удостоверенная версия T . Другими словами, несмотря на то, что T посылается простым текстом по незащищенному каналу, злоумышленник не в состоянии изменить его так, чтобы сервер не заметил подмены.

Когда Алиса подключается к системе Coda, она должна первым делом получить от сервера AS маркеры аутентификации. При этом она переходит на защищенные вызовы RPC, применяя свой пароль для генерации секретного ключа $K_{A,AS}$, который будет использоваться совместно с AS, в том числе и для взаимной аутентификации по ранее описанному механизму. AS возвращает ей два маркера аутентификации. *Свободный маркер (clear token)* $CT = [A, TID, K_S, T_{start}, T_{end}]$ идентифицирует Алису и содержит идентификатор маркера TID , сеансовый ключ K_S и две отметки времени, T_{start} и T_{end} , определяющие срок действия маркера. Кроме того, AS посылает *секретный маркер (secret token)* $ST = K_{vice}([CT]^* K_{vice})$, который представляет собой свободный маркер CT , подписанный общим для всех серверов Vice секретным ключом K_{vice} и зашифрованный тем же самым ключом.

Сервер Vice в состоянии расшифровать маркер ST и извлечь из него как маркер CT , так и сеансовый ключ K_S . Кроме того, поскольку ключ K_{vice} известен только серверам Vice, такой сервер может проверить целостность и с легкостью обнаружить подмену свободного маркера CT (подобная проверка требует знания ключа K_{vice}).

Когда бы Алиса ни создавала защищенный канал с сервером Vice, для идентификации она будет использовать секретный маркер ST , как показано на рис. 10.25. Для шифрования запроса R_A , который она посылает серверу, требуется сеансовый ключ K_S , полученный ранее от AS в свободном маркере.

Сервер, в свою очередь, сначала расшифровывает ST , используя общий секретный ключ K_{vice} , и получает CT . После этого он ищет ключ K_S , который далее используется для завершения аутентификации. Разумеется, сервер также производит проверку целостности CT и продолжает работу, только если маркер действителен.

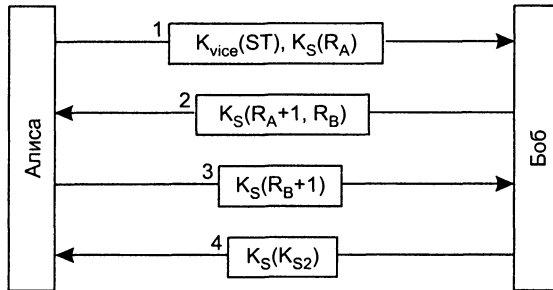


Рис. 10.25. Организация защищенного канала в Coda между клиентом (Venus) и сервером Vice

В том случае если отключенный от сервера клиент для получения доступа к файлам должен пройти процедуру аутентификации, возникают сложности. Аутентификацию провести невозможно, поскольку контакта с сервером нет. В этом случае аутентификация откладывается, а доступ предоставляется. Затем, когда клиент восстанавливает соединение с сервером (серверами), аутентификация производится в состоянии *восстановления* (см. рис. 10.23).

Контроль доступа

Обсудим кратко схему ограничения доступа в Coda. Как и в AFS, для гарантии того, что доступ к файлам имеют только авторизованные процессы, Coda использует списки контроля доступа. Для простоты и масштабируемости в файловых серверах Vice списки контроля доступа ведутся только для каталогов, но не для файлов. Все обычные файлы (то есть исключая вложенные каталоги) в одном и том же каталоге имеют одинаковые права доступа.

Coda подразделяет права доступа по типам операций (табл. 10.11). Отметим, что права исполнять файл не существует. Такая неполнота имеет свою причину: исполнение файлов производится на клиенте и лежит вне зоны ответственности файловых серверов Vice. Если клиент загрузил файл, Vice не может знать, исполняет он его или просто читает.

Таблица 10.11. Классификация операций над каталогами и файлами с точки зрения контроля доступа в Coda

Операция	Описание
read	Чтение любого файла в каталоге
write	Изменение любого файла в каталоге
lookup	Определения статуса любого файла
insert	Добавление нового файла в каталог
delete	Удаление существующего файла
administer	Изменение списка контроля доступа каталога

Coda поддерживает информацию о пользователях и группах. Кроме списков прав, которыми обладают пользователи или группы, Coda также поддерживает

списки негативных прав. Другими словами, можно явно указать, что определенный пользователь *не имеет* некоторых прав доступа. Такой подход кажется весьма удобным, поскольку позволяет немедленно лишить права доступа хулиганящего в системе пользователя, не удаляя его предварительно из различных групп.

10.3. Другие распределенные файловые системы

Помимо тех распределенных файловых систем, которые мы рассмотрели, существует еще великое множество других, но большинство из этих систем очень похожи на NFS или Coda. В этом разделе мы кратко рассмотрим три системы, каждая из которых отличается от прочих одной или несколькими особенностями. Сначала мы обсудим систему Plan 9, в которой все ресурсы трактуются как файлы. XFS является пример файловой системы без серверов. Последней мы рассмотрим файловую систему SFS, в которой информация по защите включена в имена файлов.

В отличие от тех описаний, которые мы делали для NFS и Coda, для обсуждаемых в этом разделе систем мы умышленно отбросим множество деталей, а сосредоточимся на основных принципах, которые легли в основу этих систем, чтобы дать понятие об альтернативных подходах к распределенным файловым системам.

10.3.1. Plan 9 — ресурсы как файлы

Приблизительно в 80-х годах появилась тенденция к замене централизованных систем с разделением времени сетями мощных рабочих станций, вызвавшая появление сетевых операционных систем, которые мы обсуждали в главе 1. Одной из проблем такого подхода была частая потеря прозрачности, что, в свою очередь, привело к созданию так называемых распределенных операционных систем.

Файловая система *Plan 9* была разработана для реализации идеи нескольких централизованных серверов и множества клиентских машин в качестве ответа на появление сетевых операционных систем. Однако вместо мэйнфреймов или мини-компьютеров в качестве серверов предполагается задействовать мощные и относительно дешевые микрокомпьютеры. Серверы, как и прежде, управляют централизованно. С другой стороны, клиентские машины должны быть простыми и ориентированными на выполнение минимального набора задач.

Система задумывалась в основном той же группой людей в Bell Labs, которая разрабатывала UNIX. Проект был начат в конце 80-х. Большое число современных локальных распределенных систем построены по принципу относительно простых клиентов и нескольких мощных серверов. Если обратить внимание на этот факт, можно понять, что хотя идея о централизованно администрируемой системе разделения времени кажется устаревшей, на самом деле она остается вполне актуальной и сейчас. Обзор Plan 9 можно найти в [353]. Подробная информация содержится в [42].

Plan 9 — не столько распределенная файловая система, сколько система распределения файлов. Доступ ко всем ресурсам системы (в том числе и таким, как процессы и сетевые интерфейсы) осуществляется единообразно, с использованием характерных для файлов синтаксиса и операций. Эта идея заимствована в системе UNIX, которая также старается дать ресурсам интерфейс файлов, но в Plan 9 этот подход значительно расширен и выдерживается более последовательно. Каждый сервер имеет иерархически организованное пространство имен для ресурсов, которые он контролирует. Клиент может монтировать пространство имен сервера локально, выстраивая таким образом собственное пространство имен, по аналогии с подходами, используемыми в NFS. Для того чтобы обеспечить возможность разделения имен, часть пространства имен стандартизована.

Подобный подход к построению файловой системы приводит нас к архитектуре, показанной на рис. 10.26. Plan 9 состоит из набора серверов, предоставляющих клиентам ресурсы в виде локальных пространств имен. Для доступа к ресурсам сервера клиент монтирует пространство имен сервера в свое собственное пространство имен. Хотя мы и проводим грань между клиентами и серверами, следует отметить, что в Plan 9 эта грань не всегда ощутима. Так, например, серверы часто выступают в роли клиентов других машин, в свою очередь, клиенты могут экспортировать свои ресурсы на сервер.

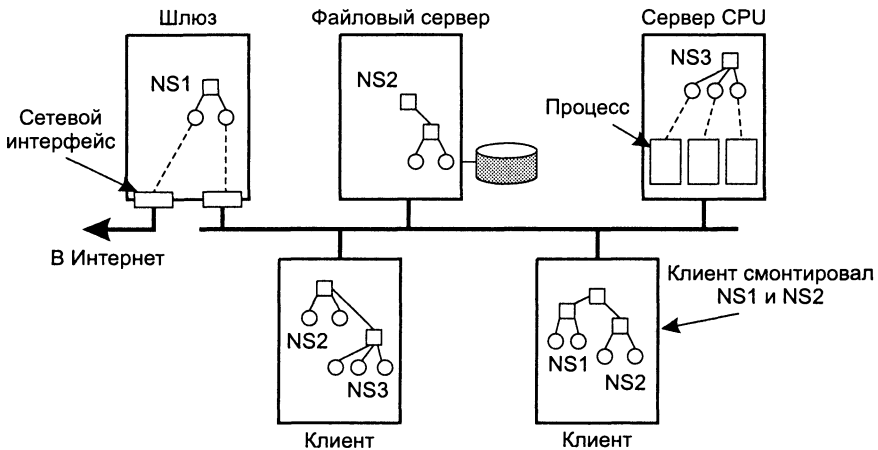


Рис. 10.26. Общая организация Plan 9

Связь

Связь в сети осуществляется в соответствии со стандартным протоколом 9P, который является протоколом, ориентированным на операции с файлами. Так, например, протокол 9P содержит операции открытия и закрытия файлов, чтения и записи блоков данных, навигации в иерархическом пространстве имен. 9P работает с надежными транспортными протоколами. В локальных сетях он использует надежный протокол дейтаграмм *Internet Link (IL)* — надежный протокол на основе сообщений, обеспечивающий доставку сообщений в порядке FIFO. Для глобальных сетей 9P использует TCP.

Интересная особенность Plan 9 состоит в способе поддержания связи на уровне сетевых интерфейсов. Сетевые интерфейсы представляют собой файловую систему, в данном случае состоящую из набора специальных файлов. Этот подход заимствован из UNIX, хотя сетевые интерфейсы в UNIX существуют в виде файлов и не имеют файловой системы. Отметим, что файловая система в данном контексте представляет собой логическое устройство блочной структуры, содержащее все данные и метаданные, составляющие набор файлов. В Plan 9, например, отдельное соединение TCP представляется в виде вложенного каталога, содержащего файлы. Именно так оно выглядит в табл. 10.12.

Таблица 10.12. Файлы, соответствующие в Plan 9 одному соединению TCP

Файл	Описание
ctl	Используется для записи управляющих команд протокола
data	Используется для чтения и записи данных
listen	Используется для получения входящих запросов на соединение
local	Предоставляет информацию на вызывающей стороне
remote	Предоставляет информацию на другом конце соединения
status	Предоставляет диагностическую информацию о текущем состоянии соединения

Файл `ctl` используется для отправки команд, управляющих соединением. Так, например, для открытия сеанса Telnet на машине с IP-адресом 192.31.231.42 с использованием порта 23 нужно, чтобы отправитель записал текстовую строку «connect 192.31.231.42!23» в файл `ctl`. Получатель должен предварительно записать в свой файл `ctl` строку «announce 23», указывая, что он принимает входящие запросы на сеанс на порт 23.

Файл `data` используется для обмена данными при помощи обычных операций `read` и `write`. Эти операции имеют стандартную семантику UNIX для операций с файлами. Так, например, для записи данных в соединение процесс просто вызывает следующую операцию:

```
res = write(fd, buf, nbytes);
```

Здесь `fd` — дескриптор файла, полученный после открытия файла `data`, `buf` — указатель на буфер, содержащий записываемые данные, а `nbytes` — число байтов, которые следует извлечь из буфера. Число байтов, которые на самом деле были записаны, возвращается и сохраняется в переменной `res`.

Файл `listen` используется для ожидания запроса на установку соединения. После того как процесс объявил о своей готовности установить новое соединение, он может осуществлять блокирующее чтение файла `listen`. Если приходит запрос, вызов возвращает дескриптор нового файла `ctl` из созданного каталога соединения.

Процессы

В Plan 9 существуют разные серверы. Каждый сервер реализует свое иерархическое пространство имен. Самый простой пример — файловый сервер Plan 9, кото-

рый представляет собой автономную систему, работающую на отдельной машине. Экспортируемый им граф именования представляет собой дерево, соответствующее набору файлов, хранящихся на нескольких дисках. Логически, как это видно из рис. 10.27, сервер построен в виде трехуровневой системы хранения данных.

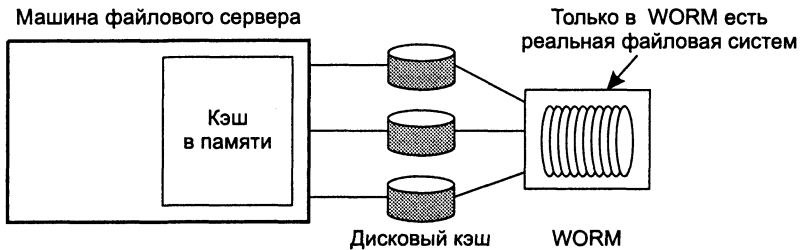


Рис. 10.27. Файловый сервер Plan 9

Самый нижний уровень образует устройство *однократной записи с множественным чтением* (*Write-Once Read-Many, WORM*), представляющее собой хранилище большого объема. Оно напоминает записываемый компакт-диск, но значительно большей емкости. Файловая система, которая используется этим устройством, представляет собой с точки зрения клиента реальную файловую систему.

Промежуточный уровень представляет собой набор магнитных дисков, которые играют роль большой системы кэширования устройства WORM. При доступе к файлу он считывается с устройства WORM и записывается на диск. Изменения временно, как и при традиционном буферном кэшировании в UNIX-системах, также сохраняются на диске. Один раз в сутки все изменения записываются на устройство WORM, которое таким образом реализует ежедневную архивацию всей файловой системы.

Самый верхний уровень сформирован из большого количества буферов памяти, которые выполняют функцию оперативного кэша магнитных дисков. Опять-таки, файлы, считанные с дисков, и их изменения сохраняются в буферах памяти, которые время от времени сбрасываются на диск.

Абсолютно другой пример сервера Plan 9 — сервер для системы работы с окнами $8\frac{1}{2}$ [354]. Концептуально этот сервер должен предоставлять клиентским программам набор файлов, связанных с устройствами типа мыши или экрана. Интерфейсы этих устройств похожи на файлы. Устройства не реализуются в виде файлов, но сделаны очень похожими на них. Так, например, программе, работающей в окне, принадлежит файл с именем `/dev/mouse` для хранения текущего положения мыши в окне. Каждое окно имеет свою версию файла `/dev/mouse`. Ввод с клавиатуры каждая программа читает из файла по имени `/dev/cons`, также отдельного для каждого окна. Другие файлы предоставляют доступ к другим службам, таким как функции управления окнами и графический вывод.

Подход, используемый в Plan 9, представляет собой интересную альтернативу службам. Так, например, сервер вызывает файл `exports`, чтобы позволить машине экспортировать локальное адресное пространство или его часть. На самом деле этот сервер принимает сообщения 9P и транслирует их в локальные систем-

ные вызовы. Для примера предположим, что машина *M* имеет несколько сетевых интерфейсов, которые доступны из локального каталога `/net`. Пусть `/net/inet` обозначает сетевой интерфейс низкоуровневого доступа к внешней сети. Если *M* экспортирует `/net`, клиент может использовать *M* в качестве шлюза, локально смонтировав `/net`, а затем открыв `/net/inet`. Записывая данные в этот файл, клиент с успехом может посылать сообщения во внешнюю сеть, хотя сама по себе клиентская машина не имеет интерфейса с этой сетью.

Подобным же образом клиент может экспортировать свои собственные файлы на удаленный сервер. В этом случае клиент может запустить программу на удаленном компьютере. Если он сделает свои файлы локально доступными для этой программы, реальные вычисления будут происходить на удаленной машине, а не у клиента. Однако для программы все будет выглядеть так, будто она выполняется в локальном адресном пространстве клиента. Отметим, что с точки зрения выполняющейся на удаленной машине программы подобный подход превращает машину клиента в файловый сервер.

Именованное

Как мы уже говорили, любой процесс в Plan 9 имеет свое внутреннее пространство имен, которое собирается из локально монтируемых удаленных пространств имен. Стоит отметить особую специфику именования в Plan 9, которая состоит в том, что несколько пространств имен можно смонтировать в одну точку. Это приведет к образованию так называемого *объединенного каталога (union directory)*. В таком каталоге различные файловые системы выглядят так, словно к ним применили логическую операцию ИЛИ (OR), хотя реально они просто упорядочиваются. Так, например, представим себе, что файловая система FSA имеет вложенные каталоги `/home` и `/usr`, а файловая система FSB — вложенные каталоги `/bin`, `/src` и `/lib`. Как показано на рис. 10.28, когда клиент монтирует эти две файловые системы в одной точке, скажем `/remote`, этот каталог будет содержать пять вложенных каталогов.

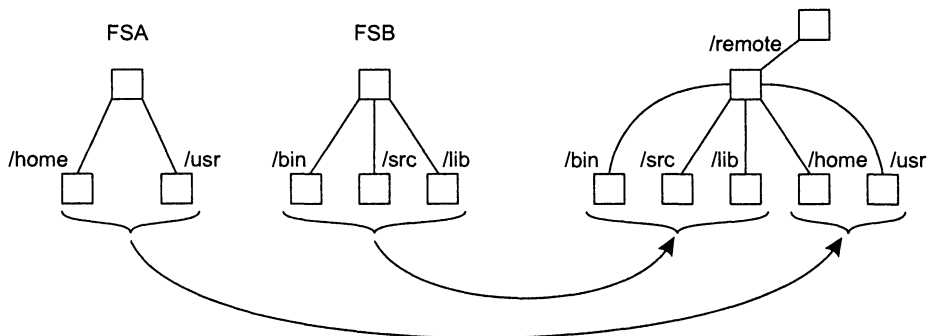


Рис. 10.28. Объединенный каталог в Plan 9

Создание объединенных каталогов происходит при монтировании файловых систем в определенном порядке. Это необходимо для разрешения конфликтов

имен. Так, если файловая система FSA также содержит вложенный каталог /bin, при монтировании FSB после FSA вложенный каталог /bin системы FSB будет помещен после одноименного вложенного каталога FSA. В результате при поиске имени /remote/bin/prog будет возвращен идентификатор файла, найденного в FSA, разумеется, если он там есть, в противном же случае поиск будет продолжен в каталоге /bin в FSB.

Когда на файловом сервере или устройстве создается файл, он помечается двумя целыми числами. *Путь (path)* — это уникальный номер файла, идентифицирующий его по отношению к данному серверу или устройству. Номер *версии (version)* увеличивается при каждом изменении файла на единицу. Для глобальной идентификации файла в пределах системы клиент, смонтировавший у себя файловую систему, должен знать тип и номер устройства сервера или устройства, на котором была создана смонтированная файловая система. Эти номера соответствуют старшему и младшему номерам устройств в UNIX. В результате все файлы глобально уникально идентифицируются четырьмя числами: двумя, идентифицирующими сервер или другое устройство, на котором хранится файл, и двумя, идентифицирующими файл в месте его хранения.

Синхронизация

Для повышения производительности многие распределенные файловые системы после открытия файла пересылают часть его содержимого клиенту. В результате может существовать несколько копий файла, изменяемых различными клиентами параллельно. В семантике сеансов учитываются изменения, внесенные клиентом, который закрывает файл последним, а изменения, внесенные остальными клиентами, пропадают.

Plan 9 реализует семантику разделения файлов UNIX, когда файловый сервер всегда сохраняет копию файла. Все операции изменения всегда пересылаются на сервер. Операции параллельно работающих над файлом клиентов обрабатываются в том порядке, который задает сервер. Изменения при этом никогда не теряются.

Кэширование и репликация

Plan 9 предоставляет пользователям минимальную поддержку кэширования и репликации. Клиенты могут кэшировать файлы, используя специальный сервер пользовательского уровня под названием *cfs*, который обычно запускается автоматически при загрузке машины. *cfs* реализует политику кэша сквозной записи; выполненные операции обновления немедленно записываются на сервер.

Чтобы избежать излишнего перемещения файлов с места на место, клиент может задействовать ранее кэшированные копии, если они продолжают оставаться корректными. Корректность проверяется путем сравнения номера версии кэшированного файла с номером версии файла, хранящегося на сервере. Если с момента последнего кэширования файл был изменен, номера версий не совпадут. В этом случае клиент считает свой кэш некорректным и получает новую копию файла с сервера.

Защита

В Plan 9 пользователи аутентифицируются по методике, похожей на протокол Нидхема—Шредера. Для организации защищенного канала связи с другим пользователем необходимо получить от службы аутентификации талон. Аутентифицируются пользователи, а не машины. Чтобы позволить машине или службе работать по поручению пользователя, Plan 9 поддерживает делегирование.

Защита файлов в Plan 9 осуществляется так же, как в UNIX. По сравнению с другими файловыми системами необычным является разве что представление Plan 9 о группах. В Plan 9 группа считается просто еще одним пользователем, который, возможно, имеет идентифицируемого лидера. Лидер группы обладает специальными правами, так, например, он может изменить права доступа группы к файлу. Если лидер отсутствует, все члены группы считаются равнозначными.

10.3.2. xFS — файловая система без серверов

Следующий пример необычных распределенных файловых систем, который мы рассмотрим, — это система *xFS* [15], разработанная в Беркли в качестве части проекта NOW [16]. *xFS* интересна своей архитектурой без серверов. Вся файловая система распределена по множеству клиентских машин. Такой подход резко контрастирует с архитектурой большинства прочих файловых систем, которые имеют жестко централизованную структуру, даже если в них для распределения и репликации файлов используется не один, а несколько серверов. Так, к подобным системам относятся AFS и Coda.

Следует отметить, что существует абсолютно другая распределенная файловая система под названием XFS (то есть с прописной «X»), разработанная одновременно с *xFS* [444].

Обзор

Система *xFS* была разработана для локальных сетей, в которых машины соединены между собой высокоскоростными каналами. Этим требованиям удовлетворяет множество существующих сетей, например внутренние сети кластеров рабочих станций (мы кратко рассматривали их в главе 1). Для полного рассредоточения данных и функций управления по машинам локальной сети разработчики *xFS* стремились добиться более высокой масштабируемости и отказоустойчивости, чем в традиционных распределенных файловых системах с файловыми серверами (возможно, реплицируемыми). Прототип *xFS* работал на рабочих станциях Sun SPARC и UltraSPARC, соединенных сетью Myrinet [70].

В архитектуре *xFS* выделяют три типа процессов. *Сервер хранения (storage server)* — это процесс, отвечающий за хранение частей файла. Совместно серверы хранения реализуют массив виртуальных дисков, аналогичный реализации дисковых массивов RAID [97]. *Менеджер метаданных (metadata manager)* — это процесс, отслеживающий, где на самом деле хранятся блоки файлов. Отметим, что блоки данных одного файла могут быть разбросаны по нескольким серверам хранения. Менеджер передает запросы клиентов соответствующим серверам

хранения. В этом смысле менеджер метаданных действует подобно серверу локализации для блоков данных, их которых состоят файлы. И наконец, *клиент* (*client*) в xFS — это процесс, который принимает запросы пользователей на выполнение операций с файлами. Каждый клиент имеет возможность кэширования и может передавать кэшированные данные другим клиентам.

Основная идея, положенная в основу xFS при проектировании этой системы, состоит в том, что роли клиентов, менеджеров и серверов могут играть любые машины. В полностью симметричной системе на каждой машине выполняются все три процесса. Однако можно также использовать выделенные машины, на которых будут выполняться только серверы хранения, в то время как на прочих машинах будут наличествовать лишь процессы клиентов или менеджеров. Подобный подход иллюстрирует рис. 10.29.



Рис. 10.29. Типичное распределение процессов xFS по нескольким машинам

Далее мы кратко рассмотрим общую структуру xFS. Детальное описание можно найти в [15]. Опыт разработки xFS вместе с описанием технических деталей изложены в [488]. Что-то похожее на подход, принятый в xFS, то есть разнесение единого хранилища на множество подключенных к сети дисков, описано в [166]. Авторы показали, каким образом можно использовать xFS-подобный способ распределения хранилищ в традиционных распределенных файловых системах, таких как NFS и AFS.

Связь

Изначально любое взаимодействие в xFS осуществлялись при помощи RPC. Однако позже произошел отход от этой практики. Тому было несколько причин, наиболее важной из которых следует считать потерю производительности. Другой проблемой вызовов RPC является тот факт, что они предназначены в основном для сквозной связи. Когда данные и управление полностью распределены среди множества машин, один запрос клиента может вызвать интенсивный обмен несколькими процессами. Серверы RPC требуют явного отслеживания ожидающих обработки запросов и отождествления приходящих ответов с соответствующими им запросами. Простое блокирование здесь не проходит.

По описанным причинам связь в xFS впоследствии была реализована посредством *активных сообщений* (*active messages*). В активных сообщениях [104, 478] сразу определяется, какой процесс будет обрабатывать сообщение на стороне приемника и какие параметры для этого необходимы. Когда сообщение доставляется приемнику, вызванный обработчик производит его полную обработку.

Пока работает обработчик, другие сообщения доставлены быть не могут. Основное достоинство такого подхода — его эффективность. Однако активные сообщения усложняют разработку средств взаимодействия. Так, обработчики не способны блокироваться. Кроме того, обработчики должны быть относительно небольшими, поскольку их работа приостанавливает любое сетевое взаимодействие на выполняющей обработчик машине.

Процессы

Давайте рассмотрим теперь различные процессы xFS в деталях. Как мы уже говорили, все серверы хранения xFS вместе реализуют массив виртуальных дисков. Точнее говоря, xFS реализует систему накопителей на основе разработанной в Беркли [387] *файловой системы со структурой журнала (Log-Structured File System, LFS)*.

В LFS различные операции записи, включая модификацию узлов индексов, блоков каталога и блоков данных, буферизуются и записываются на диск совместно, одной операцией. Все операции записи группируются в одном сегменте фиксированной длины, который добавляется к журналу. Поскольку данные могут быть разбросаны по всему журналу, основная проблема такого подхода состоит в том, как найти нужный блок файла. Для этой цели используется дополнительная таблица — *карта индексов (imap)*, которая отображает ссылки таблицы индексов на их местоположение в журнале. После того как найден индексный узел, появляется возможность найти и блоки файлов.

Для повышения производительности и доступности в xFS используется подход, применявшийся в файловой системе *Zebra* [195], в которой журнал, распределенный по нескольким серверам, организован по аналогии с организацией данных в RAID (рис. 10.30). Когда клиент добавляет к журналу очередной сегмент, он делит этот сегмент на несколько фрагментов одинакового размера, выполняя так называемую *нарезку (striping)*. Каждый из фрагментов сохраняется на собственном сервере хранения. Клиент вычисляет также *фрагмент паритета (parity fragment)*, который хранится на дополнительном сервере. Если какой-либо из серверов перестанет работать, сегмент может быть восстановлен из фрагмента паритета и фрагментов, хранящихся на работоспособных серверах.

Серверы хранения в xFS организованы в *группы нарезки (stripe groups)*. При нарезке сегмента клиент записывает его фрагменты на серверы одной группы нарезки. Этот способ отличается от принятого в *Zebra*, где части сегмента всегда записываются на все серверы хранения. Использование для хранения групп нарезки, а не всех серверов хранения облегчает в xFS добавление новых серверов хранения: отсутствует необходимость в распределении сегмента (и последующей поддержке его частей) на всех серверах. Для того чтобы определить, какой сервер в системе к какой группе нарезки относится, используется глобальная реплицируемая таблица.

Использование файловой системы со структурой журнала требует отслеживания действительных мест хранения данных. Для этой цели каждый файл в xFS имеет соответствующего ему менеджера. Этот менеджер поддерживает карту индексов — таблицу ссылок на индексные узлы, по которой можно найти в журнале

реальный индексный узел файла. Когда над файлом производится операция изменения, индексный узел файла также может быть изменен. Это изменение, в свою очередь, потребует добавления в журнал новой версии индексного узла, при этом менеджер метаданных будет уведомлен об этом и сможет внести изменения в карту индексов.



Рис. 10.30. Принцип нарезки в xFS

В xFS нет единого менеджера файлов. Управление файлами распределено между несколькими менеджерами. Другими словами, в системе может одновременно существовать несколько менеджеров, совместно обрабатывающих информацию о том, где хранятся данные из файлов. Для поиска файла необходимо, чтобы клиент связался с менеджером файлов и запросил у него файл. Для поиска менеджера по идентификатору файла используется отдельная глобальная реплицируемая таблица — *карта менеджеров* (*manager map*). Идентификатор файла соответствует ссылке на индексный узел, о котором мы говорили ранее.

Если не принимать во внимание кэширование на клиентах, клиенты в xFS исключительно просты, поскольку занимаются только отправкой запросов на чтение и запись файлов. Базовый подход к чтению блока данных b из файла f иллюстрирует рис. 10.31. Клиент начинает с поиска файла f в каталоге, и это приносит ему идентификатор файла fid (шаг 1). Полученный идентификатор используется затем для поиска менеджера метаданных, соответствующего fin в карте менеджеров (шаг 2).

После того как менеджер найден, ему передаются значения fid и b , а затем по своим внутренним таблицам менеджер отыскивает реальный индексный узел (шаг 3). После этого в журнале находится точное местоположение индексного узла. Это местоположение состоит из идентификатора группы нарезки, идентификатора сегмента и смещения в этом сегменте.

Все, что нам теперь необходимо, — это определить, на каком сервере находится найденный индексный узел. Для этого необходимо просмотреть группу нарезки, в которую записан нужный файл (шаг 4). В результате этой операции мы получаем список серверов, на которых был сохранен сегмент, содержащий индексный

узел. Используя идентификатор сегмента и смещение, клиент вычисляет, на каком именно компьютере находится индексный узел, и передает этому компьютеру адрес узла на диске (шаг 5). После этого шаги 4 и 5 повторяются с целью чтения блока *b*.

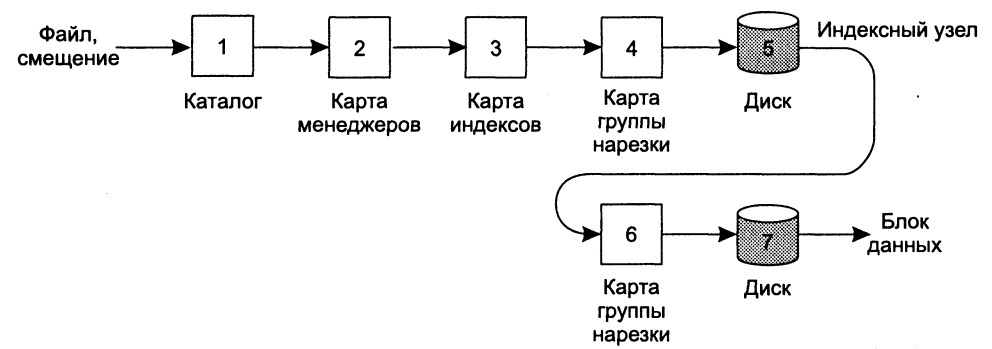


Рис. 10.31. Чтение блока данных в xFS

Именованние

Именованние в xFS не представляет собой ничего сверхъестественного, за исключением разве что различных глобальных идентификаторов, используемых для поиска местонахождения менеджеров, блоков, индексных узлов и т. п. Идентификаторы и структуры данных, применяемые в xFS, собраны в табл. 10.13 [16].

Таблица 10.13. Основные структуры данных в xFS

Структура данных	Описание
Карта менеджеров	Отображение идентификатора файла на менеджера
Карта индексов	Отображение идентификатора файла на адрес индексного узла в журнале
Индексный узел	Отображение номера блока (то есть смещения) на адрес блока в журнале
Идентификатор файла	Ссылка, используемая для индексирования в карте менеджеров
Каталог файлов	Отображение имени файла на идентификатор файла
Адрес в журнале	Группа из идентификатора группы нарезки, идентификатора сегмента и смещения сегмента
Карта группы нарезки	Отображение идентификатора группы нарезки на список серверов хранения

Кэширование

Клиенты в xFS поддерживают локальное кэширование блоков. Схема поддержания непротиворечивости похожа на соответствующую схему Coda за исключением того, что вместо кэширования файлов используется кэширование блоков. Другими словами, когда клиент хочет записать что-то в блок данных, он связывается с менеджером файлов, которому принадлежит этот блок, и просит у него раз-

решения на запись. Менеджер, в свою очередь, первым делом объявляет неправильными все копии этого блока, хранящиеся в кэшах других клиентов, после чего дает разрешение на запись. Менеджер отслеживает, где имеются кэшированные блоки файла и какой клиент затребовал разрешение на запись. Последний называется также текущим владельцем.

В течение того времени, пока владелец имеет разрешение на запись данного блока, он не связывается с менеджером, а просто использует для изменения данных свою кэшированную копию. В том случае если другой клиент пожелает получить доступ к этому же блоку, разрешение на запись отзывается. В этот момент блок, находящийся в кэше текущего пользователя, сохраняется с целью пересылки его на сервер хранения. Кроме того, он пересылается новому клиенту, который становится его следующим владельцем.

В системе xFS реализуется также *кооперативное кэширование* (*collaborative caching*). Согласно этой схеме, когда клиент хочет получить доступ к блоку данных, он связывается с соответствующим менеджером так, как это было описано. Однако вместо поиска запрашиваемого блока на сервере хранения менеджер в первую очередь проверяет, не кэширует ли этот блок один из запросивших его клиентов. Если это так, то клиент получает копию из кэша другого клиента.

Отказоустойчивость

В результате реализации серверов хранения в виде групп нарезки, которые поддерживают избыточность информации, доступность в xFS по сравнению с традиционными подходами повышена. Если на одном из серверов случается сбой, его фрагмент может быть восстановлен при помощи фрагмента паритета, хранящегося на одном из других серверов. Если будет утерян фрагмент паритета, его можно легко вычислить из существующих фрагментов. Тем не менее при сбое на нескольких серверах одной группы нарезки необходимо предпринимать специальные меры, а они-то в системе и не реализованы.

Восстановление состояния менеджеров реализовано путем записи контрольных точек с обрабатываемыми менеджером данными. Одна из наиболее сложных проблем — восстановление карты индексов. Установка контрольных точек способна частично разрешить эту проблему, но нам следует учитывать и изменения, внесенные после записи последней контрольной точки. Для этого клиенты отслеживают изменения, отосланные менеджеру после создания последней контрольной точки. Таким образом, после восстановления в контрольной точке менеджер может потребовать, чтобы клиенты повторили изменения, произошедшие с момента создания последней контрольной точки, чтобы привести карту индексов в состояние, соответствующее реальному положению дел.

Восстановление в xFS реализовано только частично. Большинство необходимых механизмов в реализацию прототипа не включены. Детали можно найти в [15].

Защита

Система xFS обеспечивает лишь минимальную защиту. Кроме реализации обычных механизмов контроля доступа, xFS требует, чтобы клиенты, менеджеры

и серверы хранения работали на доверенных, обеспечивающих защиту машинах. Однако было предложено и простое средство защиты, которое позволяет организовать доступ не облеченных доверием клиентов NFS к набору машин xFS с использованием защищенных вызовов RPC, которые мы обсуждали в пункте 10.1.8. На самом деле клиент NFS путем защищенного вызова RPC связывается с клиентом xFS. В этом случае клиент xFS играет для клиента NFS роль сервера NFS.

10.3.3. SFS — масштабируемая защита

В качестве последнего примера мы кратко рассмотрим систему с совершенно особым подходом к обеспечению защиты. В *защищенной файловой системе (Secure File System, SFS)* основной особенностью является отделение средств управления ключами от средств защиты файловой системы (см. [287]). Другими словами, SFS гарантирует, что клиент не сможет получить доступ к файлу, не владея соответствующим секретным ключом. Однако получение ключа абсолютно не зависит от доступа к файлам.

Обзор

Общая структура SFS представлена на рис. 10.32. Для того чтобы обеспечить совместимость системы с разнообразными машинами, она выполнена в виде нескольких компонентов NFS версии 3. На машине клиента помимо программы пользователя размещаются три разных компонента системы. Клиент NFS используется как интерфейс с программой пользователя и обменивается информацией с клиентом SFS. Последний представляется клиенту NFS в качестве сервера NFS. Другими словами, эти два компонента обмениваются информацией через систему ONC RPC по протоколу NFS.

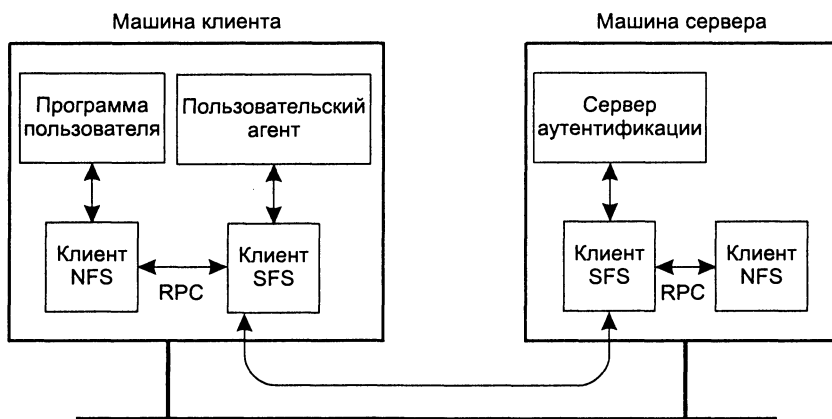


Рис. 10.32. Структура SFS

Клиент SFS отвечает за организацию защищенного канала с сервером SFS. Он отвечает также за связь с локальным *пользовательским агентом (user agent)*

SFS, который представляет собой программу, автоматически осуществляющую идентификацию пользователя. SFS не предписывает способ аутентификации пользователей. В соответствии с принципами своего создания, SFS выделяет этот процесс в отдельную задачу и использует разные агенты для разных протоколов аутентификации пользователя.

Со стороны сервера также имеется три компонента. Из соображений переносимости по-прежнему используется сервер NFS. Этот сервер поддерживает связь с сервером SFS, который для сервера NFS представляется клиентом NFS. Сервер SFS создает процесс ядра SFS, который отвечает за обработку файловых запросов клиентов SFS. Аналогично агенту SFS сервер SFS для аутентификации пользователей поддерживает связь с выделенным сервером аутентификации.

Связь

Клиент SFS и сервер SFS взаимодействуют по немного адаптированному варианту протокола NFS версии 3. Когда клиент открывает файл, он получает атрибуты файла, которые также могут быть кэшированы. В SFS кэширование этих атрибутов ограничено арендой. Когда срок аренды истекает, клиент SFS считает атрибуты файла не соответствующими действительности. Кроме того, сервер SFS позволяет клиенту самому послать обратный вызов и объявить атрибуты файла не соответствующими действительности. Фактически SFS выглядит как дополнение к NFS, реализующее отдельные важные аспекты работы с файлами, которые в настоящее время реализованы в NFS версии 4.

Процессы

Как можно видеть на рис. 10.32, SFS представляет собой традиционную систему клиент-сервер. Однако она имеет важную особенность, которая состоит в том, что клиент SFS не обладает возможностями конфигурирования под конкретное местоположение. Другими словами, клиент SFS абсолютно независим от местоположения пользователя SFS. Подобный подход совпадает с идеей о том, что пользователь должен быть в состоянии получить доступ к своим файлам из любой точки земного шара. Вместо того чтобы разрешать доступ клиентам, сервер SFS предоставляет право на доступ пользователям.

Именован

Уникальной по сравнению с другими распределенными файловыми системами SFS делает организация пространства имен. SFS поддерживает глобальное пространство имен с корнем в каталоге под названием /sfs. Клиент SFS позволяет своим пользователям создавать внутри этого адресного пространства символические ссылки.

Что более важно, SFS использует в качестве имен файлов *самосертифицирующиеся имена nymей (self-certifying pathnames)*. Эти имена, в сущности, несут всю информацию, необходимую для аутентификации сервера SFS, на котором находится файл, идентифицируемый таким именем. Самосертифицирующееся имя, как видно на рис. 10.33, состоит из трех частей.

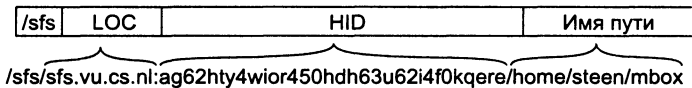


Рис. 10.33. Самосертифирующееся имя в SFS

Первая часть имени (*LOC*) определяет местоположение, которое представляет собой либо доменное имя DNS, идентифицирующее сервер SFS, либо соответствующий ему IP-адрес. SFS предполагает, что каждый сервер *S* имеет соответствующий открытый ключ K_S^+ . Вторая часть самосертифирующегося имени (*HID*) — это идентификатор хоста, который вычисляется с использованием криптографической хэш-функции *H* по месторасположению сервера и его открытому ключу:

$$HID = H(LOC, K_S^+).$$

HID представляет собой число из 32 цифр в системе счисления по основанию 32. Третья часть образована локальным именем пути сервера SFS, на котором находится файл.

Когда клиент пытается получить доступ к серверу SFS, он может аутентифицировать этот сервер, просто запросив у него открытый ключ. Используя известную функцию хэширования *H*, клиент может вычислить *HID* и сверить его со значением, содержащимся в имени пути. Если они совпадут, клиент может быть уверен, что имеет дело с сервером, носящим искомое имя. После этого сервер должен аутентифицировать пользователя, но, как мы уже говорили, это происходит независимо от SFS. В [287] описана реализация, при которой клиент использует своего локального агента для управления закрытым ключом пользователя и аутентификации на сервере.

Защита

Как отделить механизм управления ключами от механизмов защиты файловой системы? Эту проблему решает SFS. Один из способов получения ключа сервера — позволить клиенту связаться с сервером и запросить ключ, как было ранее описано. Однако можно также хранить коллекцию ключей локально, например администратором системы. В этом случае связываться с сервером не нужно. Вместо этого при разрешении имени ключ сервера отыскивается локально, после чего идентификатор хоста можно проверить по части имени пути, относящейся к местоположению.

Прозрачность именования, позволяющая упростить этот процесс, может быть достигнута при помощи символических ссылок. Так, предположим, что клиент хочет получить доступ к файлу по имени:

```
/sfs/sfs.vu.cs.nl:ag62hty4wior450hdh63u62i4f0kqere/home/steen/mbox
```

Чтобы скрыть имя хоста, пользователь может создать символическую ссылку:

```
/sfs/vucs —> /sfs/sfs.vu.cs.nl:ag62hty4wior450hdh63u62i4f0kqere
```

После этого достаточно использовать только имя `/sfs/vucs/home/steen/mbox`. При разрешении это имя будет автоматически расширено до полного имени SFS,

а найденный локально открытый ключ позволит идентифицировать сервер SFS `sfs.vu.cs.nl`.

Таким же образом SFS может поддерживаться сертифицирующими организациями. Обычно подобная организация поддерживает ссылки на серверы SFS, с которыми она работает. В качестве примера рассмотрим сертифицирующую организацию (CA), которая запускает сервер SFS по имени:

```
/sfs/sfs.certsfs.com:kty83pad72qmbna9uefdppioq7O53jux
```

Предположим, что у клиента уже установлена символическая ссылка:

```
/certsfs —> /sfs/sfs.certsfs.com: kty83pad72qmbna9uefdppioq7O53jux
```

Тогда сертифицирующая организация может использовать другую символическую ссылку:

```
/vucs —> /sfs/sfs.vu.cs.nl:ag62hty4wior450hdh63u62i4f0kqere
```

Эта ссылка указывает на сервер SFS `sfs.vu.cs.nl`. Тогда клиент может просто сослаться на `/certsfs/vucs/home/steen/mbox`, и это будет означать, что он запрашивает файловый сервер, открытый ключ которого сертифицирован организацией CA.

Эти примеры по методам управления ключами взяты из [287]. Детальное описание реализации и вопросов производительности SFS можно найти в [286]. Там же содержится подробное описание различных протоколов, используемых в SFS для обеспечения защиты.

10.4. Сравнение распределенных файловых систем

Мы обсудили пять разных файловых систем. Хотя эти системы имеют много общих черт, они также во многом различаются. Далее мы проведем сравнение этих систем, обращая особое внимание на NFS и Coda.

10.4.1. Философия

Цели разработки файловых систем часто весьма разнятся. NFS — это в основном система, предоставляющая клиентам прозрачный доступ к файловой системе конкретного удаленного сервера. В этом смысле она похожа на системы объектов, которые поддерживают только удаленные объекты. Разработчики NFS версии 4 также постарались реализовать прозрачный доступ к глобальным сетям, сведя на нет задержки на передачу данных. Этого удалось добиться путем полного кэширования файлов на клиенте и локального выполнения всех операций.

Основная цель разработки Coda состояла в обеспечении доступности даже в случае разрывов в сети и работе в автономном режиме. Система Coda также соответствует целям создания AFS, наиболее важной из которых является масштабируемость. Сочетание высокой доступности и масштабируемости выделяет Coda среди большинства других распределенных файловых систем.

Система Plan 9 в первую очередь являет собой распределенную систему разделения времени, в которой доступ ко всем ресурсам осуществляется единообразно, а именно в форме доступа к файлам. Как мы показали, система Plan 9 может быть также названа системой распределения файлов.

Система xFS разделяет те же цели, для которых разрабатывались многие другие распределенные файловые системы, в том смысле, что она также обеспечивает высокую доступность и масштабируемость. Важнее, однако, что заявленных целей xFS добивается в условиях отсутствия выделенных серверов, что, собственно, и являлось основной целью разработчиков.

Система SFS предназначена для масштабирования системы защиты. Она выполняет эту задачу путем раздельного решения вопросов управления файловой системой и вопросов ее защиты. Любому пользователю позволено запустить собственную службу SFS, не внося изменений в работу централизованных служб. Важную роль в этом играет организация пространства имен — в имя файла включается открытый ключ сервера, на котором находится файл.

Существуют и другие немаловажные различия, например, в базовых моделях файлов, поддерживаемых каждой из систем. Так, например, NFS, Plan 9 и SFS поддерживают модель удаленного доступа, в которой за файлы постоянно отвечает сервер, совершающий все операции с файлами. Можно считать, что Coda, в свою очередь, до определенной степени поддерживает модель загрузки-выгрузки. Это связано с тем агрессивным кэшированием файлов, которое эта система унаследовала от AFS. В XFS используется совершенно другой подход, в котором реализация файловой системы имеет структуру журнала.

10.4.2. Связь

При сравнении применяемых схем связи обнаруживается, что реально в большинстве распределенных файловых систем применяется та или иная форма вызовов RPC. NFS, Coda и SFS прямо используют базовую систему RPC, которая иногда может быть оптимизирована для работы в определенной ситуации.

Можно сказать, что Plan 9 также использует RPC, но реальный протокол был доработан для поддержки операций с файлами, что несколько отличает его от других протоколов.

Полезно отметить, что разработчики xFS тоже начали с организации взаимодействия при помощи вызовов RPC. Однако для повышения производительности и в плане обеспечения связи с продуктами разных производителей на смену вызовам RPC пришли активные сообщения.

10.4.3. Процессы

Приведенные примеры систем различаются по той роли, которую клиенты играют в файловой системе. В структуре систем клиент-сервер, описанной в главе 1, в NFS версии 3 клиентский процесс может, в сущности, быть «тонким». Другими словами, большая часть работы выполняется файловым сервером, в то время как клиент NFS лишь обращается к серверу, запрашивая выполнение определенных

операций. В NFS версии 4 клиенты умеют значительно больше, так, им разрешено кэшировать файлы и выполнять многие операции локально. Другими словами, в NFS версии 4 клиенты могут считаться скорее «толстыми», чем «тонкими».

Те же самые рассуждения годятся и для процесса Venus, представляющего собой клиентскую часть систем AFS и Coda. Venus выполняет на стороне клиента массу работы. Он, в частности, целиком берет на себя кэширование файлов, эмуляцию функций сервера при отключении от системы и т. п.

В противоположность ему разработчики Plan 9 постарались сохранить клиента настолько простым, насколько это возможно. В их проекте всю работу выполняет сервер, а клиенты представляют из себя всего лишь простые терминалы. Тем не менее клиентские процессы могут кэшировать файлы, хотя система продолжает отлично работать и при отсутствии какого-либо кэширования.

Клиентские процессы в xFS также можно считать «толстыми», если принять во внимание кэширование. С другой стороны, единственное, что делает клиент xFS, — это передает файловые операции серверам хранения. Однако клиенты xFS реализуют кооперативное кэширование, что повышает производительность их работы.

Подход, принятый в SFS, представляет собой нечто среднее между простым клиентом Plan 9 и усложненным вариантом клиента Coda. Клиент SFS — это относительно простой компонент, который не использует специфических особенностей своего локального состояния. Он просто предоставляет доступ к удаленным серверам SFS. При необходимости его можно дополнить специальным агентом для аутентификации пользователя или сервера.

Серверы разных систем очень отличаются друг от друга. Только Coda и xFS позволяют объединять серверы в группы. Каждая группа может содержать реплицированные файлы или, в случае xFS, отвечать за нарезку файлов. В NFS, Plan 9 и SFS, в сущности, предполагается, что файл расположен на одном нереплицируемом сервере.

10.4.4. Именованное

За исключением SFS, рассмотренные примеры файловых систем поддерживают более или менее одинаковые пространства имен, создаваемые при монтировании, как это делается и в системах UNIX. Монтирование производится дроблением каталогов, как в NFS (а также и в SFS), или при помощи файловой системы, как в Coda, Plan 9 и xFS (которая унаследовала этот способ от LFS и Zebra).

Более интересен подход к организации пространства имен. Существует два основных метода. Согласно первому из них, каждый пользователь имеет собственное пространство имен. Так построены системы NFS и Plan 9. Недостаток пространств имен, принадлежащих пользователям, состоит в том, что в них нелегко организовать совместное использование файлов, поскольку с их именами возникают сложности. Чтобы снизить остроту некоторых из этих проблем, часть пространства имен может быть стандартизована.

Второй подход состоит в том, чтобы создать глобальное общее пространство имен, как это сделано в Coda, xFS и SFS. Во всех этих системах каждый пользователь имеет право расширить глобальное пространство имен за счет своего

внутреннего локального пространства имен. Так, например, клиент SFS позволяет пользователю локально создать символическую ссылку. Эти имена являются внутренними именами пользователя и невидимы для пользователей других клиентов SFS.

Имеются также различия и в ссылках на файлы. За исключением Coda и xFS ни одна из систем не использует глобально уникальные ссылки на файлы. В Coda ссылка на файл состоит из двух частей. Первая часть — это не зависящий от местоположения идентификатор тома, вторая часть — дескриптор, который однозначно определяет файл в этом томе. Взятые вместе, эти части образуют глобально уникальное имя.

В xFS для ссылки на файлы используется дополнительный уровень косвенности, а именно менеджеры метаданных. Ссылка на файл, в сущности, представляет собой ссылку на менеджер, который затем используется для отыскания индексного узла. Эти ссылки на файл глобально уникальны.

Другие системы используют ссылки, уникальные для файловой системы, частью которой являются эти файлы (как в NFS и SFS), или для сервера, отвечающего за эти файлы, как в случае Plan 9. В результате для поиска файла эти системы нуждаются в дополнительной информации.

10.4.5. Синхронизация

Наиболее важный аспект синхронизации при обсуждении распределенных файловых систем — это семантика разделения файлов, которая в них применяется. Грубо говоря, в NFS принята семантика сеансов, то есть на сервере запоминаются только те изменения, которые внесены процессом, закрывшим файл последним. В Coda поддерживается семантика транзакций, в том смысле, что допустимы только те сеансы, которые могут быть сериализованы. Однако в случае работы в автономном режиме эта семантика не гарантируется, и может возникнуть конфликт обновлений, который затем должен быть разрешен.

Поскольку все операции с файлами в Plan 9 осуществляются на файловом сервере, Plan 9 поддерживает семантику UNIX. Аналогичная семантика поддерживается также и в системе xFS при помощи встроенного в нее механизма кооперативного кэширования, при котором только текущий владелец блока файла имеет право выполнять операции записи.

Семантика совместного использования файлов в SFS не определена. Однако поскольку система SFS в значительной степени основана на протоколах NFS, она как минимум поддерживает семантику сеансов.

10.4.6. Кэширование и репликация

Все приведенные примеры систем поддерживают кэширование на клиенте. Только в Plan 9 используется протокол непротиворечивости кэша сквозной записи, который предписывает немедленную передачу всех операций записи на сервер. Другие системы реализуют кэш обратной записи, позволяющий клиенту до сброса кэша на диск сервера выполнить несколько операций записи. В NFS (а также

в SFS) протокол поддержания целостности кэша в значительной степени определяется реализацией, хотя NFS версии 4 и предписывает сбрасывать изменения на сервер при закрытии файла.

XFS кэширует блоки данных файлов. Plan 9 кэширует данные с переменным шагом, в зависимости от характера доступа клиента. NFS (и SFS) отставляют этот вопрос открытым, хотя можно рассчитывать, что в системе NFS версии 4 в плане поддержки глобальных сетей будет существовать возможность кэширования целых файлов. В Coda кэширование целых файлов поддерживается явно, эта особенность унаследована еще от AFS.

Поддержка репликации файлов на серверах во всех системах, за исключением Coda, в основном не реализована. NFS версии 4 предоставляет минимальную поддержку репликации файловых систем через атрибут `FS_LOCATIONS`. В NFS версии 3 репликация не реализована.

Plan 9 также не предоставляет явной поддержки репликации. В xFS поддерживаются некоторые возможности репликации через нарезку. В Coda явно учитывается тот факт, что тома могут быть реплицированы, а для поддержания их непротиворечивости используется протокол ROWA.

10.4.7. Отказоустойчивость

В большинстве случаев поддержка отказоустойчивости минимальна и ограничивается использованием надежной связи. В NFS даже надежная связь требует дополнительной поддержки, так как базовая система RPC не поддерживает семантики «максимум однажды». В Coda для повышения доступности применяется кэширование и репликация. В xFS используется нарезка, которая способна защитить систему от сбоя одного сервера из группы нарезки.

Восстановление в NFS в основном производится клиентом, в том смысле, что клиент может восстанавливать ресурсы, например, блокировки, информация о которых была потеряна сервером при отказе. Coda не определяет, каким образом восстанавливается сервер, но предоставляет достаточно серьезную поддержку средств восстановления при дроблении сети. Такая же методика применяется и для повторного включения восстановившегося после сбоя сервера в группу серверов.

В XFS применяются контрольные точки. Кроме того, клиент ведет журнал, который помогает восстановить события, имевшие место с момента создания контрольной точки, когда данные менеджера были непротиворечивы, до окончания записи информации в журнал. Однако большая часть механизмов восстановления не реализована.

10.4.8. Защита

В NFS версии 4 проводится четкая грань между механизмами защиты и их реализацией. Для организации защищенных каналов NFS предоставляет стандартный интерфейс в форме `RPCSEC_GSS`, доступ к которому могут иметь многие из существующих систем защиты. До момента реализации NFS об используемом механизме можно не заботиться.

Coda и Plan 9 предоставляют защищенные каналы, причем их реализация в обеих системах основана на протоколе аутентификации Нидхема—Шредера, который мы обсуждали в главе 8. При таком подходе используются общие секретные ключи.

В xFS защищенные каналы не поддерживаются. Вместо них задача обеспечения защиты системы возлагается на разные машины. Поэтому для доступа к системе xFS с не доверенных машин необходимо предпринимать специальные меры. В настоящее время для этого применяют защищенные вызовы RPC, лежащие в основе NFS версии 3.

В SFS имеется собственный способ работы с защищенными каналами. В сущности, эта система поддерживает любые механизмы, но предпринимает специальные меры, когда дело доходит до управления ключами. В случае самосертифицирующихся имен аутентификация сервера пользователем может происходить по-разному. Для аутентификации пользователей на сервере могут использоваться специальные агенты.

NFS версии 4 поддерживает обширный список операций контроля доступа. Список контроля доступа представляет собой атрибут файла, благодаря которому можно добиться большой гибкости. Coda использует другой подход, осуществляя контроль доступа только по отношению к операциям над каталогами. Plan 9 и xFS в основном поддерживают стандартный подход UNIX, разделяя операции чтения, записи и выполнения. И наконец, SFS наследует механизмы контроля доступа от NFS версии 3.

Подведем итог. В табл. 10.14 сравниваются различные аспекты пяти распределенных файловых систем.

Таблица 10.14. Сравнение систем NFS, Coda, Plan 9, xFS и SFS

Аспект	NFS	Coda	Plan 9	XFS	SFS
Цели создания	Прозрачность доступа	Высокая доступность	Унификация	Бессерверная система	Масштабируемая безопасность
Модель доступа	Удаленный доступ	Загрузки-выгрузки	Удаленный доступ	На основе журнала	Удаленный
Связь	RPC	RPC	Специальная связь	Активные сообщения	RPC
Клиентский процесс	Тонкий и толстый	Толстый	Тонкий	Толстый	Средний
Группы серверов	Не поддерживаются	Поддерживаются	Не поддерживаются	Поддерживаются	Не поддерживаются
Дробность монтирования	Каталог	Файловая система	Файловая система	Файловая система	Каталог
Пространство имен	Клиент	Глобальное пространство	Пространство имен процесса	Глобальное пространство	Глобальное пространство

Аспект	NFS	Coda	Plan 9	XFS	SFS
Область видимости идентифика- тора файла	Файловый сервер	Глобальная видимость	Сервер	Глобальная видимость	Файловая система
Семантика совместного использо- вания	Семантика сеансов	Семантика транзакций	Семантика UNIX	Семантика UNIX	Не определено
Элемент кэширова- ния	Файл (версии 4)	Файл	Файл	Блок	Не определено
Непротиво- речивость кэша	Кэш обратной записи	Кэш обратной записи	Кэш сквозной записи	Кэш обратной записи	Кэш обратной записи
Репликация	Минималь- ная	ROWA	Отсутствует	Нарезка	Отсутствует
Отказоустой- чивость	Надежная связь	Репликация и кэширо- вание	Надежная связь	Нарезка	Надежная связь
Восстанов- ление	На основе клиента	Повторная интеграция	Не определено	Контроль- ные точки и журнал	Не определено
Защищен- ные каналы	Существую- щие механизмы	Протокол Нидхема— Шредера	Протокол Нидхема— Шредера	Имена путей отсутствуют	Самосерти- фицирую- щиеся имена путей
Контроль доступа	Множество операций	Операции с каталогами	На основе UNIX	На основе UNIX	На основе NFS

10.5. Итоги

Распределенные файловые системы представляют собой одну из важных парадигм распределенных систем. Они строятся в основном в соответствии с моделью клиент-сервер с кэшированием на клиенте и поддержкой репликации серверов для получения необходимой масштабируемости. Кэширование и репликация требуются также для повышения доступности этих систем.

Распределенные файловые системы отличаются от нераспределенных семантикой разделения файлов. В идеале файловые системы всегда позволяют клиенту прочесть данные, записанные в файл последними. Такую семантику разделения файлов (семантику UNIX) очень трудно эффективно реализовать в распределенных системах. NFS поддерживает ее «урезанный» вариант, именуемый семантикой сеансов, при которой итоговая версия файла определяется последним клиентом, закрывшим файл, открытый на запись. В Coda разделение файлов происходит в соответствии с семантикой транзакций в том смысле, что клиенты, выполняющие чтение, получают последнюю версию файла, только повторно открыв его. Семантика транзакций в Coda не обладает всеми свойствами ACID настоящих

транзакций. В том случае, если управление всеми операциями осуществляет сервер, может использоваться и базовая семантика UNIX, хотя масштабируемость в этом случае останется под вопросом.

Чтобы добиться приемлемой производительности, распределенные файловые системы обычно позволяют клиентам кэшировать файлы целиком. Кэширование целых файлов поддерживается в NFS и Coda, хотя существует возможность сохранять также и большие фрагменты файлов. После того как файл открыт и передан (частично) клиенту, все остальные операции могут осуществляться локально. Изменения сбрасываются на сервер перед закрытием файла. С другой стороны, такие системы, как Plan 9 и xFS, поддерживают блочное кэширование в комбинации с протоколом кэша обратной записи.

Общепринятая практика состоит в том, что распределенная файловая система не строится поверх транспортного уровня, а использует существующий уровень RPC, так что все операции представляют собой вызовы RPC, адресованные файловому серверу, и необходимости в передаче сообщений не возникает. Такой подход используется в NFS, Coda и SFS. Желательно, чтобы уровень RPC поддерживал семантику обращений «максимум однажды», в противном случае эта семантика должна быть явно реализована в форме части уровня файловой системы, как в случае NFS.

Coda отличается от других распределенных файловых систем тем, что поддерживает работу в автономном (отключенном от сети) режиме. Если гарантировать, что кэш клиента всегда содержит данные, которые могут потребоваться ему в ближайшем будущем, можно позволить клиенту продолжать работу, даже если он временно отключен от файлового сервера. Для поддержки такого режима работы необходимо специальное управление кэшем, именуемое накоплением. Накопление нелегко реализовать, тем не менее было показано, что его эффективная реализация вполне возможна.

Защита очень важна для любых распределенных систем, включая файловые. Система NFS сама по себе фактически не предоставляет каких-либо механизмов защиты, но, обладая стандартизованным интерфейсом, позволяет использовать в связке с ней разнообразные системы защиты, например Kerberos. В противоположность ей Coda и Plan 9 обладают собственными механизмами защиты. Аутентификация в этих системах осуществляется при помощи защищенных вызовов RPC и часто представляет собой вариации на тему протокола аутентификации Нидхема—Шредера. SFS отличается от них тем, что информация об открытом ключе сервера включается в имена файлов. Такой подход упрощает управление ключами в крупномасштабных системах. На практике SFS распространяет ключ, включая его в имя файла. Специальные средства помогают делать это прозрачно для клиента.

Вопросы и задания

1. Обязательно ли, чтобы файловый сервер, на котором реализована система NFS версии 3, не хранил информацию о состоянии?

2. NFS не имеет глобального разделяемого пространства имен. Существует ли в этом случае способ имитации пространства имен?
3. Укажите простое расширение операции NFS `lookup`, которое позволяло бы производить итеративный поиск в ситуации с сервером, экспортирующим каталоги, монтируемые к другому серверу.
4. В UNIX-подобных операционных системах открытие файла с использованием дескриптора файла может производиться только на уровне ядра. Приведите возможную реализацию дескриптора файла NFS для сервера NFS пользовательского уровня в системе UNIX.
5. При использовании автоматического монтировщика, который устанавливает символические ссылки, как было описано в этой главе, сложнее обеспечить прозрачность монтирования. Почему?
6. Представьте, что текущее состояние запрета на доступ к файлу в NFS — *WRITE*. Возможна ли ситуация, когда другой клиент сначала успешно открывает файл, а затем запрашивает блокировку на чтение?
7. Если рассматривать согласованность кэшей, о которой мы говорили в главе 6, какой тип протокола согласованности кэшей реализован в NFS?
8. Реализует ли NFS поэлементную непротиворечивость? А свободную непротиворечивость?
9. Мы установили, что NFS реализует модель удаленного доступа к файлам. Можно также считать, что она реализует модель загрузки-выгрузки. Объясните, почему.
10. В NFS атрибуты кэшируются с использованием правил согласованности кэша сквозной записи. Так ли необходимо передавать все изменения атрибутов на сервер немедленно?
11. До каких пределов кэш дублированных запросов, описанный в этой главе, в действительности способен реализовать семантику «максимум однажды»?
12. Какие меры защиты могут быть задействованы в NFS, чтобы предотвратить восстановления никогда не назначавшейся блокировки, запрос на продление которой клиент-злоумышленник пытается послать в период действия амнистии?
13. На рис. 10.14 предполагается, что клиент имеет абсолютно прозрачный доступ к Vise. Насколько это истинно на самом деле?
14. Благодаря побочным эффектам механизм RPC2 можно использовать для обработки непрерывных потоков данных. Приведите другой пример, когда имело бы смысл задействовать специальный протокол, являющийся надстройкой над RPC.
15. Какие изменения в базе данных репликации томов и базе данных размещения томов Coda требуется сделать при перемещении физического тома с сервера A на сервер B?
16. Какую семантику вызовов использует RPC2 при наличии отказов?

17. Опишите, как Coda разрешает конфликты чтения-записи в файле, который совместно используется несколькими процессами чтения и одним записи.
18. Нужно ли шифровать свободный маркер, который Алиса получает от AS при входе в систему Coda? Если да, то где должно происходить шифрование?
19. Что требуется, чтобы файл на сервере Vice имел собственный список контроля доступа?
20. Имеет ли смысл для сервера Vice предоставлять в файле только разрешение *WRITE*? (Подсказка: подумайте, что произойдет, когда клиент откроет файл.)
21. Могут ли объединенные каталоги в Plan 9 заменить переменную окружения в UNIX-системах?
22. Может ли система Plan 9 успешно использоваться в качестве глобальной?
23. Каково основное преимущество групп нарезки в xFS по сравнению с распределением сегментов по всем серверам хранения?
24. Всегда ли при использовании самосертифицирующихся имен путей клиенту можно гарантировать, что он не имеет дело с подставным сервером?

Глава 11

Распределенные системы документов

11.1. World Wide Web

11.2. Lotus Notes

11.3. Сравнение WWW и Lotus Notes

11.4. Итоги

Одной из наиболее важных причин, способствовавших популярности как сетей, так и распределенных систем, стало появление Всемирной паутины (World Wide Web). Сила Web заключается в относительной простоте парадигмы: все вокруг — это документы. В этой главе мы рассмотрим распределенные системы документов, такие как Web. Системы документов предлагают пользователям представление о документах как простом и мощном средстве обмена информацией. Модель проста для понимания и часто близка тому, с чем пользователи уже сталкивались в своей повседневной деятельности. Так, например, в офисах связь часто осуществляется путем передачи записок, отчетов, заявлений и т. п.

Web в настоящее время является наиболее важной распределенной системой документов и, похоже, останется таковой в ближайшем будущем. Фактически, когда большинство людей говорят об Интернете, они имеют в виду Web. Итак, Web — первый пример системы, который мы обсудим в этой главе.

Другой важной системой документов, появившейся раньше Web, является Lotus Notes. В противоположность Web в основе системы Notes лежат скорее не файлы, а базы данных. В настоящее время система Lotus Notes остается достаточно популярной и часто интегрируется в технологию Web, предоставляя средства разработки web-служб. Lotus Notes будет вторым примером распределенной системы документов, которую мы обсудим в этой главе. Завершается глава разделом, в котором мы сравним эти две системы.

11.1. World Wide Web

World Wide Web (WWW) можно считать гигантской распределенной системой, для доступа к связанным документам содержащей миллионы клиентов и серверов.

ров. Серверы поддерживают наборы документов, а клиенты предоставляют пользователям простой интерфейс для доступа и просмотра этих документов.

Среда Web родилась в качестве проекта Европейской физической лаборатории элементарных частиц в Женеве и предназначалась для организации доступа большого количества географически удаленных групп исследователей к совместно используемым документам при помощи простой гипертекстовой системы. Документом при этом считалось все, что можно вывести на терминал компьютера пользователя, то есть заметки, отчеты, рисунки, чертежи и т. п. Создавая перекрестные ссылки между документами, можно было включать в новый документ документы из разных проектов без необходимости в централизованных изменениях. Все, что было необходимо для создания документа, — это организация ссылок на другие документы [43].

Среда Web постепенно росла благодаря появлению во всем мире сайтов, никак не связанных с физикой высоких энергий, но особенно заметно ее популярность выросла после появления удобных графических интерфейсов пользователя, в частности интерфейса программы Mosaic [474]. В программе Mosaic для получения документа было достаточно всего лишь одного щелчка мыши. Документ запрашивался на сервере, передавался клиенту и появлялся на экране. Для пользователя отсутствовали концептуальные различия между локальным документом и документом из другой части света. В этом смысле распределение было прозрачным.

С 1994 года относящиеся к Web разработки иницируются и направляются в первую очередь консорциумом World Wide Web, совместным органом CERN и M.I.T. Этот консорциум отвечает за стандартизацию протоколов, улучшения межоперационного взаимодействия и дальнейший рост возможностей Web. Его домашняя страница находится по адресу <http://www.w3.org/>.

11.1.1. WWW

WWW — это, в сущности, гигантская система с архитектурой клиент-сервер и миллионами серверов по всему миру. Каждый сервер поддерживает набор документов, каждый документ содержится в файле (хотя в некоторых случаях документы могут генерироваться по запросу). Сервер принимает запросы на выдачу документов и передает их клиенту. Кроме того, он может принимать запросы на сохранение новых документов.

Наиболее простой способ сослаться на документ — *унифицированный указатель ресурса* (Uniform Resource Locator, URL). URL-адрес подобен межоперационной ссылке на объект (IOR) в CORBA или контактному адресу в Globe. Он определяет, где находится целевой документ. Обычно это делается путем встраивания в исходный документ DNS-имени соответствующего сервера вместе с именем файла, с помощью которого сервер может найти целевой документ в своей локальной файловой системе. Кроме того, URL определяет протокол прикладного уровня, используемый для пересылки документа по сети. Как мы увидим, таких протоколов существует несколько.

Клиент взаимодействует с web-серверами при помощи специальной программы, называемой *браузером (browser)*. Браузер отвечает за правильное отображение документа. Кроме того, браузер обрабатывает операции пользовательского ввода, основная из которых — выбор ссылки на другой документ, который затем извлекается из хранилища и выводится на экран. Эта схема приводит нас к обобщенной структуре Web, представленной на рис. 11.1.

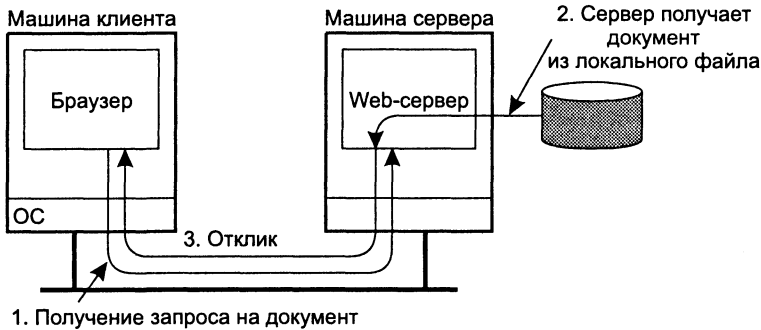


Рис. 11.1. Обобщенная структура Web

Среда Web значительно изменилась с момента своего создания около 10 лет назад. В настоящее время она представляет собой целую гамму методов и средств предоставления информации, которая может обрабатываться как web-клиентами, так и web-серверами. Далее мы детально рассмотрим, как работает web с точки зрения распределенной системы. Однако мы опустим большинство методов и средств, касающихся собственно создания web-документов, поскольку они обычно не имеют прямого отношения к распределенной природе Web. Хорошее и основательное введение в тему разработки приложений для Web можно найти в [123].

Документная модель

Web базируется на представлении всей информации в виде документов. Существует множество способов создания документов. Некоторые документы могут быть просто текстовыми ASCII-файлами, другие же представляют собой набор сценариев, которые автоматически выполняются при загрузке документа в браузер.

Однако наиболее важно для нас то, что документ может содержать ссылки на другие документы. Эти ссылки называются *гиперссылками (hyperlinks)*. Когда документ открывается в браузере, гиперссылки на другие документы выделяются особо, чтобы пользователь легко мог их заметить. Пользователь может выбрать ссылку, щелкнув на ней мышью. Выбор гиперссылки генерирует запрос на получение документа, пересылаемый на сервер, на котором хранится документ. Оттуда он возвращается на машину пользователя и открывается в браузере. Новый документ может быть показан в новом окне или заменить существующий документ в текущем окне браузера.

Большинство документов в Web описываются с помощью особого языка *разметки гипертекста (HyperText Markup Language, HTML)*. Слово «разметка» оз-

начает, что HTML содержит ключевые слова для разбивки текста документа на отдельные разделы. Так, например, любой HTML-документ имеет разделы заголовка и тела документа. Помимо заголовков HTML позволяет выделять списки, таблицы и формы. Также в определенные места документа можно вставить изображения и анимацию. Кроме этих элементов HTML содержит различные ключевые слова, позволяющие указать браузеру способ представления документа. Так, например, существуют ключевые слова для выбора определенного шрифта или размера шрифта, для представления текста жирным или курсивным начертанием, для выравнивания частей текста и т. п.

Сейчас HTML больше не является тем, чем он был раньше, — просто языком разметки. В настоящее время он включает в себя множество средств для создания красивых web-страниц. В частности, имеется возможность представления частей документа в форме сценариев. Рассмотрим простой пример HTML-документа (листинг 11.1).

Листинг 11.1. Простая web-страница, содержащая сценарий, написанный на языке JavaScript

<HTML>	<!-- Начало HTML-документа -->
<BODY>	<!-- Начало тела документа -->
<H1>Hello World</H1>	<!-- Основной текст -->
<P>	<!-- Начало нового абзаца -->
<SCRIPT type = "text/javascript">	<!-- Назначение языка сценариев -->
document.writeln("<H1>Hello World</H1>");	// Вывести текстовую строку
</SCRIPT>	<!-- Окончание секции сценария -->
</P>	<!-- Окончание секции абзаца -->
</BODY>	<!-- Окончание тела документа -->
</HTML>	<!-- Окончание секции HTML -->

Когда эта web-страница будет обработана и показана браузером, пользователь увидит текст «Hello World», повторенный дважды в отдельных строках. Первый вариант этого текста станет результатом интерпретации строки HTML:

```
<H1>HeIIo World</H1>
```

Второй вариант, в свою очередь, будет результатом работы небольшого сценария, написанного на Java-подобном языке сценариев *JavaScript*. Этот сценарий состоит из одной строки кода:

```
document.writeln("<H1>HeIIo World</H1>");
```

Хотя результат работы этих двух фрагментов HTML-кода абсолютно одинаков, ясно, что между ними имеется существенная разница. В первом случае мы имеем результат непосредственной интерпретации команд HTML, предназначенных для правильного показа размеченного текста. Второй вариант — это результат выполнения сценария, загруженного в браузер в виде части документа. Другими словами, мы видим перед собой один из вариантов мобильного кода.

Когда выполняется синтаксический разбор документа, он сохраняется внутри браузера в виде корневого дерева, которое называют деревом синтаксического разбора. В этом дереве каждый из узлов представляет собой элемент документа.

Чтобы добиться переносимости, представление дерева синтаксического разбора должно быть стандартизовано. Так, например, требуется, чтобы каждый из узлов содержал только элементы одного типа из предопределенного набора типов элементов. Точно так же каждый из узлов должен реализовывать стандартный интерфейс, содержащий методы доступа к его содержимому, возвращать ссылки на родительские и дочерние узлы и т. п. Это стандартное представление известно [259] как *объектная модель документа (Document Object Model, DOM)*. Его также часто называют *динамическим языком HTML (dynamic HTML)*.

Модель DOM предоставляет стандартный программный интерфейс для web-документов, прошедших синтаксический разбор. Этот интерфейс определен в языке IDL системы CORBA и его отображения на различные языки сценариев, такие как JavaScript, стандартизованы. Интерфейс используется для включения в документы сценариев обхода дерева синтаксического разбора, осмотра и модификации узлов, их добавления и удаления и т. п. Другими словами, с помощью сценария можно выполнять просмотр и изменение документа, частью которого он является. Понятно, что это открывает массу возможностей для динамической настройки документов.

Хотя большинство web-документов, как и раньше, пишутся на HTML, существует и другой согласованный с DOM язык [73] — *расширяемый язык разметки (eXtensible Markup Language, XML)*. В отличие от HTML XML обеспечивает только структурирование документа; он не содержит ключевых слов для форматирования документа, например выравнивания текста по центру или изображения его курсивом. Еще одно его важное отличие от HTML состоит в том, что XML можно использовать для определения произвольных структур. Другими словами, он предоставляет средства определения разных типов документов.

Определение типа документа требует, чтобы все элементы документа были сначала объявлены. Так, например, в листинге 11.2 показано определение на языке XML простой обобщенной ссылки на журнальную статью (номера строк не являются частью определения). Ссылка на статью объявляется в строке 1 как документ, состоящий из трех элементов: title (заголовок), author (автор) и journal (журнал). Знак плюс (+), следующий за элементом author, означает, что авторов может быть один или больше.

Листинг 11.2. Определения XML для ссылки на журнальную статью

```
(1) <!ELEMENT article (title, author+, journal)>
(2) <!ELEMENT title (#PCDATA)>
(3) <!ELEMENT author (name, affiliation?)>
(4) <!ELEMENT name (#PCDATA)>
(5) <!ELEMENT affiliation (#PCDATA)>
(6) <!ELEMENT journal(jname, volume, number?, month?, pages, year)>
(7) <!ELEMENT maine (#PCDATA)>
(8) <!ELEMENT volume (#PCDATA)>
(9) <!ELEMENT number (#PCDATA)>
(10) <!ELEMENT month (#PCDATA)>
(11) <!ELEMENT pages (#PCDATA)>
(12) <!ELEMENT year (#PCDATA)>
```

В строке 2 объявлен элемент `title`, состоящий из последовательности символов (соответствующих в XML простому типу данных `#PCDATA`). Элемент списка авторов подразделяется на два других элемента: `name` (имя) и `affiliation` (дополнение). Символ вопроса (?) показывает, что элемент `affiliation` не обязательный и не может присутствовать в ссылке на статью в количестве более чем один элемент на автора. Подобным же образом в строке 6 элемент `journal` также делится на меньшие элементы. Каждый элемент, который не подвергается дальнейшему разделению (то есть образует листовой узел в дереве синтаксического разбора), определяется как набор символов.

Реальная ссылка на статью в XML может быть теперь представлена в виде XML-документа, приведенного в листинге 11.3 (номера строк не входят в документ и здесь). Если предположить, что определения из листинга 11.3 хранятся в файле `article.dtd`, строка 2 говорит системе синтаксического разбора XML, где найти определение, соответствующее текущему документу. Строка 4 содержит заголовок статьи, а строки 5 и 6 содержат имена каждого из авторов. Отметим, что дополнения пропущены, в соответствии с определениями XML из листинга 11.2 это возможно.

Листинг 11.3. XML-документ, использующий определения XML из листинга 11.2

```
(1)      <?xml = version "1.0">
(2)      <!DOCTYPE article SYSTEM "article.dtd">
(3)      <article>
(4)      <title>Prudent Engineering Practice for Cryptographic Protocols</title>
(5)      <author><name>M. Abadi</name></author>
(6)      <author><name>R. Needham</name></author>
(7)      <journal>
(8)      <jname>IEEE Transactions on Software Engineering</jname>
(9)      <volume>22</volume>
(10)     <number>1</number>
(11)     <month>January</month>
(12)     <pages>6-15</pages>
(13)     <year>1996</year>
(14)     </journal>
(15)     </article>
```

Представление XML-документа пользователю требует задания правил форматирования. Самый простой подход — это встроить XML-документ в HTML-документ и использовать для форматирования ключевые слова HTML. С другой стороны, существует специальный язык форматирования, расширяемый язык стилей (*eXtensible Style Language, XSL*), и для определения формата вывода XML-документа можно использовать и его. Детальное описание того, как это сделать, можно найти в [123].

Кроме элементов, которые традиционно наличествуют в документе, таких как заголовки и абзацы, HTML и XML могут поддерживать также специальные элементы, связанные с мультимедиа. Так, например, можно не просто включать в документ изображения, но и присоединять к нему видеоклипы, аудиофайлы и даже интерактивную анимацию. Понятно, что языки сценариев играют важную роль в мультимедийных документах.

Типы документов

Помимо HTML- и XML-документов существует еще множество типов документов. Так, например, сценарий также можно рассматривать как документ. Другими примерами подобных документов могут быть документы в форматах PostScript или PDF, изображения в форматах JPEG или GIF, звуковые документы в формате MP3. Тип документа часто выражается в виде *muna MIME (MIME type)*. Спецификация MIME расшифровывается как *Multipurpose Internet Mail Extensions (многоцелевые расширения почты Интернета)* и изначально разрабатывалась для передачи информации в виде содержимого тел сообщений, составляющих часть электронной почты. В MIME различаются типы содержимого сообщений, которые описаны в [154]. Эти типы также используются и в WWW.

В MIME проводится грань между типами верхнего уровня и подтипами. Разные типы верхнего уровня перечислены в табл. 11.1 и включают текст, изображения, аудио и видео. Специальный тип Application указывает, что документ содержит данные, которые относятся к определенному приложению. На практике только это приложение способно преобразовать документ в нечто, воспринимаемое человеком.

Таблица 11.1. Шесть типов MIME верхнего уровня и некоторые распространенные подтипы

Тип	Подтип	Описание
Text	Plain	Неформатированный текст
	HTML	Текст, содержащий команды разметки HTML
	XML	Текст, содержащий команды разметки XML
Image	GIF	Изображение в формате GIF
	JPEG	Изображение в формате JPEG
Audio	Basic	Аудио, 8-бит PCM, 8000 Герц
	Tone	Заданный слышимый тон
Video	MPEG	Видео в формате MPEG
	Pointer	Представление указателя мыши для презентаций
Application	Octet-stream	Не интерпретируемая последовательность байтов
	Postscript	Печатный документ в формате PostScript
	PDF	Печатный документ в формате PDF
Multipart	Mixed	Независимые части в заданном порядке
	Parallel	Одновременно просматриваемые части

Тип Multipart используется для составных документов, то есть документов, состоящих из нескольких частей, причем каждая из частей может ассоциироваться с собственным типом верхнего уровня.

Для каждого из типов верхнего уровня может существовать несколько подтипов, некоторые из них также приведены в таблице. Тип документа, таким образом, определяется как комбинация типа верхнего уровня и подтипа, например application/PDF. В данном случае указывается, что для обработки документа,

представленного в виде PDF, необходимо специальное приложение. Обычно web-страницы имеют тип text/HTML, который отражает тот факт, что они представляют собой ASCII-текст, содержащий ключевые слова HTML.

Обзор архитектуры

Комбинация HTML или XML со сценариями образует мощное средство отображения документов. Однако вряд ли имеет смысл выяснять, где именно обрабатываются документы и каким образом происходит их обработка. Среда WWW начиналась с относительно простых систем с архитектурой клиент-сервер (см. рис. 11.1). В настоящее время эта простая архитектура была расширена за счет множества компонентов, поддерживающих усложненные типы документов, которые мы только что описали.

Одним из первых дополнений базовой архитектуры стала поддержка несложного взаимодействия с пользователями при помощи *обобщенного интерфейса шлюзов (Common Gateway Interface, CGI)*. CGI определяет стандартный способ выполнения web-сервером программ с данными пользователя в качестве входных параметров. Обычно данные пользователя вводятся в HTML-форму; она определяет программу, которая должна выполняться на стороне сервера, и значения параметров, введенные пользователем. После того как заполнение формы закончено, имя программы и собранные значения параметров пересылаются на сервер, как это показано на рис. 11.2.

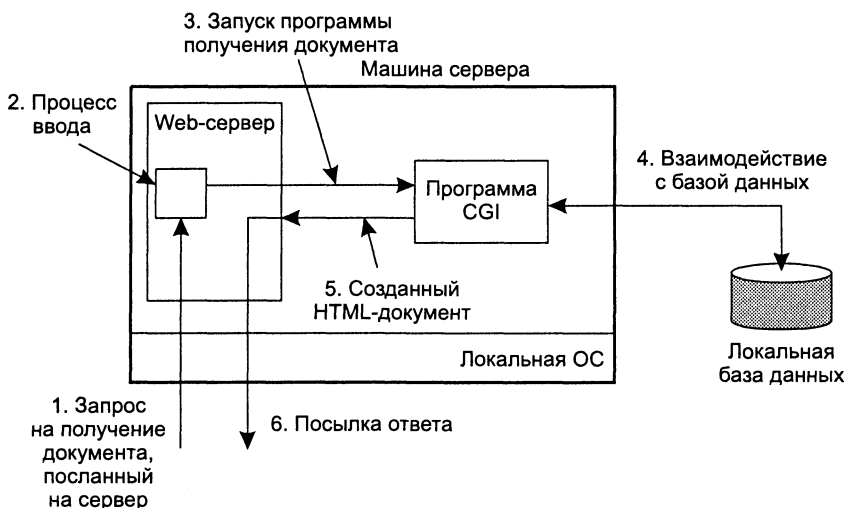


Рис. 11.2. Принцип использования программ CGI на стороне сервера

Когда сервер обнаруживает запрос, он запускает указанную в запросе программу и передает ей присланные в запросе параметры. После этого программа делает свою работу и обычно возвращает результаты в виде документа, который отправляется назад браузеру клиента, чтобы клиент мог его увидеть.

Программы CGI могут быть любой сложности. Так, например, как показано на рисунке, многие программы работают с размещенными на web-сервере базами данных. После обработки данных программа создает HTML-документ и возвращает этот документ серверу. Сервер затем пересылает документ клиенту. Интересно, что сервер как бы запрашивает документ у программы CGI. Другими словами, сервер просто делегирует обязанности по извлечению документа внешней программе.

Основная задача сервера, используемого для обработки запросов клиента, состоит в извлечении документа. При наличии программ CGI извлечение документа может быть делегировано этим программам, так что сервер остается в неведении о том, был ли документ создан «на лету» или его действительно получили из локальной файловой системы. Тем не менее серверы в наше время тоже делают значительно больше, чем просто извлекают документы.

Одно из наиболее важных усовершенствований состоит в том, что серверы могут обрабатывать извлеченные документы перед их отправкой клиенту. В частности, документ может содержать *серверный сценарий* (*server-side script*), который исполняется сервером при локальном извлечении документа. Результаты исполнения сценария вместе с другими частями документа пересылаются клиенту. Однако сам сценарий не пересылается. Другими словами, серверный сценарий изменяет документ, заменяя сценарий результатом его исполнения.

Для знакомства с примером подобного сценария рассмотрим HTML-документ из листинга 11.4 (номера строк не являются частью документа). Когда клиент запрашивает этот документ, сервер выполняет сценарий, описанный в строках 5–12. Сервер создает локальный файловый объект *clientFile* и использует его для открытия файла */data/file.txt*, из которого он читает данные (строки 6–7). До тех пор пока в файле имеются данные, сервер пересылает эти данные клиенту в виде части HTML-документа (строки 8–9).

Листинг 11.4. HTML-документ, содержащий сценарий JavaScript, исполняемый на сервере

```
(1)      <HTML>
(2)      <BODY>
(3)      The current content of <PRE>/data/file.txt</PRE> is:</P>
(4)
(5)      <SERVER type = "text/javascript">
(6)          clientFile = new File("/data/file.txt")
(7)          if(clientFile.open("r")){
(8)              while(!clientFile.eof())
(9)                  document.writeln(clientFile.readLine());
(10)         clientFile.close();
(11)     }
(12)     </SERVER>
(13)     </P>
(14)     Thank you for visiting this site.</P>
(15)     </BODY>
(16)     </HTML>
```

Сценарий, исполняемый на сервере, опознается при помощи специального нестандартного тега HTML `<SERVER>`. Серверы разных производителей используют

собственные способы распознавания исполняемых на сервере сценариев. В этом примере при запуске сценария исходный документ изменяется так, что серверный сценарий заменяется текущим содержимым файла `/data/file.txt`. Другими словами, клиент никогда не видит текста сценария. Если документ содержит сценарии, выполняемые на стороне клиента, они пересылаются клиенту обычным образом.

Кроме выполнения клиентских и серверных сценариев, клиенту можно также передавать заранее скомпилированные программы в виде апплетов. Обычно *applet* (*applet*) представляет собой небольшое автономное приложение, которое можно отослать клиенту и выполнить в пространстве адресов браузера. Наиболее часто используются апплеты в виде программ на языке Java, скомпилированных в интерпретируемый код Java. Так, Java-апплет можно включить в HTML-документ при помощи следующей строки:

```
<OBJECT codetype = "application/java" classid = "java:welcome.class">
```

Апплеты выполняются на стороне клиента. Существуют также и серверные варианты апплетов, которые называются *сервелетами* (*servlets*). Как и апплеты, сервелеты — это заранее скомпилированные программы, которые выполняются в адресном пространстве сервера. В современной практике сервелеты в основном пишутся на Java, но никаких фундаментальных ограничений на использование других языков нет. Сервелет реализует методы, которые также имеются и в HTTP — стандартном коммуникационном протоколе между клиентом и сервером. В подробностях мы рассмотрим HTTP чуть ниже.

Когда сервер получает HTML-запрос, адресованный сервелету, он вызывает метод сервелета, соответствующий этому запросу. Последний, в свою очередь, обрабатывает запрос и обычно возвращает результаты в виде HTML-документа. Основное отличие сервелетов от сценариев CGI состоит в том, что последние представляют собой отдельные процессы, в то время как сервелеты выполняются сервером.

Теперь мы имеем более полное представление об архитектуре и структуре клиентов и серверов в Web (рис. 11.3). Когда пользователь посылает запрос на получение документа, web-сервер обычно, в зависимости от предлагаемых в документе задач, делает одно из трех. Во-первых, он может извлечь документ прямо из локальной файловой системы. Во-вторых, он может запустить программу CGI, которая генерирует документ, возможно, используя данные из локальной базы данных. В-третьих, он может передать запрос сервелету.

Когда документ будет получен сервером, ему может потребоваться дополнительная обработка, в ходе которой будут выполнены содержащиеся в нем сценарии, исполняемые на сервере. На практике это требуется только для документов, напрямую извлекаемых из локальной файловой системы, то есть документов, полученных без помощи сервелетов или программ CGI. Затем документ пересылается браузеру пользователя.

После того как документ будет получен клиентом, браузер выполняет клиентские сценарии и, возможно, извлекает и выполняет содержащиеся в документе апплеты. Результаты обработки документа в виде его содержимого выводятся на терминал пользователя.

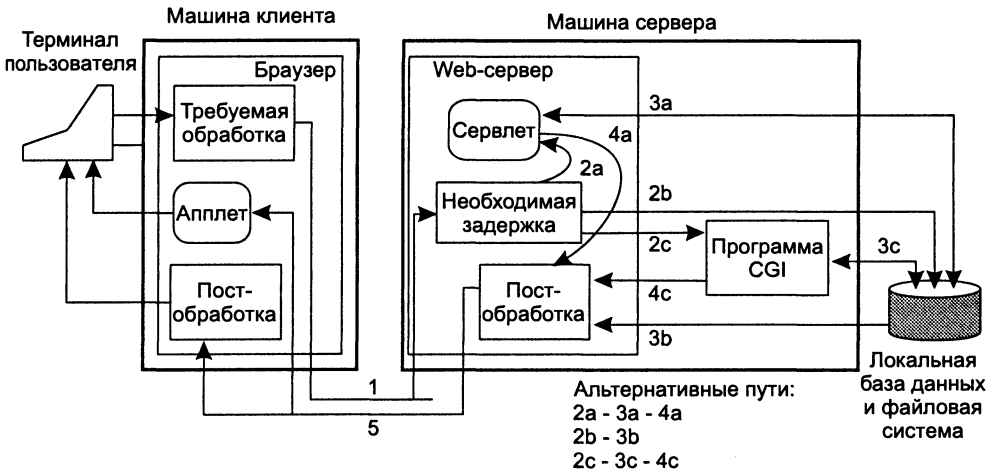


Рис. 11.3. Подробности архитектуры клиента и сервера в Web

В описанной архитектуре мы предполагали, что браузер может показывать HTML-документы и, возможно, XML-документы. Однако существует множество документов в других форматах, таких как PostScript или PDF. В этих случаях браузер просто просит сервер без всякой обработки извлечь этот документ из файловой системы и прислать его. В результате такого подхода клиент получает файл. Используя расширение файла, браузер запускает соответствующее ему приложение, которое позволяет продемонстрировать или обработать его содержимое. Поскольку подобные приложения помогают браузеру выполнять его работу — показывать документы, они называются также *приложениями-помощниками* (*helper applications*). Ниже мы рассмотрим, чем можно заменить такие приложения.

11.1.2. Связь

Любое взаимодействие между клиентом и сервером в Web происходит по *протоколу передачи гипертекста* (*Hypertext Transfer Protocol, HTTP*). HTTP — это относительно простой протокол, ориентированный на архитектуру клиент-сервер. Клиент в нем посылает запросы на сервер и ожидает получения ответа. Следует отметить важное свойство HTTP — этот протокол не фиксирует состояния. Другими словами, у него нет никаких данных об открытом соединении и он не требует, чтобы сервер поддерживал какую бы то ни было информацию о своих клиентах. Последняя версия HTTP описана в [142].

Соединения HTTP

В основе HTTP лежит протокол TCP. Когда клиент посылает серверу запрос, он устанавливает соединение TCP с сервером и посылает через это соединение свой запрос. То же самое соединение задействуется и для получения ответа. Исполь-

зуя TCP в качестве базового протокола, HTTP может не заботиться о потере запросов или ответов. Клиент и сервер могут просто допустить, что их сообщения доходят до противоположной стороны соединения. Если дела идут плохо, например соединение разорвано или истек тайм-аут, генерируется сообщение об ошибке. Однако обычно попытки восстановления после сбоя не предпринимаются.

Одной из проблем первой версии HTTP было неэффективное использование соединений TCP. Каждый web-документ собирается из коллекции разных файлов, хранящихся на одном сервере. Чтобы правильно отобразить документ, все эти файлы также требуется переслать клиенту. Каждый из них в принципе представляет собой отдельный документ, для получения которого клиент должен отправить отдельный запрос на сервер.

В HTTP версии 1.0 и предыдущих версий каждый запрос к серверу требовал отдельного соединения (рис. 11.4, а). После ответа сервера соединение разрывалось. Такие соединения называются *несохранными* (*nonpersistent*). Основной недостаток несохранных соединений состоит в существенных издержках на создание соединений TCP. В результате время, необходимое на пересылку документа со всеми его элементами клиенту, значительно возрастает.

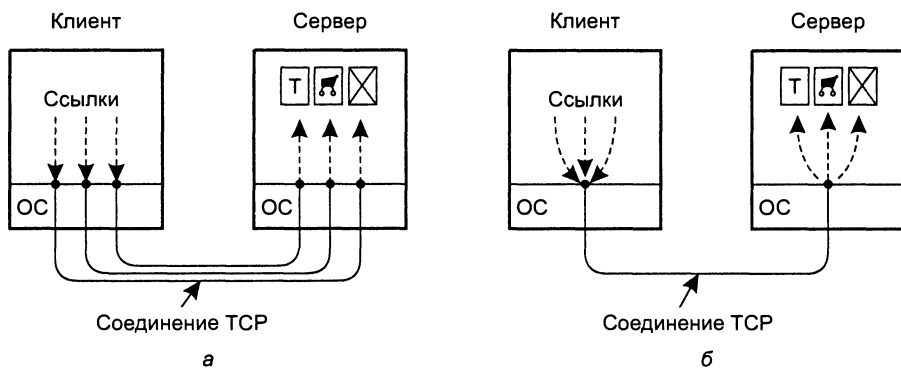


Рис. 11.4. Использование несохранных (а) и сохраняющих (б) соединений

Отметим, что HTTP не запрещает клиенту одновременно установить несколько соединений с одним и тем же сервером. Такой подход нередко используется для сокрытия задержек, связанных с временем установки соединения, а также для параллельной передачи данных от сервера клиенту.

Более эффективный подход, характерный для HTTP версии 1.1, — использование *сохранных соединений* (*persistent connections*), которые позволяют передать несколько запросов (и соответствующих им ответов), без установки отдельных соединений для каждой пары (*запрос, ответ*). Для еще большего повышения производительности клиент может посылать несколько запросов одной командой, не дожидаясь получения ответа на первый запрос. Такая отправка известна под названием *конвейеризации* (*pipelining*). Использование сохраняющих соединений продемонстрировано на рис. 11.4, б.

Методы HTTP

HTTP разрабатывался как протокол общего назначения для архитектуры клиент-сервер, ориентированный на передачу документов в обоих направлениях. Клиент может запрашивать выполнение любой из операций, перечисленных в табл. 11.2, просто послав на сервер запрос с нужной операцией (в таблице перечислены наиболее часто используемые операции).

Таблица 11.2. Операции, поддерживаемые протоколом HTTP

Операция	Описание
head	Запрос на возвращение заголовка документа
get	Запрос на возвращение клиенту документа
put	Запрос на сохранение документа
post	Данные, добавляемые к документу (набору)
delete	Запрос на удаление документа

HTTP предполагает, что каждый документ может иметь ассоциированные с ним метаданные, сохраняемые в отдельном заголовке, который может пересылаться вместе с запросом или ответом. Операция `head` посылается на сервер в том случае, если клиент не хочет получать документ, а нуждается только в метаданных документа (например, так можно узнать время последней модификации документа). Эта операция может быть использована для проверки правильности документа, кэшированного на стороне клиента. Кроме того, ее можно применить для проверки наличия документа без его передачи.

Наиболее важная операция — `get`. Она используется для извлечения документа с сервера и передачи его запросившему документ клиенту. Можно указать, что документ должен быть возвращен только в том случае, если он был изменен в определенный промежуток времени. Кроме того, HTTP позволяет документам иметь ассоциированные с ними теги, представляющие собой строки символов, и извлекать документы, только если они содержат соответствующие теги.

Операция `put` противоположна операции `get`. Клиент может потребовать от сервера сохранить документ под определенным именем (которое посылается вместе с запросом). Разумеется, сервер не будет слепо выполнять операцию `put`, он примет подобный запрос только от авторизованного клиента. О том, как работают механизмы защиты, мы поговорим позже.

Операция `post` чем-то похожа на сохранение документов за исключением того, что клиент может потребовать, чтобы данные были добавлены к документу или набору документов. Типичный пример — посылка заметки в группу новостей. По сравнению с операцией `put` особенность состоит в том, что в операции `post` указывается, в какую группу документов должна быть добавлена эта статья. Статья посылается вместе с запросом. В противоположность этому операция `put` пересылает документ и имя, под которым сервер должен его сохранить.

И, наконец, операция `delete` используется для того, чтобы сервер удалил документ с именем, которое содержится в сообщении, посланном серверу. Будет произведено удаление или нет, зависит от установленных правил защиты. Воз-

можно ситуация, когда даже сам сервер не имеет права на удаление указанного документа. В конце концов, сервер — это всего лишь пользовательский процесс.

Сообщения HTTP

Всякая связь между клиентом и сервером осуществляется посредством сообщений. HTTP различает только два вида сообщений — сообщения-запросы и сообщения-ответы. Как показано на рис. 11.5, а, сообщения-запросы состоят из трех частей. *Строка запроса (request line)* имеет стандартный вид и определяет операцию, которую клиент хочет выполнить на сервере, а также ссылку на документ, связанный с запросом. Отдельное поле требуется для указания версии HTTP, которую использует клиент. Ниже мы рассмотрим дополнительные заголовки сообщений.

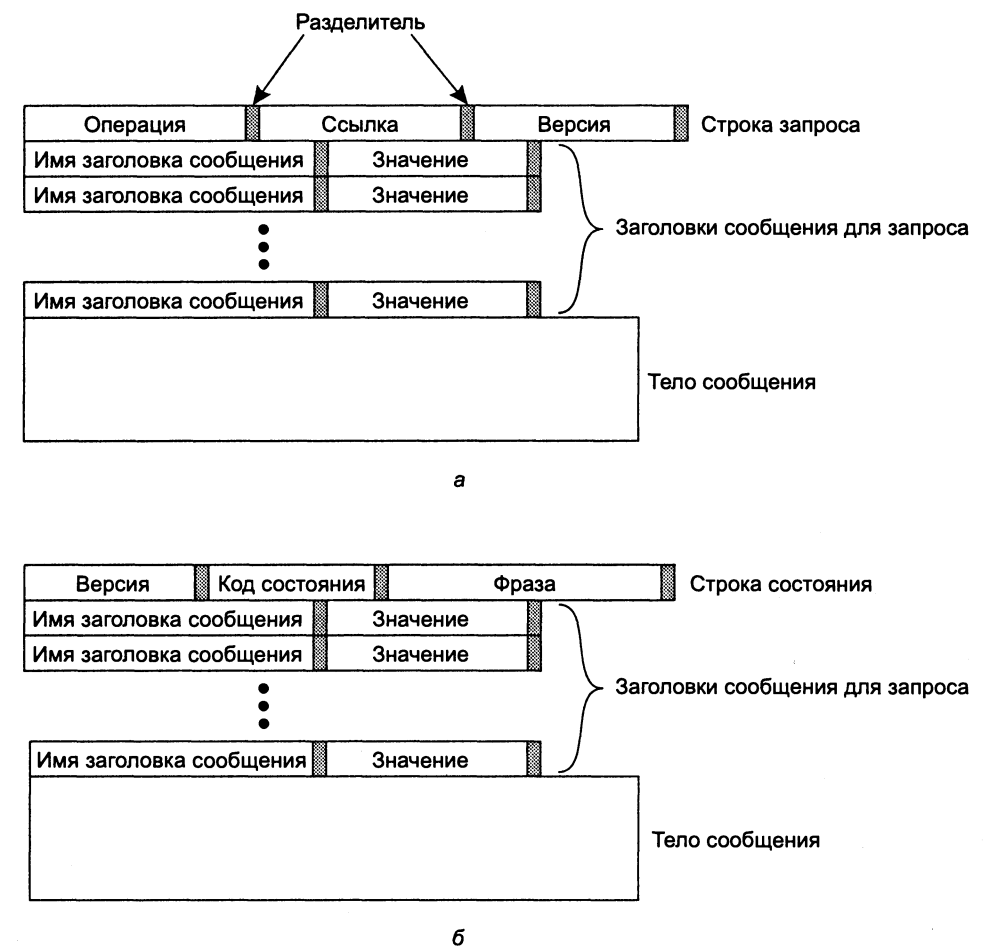


Рис. 11.5. Сообщение-запрос (а). Сообщение-ответ (б)

Ответное сообщение начинается *строкой состояния (status line)*, содержащей номер версии и код состояния из трех цифр, как показано на рис. 11.5, б. Код кратко поясняется текстовой фразой, которая также является частью строки состояния. Так, например, код состояния 200 указывает на то, что запрос был обработан и имеет ассоциированную с ним фразу «ОК». Вот другие часто используемые коды:

400 (Bad Request)
 403 (Forbidden)
 404 (Not Found)

Сообщения с запросами или ответами могут содержать дополнительные заголовки. Так, например, если клиент отправляет запрос на операцию `post` с документом, предназначенным только для чтения, сервер отвечает сообщением с кодом состояния 405 (Method Not Allowed) и заголовком сообщения `Allow`, определяющим допустимые операции (например, `head` и `get`). Другой пример: клиент может поинтересоваться, не изменялся ли документ с определенного момента времени T . В этом случае клиент отправляет запрос `get`, дополненный заголовком сообщения `If-Modified-Since` с определенным значением T .

В табл. 11.3 приведены некоторые корректные заголовки сообщений, которые могут быть посланы в запросах или ответах. Большинство заголовков говорят сами за себя, так что обсуждать их мы не будем.

Таблица 11.3. Некоторые заголовки сообщений HTTP

Заголовок	Источник	Содержимое
Accept	Клиент	Тип документов, которые клиент в состоянии обрабатывать
Accept-Charset	Клиент	Наборы символов, которые клиент может воспринимать
Accept-Encoding	Клиент	Кодировки документов, которые клиент в состоянии обрабатывать
Accept-Language	Клиент	Естественный язык, который понимает клиент
Authorization	Клиент	Список атрибутов авторизации клиента
WWW-Authenticate	Сервер	Требование системы защиты, на которое клиент должен отреагировать
Date	Оба	Дата и время отправки сообщения
ETag	Сервер	Теги, ассоциированные с возвращаемым документом
Expires	Сервер	Время, в течение которого полученная информация будет соответствовать действительности
From	Клиент	Электронный адрес клиента
Host	Клиент	TCP-адрес сервера документа
If-Match	Клиент	Теги, которые следует иметь документу
If-None-Match	Клиент	Теги, которые не следует иметь документу

продолжение ➤

Таблица 11.3 (продолжение)

Заголовок	Источник	Содержимое
If-Modified-Since	Клиент	Возвратить документ, только если за время, прошедшее с определенного момента, он был изменен
If-Unmodified-Since	Клиент	Возвратить документ, только если за время, прошедшее с определенного момента, он не был изменен
Last-Modified	Сервер	Время последнего изменения возвращаемого документа
Location	Сервер	Ссылка на документ, на которую клиент должен перенаправить свой запрос
Referer *	Клиент	Ссылка на наиболее часто запрашиваемый документ клиента
Upgrade	Оба	Прикладной протокол, на который хочет переключиться отправитель
Warning	Оба	Информация о состоянии данных, передаваемых в сообщении

Существует множество заголовков, которые клиент может послать серверу, разъясняя, в каком виде ему можно послать ответ. Так, например, клиент может иметь средства для приема ответов, сжатых при помощи архиватора *gzip*, доступного на большинстве машин под управлением *Windows* и *UNIX*. В этом случае клиент может послать вместе со своим сообщением заголовок сообщения *Accept-Encoding*, содержащий строку *Accept-Encoding: gzip*. Точно так же заголовок сообщения *Accept* можно использовать для определения того, что следует возвращать только *web-страницы* на языке *HTML*.

Имеется два заголовка сообщений, относящихся к средствам защиты, но, как мы увидим чуть позже, защита в *Web* обычно обеспечивается при помощи специальных протоколов транспортного уровня.

Заголовки сообщений *Location* и *Referer* используются для перенаправления клиента к другому документу (отметим, что слово «*Referer*» в спецификации записано с орфографической ошибкой). Перенаправление соответствует применению для локализации документов механизма передачи указателей, который мы описывали в главе 4. Когда клиент посылает запрос на документ *D*, сервер может ответить ему заголовком сообщения *Location*, определяющим, что клиент должен перенаправить свой запрос в другое место, к документу *D'*. При использовании ссылки на *D'* клиент может добавить заголовок сообщения *Referer* со ссылкой на *D*, чтобы показать, чем было вызвано перенаправление. Обычно этот заголовок сообщения идентифицирует последний запрошенный клиентом документ.

Заголовок сообщения *Upgrade* позволяет перейти на другой протокол. Так, например, клиент и сервер могут вначале использовать протокол *HTTP/1.1* — только в качестве общедоступного метода организации соединения. Сразу после этого сервер может сообщить клиенту, что он предлагает продолжать работу по безопасной версии *HTTP* — протоколу *SHTTP* [378]. Для этого сервер должен послать клиенту заголовок сообщения *Upgrade* с содержимым *Upgrade: SHTTP*.

11.1.3. Процессы

В сущности, в Web используются только два типа процессов — браузеры, при помощи которых пользователь получает доступ к документам Web и может отображать их на своем мониторе, и web-серверы, которые обрабатывают запросы браузеров. Браузеры могут, как говорилось выше, использовать приложения-помощники. Также и серверы могут быть окружены дополнительными программами, например сценариями CGI. Далее мы рассмотрим типичные для Web программы клиента и сервера.

Клиенты

Наиболее важные клиенты Web — это программы, называемые *web-браузерами* (*Web browser*), которые позволяют пользователям путешествовать по web-страницам, извлекая эти страницы с серверов и выводя их на экран пользователя. В интерфейсе браузера гиперссылки обычно выглядят так, что пользователь может легко выделять их одиночным щелчком мыши.

По сути, web-браузеры достаточно просты. Однако они должны быть в состоянии обрабатывать документы различных типов, а также предоставлять пользователям простой и удобный интерфейс. Это превращает их в сложные программные комплексы.

Одна из проблем, с которой сталкиваются разработчики web-браузеров, состоит в том, что функциональность браузеров должна легко расширяться, чтобы они могли в принципе поддерживать любые приходящие к ним от сервера типы документов. В большинстве случаев для этого используются так называемые модули расширения. *Модули расширения (plug-ins)* — это небольшие программы, которые могут динамически подгружаться браузером для обработки документа определенного типа. Документ обычно задается в виде типа MIME. Модуль расширения должен быть доступен локально, возможно, после загрузки пользователем с удаленного сервера.

Модули расширения предоставляют браузеру стандартный интерфейс и сами, в свою очередь, пользуются стандартным интерфейсом браузера, как показано на рис. 11.6. Когда браузер обнаруживает документ особого типа, для которого необходим модуль расширения, он локально загружает модуль и создает его экземпляр. Хотя модуль расширения использует только методы стандартизованных интерфейсов, общий вид браузера после инициализации определяется модулем расширения. После того как надобность в них отпадает, модули расширения удаляются из браузера.

Другой часто используемый клиентский процесс [275] носит название *web-заместитель (Web proxy)*. Изначально, как показано на рис. 11.7, этот процесс требовался для того, чтобы браузер мог работать с другими прикладными протоколами, кроме HTTP. Так, например, для получения файлов с FTP-сервера браузер мог послать HTML-запрос к своему локальному заместителю по FTP, который получал файл и возвращал его упакованным в ответное сообщение HTTP.

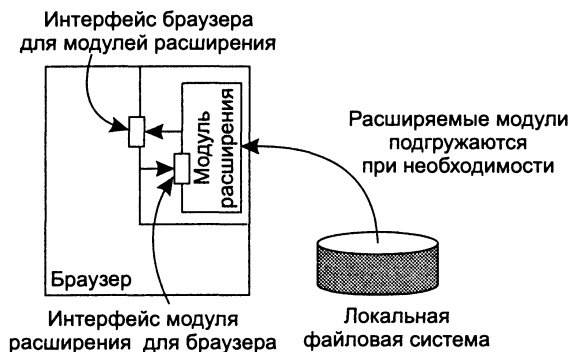


Рис. 11.6. Использование модуля расширения в web-браузере

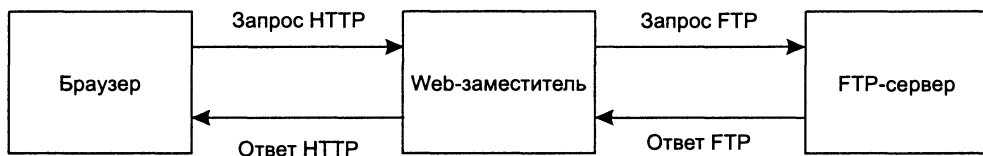


Рис. 11.7. Использование web-заместителя, если браузер не понимает протокола FTP

В настоящее время большинство web-браузеров в состоянии поддерживать множество протоколов, и по этой причине необходимости в заместителях вроде бы нет. Однако web-заместители остаются популярными, но используются теперь для абсолютно других целей, а именно для работы с локальным кэшем. Как мы увидим ниже, при запросе web-документа браузер передает этот запрос локальному web-заместителю. До того как связываться с удаленным сервером, где должен находиться запрашиваемый документ, заместитель проверяет, не содержится ли он в локальном кэше.

Серверы

Как мы говорили, web-сервер — это программа, которая обрабатывает входящие HTTP-запросы, извлекая запрашиваемые документы и возвращая их клиенту. В качестве конкретного примера мы кратко рассмотрим общую структуру *сервера Apache*, основного web-сервера на платформах UNIX.

Общая структура web-сервера Apache показано на рис. 11.8. Сервер состоит из множества модулей, которые управляются одним модулем ядра. Модуль ядра принимает входящие HTTP-запросы и поочередно передает их другим модулям. Другими словами, модуль ядра определяет поток управления для обработки запроса.

Для каждого приходящего запроса модуль ядра образует запись запроса с полями для ссылки на документ, содержащейся в HTTP-запросе, для связанных с HTTP-запросом заголовков, для заголовков ответа HTTP и т. д. Любой модуль работает с этой записью, считывая и изменяя соответствующие поля. В конце концов, когда все модули закончат обработку своих частей запроса, последний

из них вернет запрошенный документ клиенту. Отметим, что в принципе каждый запрос может идти по своему собственному маршруту.

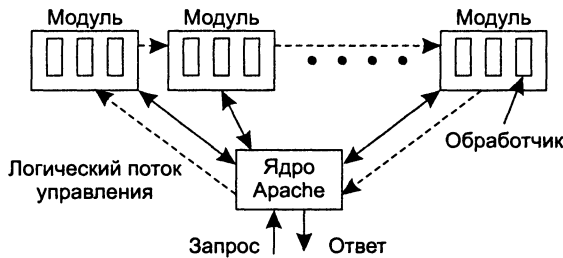


Рис. 11.8. Общая структура web-сервера Apache

Серверы Apache легко конфигурируются; чтобы содействовать обработке входящих HTTP-запросов, к ним могут подключаться самые разные модули. Для этого каждый модуль предоставляет один или несколько *обработчиков (handlers)*, которые вызываются из головного модуля. Обработчики похожи друг на друга тем, что все они получают в качестве единственного параметра указатель на запись запроса. Они также похожи тем, что одинаково могут считывать и изменять поля этой записи.

Для того чтобы в нужное время вызывался необходимый обработчик, обработка HTTP-запроса на некоторых фазах приостанавливается. Модуль может зарегистрировать для определенной фазы обработки конкретный обработчик. Когда дело доходит до некоторой фазы, модуль ядра просматривает, какие обработчики зарегистрированы для этой фазы, и вызывает один из них. Фазы перечислены ниже.

1. Разрешение ссылки на документ в локальное имя файла.
2. Аутентификация клиента.
3. Контроль доступа клиента.
4. Контроль доступа запроса.
5. Определение типа MIME ответа.
6. Общая фаза обработки.
7. Пересылка ответа.
8. Протоколирование данных по обработке запроса.

Ссылки предоставляются в виде *унифицированного идентификатора ресурса (Uniform Resource Identifier, URI)*, о котором мы поговорим ниже. Существует несколько способов разрешения URI. Во многих случаях частью URI является имя локального файла данного документа. Однако когда документ генерируется, например, при помощи программы CGI, URI будет содержать информацию о программе, которую сервер должен запустить, вместе со значениями входных параметров этой программы.

Вторая фаза — это аутентификация клиента. В принципе она подразумевает только проверку подлинности клиента, никаких попыток проверить права доступа не производится.

При контроле доступа клиента сервер проверяет, известен ли на самом деле клиент серверу и имеет ли он какие-либо права доступа. Так, в ходе этой фазы мы можем определить, принадлежит ли этот клиент к определенной группе пользователей.

В ходе контроля доступа запроса проверяется, имеет ли клиент право на работу с документом, указанным в запросе.

На следующей фазе производится определение типа MIME документа. Определить тип документа можно разными способами, например по расширениям файлов или с помощью специального файла, содержащего тип MIME документа.

Любая дополнительная обработка запроса, которая не выполнена в других фазах, выполняется в общей фазе обработки. В качестве примера задач, обычно решаемых в ходе этой фазы, можно выделить проверку синтаксиса возвращаемых ссылок или построение профиля пользователя.

Наконец, ответ пересылается пользователю. И снова сделать это можно разными способами, в зависимости от того, что было запрошено и как был создан ответ.

Последняя фаза включает в себя протоколирование всех действий, совершенных в ходе обработки запроса. Полученные журналы можно впоследствии использовать в самых разных целях.

Для выполнения этих фаз модуль ядра поддерживает список зарегистрированных обработчиков каждой фазы. Он выбирает обработчик и вызывает его. Обработчик может отклонить запрос, обработать его или сообщить об ошибке. Если обработчик отклоняет запрос, это означает, что он не в состоянии его обработать и модуль ядра должен выбрать другой обработчик, зарегистрированный для выполнения этой фазы. Если запрос может быть обработан, он обычно обрабатывается и начинается следующая фаза обработки. Если обработчиком возвращается сообщение об ошибке, обработка запроса прекращается и клиенту возвращается сообщение об ошибке.

Какие модули войдут в состав сервера Apache на самом деле, определяется во время конфигурирования. Самый простой сценарий предполагает, что модуль ядра выполняет всю обработку запросов, и в этом случае сервер «сжимается» до уровня простого web-сервера, который способен обрабатывать только HTML-документы. Для одновременного выполнения нескольких запросов модуль ядра обычно запускает для каждого из входящих сообщений новый процесс. Максимальное число процессов также задается параметром конфигурации. Детали конфигурирования и программирования сервера Apache можно найти в [257].

Кластеры серверов

Структура Web, имеющая архитектуру клиент-сервер, порождает серьезную проблему, состоящую в том, что web-сервер легко можно перегрузить. Практическое решение этой проблемы, различным образом реализуемое, состоит в том, чтобы реплицировать сервер и получить вместо одного сервера кластер рабочих станций, а для перенаправления запросов клиентов к одной из реплик использовать внешний интерфейс [150]. Этот принцип иллюстрирует рис. 11.9 и представляет собой пример горизонтального распределения, о котором мы говорили в главе 1.

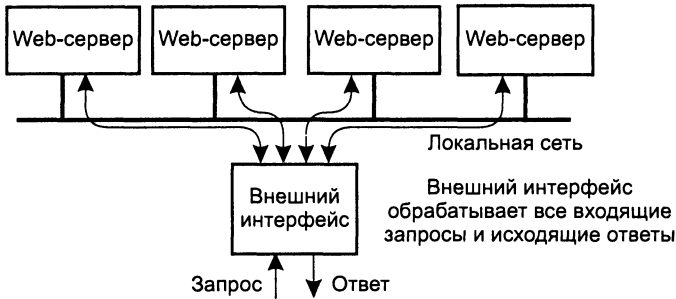


Рис. 11.9. Принципы использования кластера рабочих станций для реализации web-службы

Наиболее важная деталь подобной структуры — это организация внешнего интерфейса, который может стать опасным узким местом по производительности. Обычно вводится подразделение на внешние интерфейсы, играющие роль переключателей транспортного уровня, и внешние интерфейсы уровня приложений.

Как мы уже говорили, когда клиент посылает HTML-запрос, он устанавливает с сервером соединение ТСП. Переключатель транспортного уровня просто пересылает данные через соединение ТСП одному из серверов в зависимости от измеренной загрузки серверов. Основной недостаток такого подхода состоит в том, что переключатель не знает содержимого HTTP-запроса, который он передает через соединение ТСП. Он может только на основании данных о загрузке серверов выбрать очередную точку пересылки.

Обычно наилучшим подходом оказывается *распределение запросов с известным содержимым* (*content-aware request distribution*), при котором внешний интерфейс сначала просматривает пришедший HTML-запрос, а затем решает, на какой сервер следует переслать этот запрос. Эту схему можно использовать в сочетании с распределением содержимого в кластере серверов, как описано в [499].

Распределение запросов с известным содержимым имеет несколько достоинств. Так, например, если внешний интерфейс всегда пересылает запросы на один и тот же документ одному и тому же серверу, сервер может кэшировать этот документ, результатом чего станет резкое сокращение времени отклика. Кроме того, фактически можно разделить набор документов между серверами, а не реплицировать каждый документ на каждом из серверов. Такой подход позволяет более эффективно использовать имеющиеся емкости накопителей и выделять специальные серверы для обработки определенных видов документов, таких как аудио- или видеозаписи.

Проблема распределения запросов с известным содержимым состоит в том, что внешнему интерфейсу необходимо выполнять значительный объем работы. Для повышения производительности в [339] был предложен механизм, посредством которого соединение ТСП переключается непосредственно на сервер. В результате сервер непосредственно отвечает клиенту, минуя внешний интерфейс, как показано на рис. 11.10, а. Переключение соединения ТСП полностью про-

значно для клиента; клиент посылает свои сообщения по TCP внешнему интерфейсу (включая подтверждения и пр.), но все ответы получает напрямую от сервера, на который переключится соединение.

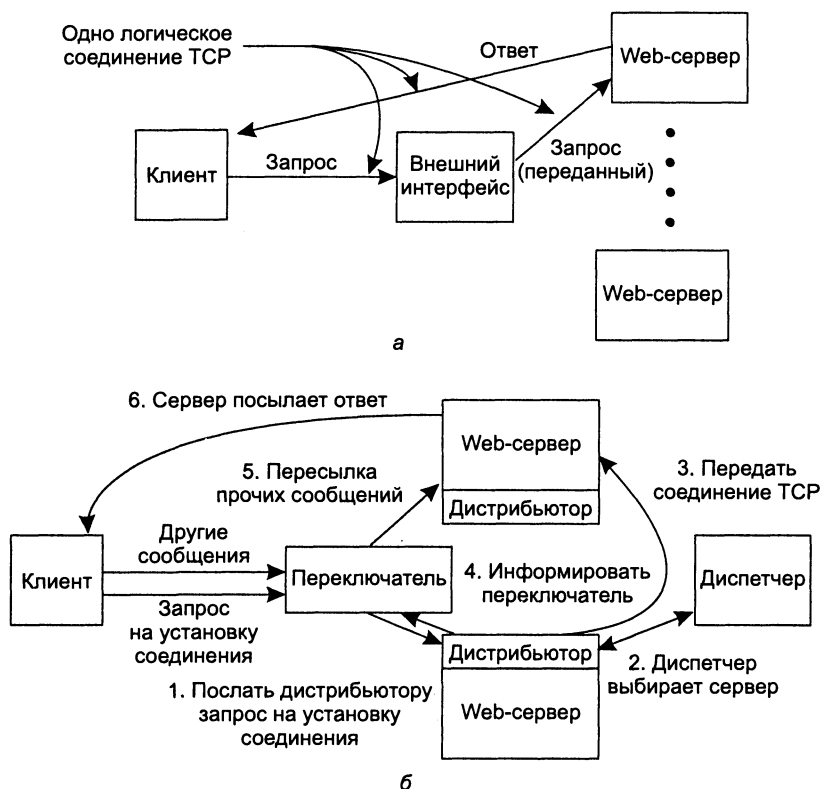


Рис. 11.10. Переключение соединения TCP (а). Масштабируемый кластер web-серверов с известным содержимым (б)

Дальнейшего улучшения можно добиться, распределяя работу внешнего интерфейса и переключателя транспортного уровня так, как излагается в [18]. В варианте с переключением соединения TCP внешний интерфейс решает две задачи. Во-первых, когда запрос только поступает, он должен решить, какой из серверов будет осуществлять дальнейшее взаимодействие с клиентом. Во-вторых, внешний интерфейс должен передать выбранному серверу клиентские сообщения, полученные по соответствующему переключаемому соединению TCP.

Эти две задачи можно распределить так, как показано на рис. 11.10, б. Диспетчер отвечает за выбор сервера, на который будет переключено соединение TCP, а дистрибьютор отслеживает входящий по TCP трафик в поисках нуждающихся в переключении соединений. Переключатель используется для передачи сообщений TCP дистрибьютору. Когда клиент в первый раз связывается со службой Web, его сообщение, предназначенное для установления соединения TCP, пере-

сылается дистрибьютору, который, в свою очередь, связывается с диспетчером, чтобы тот решил, на какой из серверов следует переключить соединение. В это время переключатель уведомляется, что ему следует пересылать все последующие сообщения TCP данного соединения на выбранный сервер.

11.1.4. Именование

В Web для ссылок на документы используется единая схема именования [44] под названием *унифицированный идентификатор ресурса (Uniform Resource Identifier, URI)*. URI существуют в двух формах. *Унифицированный указатель ресурса (Uniform Resource Locator, URL)* идентифицирует документ и содержит информацию о том, как и где можно получить к нему доступ. Другими словами, URL — это зависящая от местоположения ссылка на документ. В противоположность ему *унифицированное имя ресурса (Uniform Resource Name, URN)* представляет собой правильный идентификатор, как он был описан в главе 4. URN используется как глобальная уникальная и не зависящая от местоположения сохраняемая ссылка на документ.

Реальный синтаксис URI определяется ассоциированной с этим идентификатором *схемой (scheme)*. Имя схемы является частью URI. Было определено множество различных схем, и сейчас мы рассмотрим некоторые из них, вместе с примерами соответствующих идентификаторов URI. Лучше других известна схема `http`, но она не единственная.

Универсальные указатели ресурсов

URL содержит информацию о том, как и где можно получить доступ к документу. Как получить доступ к документу, часто становится ясно из имени схемы, которая является частью URL, как, например, `http`, `ftp` или `telnet`. Местонахождение документа часто присоединяется к URL при помощи DNS-имени сервера, к которому следует отправлять запросы, хотя для этой цели может быть использован IP-адрес. Номер порта, который выделен на этом сервере для отслеживания подобных запросов, также является частью URL. Если он пропущен, используется стандартный адрес. И наконец, URL содержит также имя документа, который следует искать на сервере. Все это приводит к общей структуре URL, представленной на рис. 11.11.

а

Схема	Имя сервера	Имя файла
http	:// www.cs.vu.nl	/home/steen/mbox

б

Схема	Имя сервера	Порт	Имя файла
http	:// www.cs.vu.nl	: 80	/home/steen/mbox

в

Схема	Имя сервера	Порт	Имя файла
http	:// 130.37.24.11	: 80	/home/steen/mbox

Рис. 11.11. Характерные структуры URL. Использование только DNS-имени (а). Комбинация DNS-имени и номера порта (б). Комбинация IP-адреса и номера порта (в)

Разрешение указателей URL, подобных приведенным на рис. 11.11, проводится напрямую. Если сервер указан в виде DNS-имени, это имя следует разрешить в IP-адрес сервера. Используя номер порта, содержащегося в URL, клиент может связаться с сервером по протоколу, имя которого он получает из схемы, и передать ему имя документа, представляющее собой последнюю часть URL.

В табл. 11.4 представлено несколько примеров URL. URL со схемой `http` используется для пересылки документов по протоколу HTTP, как мы сейчас описали. Точно так же URL со схемой `ftp` используется для передачи файлов по протоколу FTP.

Таблица 11.4. Примеры URL

Название	Назначение	Пример
<code>http</code>	HTTP	<code>http://www.cs.vu.nl:80/globe</code>
<code>ftp</code>	FTP	<code>ftp://ftp.cs.vu.nl/pub/minix/README</code>
<code>file</code>	Локальные файлы	<code>file:/edu/book/work/chp/11/11</code>
<code>data</code>	Непосредственная передача данных	<code>data:text/plain;charset=iso-8859-7,%e1%e2%e3</code>
<code>telnet</code>	Удаленный вход в систему	<code>telnet://flits.cs.vu.nl</code>
<code>tel</code>	Телефон	<code>tel:+31201234567</code>
<code>modem</code>	Модем	<code>modem:+31201234567;type=v32</code>

Непосредственная отправка документов поддерживается при помощи URL со схемой `data` [284]. В таком случае к URL присоединяется сам документ, подобно присоединению данных файла к индексному узлу [307]. В качестве примера в таблице показан указатель URL, содержащий простой текст из греческих букв $\alpha\beta\gamma$.

Помимо ссылок на документы URL часто применяют для других целей. Так, URL со схемой `telnet` используется для организации сеанса `telnet` с сервером. Существуют также указатели URL для связи при помощи телефонных каналов, как описано в [468]. Пример URL со схемой `tel`, приведенный в таблице, содержит телефонный номер и просто позволяет клиенту сделать звонок по телефону. В данном случае клиентом обычно является устройство, например мобильный телефон. Пример URL со схемой `modem` можно использовать для организации модемного соединения с другим компьютером. В этом примере URL указывает, что удаленный модем поддерживает стандарт связи V32 ITU-T.

Универсальные имена ресурсов

В противоположность URL имена URN представляют собой не зависящие от местоположения ссылки на документы. Как показано на рис. 11.12, в структуре URN можно выделить три части [299]. Отметим, что имена URN образуют подмножество всех идентификаторов URI и имеют схему `urn`. Однако правила синтаксиса третьей части URN определяет идентификатор пространства имен.

"urn"	Пространство имен	Имя ресурса
urn	: ietf	: rfc:2648

Рис. 11.12. Обобщенная структура URN

Типичными примерами имен URN являются имена для идентификации книг по их номерам ISBN, например `urn:isbn.0-13-349945-6` [276], или имена для идентификации документов по их номерам IETF, например `urn:ietf.rfc:2648` [300]. *IETF* — это сокращение от имени организации, отвечающей за подготовку стандартов для Интернета, *Internet Engineering Task Force* (*целевая группа инженерной поддержки Интернета*).

Хотя определить пространство имен URN относительно просто, разрешить URN обычно гораздо сложнее, поскольку имена URN, рожденные в различных пространствах имен, могут значительно отличаться по структуре. Таким образом, вводя новое пространство имен, следует задать для него механизм разрешения имен. К сожалению, ни одно из созданных в настоящее время пространств имен URN не содержит подсказок на механизм разрешения имен.

11.1.5. Синхронизация

Синхронизация — не самый важный вопрос в Web в основном по двум причинам. Во-первых, жесткая организация Web, в которой серверы никогда не обмениваются информацией с другими серверами (так же как клиенты с другими клиентами), означает, что синхронизировать практически нечего. Во-вторых, Web может считаться системой, предназначенной главным образом для чтения. Изменения обычно вносятся одним человеком, а в таких условиях трудно ожидать конфликта двойной записи.

Однако все меняется, и постепенно становится актуальной поддержка совместной работы над web-документами. Другими словами, теперь и в Web требуются механизмы одновременного обновления документов группой совместно работающих пользователей или процессов.

По этой причине было создано расширение HTTP [490], протокол *WebDAV* (*Web Distributed Authoring and Versioning* — *распределенная подготовка документов в Web с контролем версий*), который поддерживает простые средства блокировки разделяемых файлов, а также создания, удаления, копирования и перемещения документов, находящихся на удаленных web-серверах. Мы кратко опишем способ синхронизации, принятый в WebDAV. Дополнительную информацию можно найти в [489]. Подробная спецификация WebDAV содержится в [171].

Для синхронизации параллельного доступа к совместно используемому документу WebDAV поддерживает простой механизм блокировок. В нем имеются два типа блокировок записи. Блокировка исключительной записи может быть предоставлена одному клиенту и в течение срока действия запрещает всем остальным клиентам изменять документ. Существует также и блокировка совместной записи, которая позволяет нескольким клиентам изменять документ одновременно. Поскольку такая блокировка сопровождается дроблением документа, блоки-

ровка совместной записи удобна, когда клиенты изменяют разные части одного документа. Однако клиенты должны отслеживать возможные конфликты двойной записи самостоятельно.

Блокировка осуществляется путем отправки маркера блокировки запросившему ее клиенту. Сервер регистрирует, какой из клиентов владеет маркером блокировки. Когда клиент пожелает модифицировать документ, он посылает серверу HTML-запрос `post` вместе с маркером блокировки. Маркер удостоверяет тот факт, что клиент имеет право записи документа и поэтому сервер должен выполнить запрос.

Важным является то, что во время, пока клиент сохраняет блокировку, поддерживать соединение между клиентом и сервером не нужно. Клиент, получив блокировку, может просто отсоединиться от сервера, соединившись с ним вновь, когда пожелает послать HTML-запрос.

Отметим, что если на клиенте, который хранит маркер блокировки, происходит отказ, сервер должен так или иначе восстановить блокировку. WebDAV не определяет, как сервер должен обрабатывать подобные ситуации, оставляя это конкретным реализациям. Выбор наилучшего решения сильно зависит от типа документов. Причина подобного подхода заключается в том, что общего способа решения проблемы зависших блокировок не существует.

11.1.6. Кэширование и репликация

Кэширование на клиенте, применяемое для повышения производительности в Web, всегда играло важную роль в структуре web-клиентов. Такое положение сохраняется и поныне [39]. Кроме того, в попытках разгрузить перегруженные web-серверы общепринятой практикой является репликация web-сайтов с размещением их копий по всему Интернету. Совсем недавно появилась сложная технология репликации в Web, и кэширование понемногу стало отходить на второй план. Давайте поближе познакомимся с этими вопросами.

Кэширование с помощью web-заместителя

Кэширование на клиенте обычно осуществляется в двух местах. Во-первых, большинство браузеров оснащены простейшими средствами кэширования. Когда документ поступает в браузер, он сохраняется в его кэше, откуда и загружается в следующий раз. Клиенты обычно могут настраивать кэш, указывая, когда производить проверку его непротиворечивости. Мы еще коснемся этого вопроса чуть ниже.

Во-вторых, клиент часто запускает web-заместитель. Как мы говорили ранее, web-заместитель получает запросы от локальных клиентов и пересылает их web-серверам. По приходу ответа результаты запроса передаются клиенту. Преимущество такого подхода состоит в том, что заместители могут кэшировать результат и при необходимости возвращать его и другим клиентам. Другими словами, web-заместитель может реализовывать разделяемый кэш.

Кроме кэширования в браузерах и заместителях можно также создать кэш, который будет покрывать регионы или целые страны, что приводит нас к иерар-

хической схеме кэширования. Подобные схемы используются в основном для снижения трафика, но проигрывают неиерархическим схемам из-за больших задержек. Увеличение времени ожидания вызывается частой необходимостью обращаться к нескольким кэширующим серверам, в то время как в неиерархических схемах достаточно лишь одного.

В Web введены различные протоколы поддержания непротиворечивости кэша. Чтобы гарантировать, что документ, полученный из кэша, соответствует действительности, некоторые web-заместители сначала посылают серверу HTML-запрос `get` с дополнительным заголовком запроса `If-Modified-Since`, определяя время последней модификации кэшируемого документа. Сервер возвращает документ, только если за указанное время он был изменен. В противном случае web-заместитель просто возвращает локальному клиенту его кэшированную версию. Если следовать терминологии, которую мы ввели в главе 6, это соответствует протоколу извлечения.

К сожалению, эта стратегия требует, чтобы заместитель связывался с сервером при каждом запросе. Чтобы повысить производительность за счет некоторого снижения требований к непротиворечивости, популярный web-заместитель Squid [91] устанавливает срок устаревания документа T_{expire} , зависящий от того, как давно изменялся кэшируемый документ. В частности, если $T_{\text{last_modified}}$ — время последней модификации кэшируемого документа (записанное владельцем), а T_{cached} — время кэширования, то:

$$T_{\text{expire}} = \alpha (T_{\text{cached}} - T_{\text{last_modified}}) + T_{\text{cached}}.$$

Здесь α , равное 0,2, — это значение, полученное эмпирическим путем. До момента T_{expire} документ считается непротиворечивым, и заместитель с сервером не связывается. После момента T_{expire} заместитель просит у сервера свежую копию, даже если документ не изменялся. При $\alpha = 0$ стратегия оказывается той же самой, что и в предыдущем обсуждавшемся случае.

Отметим, что не изменявшийся долгое время документ не проверяется на модификацию так же часто, как документ, изменявшийся недавно. Такая стратегия впервые была предложена в файловой системе Alex [87]. Очевидный недостаток такого подхода состоит в том, что заместитель может возвратить неверный документ, то есть документ, более старый, чем версия, находящаяся на сервере. Еще печальнее, что клиент не в состоянии обнаружить тот факт, что он получил устаревший документ.

Альтернативой протоколу на основе извлечения является схема, при которой сервер уведомляет заместителя об изменении документа, посылая ему сообщение о недействительности хранящегося в кэше документа. Недосток такого подхода состоит в том, что сервер вынужден проверять множество web-заместителей, а это неминуемо приведет к проблемам масштабируемости. Однако в [83] показано, комбинируя аренду и сообщения о недействительности документа можно удерживать нагрузку на сервер в допустимых пределах.

Одной из проблем, с которыми сталкиваются разработчики кэшей web-заместителей, состоит в том, что кэширование на заместителе имеет смысл, только если документ действительно разделяется несколькими клиентами. На практике

оказывается, что коэффициент кэш-попаданий не превышает значения 50 % и то, если кэш достаточно велик. Один из способов построения очень больших кэшей, способных обслуживать огромное число клиентов, состоит в использовании кооперативных кэшей, принцип действия которых иллюстрирует рис. 11.13.

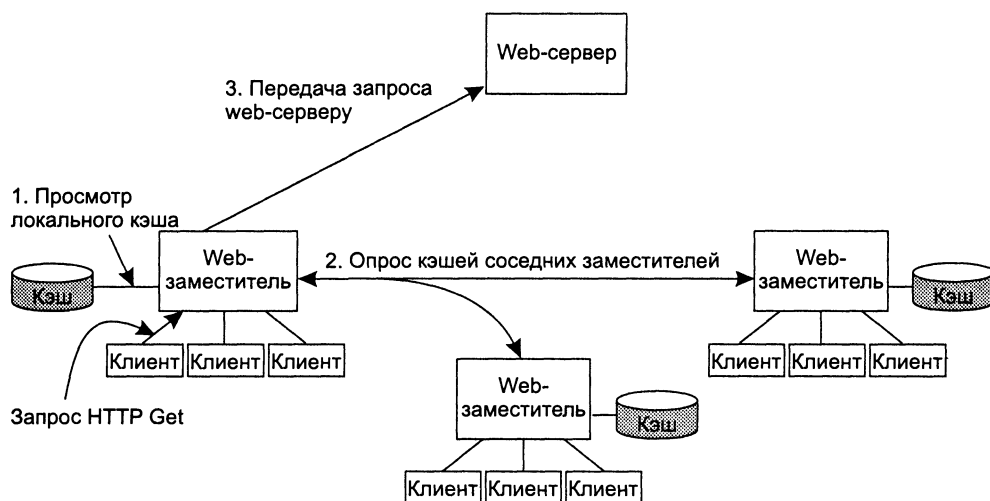


Рис. 11.13. Кооперативное кэширование

При *кооперативном кэшировании* (*cooperative caching*), когда бы ни случился кэш-промах на web-заместителе, заместитель первым делом проверяет несколько соседних заместителей на предмет того, не содержит ли один из них искомым документ. Если эта проверка не дает результатов, заместитель пересылает запрос на web-сервер, на котором находится этот документ [455]. В [495] показано, что кооперативное кэширование может оказаться эффективным для относительно малых групп клиентов (от десятков до тысяч пользователей). Однако эти группы могут быть обслужены также и кэшем одного заместителя, который значительно дешевле с точки зрения связи и затрачиваемых ресурсов.

Другая проблема с кэшем web-заместителя состоит в том, что его можно использовать только для статических документов, то есть для документов, которые не генерируются web-серверами в ответ на запрос клиента. Подобные документы уникальны в том смысле, что тот же самый запрос клиента в следующий раз может привести к другим результатам.

В некоторых случаях имеется возможность перенести генерацию документа с сервера в кэш заместителя [82], что в результате дает так называемый *активный кэш* (*active cache*). При таком подходе сервер возвращает не генерируемый документ, а апплет, который кэшируется заместителем. После этого за генерацию ответа пользователю отвечает уже заместитель, который и запускает апплет. Сам по себе апплет хранится в кэше до следующего аналогичного запроса, после чего выполняется снова. Отметим, что апплет тоже может устареть, если будет изменен на сервере.

Для активных кэшей существует несколько приложений. Так, например, если генерируемый документ получается слишком большим, то в следующий раз, когда потребуются его обновить, кэшированный апплет может организовать передачу от сервера только разницы между новым и старым вариантами документа.

В качестве другого примера рассмотрим документ, содержащий список рекламных баннеров, один из которых демонстрируется при каждом вызове документа. В этом случае при первом запросе документа сервер пошлет заместителю документа и полный список баннеров вместе с апплетом выбора баннера. При следующем запросе документа заместитель может сам обработать этот запрос и сформировать ответ, вернув на этот раз другой баннер.

Репликация серверов

В Web используются два основных способа репликации серверов. Во-первых, как мы говорили ранее, сильно нагруженные web-сайты для сокращения времени отклика используют кластеры web-серверов. Как правило, этот вид репликации прозрачен для клиентов. Во-вторых, имеется широко распространенный непрозрачный вид репликации, заключающийся в создании точной копии web-сайта на другом сервере, которая называется *зеркалом* (*mirror*). Клиент при этом получает возможность выбирать, через какой сервер получить доступ к web-сайту.

Недавно появился и постепенно набирает популярность третий вариант репликации. Он следует стратегии инициирования реплик серверов, которую мы обсуждали в главе 6. При таком подходе коллекция разбросанных по Интернету серверов предоставляет службы хостинга web-документов, организованные в форме реплицированных на серверах копий, в которых учтены характеристики доступа клиентов. Коллекция таких серверов называется *сетью доставки содержимого* (*Content Delivery Network, CDN*), или (реже) *сетью распределения содержимого* (*content distribution network*).

В CDN задействуют разные технологии репликации и распределения документов. Мы представили в главе 6 одно из решений, используемое в службе web-хостинга RaDaR [366]. В RaDaR сервер отслеживает число запросов, приходящих от клиентов отдельного региона. Если запросы представляют собой значительную долю всех запросов к серверу, принимается решение поместить копию документа на сервер этого региона.

Другой популярный подход реализован в системе Akamai [260]. Основная идея состоит в том, что каждый документ в Web состоит из главной HTML-страницы, в которую встроены другие документы, такие как изображения, видео- и аудиофайлы. Чтобы показать весь документ, необходимо, чтобы встроенные документы также были доставлены в браузер пользователя. Если предположить, что эти документы изменяются редко, то их имеет смысл кэшировать или реплицировать.

На любой встроенный документ можно сослаться обычным образом при помощи указателя URL (см. рис. 11.11). В CDN Akamai этот URL изменяется так, чтобы он ссылался на *виртуальный образ* (*virtual ghost*), представляющий собой ссылку на реальный сервер CDN. Измененный указатель URL разрешается так, как показано на рис. 11.14.

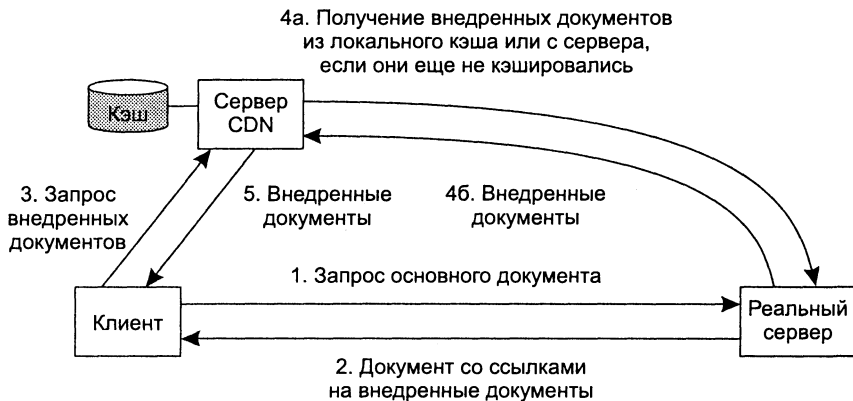


Рис. 11.14. Принцип работы сети CDN Akamai

Имя виртуального образа включает в себя DNS-имя `ghosting.com`, которое разрешается стандартной системой именования DNS в DNS-сервер CDN, ближайший к клиенту, пославшему запрос с разрешенным DNS-именем. Выбор ближайшего DNS-сервера CDN выполняется путем нахождения IP-адреса клиента в базе данных, содержащей «карту» Интернета. Разрешение имен, производимое на DNS-сервере CDN, позволяет получить адрес одного из обычных серверов CDN, выбираемый по критерию близости к клиенту и некоторым другим критериям, например текущей загрузке.

И, наконец, клиент пересылает запрос на внедренный документ на выбранный сервер CDN. Если этот сервер не содержит запрошенного документа, он получает его с исходного web-сервера (шаг 4 на рисунке), локально кэширует и передает клиенту. Если документ уже находится в кэше сервера CDN, он возвращается клиенту немедленно.

Важный момент в работе CDN Akamai и некоторых других сетях CDN — это обнаружение ближайшего сервера CDN. Один из возможных подходов предполагает использование службы локализации, подобной описанной в главе 4. Только что описанный подход сети CDN Akamai предполагает наличие службы DNS и поддержку «карты» Интернета. Альтернативный подход, описанный в [11], предполагает назначение одного и того же IP-адреса нескольким серверам DNS с тем, чтобы базовый сетевой уровень автоматически посылал запрос на поиск имени на ближайший сервер, используя протокол маршрутизации кратчайшего пути.

11.1.7. Отказоустойчивость

Отказоустойчивость в Web в основном реализуется путем кэширования на клиенте и репликации серверов. В HTTP нет никаких специальных средств повышения отказоустойчивости или восстановления после отказов. Отметим, однако, что высокая доступность Web достигается в основном путем избыточности, использование которой вообще является традиционным приемом в случае ответст-

венных служб, таких как DNS. Так, например, DNS позволяет возвращать в ответ на запрос о разрешении имени несколько адресов.

11.1.8. Защита

С учетом открытости Интернета понятна важность разработки системы защиты, которая могла бы оградить клиентов и серверы от разнообразных атак. Большинство вопросов защиты в Web связаны с организацией защищенного канала между клиентом и сервером. Основным механизмом организации защищенного канала в Web является *уровень защищенных сокетов (Secure Socket Layer, SSL)*, изначально предложенный компанией Netscape. Хотя SSL никогда не был формально стандартизирован, его поддерживает большинство клиентов и серверов Web. Недавно измененный вариант SSL, известный как протокол *защиты транспортного уровня (Transport Layer Security, TLS)*, был официально зафиксирован в виде спецификации RFC 2246 [125].

Как показано на рис. 11.15, TLS — это не зависящий от приложения протокол защиты, который логически располагается поверх транспортного протокола. Для простоты реализации TLS (и SSL) обычно основаны на TCP. Как мы увидим далее, TLS может поддерживать множество протоколов верхнего уровня, включая HTTP. Так, например, используя TLS, можно реализовывать защищенные варианты FTP или Telnet.



Рис. 11.15. Положение TLS в стеке протоколов Интернета

Сам по себе протокол TLS организован в виде двух уровней. Ядро протокола образует *уровень протокола записей TLS (TLS record protocol layer)*, который реализует защищенный канал между клиентом и сервером. Точные характеристики канала определяются при его создании и могут включать в себя дробление и сжатие сообщений, применяемые вместе с аутентификацией, контролем целостности и конфиденциальности сообщений.

Организация защищенного канала выполняется в два этапа, как показано на рис. 11.16. Сначала (это первые два сообщения на рисунке) клиент уведомляет сервер об используемом криптографическом алгоритме, а также о поддерживаемых методах сжатия. Итоговый выбор всегда остается за сервером, который сообщает о нем клиенту.



Рис. 11.16. Протокол TLS со взаимной аутентификацией

На втором этапе происходит аутентификация. Сервер всегда требует аутентифицировать себя, поэтому он передает клиенту сертификат, содержащий его открытый ключ, подписанный сертифицирующей организацией (CA). Если сервер требует, чтобы клиент также был аутентифицирован, клиенту тоже нужно послать серверу сертификат (сообщение 4 на рисунке).

Клиент создает случайное число, которое будет использоваться обеими сторонами для построения сеансового ключа, и посылает это число серверу, шифруя его открытым ключом сервера. Кроме того, если необходима аутентификация клиента, клиент также подписывает это число своим закрытым ключом (сообщение 5 на рисунке). В действительности посылаются отдельные сообщения с шифрованным и подписанным вариантами случайного числа, что имеет тот же эффект. После этого сервер может идентифицировать клиента и установить защищенный канал.

11.2. Lotus Notes

Рассмотрим теперь абсолютно другую распределенную систему документов. Lotus Notes — это ориентированная на базы данных система, разработанная корпорацией Lotus Development, но в настоящее время продажами и разработкой этой системы занимается компания IBM, которая купила Lotus Notes в конце 1990-х годов. Система работает под управлением различных платформ семейств Windows и UNIX. Большая часть внутренней структуры Lotus Notes является корпоративной тайной, и открытая документация по этой теме отсутствует. Не так давно руководства и справочники, а также материалы по архитектуре этой системы были опубликованы в Интернете (по адресу <http://www.notes.net/doc>). Обзор системы Lotus Notes можно найти в [272].

11.2.1. Обзор

Как и Web, система Lotus Notes представляет собой систему (потенциально очень большую) с архитектурой клиент-сервер. Lotus Notes изначально разрабатывалась для работы в локальных сетях, но в настоящее время может работать

и в глобальных сетях, например в Интернете. Общая структура системы показана на рис. 11.17. Система Lotus Notes образована из четырех основных компонентов: клиентов, серверов, баз данных и программ промежуточного уровня. Каждый клиент или сервер может иметь несколько локальных баз данных. Каждая база данных формирует коллекцию *заметок* (*notes*), которые являются ключевыми элементами данных в любой системе Lotus Notes. Мы еще вернемся к заметкам и базам данных из них чуть ниже.

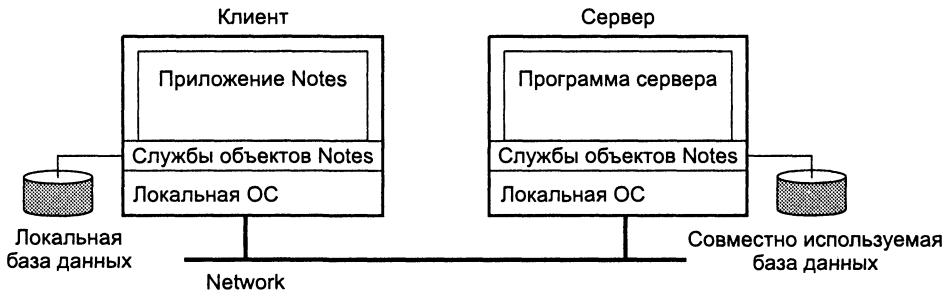


Рис. 11.17. Общая структура системы Lotus Notes

Клиент запускает приложения, требующие доступа к базам данных, сравнимые по функциональности с web-браузерами. Основное их отличие от браузеров состоит в том, что пользователи могут не только читать информацию из баз данных, но и изменять ее. Кроме того, имеется отдельный набор утилит, позволяющих пользователям создавать и поддерживать собственные базы данных.

Серверы Lotus Notes известны под названием *серверов Domino* (*Domino servers*). Сервер управляет коллекцией баз данных. Его главная задача — предоставлять доступ к этим базам данных удаленным клиентам и другим серверам. Основная программа сервера, таким образом, состоит из модулей, которые ожидают запросов, поступающих по сети, поддерживают соединения и сеансы с удаленными процессами, обрабатывают информацию, содержащуюся в локальных базах данных. Кроме того, имеются и другие задачи, связанные с управлением базами данных, которые нередко выполняются отдельными программами на машине сервера.

Документы в Web обычно реализуются в виде файлов. В противоположность им Lotus Notes хранит и обрабатывает документы посредством баз данных. Это главное и очень важное отличие этих двух систем, которое в основном и определяет разницу между web-серверами и серверами Domino.

Клиенты, серверы и базы данных связываются воедино при помощи выделенного компонента промежуточного уровня, который называют *службами объектов Notes* (*Notes Object Services, NOS*). Этот компонент промежуточного уровня реализуется поверх базовых операционных систем и сетей, позволяя клиентам и серверам связываться между собой и использовать локальные и разделяемые базы данных. Для этой цели он содержит компоненты для удаленного вызова процедур, средства хранения данных, механизмы обратного вызова и т. п.

Документная модель

Основной элемент данных в системе Lotus Notes — это *заметка (note)*, которая представляет собой, в сущности, список элементов. *Элемент (item)* — это ячейка, в которой хранятся соответствующие заметке данные. Каждый элемент имеет тип, который определяет, какого рода данные могут в нем храниться. Существуют текстовые элементы для хранения простого текста, числовые элементы для чисел с плавающей точкой, элементы для хранения даты и времени, элементы формул для хранения скомпилированных команд Notes, элементы сценария для хранения сценариев и т. д.

Каждая заметка может также иметь список ассоциированных с ней дочерних заметок. Кроме того, любая заметка может иметь максимум одного ассоциированного с ней родителя. Таким образом, мы приходим к иерархии заметок. Эта иерархия отражает одну из специфических целей систем Notes, а именно создание такой системы, которая позволила бы пользователям посылать заметки, а затем отправлять ответы на эти заметки. Связь между первоначальной заметкой и ответом на нее поддерживается в виде соотношения родитель—потомок примерно так же, как это происходит в системах сетевых новостей.

В системе производится разделение заметок на *заметки с данными (data notes)* и другие типы заметок. Заметку с данными можно сравнить с web-документом: она воспроизводит представляемый пользователю документ, который может включать в себя различные элементы данных — аудио- и видеозаписи, изображения, текст, значки и т. д. Заметки с данными также называют просто документами.

Существует множество других типов заметок, часть которых перечислена в табл. 11.5. Все их можно грубо разделить на заметки конструкции и заметки администрирования. Заметки конструкции используются для представления документов и работы с ними, заметки администрирования — для управления базами данных Notes.

Таблица 11.5. Примеры заметок

Тип заметки	Категория	Описание
Document	Данные	Предназначенный для пользователя документ (как и web-страница)
Form	Конструкция	Структура для создания, редактирования и просмотра документа
Field	Конструкция	Определяет поле, совместно используемое в формах и подчиненных формах
View	Конструкция	Структура для показа коллекции документов
ACL	Администрирование	Содержит список контроля доступа базы данных
RepFormula	Администрирование	Определяет репликацию базы данных

Заметка Form напоминает формы, знакомые пользователям Web или пользователям традиционных баз данных. Форма определяет, как должен выглядеть документ, включая положение и вид его элементов, которые становятся полями

формы. Некоторые поля могут совместно использоваться разными формами. Эта возможность задается при помощи заметки Field. Также из мира баз данных пришла заметка View, которая реализует так называемое представление, определяющее, как должна быть показана пользователю коллекция документов. Например, представление может определять, что коллекция документов должна выглядеть как таблица с элементами документов в качестве столбцов.

Существуют также и многочисленные заметки администрирования, например, заметка ACL используется для хранения списка контроля доступа. Как мы увидим ниже, Notes предоставляет возможности репликации баз данных. Как именно будет происходить репликация, записано в заметке Rep1Formula, которая представляет собой еще один пример заметки администрирования.

Как и документы в Web, заметки могут ссылаться друг на друга при помощи гиперссылок, которые в Notes называются *ссылками на заметки (notelinks)*. Ссылка на заметку определяет базу данных и заметку, содержащуюся в этой базе. Другими словами, возможно существование перекрестных ссылок на заметки из разных баз данных. Можно также ссылаться на заметки с использованием указателей URL, которые обычно встраиваются в ссылку на заметку вместе со ссылкой на сервер, управляющий соответствующей базой данных. Мы вернемся к этому вопросу, когда будем обсуждать правила именования Notes.

Следует понимать, что заметки похожи на web-документы, но построены совершенно иным образом. В частности, имеется четкое разделение на содержание заметки (которое определяется ее элементами) и способ ее представления пользователям (который определяется элементами конструкции). Это разделение можно сравнить с обсуждавшимся ранее разделением между XML и XSL.

11.2.2. Связь

Как и множество других распределенных систем, Lotus Notes использует для связи между клиентами и серверами базовую систему RPC. Система RPC Notes полностью прозрачна для клиентов и на практике является частью промежуточного уровня сетевой операционной системы (NOS). Когда сервер получает запрос RPC, он запускает отдельное задание, которое выполняет всю работу по взаимодействию, относящуюся к данному вызову RPC, как это происходит в других системах RPC, ориентированных на установление соединения.

Notes имеет переносимые средства для организации взаимодействия между процессами, чтобы процессы, выполняющиеся на одной машине, могли обмениваться информацией друг с другом. Однако в отличие от распределенных систем, таких как CORBA и DCOM, большого числа средств взаимодействия между процессами на разных машинах в Notes не предусмотрено. В основном связь осуществляется через общеизвестные интерфейсы, реализованные при помощи системы RPC Notes.

Исключение составляет подсистема Notes для работы с электронной почтой. Почтовые сообщения Notes всегда посылаются в формате MIME, и подсистема сама выбирает один из многочисленных протоколов пересылки почты, например SMTP. Существует также и специальный почтовый протокол Notes.

Для того чтобы облегчить разработку приложений верхнего уровня, например систем документооборота, в Notes имеются различные средства автоматической отправки писем в качестве отклика на определенные события, происходящие с базой данных. Так, например, изменение отдельного элемента в заметке может привести к отправке электронного письма, содержащего копию изменившейся заметки. Точно так же и приходящие сообщения электронной почты могут обрабатываться автоматически, приводя, в свою очередь, к изменениям в базе данных Notes.

11.2.3. Процессы

Как и в Web, в Notes имеется четкое разделение машин на клиенты и серверы. Клиентское программное обеспечение состоит из программ, которые позволяют пользователю взаимодействовать с серверами, и включает в себя, кроме всего прочего, и web-браузер. Также имеется отдельная программа для разработки и выполнения приложений Notes. Эта программа похожа на современные средства разработки приложений баз данных; она позволяет пользователю создавать заметки и соответствующие им формы, представления, события, задачи и т. п.

Программное обеспечение серверов состоит из основной программы и множества встроенных в нее заданий, выполняющих обработку входящих запросов, открытие и закрытие локальных баз данных, поддержание непротиворечивости данных и управление кластерами серверов (позже мы кратко обсудим этот вопрос). Кроме этих встроенных заданий на сервере имеется множество дополнительных *заданий сервера* (*server tasks*), имеющих вид отдельных процессов, которые выполняются на одной машине с главным сервером и находятся под полным его контролем. Главный сервер и его задания образуют сервер Domino (рис. 11.18).

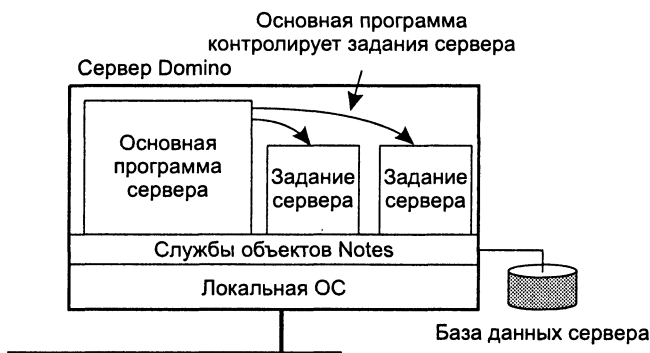


Рис. 11.18. Общая структура сервера Domino

Взаимодействие между процессами в сервере Domino осуществляется при помощи уровня NOS, который мы уже описывали. NOS предоставляет разнообразные средства связи между процессами, а также средства доступа к локальным базам данных и управления ими.

Для повышения отказоустойчивости и производительности серверы Domino могут объединяться в *кластеры (clusters)*, являющиеся коллекциями двух-шести машин, на которых работают одинаковые серверы с идентичными копиями баз данных. Кластер Notes похож на кластер web-серверов, но организован несколько по-иному. С одной стороны, кластеры не прозрачны для клиентов, клиенты знают, что работают с кластером и могут выбирать сервер, к которому пойдут их запросы.

Клиент узнает о кластере следующим образом. Обычно для доступа к заметке в определенной базе данных клиент должен получить ссылку на эту заметку, которая содержит и ссылку на сервер, связанный с этой базой данных. Когда клиент первый раз связывается с сервером базы данных, сервер возвращает ему список серверов, входящих в тот же кластер. Если сервер, к которому клиент обратился вначале, слишком загружен, клиент выбирает другой сервер кластера.

Как показано на рис. 11.19, этому второму серверу предлагается создать упорядоченный список доступных серверов кластера. Порядок представления серверов в списке определяется текущей загрузкой каждого сервера, причем наименее загруженный сервер попадает в начало списка. После этого клиент связывается с первым по списку сервером и выясняет, может ли тот обработать его запрос. Если нет, он повторяет те же операции со следующим по списку сервером.

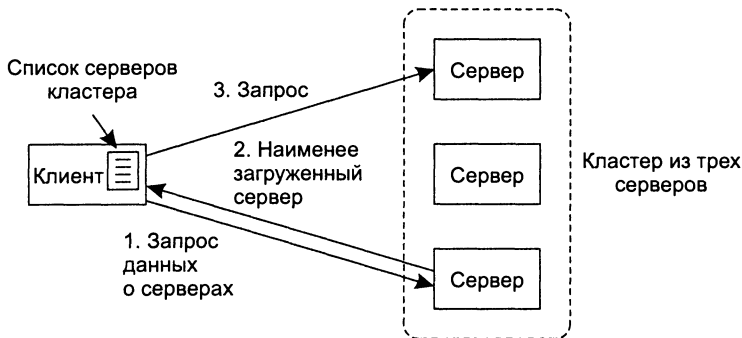


Рис. 11.19. Обработка запросов к кластеру серверов Domino

Загрузка сервера W вычисляется для каждого сервера как функция от времени отклика, возвращающая значение от 0 до 100. Так, если $T_{response}$ — текущее время отклика, а T_{opt} — оптимальное время отклика, то W вычисляется по формуле:

$$W = \max\{0 \dots 100 - (T_{response}/T_{opt})\}.$$

Меньшее значение W указывает на то, что этот сервер загружен сильнее, чем сервер с большим значением W . Текущее время отклика вычисляется по предdefinированному набору функций один раз в минуту. Если запросы, нуждающиеся в обработке, отсутствуют, W будет иметь значение, равное 100, указывая на оптимальное время отклика для следующего запроса.

11.2.4. Именование

Поскольку система Notes ориентирована на работу с базами данных, именование в ней играет иную по сравнению с другими распределенными системами роль. Так, поскольку заметки хранятся в базах данных, для доступа к ним не годятся механизмы разрешения имен. Вместо этого используются либо традиционные для баз данных операции, либо упоминавшиеся ранее представления и ссылки.

Сама база данных представлена единственным файлом, именуемым согласно соглашениям об именовании базовой файловой системы, поверх которой работает Notes. Помимо баз данных и других файлов в Notes имеется немало других требующих именования сущностей, включая пользователей, серверы, открытые ключи и т. д. Эта информация содержится в специальной базе данных, которая называется *каталогом Domino (Domino directory)* и доступ к которой осуществляется так же, как и к любой другой базе данных Notes.

Более традиционные имена в виде строк символов поддерживаются двумя способами. Во-первых, Notes имеет службы *различимых имен (distinguished names)* на основе LDAP. Эти службы каталогов могут использоваться не только для доступа к базам данных, но и, как мы говорили в главе 4, для организации иерархического пространства имен. Отметим, что службы LDAP часто реализуются посредством баз данных Notes.

Второй вариант поддержки имен в виде строк символов предполагает использование URL. Этот механизм требуется для доступа к серверам Domino через Web. Кроме того, он необходим для реализации web-серверов на базе Notes. Общий вид URL в заметках приведен на рис. 11.20 и напоминает URL для CGI.



Рис. 11.20. URL для доступа к базе данных Notes

Указатель URL в Notes состоит из трех основных частей. Первая часть — DNS-имя сервера Domino, который обеспечивает обработку запросов HTTP. Вторая часть — путь к базе данных Notes, размещенной на этом сервере. Третья часть, собственно, и реализует доступ к заметке в заданной базе данных. Она состоит из идентификатора (который мы кратко обсудим), имени операции и (необязательно) параметров этой операции. Сервер Domino отвечает за осуществление указанной операции над заданной заметкой.

Идентификаторы

Хотя имена в виде строк символов и играют в Notes важную роль, большинство вопросов, относящихся к именованию, решается при помощи машинных идентификаторов. Notes поддерживает несколько видов идентификаторов (табл. 11.6).

Наиболее важный из них — *универсальный идентификатор (Universal ID, UNID)*, используемый как глобальный уникальный идентификатор заметок. UNID — это 16-байтовое значение, которое приписывается каждой заметке. В случае репликации заметок все реплики имеют один и тот же идентификатор UNID.

Таблица 11.6. Некоторые идентификаторы Notes

Идентификатор	Область видимости	Описание
Универсальный идентификатор	Глобальная	Глобально уникальный идентификатор, присваиваемый каждой заметке
Идентификатор инициатора	Глобальная	Идентификатор заметки, включающий информацию об истории
Идентификатор базы данных	В пределах сервера	Меняющийся со временем идентификатор базы данных
Идентификатор заметки	В пределах базы данных	Идентификатор заметки, зависящий от экземпляра базы данных
Идентификатор реплики	Глобальная	Отметка времени, используемая для идентификации копий одной базы данных

От UNID ведет свое начало *идентификатор инициатора (Originator ID, OID)*, который используется для идентификации отдельного экземпляра заметки. OID содержит UNID заметки, а также 2-байтовый последовательный номер и 8-байтовую отметку времени. Он требуется для обнаружения и разрешения тех конфликтов, которые могут возникнуть при одновременном изменении реплицированных заметок. Мы вернемся к OID, когда будем рассматривать вопросы репликации в Notes.

Идентификатор базы данных (database ID) не является истинным идентификатором; это скорее отметка времени, указывающая на момент создания базы данных или ее последнего восстановления после сбоя сервера. Его можно рассматривать как идентификатор только в пределах сервера Domino, который за него отвечает.

Чтобы однозначно идентифицировать заметку в пределах конкретного экземпляра базы данных, используется *идентификатор заметки (note ID)*. Идентификатор заметки можно сравнить с идентификатором файла в файловой системе в том смысле, что он идентифицирует заметку только в пределах базы данных. В каждой базе данных имеется таблица соответствия, в которой идентификатор заметки отображается на ее физическое местоположение в базе данных. Таким образом, при перестроении базы данных необходимо изменять только таблицу соответствия, в то время как все идентификаторы заметок остаются прежними.

И, наконец, *идентификатор реплики (replica ID)* используется для идентификации группы баз данных, участвующих в репликации. Если две базы данных имеют одинаковый идентификатор реплики, значит, все изменения, сделанные в одной базе данных, будут распространены и на вторую. В принципе идентификаторы реплик глобально уникальны, но поскольку они базируются только на отметках времени, их уникальность не гарантирована. Как мы позже увидим, нет никакой нужды пытаться делать базы данных абсолютно идентичными, хотя во

многих случаях дело обстоит именно таким образом. Идентификатор реплики используется только для идентификации набора баз данных, изменения в которые вносятся одинаковым образом.

11.2.5. Синхронизация

Подобно многим другим распределенным системам, Notes поддерживает разнообразные механизмы блокировки, которые гарантируют исключительный доступ к базам данных. Кроме того, Notes поддерживает транзакции. Однако эти механизмы синхронизации реализованы нераспределенным образом в том смысле, что ограничены пределами одного сервера Domino. Поддержка распределенных блокировок или транзакций отсутствует.

11.2.6. Репликация

В отличие от многих других коммерческих систем, в системе Notes репликация поддерживалась с самого начала [229]. В Notes используется слабая форма распространения изменений, гарантирующая потенциальная непротиворечивость, при которой конфликтов обновления не бывает. Репликация в основном применяется к документам, то есть заметкам с данными, и производится следующим образом.

Ключевую роль в репликации играют *связующие документы (connection documents)*, представляющие собой особые заметки, содержащиеся в каталоге Domino и описывающие, когда, как и что реплицировать. Каждый сервер Domino имеет одно или более соответствующих заданий на репликацию, которые поддерживают схему репликации, определенную в связующих документах. Существует четыре возможных схемы репликации, перечисленные в табл. 11.7. Стандартной схемой считается схема *извлечения-продвижения (pull-push)*, в этом случае задание на репликацию устанавливает соединение с целевым сервером, а затем передает (*продвигает*) свои изменения на этот сервер. Кроме того, задание на репликацию считывает (*извлекает*) изменения, имевшие место на целевом сервере. Отметим, что эта схема полностью соответствует схеме извлечения-продвижения эпидемических алгоритмов, которые мы рассматривали в главе 6. Репликация в Notes в значительной степени основана на этих алгоритмах.

Таблица 11.7. Схемы репликации в Notes

Схема	Описание
Извлечение-продвижение	Задание на репликацию считывает изменения с целевого сервера и передает на него собственные изменения
Двойное извлечение	Задание на репликацию считывает изменения с целевого сервера и отвечает на приходящие от него запросы, передавая собственные изменения
Продвижение	Задание на репликацию передает собственные изменения на целевой сервер, никак не реагируя на изменения, имевшие место на этом сервере

Схема	Описание
Извлечение	Задание на репликацию считывает изменения, имевшие место на целевом сервере, не пытаясь передать ему собственные изменения

Кроме того, репликация может происходить по схеме *двойного извлечения* (*pull-pull*), при которой два задания на репликацию на разных серверах считывают изменения, появившиеся на каждом из серверов. Существуют также отдельные схемы *продвижения* (*push*) и *извлечения* (*pull*), при работе по которым изменения распространяются в одном направлении, соответственно, к целевому серверу или от него.

Существует три операции изменения, которые следует различать: модификация, добавление и удаление документа. С точки зрения репликации самое главное — знать, что документ модифицирован, а также когда он был модифицирован. Модифицированный документ должен быть разослан на все реплики. Запись изменений в заметку заканчивается изменением последовательного номера и отметки времени в идентификаторе инициатора документа. Предыдущие значения копируются в журнал истории документа.

Добавление нового документа сопровождается введением нового глобально уникального идентификатора инициатора. Чтобы зафиксировать факт удаления документа, на место удаленного документа в базе данных помещается *заглушка удаления* (*deletion stub*). Эта заглушка удаления аналогична свидетельству о смерти в эпидемических алгоритмах. Отметим, что в принципе заглушка удаления не может быть уничтожена до тех пор, пока не уничтожены все копии соответствующего документа.

Когда происходит репликация, все изменения распространяются на реплики баз данных. Напомним, что мы считаем репликами все базы данных, имеющие одинаковый идентификатор реплики. Идентичны ли две реплики на самом деле, то есть имеют ли они одинаковую внутреннюю организацию, не важно. Единственное, что нас действительно интересует, — это идентичность копий документов, которые содержатся в этих базах данных. Эти копии можно опознать по одинаковому универсальному идентификатору (UNID).

Разрешение конфликтов

Фундаментальная проблема метода репликации Notes состоит в том, что этот метод может (и будет) порождать конфликты. Если копии одного и того же документа изменяются независимо друг от друга, будут возникать конфликты двойной записи. При распространении изменений они должны быть разрешены. Notes обнаруживает и разрешает эти конфликты следующим образом.

Представим себе репликацию, производимую по схеме извлечения-продвижения, которую мы обсуждали ранее, с двумя репликами, *A* и *B*. Изменение в *B* необходимо считать (*извлечь*) в *A*, после чего следует передать (*продвинуть*) в *B* изменения, сделанные в *A*. В процессе репликации для каждой реплики создается список идентификаторов инициаторов.

Если список, составленный на сервере *B*, содержит универсальный идентификатор, не входящий в список, составленный на сервере *A*, это, по всей видимости, означает, что на *B* появился новый документ, который сервер *A* должен считать. Проведя аналогичное сравнение, мы обнаружим, что на *B* нужно передать новый документ, появившийся на *A*. Если оба списка содержат одинаковые идентификаторы инициаторов, значит, эти идентификаторы указывают на документ, который считается одинаковым на *A* и на *B*. В этом случае передавать с сервера на сервер какие-либо изменения не нужно.

Рассмотрим теперь, что происходит, если заметка *N* имеется в списках, составленных как на *A*, так и на *B*, но соответствующие ей идентификаторы инициаторов различны. Другими словами, существует две разных копии, N_A и N_B , заметки *N*. Эти две копии имеют одинаковые универсальные идентификаторы, но разные идентификаторы инициаторов. В этом случае задание на репликацию *A* должно просмотреть журнал истории каждой из копий. Если одна из историй является частью другой, значит, конфликт отсутствует, поскольку обновлять следует только одну копию. В этом случае обновленную копию следует переслать на другую реплику, после чего их истории и идентификаторы инициаторов можно сделать одинаковыми.

Однако, если две истории различны и ни одна из них не является частью другой, следует разрешить возникший между заметками конфликт. Здесь необходимо сделать следующие уточнения. Notes позволяет объединять документы с отслеживанием составляющих заметку элементов. В том случае, если элемент изменяется, последовательный номер, являющийся частью его идентификатора *OID*, увеличивается на единицу. Таким образом можно определить, какой именно элемент был изменен.

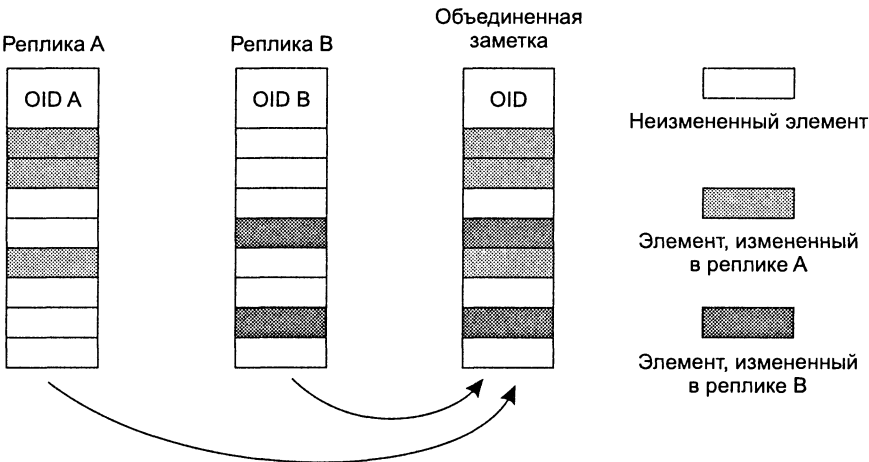


Рис. 11.21. Безопасное объединение двух документов с конфликтующими идентификаторами инициаторов

Представим теперь, что две истории абсолютно одинаковы до последовательного номера *k*. Если все изменения в N_A , имевшие место после последовательно-

го номера k , пришлось не на те же элементы, которые изменялись в N_B после номера k , то эти два документа можно без опаски объединить в один, как показано на рис. 11.21. Такую возможность мы получаем из-за отсутствия конфликтующих модификаций. Другими словами, изменения, сделанные в A , не конфликтуют с изменениями, сделанными в B .

Во всех остальных случаях мы отмечаем неразрешимый конфликт. В этом случае Notes выбирает один из документов «победителем», делая другой «проигравшим». Победителем объявляется копия с большим последовательным номером в идентификаторе инициатора. Если последовательные номера одинаковы, ничья разрешается в пользу документа, измененного последним (имеющего в идентификаторе инициатора самую позднюю отметку времени).

Репликация в кластере

Репликация, которую мы до сих пор обсуждали, относилась в основном к рассеянным коллекциям серверов. В случае кластеров серверов, которые всегда находятся в одной локальной сети, применяются другие подходы. Вместо явного планирования репликации при помощи связующих документов изменения просто немедленно передаются на все реплики кластера.

Для этой цели каждый сервер поддерживает очередь событий модификации, в которую записываются разнообразные локальные операции с базой данных. Один раз в секунду специальное задание на репликацию просматривает очередь в поисках изменений, которые следует распространить на другие серверы кластера. Эти изменения пересылаются на другие реплики, а соответствующие им события удаляются из очереди.

11.2.7. Отказоустойчивость

С точки зрения поддержки отказоустойчивости в распределенных системах Notes не имеет каких-либо определенных механизмов маскировки отказов. Наиболее явная поддержка отказоустойчивости состоит в том, что в реализацию базы данных на уровне одного сервера входит подсистема восстановления. Эта подсистема использует журнал с упреждающей записью, в который записывается перед выполнением каждая операция с базой данных, входящая в транзакцию. Мы уже описывали протоколирование транзакций в главе 5 и не будем повторяться.

11.2.8. Защита

Основу механизмов защиты в любой системе Notes составляет аутентификация пользователей. После того как пользователь аутентифицирован, ему предоставляется доступ к заметкам в базах данных на серверах. Существует сложный механизм управления доступом к заметкам и частям заметок. Далее мы более подробно рассмотрим сначала вопросы аутентификации, а затем и проблемы управления доступом в Notes.

Аутентификация

Для аутентификации пользователей и серверов Notes использует криптографию с открытым ключом. Аутентификация производится путем передачи сертификатов, содержащих открытый ключ. Основой этой схемы является доверие к сертификатам. Другими словами, получатель сертификата должен быть уверен, что открытый ключ, который в нем содержится, действительно принадлежит аутентифицируемому отправителю. В Notes уделяется особое внимание подтверждению правильности сертификатов. Чтобы понять, как происходит подтверждение правильности, рассмотрим, как эти сертификаты создаются.

С точки зрения защиты Notes различает три типа сущностей: пользователи, серверы и сертифицирующие организации. Каждая из этих сущностей известна системе по ее *удостоверению (identity)*, которое реализовано в виде файла, содержащего ключи и сертификаты. Так, например, когда в системе регистрируется новый пользователь, для него создается два набора пар (*открытый ключ, закрытый ключ*). Одна пара, 512-битные ключи RSA, используется для шифрования данных за пределами Соединенных Штатов; другая пара, 630-битные ключи, создается для шифрования данных в Соединенных Штатах, а также для аутентификации и подписания сообщений. Кроме того, создается сертификат, который подписывается сертифицирующей организацией, известной в той системе Notes, к которой подключается пользователь. Понятно, что использовать эту подпись могут только авторизованные администраторы.

Вся эта информация хранится в файле удостоверения пользователя системы Notes. Конфиденциальная информация, такая как закрытые ключи, шифруется с использованием ключа, генерируемого на основе пароля пользователя. Серверы Domino также имеют похожие файлы удостоверений, которые хранятся где-то внутри системы. Файлы удостоверений сертифицирующих организаций обычно хранятся в физически защищенном месте.

Notes поддерживает иерархию сертификатов. Другими словами, при добавлении пользователя в систему системный администратор может создать для него новый сертификат, в котором содержится открытый ключ, соответствующий закрытому ключу администратора. Продолжая подобную линию рассуждений, для подтверждения правильности сертификата системного администратора можно использовать сертификат более высокого уровня полномочий и т. д.

Предположим теперь, что Алиса хочет аутентифицировать себя для Боба. Давайте сначала рассмотрим вариант, когда Алиса и Боб работают в одной организации, скажем, в университете Franeker (FU), но Алиса — на факультете вычислительной техники (Department of Computer Science, CS), а Боб — на электротехническом факультете (Department of Electrical Engineering, EE). Алиса и Боб имеют в своих файлах удостоверений открытый ключ университета Franeker.

Алиса посылает свои сертификаты Бобу, а Боб проверяет открытые ключи, содержащиеся в этих сертификатах, придерживаясь следующих правил:

- ◆ все открытые ключи, хранящиеся в файле удостоверения Боба, являются доверенными для Боба;

- ♦ является доверенным открытым ключ из сертификата, подписанного организацией, открытый ключ которой Боб хранит в своем файле удостоверения;
- ♦ является доверенным открытым ключ, содержащийся в сертификате, подписанном организацией, использующей доверенный открытый ключ.

Алиса может передать Бобу два сертификата: сертификат $[A, K_A^+]_{EE}$, содержащий ее открытый ключ и подписанный EE , и сертификат $[EE, K_{EE}^+]_{FU}$, содержащий открытый ключ EE , но подписанный FU , как показано на рис. 11.22. Боб будет доверять K_{EE}^+ , поскольку он может проверить правильность соответствующего сертификата, используя открытый ключ FU , который хранится в его файле удостоверения. Согласно первому правилу, K_{FU}^+ является доверенным ключом Боба. Применение второго правила позволяет Бобу поверить в правильность K_{EE}^+ .

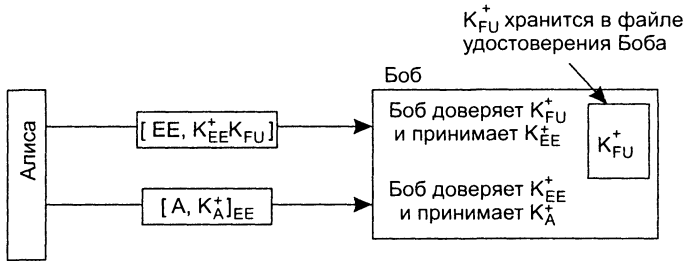


Рис. 11.22. Подтверждение правильности открытых ключей в Notes

Теперь Боб может проверить правильность сертификата $[A, K_A^+]_{EE}$. На основании третьего правила Боб может проверить, что факультет EE действительно выдал Алисе открытый ключ K_A^+ , и по этой причине он верит, что открытый ключ, присланный Алисой, действительно принадлежит ей.

Одна из проблем описанной схемы состоит в том, что Алиса должна передать сертификат Бобу, то есть кому-то, чья подпись менее представительна, чем у организации, открытый ключ которой Алиса и Боб хранят в своих файлах удостоверений. В нашем примере это открытый ключ университета Franeker. Однако если Алиса и Боб работают в разных университетах, проверка сертификатов Алисы будет невозможна.

Решение проблемы состоит в использовании *перекрестных сертификатов* (*cross certificates*), которые явно описывают обоюдное доверие. Так, например, если Алиса работает в техническом университете Lommel (Technical University of Lommel, TUL), то чтобы заставить Боба поверить ее собственному открытому ключу, она может использовать сертификат, содержащий открытый ключ TUL, подписанный FU.

После подтверждения сертификатов производится взаимная аутентификация с посылкой запроса, зашифрованного открытым ключом получателя. Получатель должен доказать, что обладает соответствующим закрытым ключом, вернув запрос отправителю так, как мы описывали в главе 8. Дополнительные подробности по вопросу аутентификации в Notes можно найти в [317].

Управление доступом

Notes предоставляет разнообразные механизмы управления доступом к компонентам системы, и здесь мы лишь кратко коснемся этого вопроса. Систему Notes можно считать состоящей из нескольких частей, перечисленных в табл. 11.8. Для серверов *списки контроля доступа (Access Control Lists, ACL)* явно определяют, каким пользователям, группам пользователей и серверам разрешено посылать запросы на сервер и на какой конкретно порт этого сервера. Списки контроля доступа — одно из базовых средств контроля доступа в Notes.

Таблица 11.8. Субъекты контроля доступа в Notes

Субъект	Описание
Серверы	Списки ACL определяют права доступа к серверам и портам
Рабочие станции	Списки ECL определяют права на выполнение сценариев и другого исполняемого кода
Базы данных	Списки ACL определяют права различных типов пользователей
Файлы	Списки ACL используются для контроля доступа через web-клиентов
Заметки конструкции	Списки ACL управляют представлением документов
Документы	Списки ACL управляют доступом к документам на чтение и запись

Для рабочих станций Notes поддерживает *списки контроля исполнения (Execution Control Lists, ECL)*, явно описывающие ограничения, налагаемые на исполняемый код, который может быть загружен на рабочую станцию при чтении заметок. Так, например, исполняемому коду ECL может ограничить доступ к локальной файловой системе рабочей станции, к средствам отправки почты, к определенным сетевым адресам и т. п. ECL можно рассматривать как некоторое расширение модели сита, описанной в главе 8, с предоставлением конкретному загружаемому коду определенных прав и запрету аналогичных действий для прочего кода. Однако следует отметить, что во многих случаях соблюдение загруженным кодом налагаемых на него ограничений основано исключительно на доверии. Другими словами, проверка выполняемых инструкций средствами защиты не обеспечивается.

Существуют также разнообразные средства управления доступом к базам данных. Каждая база данных имеет соответствующий ей список ACL, который явно определяет, кто и что может делать. Существует семь уровней доступа к базам данных, от уровня менеджера, имея который пользователь может добавлять, удалять и изменять все, что угодно, до уровня «нет доступа», который позволяет пользователю выполнять операции чтения и записи только в открытых документах. Контроль доступа к базе данных подразумевает также разнообразные дополнительные привилегии доступа по отношению к различным типам заметок, которые могут храниться в базе данных. Так, например, существует разделение среди операций с документами, особенно в отношении заметок конструкции.

Контроль доступа к файлам организован так же, как в web-клиентах. Кроме того, также организован и доступ к файлам, не являющимся файлами баз данных.

Уточнение правил доступа к базам данных может производиться путем определения ролей для задания прав доступа к элементам конструкции. Так, например, определенная группа пользователей может получить право на внесение изменений в набор хранящихся в базе данных заметок конструкции. Вместо того чтобы составлять список пользователей, менеджер базы данных имеет возможность определить роль и задать права для этой роли. Затем можно предоставлять пользователям права на исполнение этой роли.

И, наконец, каждый документ также имеет соответствующий ему список ACL. Доступом можно управлять и на уровне элементов, причем для лучшей защиты каждый элемент может иметь собственный секретный ключ. Можно также потребовать, чтобы определенная часть документов была скрыта от неавторизованных пользователей или чтобы при пересылке документов в удаленное место они в обязательном порядке подписывались.

Более полное описание системы управления доступом Notes можно найти в [272].

11.3. Сравнение WWW и Lotus Notes

World Wide Web и Lotus Notes — две наиболее распространенные распределенные системы документов, однако они очень отличаются друг от друга.

Философия

Обе системы, Web и Lotus Notes, используют простую модель клиент-сервер, в которой одиночный сервер управляет набором документов, доступных удаленным клиентам. Полный набор документов разбросан по многим и многим серверам. В этом смысле оба подхода идентичны. Однако сравнивая лежащие в их основе документные модели, мы обнаруживаем между ними значительную разницу.

Среда Web недалеко ушла от традиционной модели, в которой документ представляет собой, в сущности, текстовый файл, содержащий разнообразные команды разметки на языке HTML или одном из новых языков (например, XML). Однако эти документы могут включать в себя и документы других типов, такие как изображения, аудио- и видеофрагменты. Кроме того, по мере роста популярности Web появилась возможность включать в документ более сложные элементы, такие как сценарии и апплеты.

По мере того как Web продолжает развиваться, HTML как язык разметки постепенно заменяется языком XML, который позволяет лучше описать структуру документа. Кроме того, XML отделяет структуру документа от его представления, в то время как в HTML они смешаны «в кучу».

Модель, лежащая в основе Lotus Notes, значительно отличается от модели, принятой в Web. Основным элементом данных в Notes является список элементов (возможно, очень небольших), известный под названием заметки. В противоположность Web заметка — это структура данных, которая привязана к базе данных. Изменение и отображение заметок осуществляется исключительно механизмами баз данных, такими как формы и представления. Эти механизмы,

в свою очередь, определяются при помощи специальных заметок — элементов конструкции.

Другое отличие между Web и Lotus Notes состоит в том, что в Notes делается попытка при помощи заметок управлять всей системой. Так, например, существуют специальные заметки для управления репликацией, обеспечения защиты и т. п. В Web для этой цели используются выделенные механизмы.

Связь

Взаимодействие в Web происходит по специальному протоколу HTTP, который определяет все допустимые операции с документами, в основном ограничивающиеся получением документа с сервера и помещением его на сервер.

Взаимодействие в Lotus Notes поддерживается при помощи традиционной подсистемы RPC. Кроме того, в Notes делается упор на механизмы связи верхнего уровня, в основном электронную почту, для чего в этой системе предусмотрены разнообразные средства автоматического приема, обработки и отправки почты.

Следует отметить, что взаимодействие между процессами на одной машине также осуществляется по-разному. В Web поведение клиентов и серверов в этом отношении жестко определяется базовой операционной системой в том смысле, что любая связь между процессами реализуется механизмами операционной системы. С другой стороны, в Notes для маскировки разницы между операционными системами клиентов и серверов используется специальный уровень, NOS. Такой подход улучшает переносимость многих приложений Notes.

Процессы

Сравнивая Web и Lotus Notes с точки зрения процессов, мы можем обнаружить, что многие задачи решаются в них сходным образом. Наиболее важными клиентами для Web являются браузеры, предоставляющие пользователям графический интерфейс для получения и просмотра документов. Современные браузеры имеют дополнительные средства для редактирования документов. Кроме того, возможности большинства браузеров можно расширять динамически при помощи встраиваемых модулей расширения.

В Lotus Notes клиенту также предоставляется графический интерфейс для просмотра заметок, хранящихся в удаленной базе данных. Кроме того, клиенты Notes обычно имеют специальные приложения для редактирования заметок при помощи форм, представлений и т. п. Поскольку клиент Notes предназначен для работы со всеми типами заметок (заранее определенными), необходимость в наделении его дополнительной функциональностью не возникает.

Организация серверов этих двух систем также имеет сходные черты за исключением того, что традиционный web-сервер обычно больше приспособлен для работы с файловой системой. В противоположность ему сервер Notes Domino ближе к традиционным серверам баз данных.

Гибкость серверной части Web реализуется при помощи программ CGI. Эти программы обычно запускаются в виде отдельного процесса и могут взаимодействовать с базами данных. Таким образом, возможности web-сервера приближаются к возможностям сервера Domino. Кроме того, web-серверы могут поддерживать динамически загружаемые модули, известные как сервлеты.

Сервер Domino представляет собой главную программу, которая работает как автономный процесс. Параллельно главной программе может выполняться множество дополнительных заданий, реализованных в виде заданий сервера. Задания сервера можно сравнить с программами CGI в Web.

И Web, и Notes могут работать с кластерами серверов, когда для выполнения запросов клиентов несколько серверов объединяются в единую структуру. Основная разница между двумя системами состоит в степени прозрачности такого объединения. На практике web-клиенты могут оставаться в полном неведении о том, что они направляют запросы к кластеру серверов. В противоположность им клиенты Notes должны составить список входящих в кластер серверов и при необходимости перенаправлять свои запросы на тот или другой сервер кластера.

Именованние

Разница в системах именования между Web и Notes напрямую связана с файловой организацией Web и использованием баз данных в Notes. В Web документы именуются при помощи URN или URL. Указатели URL, используемые в Web в настоящее время, обычно содержат имя, соответствующее локальному имени файла документа, сохраненного на сервере. В противоположность URL имена URN предназначены для использования в качестве идентификаторов, никак не связанных с местоположением. Наглядным примером URN может служить номер ISBN, при помощи которого идентифицируют книги.

В Notes документы (и другие типы заметок) имеют связанный с ними универсальный идентификатор (UNID), являющийся правильным идентификатором. Универсальный идентификатор используется для именования заметки и доступа к ней. Для поиска записок в базе данных нет нужды в осмысленном (с точки зрения человека) идентификаторе из строки символов. Однако Notes имеет также интерфейс именования для Web. Он построен на базе указателя URL и включает в себя универсальный идентификатор заметки, а также имена сервера и базы данных, в которых хранится указанная заметка.

Синхронизация

Что касается синхронизации, то ее поддержка и в Web, и в Notes минимальна. В обеих системах синхронизация, в сущности, сводится к локальным механизмам блокировки. Поддержка блокировки нескольких документов, размещенных на разных серверах, отсутствует. Лишь в последнее время появились предложения для Web по поддержке коллективного редактирования с блокировкой совместно используемых документов.

Кэширование и репликация

Если сравнивать средства кэширования и репликации, то между Web и Notes обнаружатся серьезные различия. Кэширование всегда играло в Web особую роль — оно повышало масштабируемость всей системы. В настоящее время традиционными стали иерархические схемы кэширования, в которых кэширование производится как на уровне пользователей, так и на уровне организаций, регионов и стран. Кэширование эффективно потому, что большинство документов в Web

очень редко изменяются. Однако ситуация быстро меняется, что делает кэширование, особенно иерархическое кэширование, все менее эффективным.

Для Web были предложены и реализованы различные протоколы поддержания непротиворечивости кэша. Многие системы поддерживают схемы слабой непротиворечивости, когда кэшированный документ считается правильным до тех пор, пока не закончится срок аренды. За это время документ будет изменен, но кэшированные копии не получают никакой информации об этом событии.

Мы не можем ничего сказать о роли кэширования на клиенте в Notes, поскольку оно не документировано. Можно предположить, что документы используются совместно и при этом часто изменяются. По этой причине кэширование в Notes должно быть менее эффективно, нежели в Web.

Репликация в Web традиционно поддерживается в форме создания зеркал для сайта целиком. Однако этот механизм слишком негибок. С появлением сетей доставки содержимого (CDN) стала расти популярность динамической репликации. Сети CDN позволяют динамически реплицировать документы на серверах, расположенных близко к пользователю. Непротиворечивость реплик обеспечивается технологиями CDN.

В противоположность кэшированию репликация играет в Notes важную роль. Она осуществляется при помощи эпидемических алгоритмов со слабой формой распространения изменений. Такой подход приводит к конфликтам двойной записи, которые требуется разрешать вручную. Хотя Notes старается разрешать конфликты автоматически, система не в состоянии справиться со всеми возможными ситуациями. Если для однозначного разрешения конфликта требуется вмешательство оператора, Notes выбирает победивший, с точки зрения системы, документ и при этом считает все остальные документы проигравшими, но сохраняет их, чтобы пользователи впоследствии могли разрешить конфликт вручную так, как им покажется правильным.

Отказоустойчивость

Для обеспечения отказоустойчивости Web и Notes используют сходные методики. В Web надежность системы связи обеспечивается исключительно протоколом TCP. Весьма интересен тот факт, что различного рода кластеры web-серверов повышают доступность и увеличивает производительность каждого входящего в кластер web-сервера. Были предложены и реализованы несколько подобных решений, большинство из которых основаны на использовании выделенного сервера внешнего интерфейса, передающего запросы на один из серверов кластера.

В Notes используются специальные механизмы поддержки кластеров серверов Domino с репликацией баз данных по достаточно строгому протоколу непротиворечивости. Согласно этому протоколу изменения каждую секунду рассылаются всем серверам кластера; в остальном этот протокол идентичен упоминавшимся выше протоколам репликации со слабой формой распространения изменений.

Механизмы восстановления обеих систем, если они вообще существуют, ограничены восстановлением одиночного сервера и никак не адаптированы для нужд распределенных систем.

Защита

Защита играет особую роль как в Web, так и в Notes. В Web защита обычно реализуется средствами защиты транспортного уровня (TLS), который позволяет организовать защищенный канал между клиентом и сервером, а затем в ходе сеанса связи обеспечивать контроль доступа. Авторизация обычно производится сервером.

В Notes используются сертификаты аутентификации. При этом особое внимание уделяется проверке подлинности сертификатов. Для того чтобы решить, можно ли доверять открытому ключу, содержащемуся в сертификате, в системе поддерживается специфическая модель доверия. Обычно доверие устанавливается в соответствии с иерархической схемой именования, которая частично конфигурируется при настройке сервера. С помощью перекрестных сертификатов две разные системы Notes могут выразить друг другу взаимное доверие, что позволяет их клиентам задействовать серверы каждой из этих систем.

Управление доступом в Notes реализовано при помощи специальных записей, содержащих списки ACL. Таким образом можно определить самые разные права доступа вкпе с различными уровнями доступа, что открывает широкие возможности для индивидуальной настройки механизмов контроля доступа.

Результаты сравнения систем Web и Notes представлены в табл. 11.9.

Таблица 11.9. Сравнение Web и Lotus Notes

Аспект	WWW	Notes
Базовая модель	Гипертекст	Список текстовых элементов (заметок)
Расширения	Мультимедиа, сценарии	Мультимедиа, сценарии
Модель хранения	На основе файлов	На основе базы данных
Сетевое взаимодействие	HTTP	RPC, электронная почта
Взаимодействие между процессами	Определяется операционной системой	Службы NOS
Клиентский процесс	Браузер, редактор	Браузер, редактор конструкции
Клиентские расширения	Модули расширения	Предоставляются операционной системой клиента
Серверный процесс	Напоминает файловый сервер	Напоминает сервер баз данных
Серверные расширения	Сервлеты, программы CGI	Задания сервера
Кластеры серверов	Прозрачные	Непрозрачные
Именованье	URN, URL	URL, идентификаторы
Синхронизация	В основном локальная	В основном локальная
Кэширование	Расширенное	Не документировано
Репликация	Создание зеркал, CDN	Слабая
Отказоустойчивость	Надежная связь и кластеры	Кластеры
Восстановление	Явная поддержка отсутствует	Одиночного сервера

продолжение ➤

Таблица 11.9 (продолжение)

Аспект	WWW	Notes
Аутентификация	В основном TLS	Проверка подлинности сертификатов
Контроль доступа	В зависимости от сервера	Расширенный, списки ACL

11.4. Итоги

Можно утверждать, что распределенные системы документов, особенно World Wide Web, являются наиболее популярными среди конечных пользователей сетевыми приложениями. Понятие о документе как средстве передачи информации понятно для людей, постоянно имеющих дело с реальными документами. Каждый понимает, что такое бумажный документ, так что распространение этого понятия на электронные документы для большинства людей трудностей не вызывает.

Существуют разные способы моделирования документов, самое важное в которых — способ связывания документов друг с другом. Модель гипертекста, которую поддерживают как Web, так и Lotus Notes, существенно способствовала популярности распределенных систем документов. В этой модели легко можно активизировать ссылку на другой документ. Результатом является получение документа, на который указывает ссылка, и вывод его на экран пользователя.

Документы обычно хранятся на серверах, которые предоставляют клиентам доступ к этим документам. Если ссылки на документы могут пересекать границы между серверами, то есть при наличии глобальной системы ссылок на документы, относительно просто организовать глобальное распределение документов. В простейшем случае ссылка содержит имя сервера, на котором хранится документ, и клиент, желающий получить документ, напрямую обращается к этому серверу.

Важным аспектом архитектуры распределенных систем документов являются механизмы доступа к документам. От того, какой механизм доступа предлагается в данной системе, зависит уровень ее масштабируемости. Документы обычно редактируются их единственным владельцем или небольшой группой пользователей. С другой стороны, читать документ может достаточно большое число пользователей. Такой вариант доступа облегчает кэширование и репликацию, поскольку позволяет задействовать слабые формы распространения изменений. Другими словами, гарантировать хорошую производительность системы относительно просто, поскольку копии документов можно разместить неподалеку от пользователей и при этом не создать серьезных проблем с синхронизацией.

Защиту в рассматриваемых системах нужно обеспечивать, как и в любых других. Меры по обеспечению защиты, как обычно, состоят в организации защищенного канала и последующем контроле доступа.

Вопросы и задания

1. До какой степени электронная почта соответствует модели документов Web?
2. В Web клиент сначала получает файл, а затем открывает его и выводит на экран. Что можно сказать о таком подходе применительно к файлам мультимедиа?

3. Почему при наличии сохраненных соединений производительность значительно выше по сравнению с несохраненными соединениями?
4. Объясните разницу между модулями расширения, апплетами, сервлетами и программами CGI.
5. Опишите структуру службы именования общего назначения, преобразующей URN в URL, то есть службы, которая разрешает имя URN в указатель URL копии документа, идентифицируемого именем URN.
6. Достаточно ли клиенту в системе WebDAV для получения права на запись предъявить серверу только маркер блокировки?
7. Несмотря на то что web-заместитель вычисляет время истечения срока годности документа, это независимо от него делает еще и сервер. В чем достоинство такого подхода?
8. Какому протоколу распространения соответствует сеть CDN Akamai — извлечения или продвижения?
9. Опишите простую схему, при использовании которой сервер CDN Akamai может, не проверяя документ на исходном сервере, обнаружить, что срок годности кэшированного документа истек.
10. Имеет ли смысл вместо использования одной или нескольких глобальных стратегий репликации поддерживать собственную стратегию репликации для каждого web-документа?
11. URL в Notes содержит идентификатор документа. Насколько это лучше по сравнению с использованием в URL имен файлов, как это принято в Web?
12. Предполагает ли схема URL в Notes глобально уникальные имена операций?
13. Рассмотрим кластер серверов Domino и предположим, что клиент Notes в первый раз связался с сильно загруженным сервером Domino этого кластера. Опишите дальнейшее развитие событий.
14. В этой главе был описан конфликт между двумя измененными документами Notes. Возможно также возникновение конфликта между удаленной и измененной копиями. Опишите, как возникает подобный конфликт и как его можно разрешить.
15. Какова вероятность конфликта двойной записи в Notes для случая репликации в кластере?
16. Приведите решение по аутентификации в Notes, альтернативное использованию перекрестных сертификатов.
17. Выполнение загруженного кода в Notes часто базируется на доверии. Насколько это безопасно?

Глава 12

Распределенные системы согласования

12.1. Знакомство с моделями согласования

12.2. TIB/Rendezvous

12.3. Jini

12.4. Сравнение TIB/Rendezvous и Jini

12.5. Итоги

В предыдущих главах мы рассматривали различные подходы к распределенным системам, в которых в качестве субъекта распределения присутствовал некий тип данных. Этот тип данных — объект, файл или документ — является, по сути, порождением нераспределенных систем, который адаптируется для распределенных систем таким образом, чтобы многие проблемы, вызываемые распределенной архитектурой, были незаметны ни пользователям, ни разработчикам.

В этой главе мы рассмотрим новое поколение распределенных систем, которые изначально предполагают распределенность многих компонентов системы, и реальные проблемы таких систем вытекают из необходимости согласовывать взаимодействие разных компонентов. Другими словами, вместо того чтобы решать проблемы прозрачного распределения компонентов, мы сосредоточимся на согласовании этих компонентов.

Некоторые вопросы согласования уже рассматривались в предыдущих главах, особенно при обсуждении систем на базе событий. Как можно заметить, во многие традиционные распределенные системы постепенно включаются механизмы, играющие ключевую роль в системах согласования.

Перед тем как рассматривать практические примеры систем, мы познакомимся с моделями согласования и самим понятием согласования в распределенных системах. Затем мы обсудим две системы согласования: Rendezvous Bus компании TIBCO и Jini компании Sun Microsystems. Эти примеры помогут нам понять особенности многих распределенных систем согласования, существующих реально или в виде исследовательских проектов.

12.1. Знакомство с моделями согласования

Основным подходом, который используется в системах согласования, является отделение собственно вычислительных процессов от механизмов их согласования. Если рассматривать распределенную систему как набор процессов (возможно, многопоточных), то станет понятно, что вычислительная часть распределенной системы образована группой процессов, каждый из которых осуществляет конкретные вычислительные операции, причем эти операции в принципе могут выполняться независимо от других процессов.

В этой модели согласующая часть распределенной системы поддерживает все взаимодействие между процессами и организует их взаимную кооперацию. Она образует тот «клей», который связывает воедино деятельность, выполняемую разными процессами [164]. В распределенных системах согласования основное внимание уделяется согласованию процессов.

В [79] представлены результаты таксономии моделей согласования мобильных агентов, результаты которой в равной степени применимы и ко многим другим типам распределенных систем. Адаптируя приведенную там терминологию к распределенным системам в целом, можно разделить модели по двум измерениям — времени и ссылкам, как показано на рис. 12.1.

		Время	
		Связное	Несвязное
Ссылки	Связные	Прямое согласование	Согласование через почтовый ящик
	Несвязные	Согласование на встрече	Генеративная связь

Рис. 12.1. Таксономия моделей согласования

В том случае, если процессы обладают связностью ссылок и времени, согласование осуществляется напрямую. Назовем его *прямым согласованием* (*direct coordination*). Связность ссылок обычно имеет вид явной идентификации собеседника в процессе взаимодействия. Так, например, процесс может взаимодействовать с другим процессом только в том случае, если он знает идентификатор процесса, с которым хочет обменяться информацией. Временная связность означает, что оба взаимодействующих друг с другом процесса активны одновременно. Такая связность имеет место при нерезидентной связи на основе сообщений, которую мы рассматривали в главе 2.

Другой тип согласования наблюдается в том случае, если процессы не связаны по времени, но связаны по ссылкам. Мы будем называть его *согласованием через почтовый ящик* (*mailbox coordination*). В этом случае для взаимодействия вовсе не нужно, чтобы два взаимодействующих друг с другом процесса выполнялись одновременно. Вместо этого взаимодействие происходит путем отправки сообщений в почтовый ящик, может быть, используемый совместно. Эта ситуация аналогична сохранной связи на основе сообщений, которую мы также рассмат-

ривали в главе 2. При этом необходимо явно указать адрес почтового ящика, в котором должны храниться сообщения. Это, собственно, и есть ссылочная связность.

Комбинация связности по времени и несвязности по ссылкам образует группу моделей *согласования на встрече* (*meeting-oriented coordination*). В несвязной по ссылкам системе процессы не имеют полной информации друг о друге. Другими словами, когда процесс хочет согласовать свою деятельность с другими процессами, он не может обратиться к ним напрямую. Взамен используется метафора встречи, на которой собираются процессы, чтобы скоординировать свою деятельность. Модель предполагает, что процессы, участвующие во встрече, выполняются одновременно.

Системы согласования на встрече часто реализуются на базе событий, схожих с теми, которые используются в распределенных системах объектов. В этой главе мы рассмотрим другой механизм реализации встреч, реализуемый так называемыми *системами публикации/подписки* (*publish/subscribe systems*). В этих системах одни процессы могут подписываться на сообщения, содержащие информацию по определенной теме, а другие процессы — публиковать (то есть создавать) такие сообщения. Большинство систем публикации/подписки требуют, чтобы взаимодействующие процессы были активны одновременно, следовательно, они связаны по времени. Однако взаимодействующие таким образом процессы могут оставаться неизвестными друг другу.

Наиболее широко известный вариант согласования — это сочетание несвязных по времени и по ссылкам процессов, представленный *генеративной связью* (*generative communication*), которая впервые была реализована в программной системе Linda [163]. Основная идея генеративной связи состоит в том, что набор независимых процессов может использовать разделяемое сохранное пространство данных, организуемое с помощью кортежей. Кортежи — это именованные записи, содержащие несколько (но, возможно, и ни одного) типизированных полей. Процесс может помещать в разделяемое пространство данных записи любого типа (то есть генерировать связующие записи). В отличие от электронных досок объявлений, в этом случае нет необходимости точно задавать структуру кортежей. Для разделения кортежей в соответствии с информацией, которая в них содержится, достаточно их имен.

Интересная особенность этих разделяемых пространств имен состоит в том, что они реализуют механизмы ассоциативного поиска кортежей. Другими словами, когда процесс хочет извлечь кортеж из пространства данных, ему достаточно определить значения полей, которые его интересуют. Любой кортеж, удовлетворяющий описанию, будет извлечен из пространства данных и передан процессу. Если ничего найдено не будет, процесс может заблокироваться до прихода очередного кортежа. Мы отложим детальное описание этой модели согласования до обсуждения системы Jini. В [106] представлен обзор моделей генеративной связи в контексте приложений для Интернета.

Большинство исследований по вопросам согласования в основном посвящено программным моделям и соответствующим языкам [341]. Более поздние иссле-

дования относятся к следующему уровню абстракции — архитектуре программ и систем [41, 161]. Один из важных вопросов в архитектуре программ и систем — отделение компонентов от механизмов их взаимодействия. Это отделение приводит к понятию архитектурного стиля и языков описания архитектуры. Дополнительную информацию по этим темам можно найти в [289, 411].

12.2. TIB/Rendezvous

В качестве первого примера распределенной системы согласования мы рассмотрим систему TIB/Rendezvous, разработанную компанией TIBCO. Обзор этой системы можно найти в [460]; детали ее строения описаны в разнообразных технических руководствах, входящих в комплект поставки.

12.2.1. Обзор

Система TIB/Rendezvous [323, 423] изначально была описана в понятиях *информационной шины (information bus)*, минимальной коммуникационной системы группы процессов, основанной на следующих принципах. Во-первых, степень зависимости коммуникационной системы от приложений ядра очень низка. Так, например, из ядра полностью исключена сложная семантика упорядочения сообщений, поскольку предполагается, что эти вопросы решаются на прикладном уровне [99]. Мы касались этого вопроса в главе 5. Точно так же в базовой системе отсутствует встроенная поддержка атомарных транзакций. Как мы увидим, эта поддержка может быть реализована при помощи дополнительной службы.

Второй принцип построения системы состоит в том, что сообщения описывают себя сами. На практике это означает, что приложение может проверить входящее сообщение, чтобы определить, какова его структура и какие данные оно содержит. Заметим, что в противоположность этому правилу в большинстве коммуникационных систем предполагается, что процесс уже осведомлен о формате входящих сообщений и ему остается лишь верно интерпретировать их содержание.

Третий принцип построения заключается в том, что процессы не имеют ссылочной связности. Причина введения этого принципа вызвана тем, что обслуживание работающей системы не должно вести к ее остановке, а также необходимо упростить добавление новых процессов «на лету». Эти требования проще выполнить в том случае, если процессы не ссылаются явным образом друг на друга. Ссылочная несвязность обеспечивается путем использования адресации по теме, о которой мы поговорим далее.

Модель согласования

Система TIB/Rendezvous основана на такой модели согласования, когда процессы в основном не обладают ссылочной связностью, но могут получать ее на короткий срок. Как говорилось выше, эта модель напоминает согласование на встрече. Кроме того, система предоставляет также службу, при помощи которой

процессы получают возможность оставаться несвязными по времени. Детали этой модели будут рассмотрены позже.

Ключевой механизм, лежащий в основе модели согласования TIB/Rendezvous, — это *адресация по теме (subject-based addressing)*. При таком подходе процесс, который собирается послать сообщение, не может определить точное место назначения. Вместо этого он именует сообщение *названием темы (subject name)*, после чего посылает его в коммуникационную систему для пересылки по сети.

Получатели, в свою очередь, не выясняют, от каких процессов они собираются получать сообщения. Вместо этого они сообщают коммуникационной системе, какие темы их интересуют. Коммуникационная система гарантирует, что получателю будут доставлены только те сообщения, которые несут данные по теме, интересующей получателя.

Отправка сообщения методом адресации по теме также называется *публикацией (publishing)*. Чтобы получить сообщение по определенной теме, процесс должен подписаться на эту тему. Принципы организации систем публикации/подписки, использующих метод адресацию по теме, показаны на рис. 12.2.

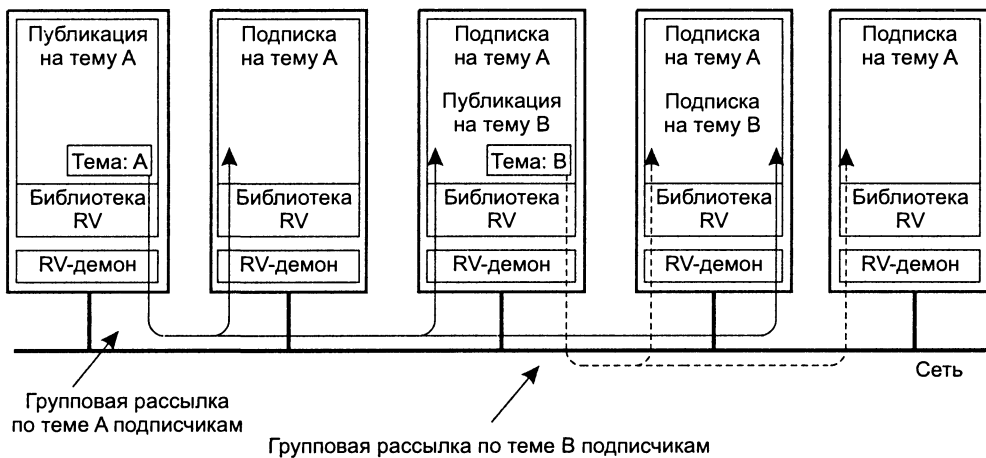


Рис. 12.2. Принципы организации системы публикации/подписки, реализованной в TIB/Rendezvous

Архитектура

Архитектура системы TIB/Rendezvous относительно проста. Ее основой является сеть с поддержкой групповой рассылки, хотя по возможности допускается использование более эффективных средств связи. Так, например, если известно точное местонахождение подписчика, обычно выполняется сквозная передача сообщений. На каждом узле этой сети работает *демон контактов (rendezvous daemon)*, который отвечает за то, чтобы сообщения отсылались и получались в соответствии с темой. Когда сообщение публикуется, демон контактов осуществляет его групповую рассылку всем узлам сети. Обычно групповая рассылка реализуется

ся средствами базовой сети, такими как групповая рассылка по IP-адресам или аппаратная широковещательная рассылка.

Процессы, подписываясь на определенную тему, передают свою подписку локальному демону. Демон создает таблицу пар (*процесс, тема*) и при доставке сообщения с темой S просто просматривает эту таблицу в поисках локальных подписчиков, пересылая его каждому из процессов. Если на тему S на данном узле никто не подписался, сообщение немедленно уничтожается.

Чтобы система могла вырасти до размера больших сетей, таких как сети WAN, используются также *маршрутизирующие демоны контактов* (*rendezvous router daemons*). Обычно каждая локальная сеть имеет одного маршрутизирующего демона контактов. Этот демон связывается с маршрутизирующим демоном другой удаленной сети, как показано на рис. 12.3. Маршрутизирующие демоны образуют сквозную *оверлейную сеть* (*overlay network*), в которой маршрутизаторы соединены между собой попарно соединениями TCP. Вторичная сеть — это сеть маршрутизаторов прикладного уровня. Мы уже говорили о сетях такого типа, когда обсуждали системы очередей сообщений в главе 2.

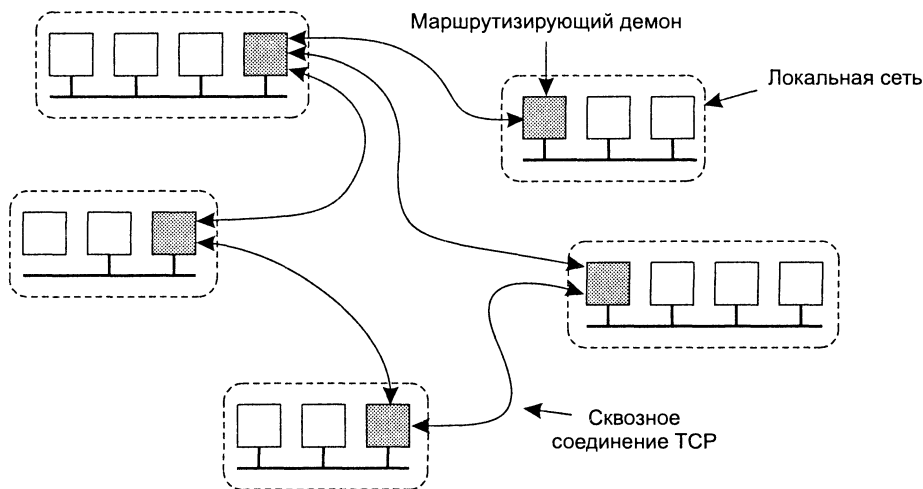


Рис. 12.3. Общая архитектура глобальной сети TIB/Rendezvous system

Каждый маршрутизатор знает топологию вторичной сети и вычисляет дерево групповой рассылки для публикации сообщений в других сетях. Маршрутизаторы рассылают только те сообщения, которые публикуются в их локальной сети. Сообщения из других сетей пересылаются по дереву групповой рассылки той сети, из которой они первоначально исходили.

В целях повышения производительности всякий раз при публикации сообщения оно доставляется только в те удаленные сети, которые были сконфигурированы на прием подобных сообщений и имеют текущих подписчиков этих сообщений. Детали такого рода ограничений мы рассмотрим ниже.

12.2.2. Связь

Основой системы TIB/Rendezvous являются ее средства связи. Мы уже познакомились с работой механизма передачи сообщений, а теперь рассмотрим детали его функционирования и изучим, в частности, вопросы поддержки глобальной связи. Идеология публикации/подписки хорошо сочетается с программированием на базе событий, которое TIB/Rendezvous поддерживает. О событиях мы также поговорим в этом разделе.

Основы передачи сообщений

Как мы говорили, связь в TIB/Rendezvous осуществляется путем отправки и получения самоопределяющихся сообщений с использованием адресации по теме. Каждое сообщение содержит несколько (или ни одного) полей, динамически созданных процессом-отправителем. Сами поля представляют собой записи с атрибутами, перечисленными в табл. 12.1.

Таблица 12.1. Атрибуты полей сообщения TIB/Rendezvous

Атрибут	Тип	Описание
Name	String	Имя поля, возможно, NULL
ID	Integer	Уникальный номер поля в сообщении
Size	Integer	Общий размер поля (в байтах)
Count	Integer	Число элементов в случае массива
Type	Константа	Константа, идентифицирующая тип данных
Data	Любой тип	Некоторые данные, хранящиеся в поле

Каждое поле обычно имеет строковый атрибут Name, который представляет собой просто имя поля. Различные поля одного сообщения могут иметь одинаковые имена. Можно также при помощи атрибута ID сопоставить каждому полю сообщения отдельный идентификатор поля. Идентификатор поля — это 2-байтовое целое число, которое должно быть уникальным для каждого поля сообщения.

Общий размер поля, выраженный в байтах, задается при помощи атрибута Size. В том случае, если поле содержит массив, в атрибуте Count хранится число элементов массива.

Атрибут Type — это константа, соответствующая одному из простых типов данных TIB/Rendezvous. Если в поле хранится массив, атрибут Type содержит тип элементов массива. TIB/Rendezvous предоставляет также средства для обмена данными с произвольным пользовательским типом. И наконец, атрибут Data содержит собственно данные, хранящиеся в поле.

Каждое сообщение может содержать произвольное число полей. Перед тем как сообщение будет отослано, необходимо указать его тему, чтобы операции отбора правильно обрабатывали это сообщение. Сама по себе тема выглядит как еще одна строка символов. Также можно указать, что темой сообщения является ответ на другое сообщение. Это название темы может быть использовано полу-

чателем для ответа процессу, отправившему исходное сообщение. Разумеется, отправитель считается подписанным на ответы на свое сообщение.

В целях повышения производительности каждый процесс может использовать специально созданное имя темы, называемое *именем входа (inbox name)*. Это имя зависит от процесса. Если издатель P использует при публикации сообщения имя входа другого процесса Q , это сообщение будет послано Q напрямую средствами сквозной связи, а не групповой рассылки.

Для отправки и приема сообщений процессы создают так называемые транспорты. *Транспорт (transport)* — это локальный объект, подобный сокетам Беркли в UNIX, о которых мы говорили в главе 2. Транспорт используется для отправки сообщений демонам контактов, прослушивающим определенный порт, по одному из стандартных протоколов связи. Так, например, существуют различные транспорты для широковещательных рассылок, групповых рассылок канального уровня, групповых рассылок по IP-адресам.

Транспорт в состоянии выполнять лишь три простейших операции. Операция `send` передает сообщение локальному демону контактов, который, в свою очередь, отвечает за корректную ее публикацию. Операция `send` является неблокирующей.

Также неблокирующей является операция `sendreply`, которая может вызываться в ответ на получение ответного сообщения, о котором мы говорили чуть выше.

И, наконец, существует блокирующая операция `sendrequest`, которая посылает локальному демону контактов сообщение для передачи по базовой сети и блокируется до получения ответного сообщения. Ответ передается на имя входа (созданное динамически) средствами сквозной связи.

Отметим, что специальной операции `receive` не существует. Как мы покажем далее, ответ на сообщение обычно обрабатывается при помощи стандартных средств обработки сообщений. Когда приходит сообщение по той теме, на которую процесс подписан, система событий TIB/Rendezvous вызовет функцию обратного вызова, предоставляемую подписавшимся процессом. Детали подобного подхода мы рассмотрим позже.

События

Для того чтобы получать сообщения по темам, на которые данный процесс был подписан, подписчик использует механизм событий. В принципе другого способа обрабатывать входящие сообщения нет. Операция `sendrequest` представляет собой исключение из этого правила.

Подписка на тему осуществляется посредством создания *слушателя событий (listener event)*. Слушатель событий представляет собой локальный объект, соответствующий способу доставки и теме, на которую подписан создавший его процесс. Слушатель событий также содержит ссылку на функцию обратного вызова, которую предоставляет подписчик. Эта функция вызывается при *диспетчеризации (dispatching)* события. Диспетчеризация событий для входящих сообщений происходит в соответствии со схемой на рис. 12.4.

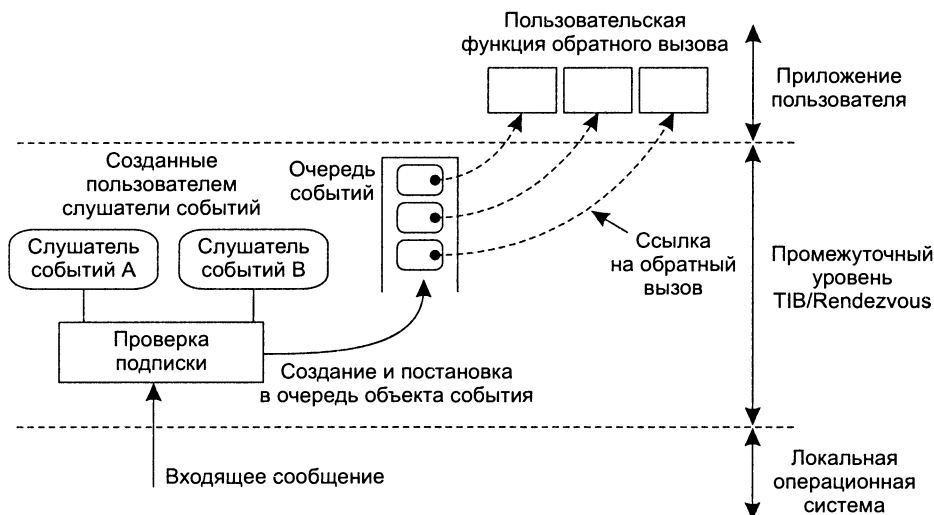


Рис. 12.4. Обработка подписок слушателем событий в TIB/Rendezvous

Когда приходит сообщение, тема которого соответствует теме, указанной в слушателе событий порта, создается и включается в локальную очередь событий системы TIB/Rendezvous *объект события (event object)*. Такой объект события содержит ту же информацию, что и слушатель событий. Последний используется для прослушивания большинства входящих сообщений. Каждая очередь событий имеет как минимум один соответствующий ей поток диспетчеризации, который удаляет событие из верхушки очереди, после чего вызывает нужную функцию обратного вызова. При вызове этой функции передается входящее сообщение, соответствующее прошедшему диспетчеризацию событию.

Если приходит очередное сообщение с темой, указанной в слушателе событий, то создается и добавляется в локальную очередь другой объект события. При этом не имеет значения, ожидает поток диспетчеризации прихода нового объекта события или занят обработкой предыдущего. Чтобы отменить подписку, подписанный процесс должен лишь удалить соответствующий слушатель событий. Любые ожидающие обработки объекты событий, ассоциированные с этим слушателем, также будут уничтожены. Рисунок 12.5 иллюстрирует процедуру обработки трех входящих сообщений, соответствующих одному слушателю событий. Каждая горизонтальная линия соответствует временному интервалу, занимаемому подпиской или обработкой сообщения.

Здесь следует сделать важное замечание. Если входящее сообщение имеет тему, указанную в двух или более слушателях событий, то каждый из них создает соответствующий объект события и добавляет его в очередь. Другими словами, одно входящее сообщение ведет к появлению нескольких объектов событий.

По умолчанию для обработки всех событий используется одна очередь. Однако, чтобы управлять обработкой событий более аккуратно, можно создать несколько очередей событий и явно распределить события по этим очередям. Наличие

нескольких очередей событий позволяет обрабатывать объекты событий независимо друг от друга, поскольку каждая из очередей имеет собственный поток диспетчеризации.

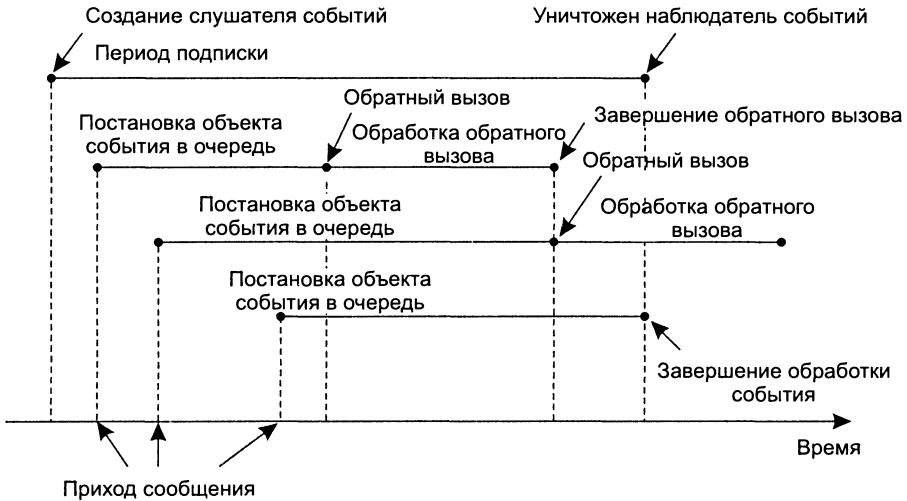


Рис. 12.5. Обработка входящих сообщений в TIB/Rendezvous.

Кроме того, можно сгруппировать несколько очередей событий в *группу очередей* (*queue group*). Каждой очереди в группе можно назначить приоритет. Диспетчер группы очередей будет постоянно отслеживать объекты событий, попадающие в очередь с максимальным приоритетом. Если эта очередь пуста, он займется объектами событий из следующей по приоритету очереди. На рис. 12.6, а показана группа из четырех очередей событий. Для того же набора объектов событий на рис. 12.6, б приведена семантически эквивалентная группе единая очередь событий.

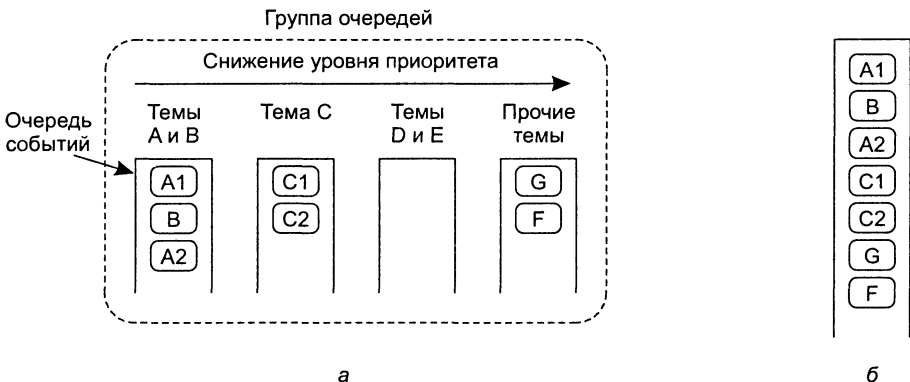


Рис. 12.6. Приоритезация событий в группе очередей событий (а). Очередь событий, семантически эквивалентная указанной группе очередей (б)

Если ни одного объекта событий нет, диспетчер блокируется. При обращении к функции обратного вызова для выбранного объекта события диспетчер остается активным до завершения вызова. Он не прекращает работу и в том случае, если в группу очередей добавляется объект события с приоритетом более высоким, чем у обрабатываемого объекта.

12.2.3. Процессы

Базовая организация системы TIB/Rendezvous довольно проста. На каждом хосте обычно работает один демон контактов, который, как было сказано выше, отвечает за все локальное сетевое взаимодействие. При необходимости демон контактов заменяется маршрутизирующим демоном, который отвечает за весь глобальный трафик. Маршрутизирующий демон имеет расширенную, по сравнению с демоном контактов, функциональность.

Каждая программа клиента выполняется на его хосте в виде отдельного процесса. Все взаимодействие осуществляется при помощи библиотеки функций, которая скомпонована с приложением клиента и, в свою очередь, пересылает сообщения локальному демону. Детальное описание этой библиотеки и подробности интерфейса программ на языке C можно найти в [459].

12.2.4. Именование

До сих пор в этой книге мы считали имена средством обращения к сущностям, которые созданы и управляются распределенной системой. Типичными примерами таких имен были имена ресурсов, процессов, файлов, сетевых соединений и т. п.

Основная особенность системы публикации/подписки, такой как TIB/Rendezvous, состоит в адресации имен тем. Именование с целью адресации в этом случае несет в себе несколько иной смысл. Имена тем непосредственно не указывают на сущности, как на особые образования, находящиеся где-то в системе и требующие обработки. В этом случае имена требуются, чтобы связать группу процессов-отправителей (издателей) с группой процессов-получателей (подписчиков).

В TIB/Rendezvous имя темы — это последовательность строковых меток, разделенных точками, похожее на DNS-имя. Имя темы должно всегда начинаться и заканчиваться меткой, пустые метки недопустимы. Имя темы ограничено 255 символами, а значит, метка может иметь длину до 252 символов. Примеры правильных и неправильных имен тем представлены в табл. 12.2.

Таблица 12.2. Примеры правильных и неправильных имен тем

Пример	Правильность
Books.ComputerSystems.DistributedSystems	Да
.ftp.cs.vu.nl	Нет, поскольку начинается с точки
ftp.cs.vu.nl	Да
NEWS.res.comp.os	Да

Пример	Правильность
Maarten..vanSteen	Нет, поскольку имеется пустая метка
Maarten.R.vanSteen	Да

Для указания набора тем сообщений, которые хотели бы получать подписчики, могут использоваться символы-заменители. Имеется два символа-заменителя. Звездочка (*), указанная вместо метки, обозначает, что на этом месте может находиться любая метка. Символ «больше» (>) означает, что в оставшейся части имени темы могут находиться любые метки. Отметим, что оба символа-заменителя указываются вместо меток, частями меток они быть не могут. Примеры символов-заменителей представлены в табл. 12.3.

Таблица 12.3. Примеры символов-заменителей в именах тем

Имя с символом-заменителем	Реальное имя темы
*.cs.vu.nl	ftp.cs.vu.nl; www.cs.vu.nl
nl.vu.	nl.vu.cs.ftp; nl.vu.cs.zephyr; nl.vu.few.www
NEWS.comp.*.books	NEWS.comp.os.books; NEWS.comp.ai.books; NEWS.comp.se.books; NEWS.comp.theory.books

Маршрутизирующие демоны можно сконфигурировать так, чтобы они фильтровали входящие и исходящие сообщения в соответствии с символами-заменителями. С точки зрения входящих сообщений это выглядит так, словно маршрутизирующий демон подписан на группу тем и пропускает в локальную сеть только сообщения с темами, входящими в эту группу. Таким образом, если клиентская программа подписана на все темы (в качестве имени темы указан символ >), а маршрутизирующий демон принимает только новостные сообщения (в качестве имени темы указана строка NEWS.>), то и клиент будет получать только новостные сообщения.

Таким же образом могут быть ограничены и исходящие сообщения. В результате маршрутизирующий демон подписывается на группу исходящих сообщений. Каждое сообщение, удовлетворяющее условиям подписки, передается маршрутизатором в удаленные сети. Все прочие сообщения остаются в локальной сети.

Подобная двусторонняя фильтрация входящих и исходящих сообщений может существенно снизить трафик в глобальных сетях. По своей природе двусторонняя фильтрация напоминает алгоритмом лавинной маршрутизации протокола рассылки новостей NNTP [36, 223]. Отметим, что в подобной схеме групповой рассылки предполагается, что темы известны заранее. Решение состоит в том, чтобы позволить издателям и подписчикам вместо предопределенных тем использовать содержимое письма, как описано в [35].

12.2.5. Синхронизация

Ядро TIB/Rendezvous предоставляет минимальную поддержку синхронизации процессов. Единственное, что хоть как-то гарантируется, — доставка сообщений от

одного источника в порядке FIFO, то есть тот факт, что сообщения с одним и тем же транспортным протоколом будут доставлены в порядке их отправления.

Вдобавок к ядру TIB/Rendezvous имеет выделенную службу транзакций, которая поддерживает *транзакционный обмен сообщениями* (*transactional messaging*). Для транзакционного обмена сообщениями посылаемые и получаемые сообщения можно рассматривать как части транзакции. Транзакционный обмен сообщениями реализуется в виде отдельного уровня поверх ядра системы TIB/Rendezvous, как показано на рис. 12.7. Транзакции всегда производятся с операциями, относящимися к одному процессу. Объединение в одной транзакции операций, относящихся к разным процессам, на уровне транзакций не поддерживается, для этого требуется выделенный менеджер транзакций, подобный описанному в главе 7.

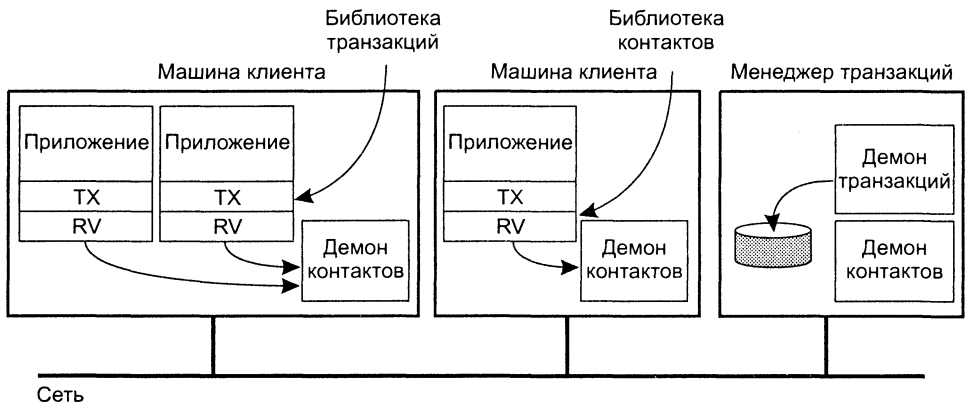


Рис. 12.7. Организация транзакционного обмена сообщениями в отдельном уровне TIB/Rendezvous

Важную роль в транзакциях играет *демон транзакций* (*transaction daemon*), который отвечает в основном за сохранение опубликованных сообщений до момента их доставки всем подписчикам. Может существовать несколько демонов транзакций, каждый из которых отвечает за часть полного набора тем. Сохраняются только те сообщения, которые являются частью транзакции. Далее мы поясним эту модель.

Процесс *P*, как показано на рис. 12.8, может объединить несколько операций издания и подписки в одной транзакции. Транзакция начинается посылкой нескольким демонам транзакций сообщения, в котором им предлагается открыть сеанс с процессом *P* для отправки и приема сообщений в качестве части транзакции. После этого, когда *P* публикует сообщение по теме *S*, оно пересылается демону, отвечающему за тему *S*, где и хранится до тех пор, пока *P* либо подтвердит, либо прервет транзакцию. Отметим, что до подтверждения транзакции опубликованные сообщения не передаются другим процессам.

Сообщения *подтвержденной* транзакции, на которые подписан процесс *P*, пересылаются демоном транзакций процессу *P* обычным порядком. Как мы уже го-

ворили, демоном сохраняются сообщения еще не подтвержденных транзакций. Когда *P* получает сообщение, он уведомляет об этом демона транзакций, который его хранит. Это уведомление гарантирует, что *P* снова не получит то же самое сообщение после подтверждения транзакции.

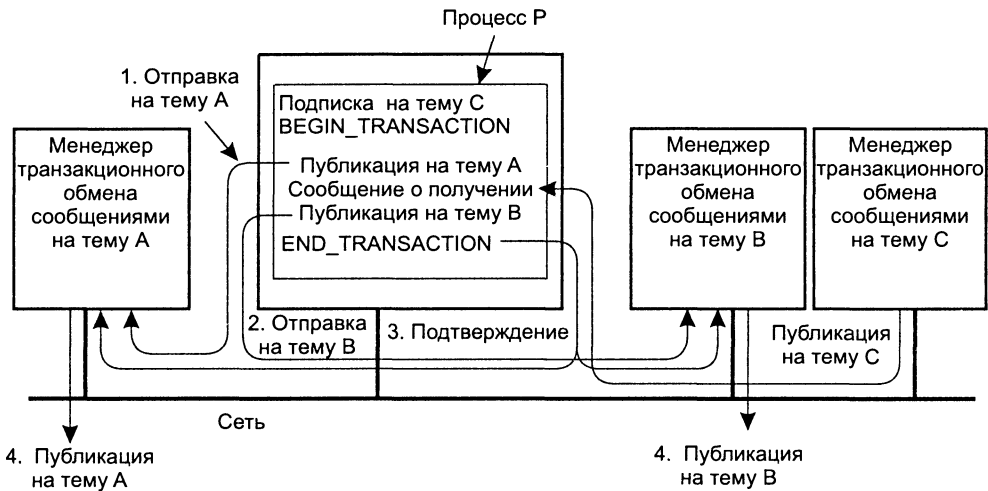


Рис. 12.8. Транзакции в TIB/Rendezvous

Подписчики получают доступ к сообщениям, пересылавшимся в транзакции, только после ее подтверждения. При прерывании транзакции демон транзакций просто отбрасывает посланные сообщения, и их публикация на этом завершается. Сообщение из подтвержденной транзакции передается подписчикам. Если подписчиков нет (неважно, принимали они участие в транзакционном обмене сообщениями или нет), сообщение отбрасывается. Сообщение сохраняется демоном до тех пор, пока его не получат все подписчики. Новые подписчики могут добавляться, пока опубликованное сообщение хранится демоном и пока не подтверждена транзакция, соответствующая этому сообщению.

Если в транзакцию вовлекается несколько демонов транзакций, TIB/Rendezvous использует для принятия решения о подтверждении или прерывании транзакции протокол двухфазного подтверждения.

Транзакционный обмен сообщениями может сочетаться с операциями с базами данных, а также с обычным обменом сообщениями (без использования транзакций). Такая гибкость может усложнить понимание семантики транзакций в TIB/Rendezvous. Мы не будем вдаваться в детали, их можно найти в [458].

12.2.6. Кэширование и репликация

Никаких специальных средств кэширования или репликации в системе TIB/Rendezvous не предусмотрено. Решение этих вопросов полностью возложено на приложения. Так, в частности, необходимо предпринимать особые меры против

двойной публикации одного и того же сообщения репликами одного процесса. TIB/Rendezvous не в состоянии обнаружить, что все эти сообщения — копии одного и того же сообщения и все они доставляются подписчикам.

Для поддержания новых подписчиков можно создать фоновый процесс, который подобно серверу будет кэшировать последние n сообщений по определенной теме. Когда процесс начнет работу в системе, он может попросить кэширующий сервер предоставить ему последние сообщения по определенной теме, чтобы быстрее понять, о чем идет речь.

12.2.7. Отказоустойчивость

В TIB/Rendezvous предполагаются два способа поддержания отказоустойчивости. Во-первых, система предоставляет средства надежной связи, которые гарантируют, что публикуемое сообщение не будет потеряно из-за отказов связи. Во-вторых, система для предотвращения отказов поддерживает группы процессов. Далее мы рассмотрим эти механизмы более подробно.

Надежная связь

В TIB/Rendezvous предполагается, что коммуникационные механизмы базовой сети ненадежны. Чтобы компенсировать эту ненадежность, демон контактов, отправивший сообщение другому демону, должен сохранять сообщение еще как минимум 60 с. При отправке сообщения демон добавляет к нему последовательный номер (от темы этот номер не зависит). Принимающий сообщение демон может, просматривая последовательные номера, обнаружить, что какое-то сообщение пропало (напомним, что сообщения посылаются всем демонам). Если обнаружится, что одно из сообщений пропало, отправившему его демону посылается просьба повторить пересылку этого сообщения.

Такая форма надежной связи не исключает потерю сообщений. Так, например, если принимающий демон запрашивает повторную отправку сообщения, которое было опубликовано более чем 60 с назад, отправитель обычно не в состоянии восстановить это сообщение. В нормальных условиях издатель и подписчик должны быть уведомлены о подобной ошибке связи. Обработка ошибки возлагается на приложения.

Значительно больше в вопросе обеспечения надежной связи TIB/Rendezvous полагается на те механизмы, которые предоставляет базовая сеть. С недавних пор TIB/Rendezvous предоставляет надежную групповую рассылку на базе групповой рассылки (ненадежной) по IP-адресам. Для этого в TIB/Rendezvous используется протокол групповой рассылки транспортного уровня, известный как *прагматическая общая групповая рассылка (Pragmatic General Multicast, PGM)*, описанный в [427]. Мы кратко рассмотрим его.

PGM не дает абсолютной гарантии того, что в процессе групповой рассылки сообщение рано или поздно дойдет до каждого получателя. На рис. 12.9, *a* показана ситуация, когда сообщение рассылается по всему дереву, но до двух получателей не доходит. Идея PGM состоит в том, что получатель сам обнаруживает потерю сообщений. Если таковые обнаружатся, отправителю посылается запрос

на повторную отправку. Этот запрос посылается обратно по ветвям дерева групповой рассылки, в корне которого находится отправитель, как это показано на рис. 12.9, б. Когда запрос на повторную отправку достигает промежуточного узла, может обнаружиться, что узел хранит запрашиваемое сообщение в своем кэше. В этом случае он и совершает повторную отправку. В противном случае узел просто пересылает запрос следующему узлу на пути к отправителю. Если ни один из промежуточных узлов не смог выполнить повторную отправку сообщения, это, в конце концов, делает отправитель.

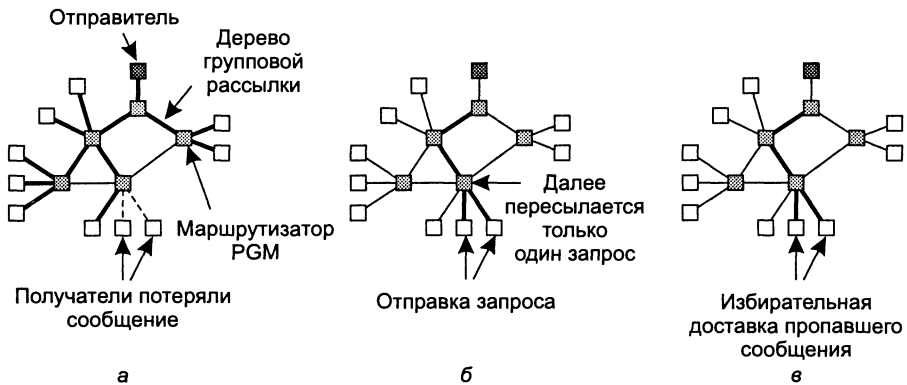


Рис. 12.9. Принцип работы PGM. Сообщение посылается по дереву групповой рассылки (а). Маршрутизатор пересылает только один запрос на каждое сообщение (б). Сообщение заново отправляется только тем получателям, которые этого требуют (в)

PGM выполняет некоторые операции, призванные обеспечить масштабируемость надежной групповой рассылки. Во-первых, промежуточные узлы, получающие несколько запросов на повторную отправку одного и того же сообщения, пересылают отправителю только один запрос. Это позволяет защитить отправителя от «обвала» сообщений об ошибке. Мы уже говорили об этой проблеме в главе 7, когда обсуждали проблемы масштабируемости для надежной групповой рассылки.

Второе, что делает PGM, — запоминает путь, по которому проходит запрос от получателя до отправителя, как показано на рис. 12.9, в. Когда отправитель, наконец, посылает запрошенное сообщение во второй раз, PGM заботится о том, чтобы оно попало только к тем получателям, которые отправили запрос. Таким образом, получателям, уже получившим это сообщение, лишнее сообщение не причинит неудобств.

Помимо надежной базовой схемы и надежной групповой рассылки по протоколу PGM система TJB/Rendezvous может обеспечить дополнительные меры надежности путем *доставки сертифицированных сообщений (certified message delivery)*. В этом случае процесс использует для отправки или получения сообщений отдельный транспорт, имеющий встроенный *журнал регистрации (ledger)* отосланных и принятых сертифицированных сообщений. Процесс, собирающийся принимать сертифицированные сообщения, регистрирует себя у отправителя

этих сообщений. Регистрация позволяет транспорту обеспечить дополнительные механизмы надежности, которых не имеют демоны контактов. Большая часть этих механизмов скрыта от приложений и обеспечивается средствами транспорта.

Если журнал регистрации реализован в виде файла, становится возможным обеспечивать надежную доставку сообщений даже при отказах процессов. Так, например, при отказе принимающего процесса все сообщения, которые он пропустил до момента восстановления, будут сохранены в журнале регистрации отправителя. После восстановления получатель просто связывается с журналом регистрации и требует повторной передачи потерянных сообщений.

Отказоустойчивые группы процессов

Чтобы иметь возможность маскировать отказы процессов, TIB/Rendezvous использует простое средство автоматической активизации или деактивации процессов. В данном контексте процесс считается активным, если он обычным образом отвечает на все приходящие сообщения, и неактивным в противном случае. Коротко говоря, неактивный процесс — это работающий процесс, который может обрабатывать только некоторые события.

Процессы могут быть собраны в группу, в которой каждый процесс имеет соответствующий ему уникальный ранг. Ранг процесса в группе определяется весом, назначенным ему вручную, причем два процесса в одной группе не могут иметь одинаковый ранг. Каждой группе TIB/Rendezvous разрешено иметь несколько (в зависимости от группы) активных процессов, которые называются *активной целью* (*active goal*) группы. Во многих случаях активная цель ограничивается единственным процессом, а все взаимодействие в группе обеспечивается протоколом на базе первичной копии, о котором мы говорили в главе 6.

Активный процесс регулярно посылает сообщение прочим членам группы, чтобы объявить, что он жив и работает. Если очередное сообщение от процесса не пришло, промежуточный уровень TIB/Rendezvous автоматически активизирует процесс, имеющий самый высокий ранг из тех, которые были неактивны. Активизация сопровождается оповещением о действиях, которые должен реализовать каждый из членов группы. Точно так же, если ранее не функционировавший процесс восстанавливается после сбоя и становится активным, активный процесс с минимальным рангом автоматически деактивируется.

Чтобы сохранить непротиворечивость активных процессов, необходимо особым образом «пробуждать» неактивные процессы перед их активизацией. Простой выход из положения состоит в том, чтобы подписать неактивный процесс на все те сообщения, которые получают прочие члены группы. Он будет обрабатывать входящие сообщения обычным образом, но без публикации ответов. Отметим, что эта схема близка к активной репликации, о которой мы говорили в главе 6.

12.2.8. Защита

Защита в TJB/Rendezvous сведена к организации защищенных каналов между издателем и подписчиком. В результате такого подхода утрачивается одна из

важных черт систем публикации/подписки, а именно отсутствие ссылочной связности между отправителями и получателями. Как мы говорили ранее, появление связности сводит всю модель к прямому согласованию. При выполнении взаимной аутентификации, как это происходит в случае защищенных каналов, явная связность отправителя и получателя неизбежна.

Организация защищенных каналов в TIB/Rendezvous начинается с публикации зашифрованных данных. Публикуемые данные содержат удостоверение отправителя, которое создает основу для переговоров между отправителем и получателем. Если сообщение было отправлено нескольким подписчикам, каждый из них должен организовать с отправителем защищенный канал. Ключ, используемый для расшифровки сообщения, одинаков для всех подписчиков.

Организация защищенного канала между, скажем, Алисой и Бобом основана на обмене ключами по Диффи–Хеллману в сочетании с системой открытых ключей (рис. 12.10). Мы предполагаем, что Алиса и Боб уже получили сертификаты, содержащие открытые ключи друг друга. Как показано на рисунке, проверка этих сертификатов является частью протокола организации защищенного канала. Кроме того, мы предполагаем, что Алиса и Боб, используя протокол обмена ключами Диффи–Хелмана, создали общий секретный ключ $K_{A,B}$. Мы описывали эту процедуру в главе 8.

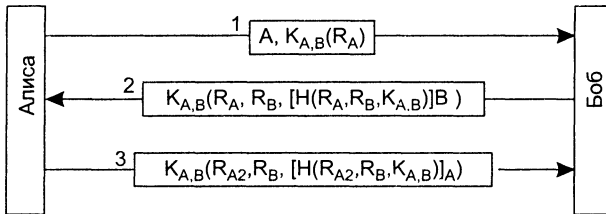


Рис. 12.10. Организация защищенного канала в TIB/Rendezvous

Чтобы организовать защищенный канал, Алиса и Боб должны аутентифицировать друг друга. Алиса начинает с посылки Бобу «одноразового» (nonce) случайного числа R_A , зашифрованного общим ключом $K_{A,B}$ (сообщение 1). Для ответа Бобу нужно передать Алисе другое случайное число R_B . Он это делает, вычисляя значение хэш-функции $H(R_A, R_B, K_{A,B})$ и подписывая его своим закрытым ключом. Подписанное значение хэш-функции вместе со случайными числами R_A и R_B возвращается Алисе, причем все это шифруется ключом $K_{A,B}$ (сообщение 2).

Когда Алиса получает сообщение 2, она может определить, действительно ли оно прислано Бобом, вычислив значение хэш-функции и сверив его со значением, которое подписал Боб. Для этой цели ей необходимо проверить сертификат Боба, в котором содержится открытый ключ. Проверка сертификата сводится к проверке того, действительно ли сертификат подписан сертифицирующей организацией, которой Алиса доверяет.

Аутентифицировав Боба, Алиса должна дать ему возможность аутентифицировать себя. Для этого она использует другое «одноразовое» случайное число, R_{A2} , и составляет сообщение, аналогичное тому, которое она получила от Боба

(сообщение 3). Теперь и Боб может аутентифицировать Алису, и, таким образом, организацию защищенного канала можно считать завершенной.

12.3. Jini

Следующим примером системы согласования, который мы рассмотрим, является система Jini компании Sun Microsystems. Отнесение Jini к системам согласования основано в первую очередь на том, что эта система способна поддерживать генеративную связь при помощи Linda-подобной службы под названием JavaSpace, на которой и будет сосредоточено наше внимание. Существует множество служб и средств, которые делают Jini больше, чем просто системой согласования. Однако все это только придает должный вес выбору этой системы в качестве примера. Спецификация Jini, распространяемая Sun Microsystems, имеется в [443, 483]. Хорошее описание базовых механизмов Jini имеется в [230].

12.3.1. Обзор

Jini — это распределенная система, состоящая из разных, но взаимосвязанных элементов. Она жестко привязана к языку программирования Java, хотя многие из ее принципов равно могут быть реализованы и при помощи других языков. Важной частью системы является модель согласования генеративной связи. Прежде чем говорить об общей архитектуре системы Jini, давайте обсудим эту модель.

Модель согласования

Jini обеспечивает как временную, так и ссылочную несвязность процессов при помощи Linda-подобной системы согласования *JavaSpace* [155, 442]. *JavaSpace* — это разделяемое пространство данных, в котором хранятся кортежи. Кортежи представляют собой типизованные наборы ссылок на объекты Java. В одной системе Jini могут сосуществовать несколько пространств *JavaSpace*.

Кортежи хранятся в сериализованной форме. Другими словами, когда бы процессу ни потребовалось сохранить кортеж, сначала выполняется маршалинг кортежа, причем также подразумевается маршалинг всех его полей. В результате если кортеж содержит два разных поля, ссылающихся на один и тот же объект, то кортеж, сохраняемый в реализации *JavaSpace*, будет содержать две подвергнутые маршалингу копии этого объекта.

Кортеж помещается в пространство *JavaSpace* при помощи операции *write*, которая сначала выполняет маршалинг кортежа, а затем сохраняет его. Каждый раз при выполнении для кортежа операции *write* в *JavaSpace* сохраняется новая подвергнутая маршалингу копия этого кортежа (рис. 12.11). Мы можем ссылаться на каждую такую копию, как на *экземпляр кортежа (tuple instance)*.

Интересный аспект генеративной связи в Jini — это способ чтения экземпляров кортежа в *JavaSpace*. Чтобы прочесть экземпляр кортежа, процесс предоставляет другой кортеж и использует его как *эталон (template)*, соответствующий

считываемым экземплярам кортежа, хранящимся в JavaSpace. Как и любой другой кортеж, эталонный кортеж представляет собой типизованный набор ссылок на объекты. В JavaSpace можно прочитать только экземпляры тех кортежей, которые имеют одинаковый с эталоном тип. Поля в эталонном кортеже также содержат либо ссылки на реальные объекты, либо значение NULL.

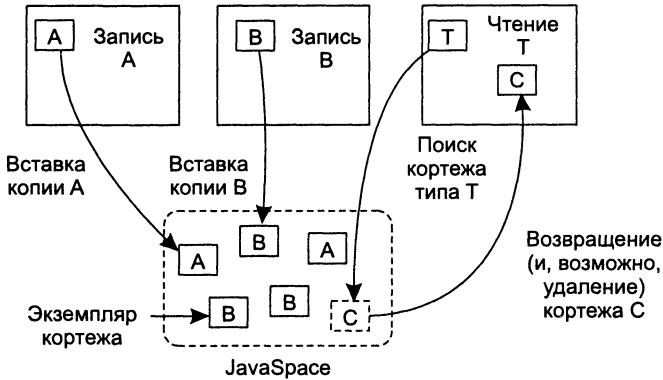


Рис. 12.11. Обобщенная организация пространства JavaSpace в Jini

Чтобы сопоставить экземпляр кортежа в JavaSpace эталонному кортежу, обычным образом выполняется маршалинг эталонного кортежа, включая маршалинг полей со значением NULL. Для каждого экземпляра кортежа того же типа, что и эталон, производится сравнение, поле за полем, с подвергнутыми маршалингу полями эталонного кортежа. Два поля совпадают, если оба они содержат копии одной и той же ссылки или если поле в эталонном кортеже равно NULL. Экземпляр кортежа совпадает с эталонным кортежем, если попарно совпадают соответствующие поля.

Когда обнаруживается экземпляр кортежа, совпадающий с эталонным кортежем (который является частью операции read), выполняется демаршалинг этого экземпляра, и он возвращается процессу, инициировавшему чтение. Для чтения может быть использована также операция take, которая заодно удаляет экземпляр кортежа из пространства JavaSpace. Обе операции блокируют вызвавший их процесс до обнаружения нужного экземпляра кортежа. Максимальное время блокирования можно предсказать. Кроме того, существуют реализации, немедленно возвращающие управление, если нужного кортежа не существует.

По сравнению с моделью публикации/подписки, используемой в TIB/Rendezvous, процессы, применяющие JavaSpace, не должны выполняться одновременно. На самом деле, если пространство JavaSpace реализовано как сохранное хранилище, всю систему Jini можно выключить и запустить снова, не потеряв ни одного экземпляра кортежа. К сожалению, реализация генеративной связи дорогостояща. По этой причине разработка высокоэффективных реализаций JavaSpace или любой другой Linda-подобной системы представляет собой немалую проблему. Обычно при попытке реализовать эту модель в глобальной сети источни-

ком проблем масштабируемости является генеративная связь. Ниже мы еще вернемся к этому вопросу.

Архитектура

JavaSpace образует лишь одну из частей системы Jini. Как и TIB/Rendezvous, Jini существует в виде компактного набора полезных средств и служб, на базе которых можно разрабатывать распределенные приложения. Распределенное приложение на базе Jini часто представляет собой свободное сообщество устройств, процессов и служб. Все взаимодействие в существующих системах Jini построено на обращениях RMI языка Java.

Архитектура системы Jini может быть представлена в виде трех уровней, как это показано на рис. 12.12. Самый нижний уровень образует инфраструктура. На этом уровне располагаются базовые механизмы Jini, в том числе и те, которые поддерживают взаимодействие посредством обращений RMI языка Java. Следует отдельно отметить одно важное свойство модели Jini: клиенты легко могут найти нужную им службу. Службы предоставляются как обычными процессами, так и устройствами, на которых программное обеспечение Jini, в том числе и виртуальная машина Java, выполняться не может. Поэтому регистрирующие и поисковые службы также относятся к инфраструктуре Jini.

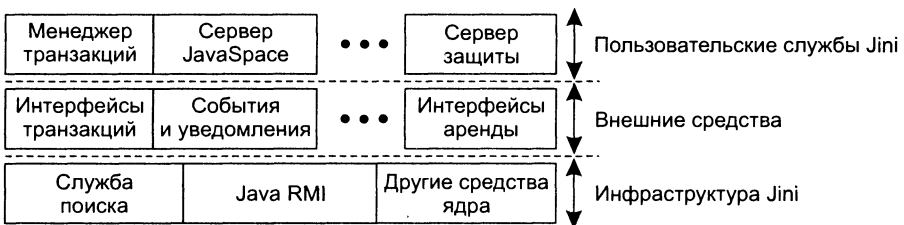


Рис. 12.12. Многоуровневая архитектура системы Jini

Второй уровень образуют средства общего назначения, которые дополняют базовую инфраструктуру и могут быть использованы для более эффективной реализации служб. В число этих средств в настоящее время входят подсистемы событий и уведомлений, средства аренды ресурсов и описания стандартных интерфейсов транзакций.

Самый верхний уровень состоит из клиентов и служб. В противоположность остальным двум уровням Jini не определяет состав этого уровня однозначным образом. В настоящее время система поддерживает несколько служб верхнего уровня, среди которых сервер JavaSpace и менеджер транзакций, реализующий интерфейсы транзакций Jini. Программам верхнего уровня, кроме того, нередко разрешается напрямую использовать механизмы инфраструктуры Jini.

Далее мы подробнее рассмотрим систему Jini, разбирая каждый из семи базовых принципов распределенных систем, которые мы рассматривали в первой части книги. При этом мы сосредоточим свое внимание на особенностях применения этих принципов в Jini, системе согласования, в которой ключевую роль играют пространства JavaSpace.

12.3.2. Связь

Как уже упоминалось, основой механизмов взаимодействия в Jini являются обращения RMI языка Java. Мы обсуждали RMI в главе 2 и повторяться не будем. Кроме генеративной связи, присущей модели JavaSpace (мы говорили об этом чуть выше), в число механизмов взаимодействия Jini входит простая подсистема событий и уведомлений, которую мы сейчас рассмотрим.

События

Модель событий Jini относительно проста. Если в рамках объекта происходит событие, которое может быть интересно клиенту, клиенту разрешается зарегистрировать себя на этом объекте. Когда факт наступления события будет зафиксирован, то есть когда событие произойдет, объект уведомит об этом зарегистрированного клиента. С другой стороны, клиент может потребовать от объекта, чтобы тот послал уведомление о наступлении события в другой процесс. В этом случае объекту пересылается удаленная ссылка на *объект-слушатель* (*listener object*), обратный вызов которого можно выполнить при наступлении события (путем обращения RMI языка Java).

Регистрация всегда арендуется. Когда срок аренды истекает, зарегистрированный клиент (или процесс, которому уведомления отправлялись по поручению клиента) перестает получать уведомления. Благодаря аренде регистрация не может сохраниться вечно, например, в результате отказа зарегистрированного клиента. Мы обсудим аренду в Jini чуть позже.

Уведомление о событии реализуется объектом путем удаленного вызова объекта-слушателя, зарегистрированного для этого события. При следующем наступлении события объект-слушатель вызывается снова. Поскольку система Jini сама по себе не предоставляет никаких гарантий того, что уведомление о событии будет доставлено объекту-слушателю, уведомления обычно имеют порядковые номера, чтобы объект-слушатель имел представление об очередности событий.

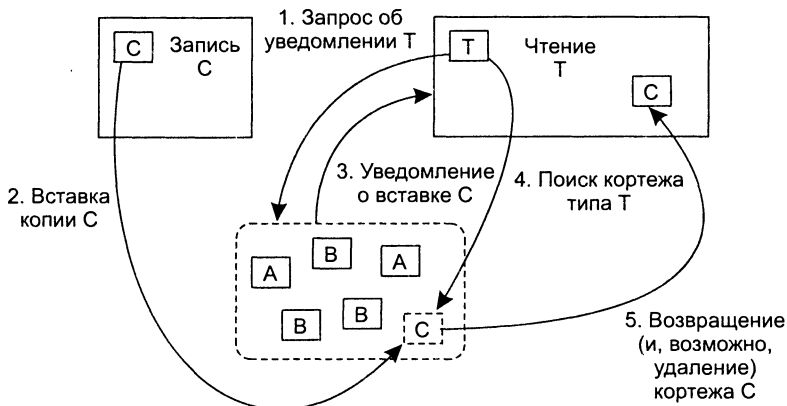


Рис. 12.13. Использование событий в JavaSpace

События также могут использоваться и в JavaSpace. Так, в частности, клиент может попросить уведомлять его о том, что в JavaSpace записан определенный экземпляр кортежа. Для этого клиент вызывает операцию `notify`, реализованную во всех пространствах JavaSpace. Эта операция получает в качестве исходных данных эталонный кортеж, который сличается с экземплярами кортежей, хранящимися в JavaSpace, точно так же, как это происходит в ходе операций `read` или `take`. Использование событий в JavaSpace иллюстрирует рис. 12.13. Отметим, что пока клиент получает уведомление о сохранении экземпляра кортежа и приступает к его чтению, другой процесс может уже прочесть этот экземпляр кортежа и удалить его из JavaSpace. Подобные условия гонок — не редкость для генеративной связи и избежать их обычно нелегко.

12.3.3. Процессы

В процессах системы `Jini` нет ровно ничего особенного. Однако реализация сервера JavaSpace часто требует особого внимания. Давайте кратко рассмотрим некоторые вопросы, связанные с реализацией сервера JavaSpace. Мы сконцентрируем свое внимание на распределенных реализациях серверов JavaSpace, то есть таких реализациях, в которых набор экземпляров кортежей может быть разбросан по нескольким машинам. Последний обзор методов реализации исполняющих систем с кортежами имеется в [388].

Эффективная распределенная реализация JavaSpace должна решать две проблемы:

- ♦ Как эмулировать ассоциативную адресацию без глобального поиска.
- ♦ Как распределить экземпляры кортежей по машинам и как их потом обнаружить.

Ключ к решению обеих проблем состоит в том, чтобы считать каждый кортеж структурой данных определенного типа. Разделив пространство кортежей на подпространства, в каждом из которых находятся кортежи одного типа, мы упрощаем программирование и делаем возможной некоторую оптимизацию. Так, например, поскольку кортежи типизированы, то уже на этапе компиляции можно определить, в каком подпространстве производится данная операция `write`, `read` или `take`. Такое дробление означает, что поиск экземпляра кортежа будет производиться не во всем наборе кортежей, а только в его части.

Кроме того, каждое из этих подпространств можно организовать в виде хэш-таблицы, использующей *i*-е поле или его часть в качестве ключа хэш-таблицы. Напомним, что каждое поле экземпляра кортежа — это подвергнутая маршалингу ссылка на объект. `Jini` не определяет способ маршалинга, поэтому маршалинг ссылок в конкретной реализации может выполняться так, чтобы первые несколько байтов ссылки после маршалинга определяли тип объекта, на который она указывает. Таким образом, вызов операций `write`, `read` или `take` может осуществляться путем вычисления значения хэш-функции *i*-го поля, в результате чего мы получим положение экземпляра кортежа в таблице. Зная подпространство и положение экземпляра кортежа в таблице, можно ничего не искать. Разумеется,

если i -е поле при операции `read` или `take` равно `NULL`, хэширование невозможно, и придется, по всей вероятности, провести полный поиск в подпространстве. Однако если аккуратно выбрать поля для хэширования, то обычно такого поиска можно избежать.

Используется также и дополнительная оптимизация. Например, схема хэширования, описанная выше, распределяет кортежи в заданных подпространствах по контейнерам, чтобы ограничить поиск одним контейнером. Можно разместить разные контейнеры на различных машинах, чтобы более разумно распределить загрузку и воспользоваться преимуществами локальных операций. Если функцией хэширования является модуль числа машин, число контейнеров будет расти в линейной зависимости от размера системы [63].

Давайте теперь кратко рассмотрим различные способы реализации описанной схемы на машинах разных типов. В мультипроцессоре подпространства кортежей можно реализовать в виде хэш-таблиц в глобальной памяти, по одной на каждое подпространство. При осуществлении операции `JavaSpace` соответствующее подпространство блокируется, к нему добавляется или удаляется экземпляр кортежа, после чего подпространство разблокируется.

В мультикомпьютере наилучший выбор зависит от коммуникационной системы. Если в нашем распоряжении имеется надежная широковещательная рассылка, можно всерьез рассматривать идею о полной репликации всех подпространств на всех машинах, как показано на рис. 12.14. При осуществлении записи (`write`) новый экземпляр кортежа рассылается по всем машинам и добав-

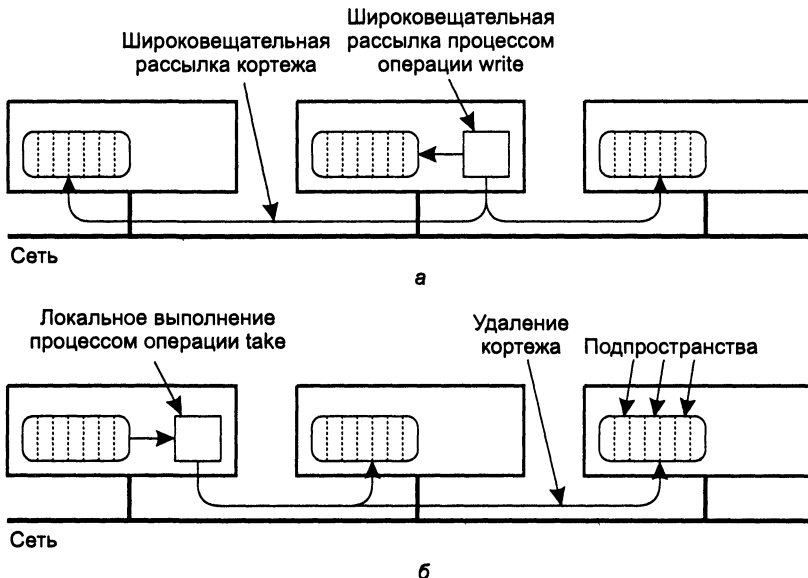


Рис. 12.14. Пространство `JavaSpace` можно распределить между машинами (дробление `JavaSpace` на подпространства показано пунктирными линиями). Рассылка кортежей при выполнении операции `write` (а). Операции `read` локальны, но удаление экземпляров при выполнении операции `take` также должно распространяться на все машины (б)

ляется на каждой из них в соответствующее подпространство. Поиск при выполнении операции `read` или `take` производится в локальном подпространстве. Однако успешное выполнение операции `take` требует удаления экземпляра кортежа из `JavaSpace`, а для того, чтобы удалить этот экземпляр со всех машин, требуется протокол удаления экземпляров кортежей. Чтобы предотвратить условия гонок и взаимные блокировки (тупики), следует использовать протокол двухфазного подтверждения.

Эта структура проста, но при ее масштабировании (если система будет наращивать число экземпляров кортежей и размеры сети) могут возникнуть проблемы. Так, например, реализация этой схемы в глобальной сети будет стоить безумно дорого.

Другой вариант состоит в том, чтобы производить операцию `write` локально, сохраняя экземпляр кортежа только на той машине, на которой он был создан, как показано на рис. 12.15. Для выполнения операции `read` или `take` требуется широковещательная рассылка эталонного кортежа. Каждая машина, получившая его, проверяет, хранится ли на ней подобный кортеж, и, если да, отправляет соответствующий отклик.

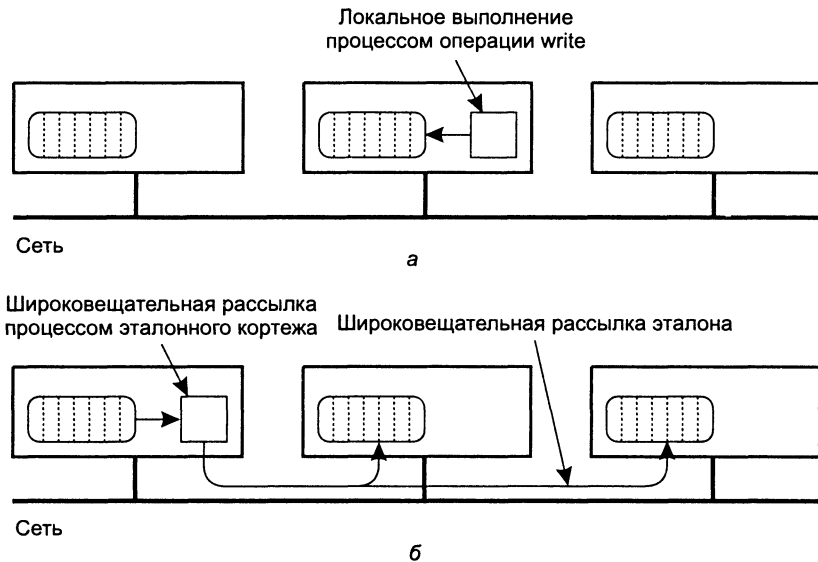


Рис. 12.15. Не реплицированное пространство `JavaSpace`. Операция `write` выполняется локально (а). Операции `read` и `take` требуют широковещательной рассылки эталона для поиска экземпляра кортежа (б)

Если экземпляр кортежа отсутствует или широковещательная рассылка не дошла до машины с нужным кортежем, то машина, пославшая запрос, продолжает повторять рассылку, увеличивая интервалы между сообщениями до тех пор, пока соответствующий экземпляр кортежа не материализуется и запрос не будет удовлетворен. Если запрос приводит к передаче двух или более экземпляров кор-

тежа, соответствующих эталону, они трактуются как локальные операции write. В результате эти экземпляры перемещаются с машины, на которой они находились, на машину, которая посылала запрос. На практике исполняющая система может перемещать кортежи с машины на машину даже с целью балансирования загрузки. В [84] подобная методика описана для реализации пространства кортежей Linda в локальной сети.

Описанные два метода можно объединить, получив в результате систему с частичной репликацией. В качестве простого примера представим себе, что все машины логически выстроены в прямоугольную решетку, как показано на рис. 12.16. Когда процесс с машины *A* хочет выполнить операцию write, он производит широковещательную рассылку кортежа или рассылает его сквозными сообщениями всем машинам в своей строке решетки. Когда процесс с машины *B* хочет выполнить операцию read или take для экземпляра кортежа, он производит широковещательную рассылку эталонного кортежа на все машины в своем столбце. Вследствие такой геометрии всегда найдется хотя бы одна машина (в нашем примере — *C*), имеющая и экземпляр кортежа, и эталонный кортеж, и эта машина обнаружит соответствие и пошлет экземпляр кортежа запросившему его процессу. Такой подход напоминает репликацию на основе кворума, которую мы обсуждали в главе 6. Он был использован при реализации системы Linda [7] и пространства кортежей в кластере [461].

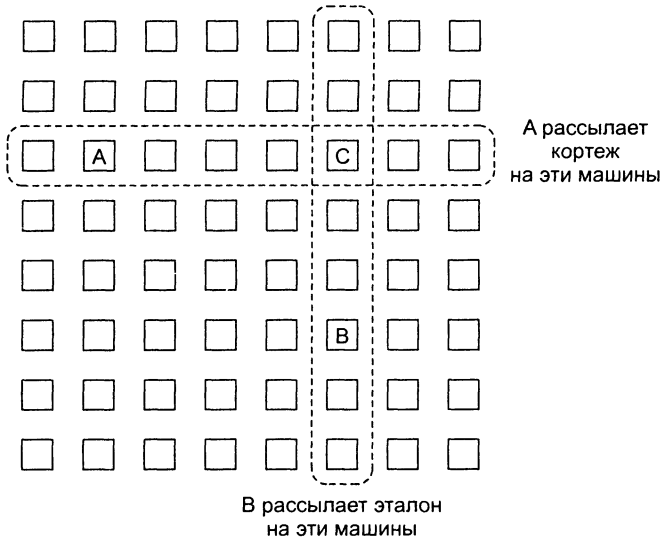


Рис. 12.16. Частичная рассылка обычных и эталонных кортежей

Реализации, которые мы обсуждали, имеют серьезные проблемы масштабируемости. Их причина в том, что и для вставки кортежа в пространство кортежей, и для его извлечения оттуда требуется широковещательная рассылка. Глобальных реализаций пространства кортежей не существует. В лучшем случае можно создать в одной системе несколько пространств кортежей так, что каждое

пространство кортежей можно будет реализовать в виде одного сервера или локальной сети. Подобный подход используется, например, в пространствах PageSpace [107] и WCL [389]. В WCL, например, каждый сервер пространства кортежей отвечает за все пространство кортежей. Другими словами, процесс всегда будет направляться ровно одним сервером. Однако для повышения производительности можно перенести пространство имен на другой сервер. Вопрос же создания эффективной глобальной реализации пространств кортежей по-прежнему остается открытым.

12.3.4. Именование

В Jini отсутствует служба именования в традиционном понимании этого термина, такая, например, как в распределенных системах объектов или в распределенных файловых системах. Подобную службу именования можно легко реализовать в архитектуре Jini в виде приложения верхнего уровня, но в ядро Jini она не входит. Тем не менее в состав Jini входит служба, которая позволяет клиентам находить зарегистрированные службы верхнего уровня, используя средства поиска по атрибутам. Далее мы рассмотрим эту поисковую службу.

Служба поиска Jini

Как мы говорили, одной из целей создания Jini была разработка системы, которая позволяла бы клиентам с легкостью обнаруживать новые службы по мере их подключения. В принципе JavaSpace в состоянии реализовать такую службу. Всякий раз при подключении службы к существующей системе она добавляет в JavaSpace описывающий ее экземпляр кортежа. Клиенты, которые ищут определенную службу, просят, чтобы пространство JavaSpace уведомляло их при добавлении службы, соответствующей заявленным требованиям.

Не полагаясь на JavaSpace, Jini имеет собственную специализированную *службу поиска (lookup service)*, являющуюся частью инфраструктуры нижнего уровня (см. рис. 12.12). Все службы верхнего уровня должны регистрироваться в ней, отослав службе поиска набор пар (*атрибут, значение*), которые описывают, например, какие услуги предоставляет служба и как с ней можно связаться. Клиент может найти службу, предоставив службе поиска эталон, схожий с эталонным кортежем JavaSpace. Служба поиска возвращает информацию о подходящих под эталон службах. Давайте кратко рассмотрим, как работает эта служба поиска.

Каждая служба имеет связанный с ней *идентификатор службы (service identifier)*, который представляет собой глобальное уникальное 128-битное значение, генерируемое службой поиска. Служба использует этот идентификатор для регистрации *элемента службы (service item)* в службе поиска. Элемент службы представляет собой запись с тремя полями, перечисленными в табл. 12.4.

Поле *ServiceID* содержит идентификатор службы, присвоенный этой службе службой поиска. Идентификатор используется как уникальный ключ для службы поиска. Не может существовать двух элементов службы, которым служба поиска выдала бы одинаковые идентификаторы.

Таблица 12.4. Структура элемента службы

Поле	Описание
ServiceID	Идентификатор службы, ассоциированный с этим элементом
Service	Ссылка на объект (возможно, удаленный), реализующий службу
AttributeSets	Набор кортежей, описывающих службу

Поле *Service* содержит ссылку на объект. Во многих случаях это ссылка на удаленный объект, а значит, клиент, получивший при помощи службы поиска указанную ссылку, может немедленно обратиться к этому объекту путем обращения RMI языка Java. Напомним, что в главе 2 мы говорили, что удаленная ссылка на объекты Java часто реализуется путем маршалинга заместителя. Остается лишь выполнить его демаршалинг, после чего владелец может обратиться к объекту.

Поле *AttributeSets* представляет собой набор кортежей, схожих с кортежами JavaSpace. Каждый кортеж, в сущности, соответствует объекту Java, а каждое поле кортежа описывает пару (*атрибут, значение*) этого объекта. Используя те же методы, что и в JavaSpace, клиент может выполнять поиск определенного экземпляра кортежа с помощью эталонного кортежа. Служба поиска выберет только те кортежи, которые соответствуют эталону.

Как и в случае с JavaSpace, клиент может потребовать от службы поиска уведомлять его о случаях добавления элементов службы, соответствующих эталонному кортежу клиента.

Чтобы способствовать развитию системы Jini, было создано несколько встроенных кортежей, которые можно использовать для регистрации служб (табл. 12.5). Для этих кортежей все атрибуты представлены в виде текстовых строк, но возможны и другие способы представления. Это дает нам необходимую в работе гибкость.

Таблица 12.5. Примеры встроенных кортежей для элементов служб

Тип кортежа	Атрибуты
ServiceInfo	Имя, производитель, продавец, версия, модель, серийный номер
Location	Этаж, комната, здание
Address	Улица, организация, организационная единица, местность, штат или район, почтовый код, страна

Мы безоговорочно предполагали, что существует лишь одна служба поиска. Однако Jini допускает одновременное наличие нескольких таких служб. Каждая служба может отвечать за группу служб. Таким образом, загрузка одной службы поиска может быть распределена по нескольким машинам.

В нашем описании мы пропустили один важный вопрос, а именно, как найти службу поиска? Стандартно применяемый во многих распределенных системах способ состоит в том, чтобы сконфигурировать поисковый сервер на общеизве-

стный адрес. В Jini используется другой подход. Клиенту предлагается для локализации службы поиска произвести групповую рассылку сообщения с запросом к этой службе. Если не принимать специальных мер, этот подход эффективен только для локальных сетей.

Кроме того, служба поиска и сама регулярно объявляет о своем местоположении также путем групповой рассылки. Клиент может определить местоположение службы поиска, а потом, когда ему понадобится найти определенную службу, воспользоваться этими сведениями. Детали соответствующих протоколов можно найти в [483], там же содержится полная спецификация интерфейсов служб поиска.

Аренда

С вопросами именования связаны вопросы управления ссылками на объекты. Как мы говорили в главе 4, управление ссылками состоит в том, что объекты отслеживают, кто на них ссылается, что приводит нас к так называемым спискам ссылок. Чтобы удерживать размеры этих списков в разумных пределах, а также иметь возможность обрабатывать случаи отказов ссылающихся на объект процессов, удобно использовать аренду. По истечении срока аренды ссылки становятся недействительными и удаляются из списка ссылок. Когда список на объект становится пустым, объект может без опасений самоуничтожиться.

В системе Jini аренда активно используется для удаления объектов, на которые никто не ссылается. Так, например, когда процесс записывает кортеж в JavaSpace, он получает аренду, определяющую, как долго будет храниться этот кортеж и когда он должен быть уничтожен. В этом случае операция записи позволяет объекту, который ее осуществляет, указать желаемое время аренды.

Аренда никогда не дает полной гарантии. Когда процесс устанавливает аренду, он действительно обещает сделать все возможное для сохранения арендованного объекта в течение как минимум определенного времени. Процесс, которому предоставлена аренда, всегда может попросить о ее продлении. Однако арендодатель вправе решать, согласиться на продление аренды или нет.

Интерфейсы аренды стандартизованы. При использовании аренды в Jini всюду соблюдаются одни и те же спецификации. Детальное описание интерфейсов можно найти в [483].

12.3.5. Синхронизация

Jini поддерживает несколько механизмов синхронизации. Один из важных классов подобных механизмов, а именно блокирующие операции `read` и `take`, реализован как часть `JavaSpace`. Эти операции позволяют реализовать множество различных шаблонов синхронизации. Примеры подобных шаблонов в отношении параллельных программ можно найти в [85]. Помимо синхронизации посредством этих двух операций Jini поддерживает также понятие транзакций, о которых мы сейчас и поговорим.

Транзакции

Чтобы способствовать выполнению последовательностей операций над несколькими объектами, Jini поддерживает транзакции, использующие протокол двухфазного подтверждения. Эта поддержка, в сущности, сводится к предоставлению набора интерфейсов, реализация оставлена другим. Однако Jini можно сконфигурировать для работы со стандартным менеджером транзакций.

Такой подход имеет важную особенность. Дело в том, что свойства ACID транзакции самой системой Jini не обеспечиваются. Взамен предполагается, что эти свойства совместно реализуются процессами, принимающими участие в транзакции. Однако взаимодействие между процессами соответствует шаблону транзакций с двухфазным подтверждением.

Общую модель транзакций в Jini иллюстрирует рис. 12.17. Клиент *A* может начать транзакцию, послав запрос менеджеру транзакций, который вернет идентификатор транзакции. Как и во многих других случаях, клиент должен определить, сколько времени может пройти до подтверждения или прерывания транзакции. Кроме того, менеджер может назначить только что созданной транзакции аренду и прервать транзакцию по истечении срока аренды независимо от состояния транзакции.

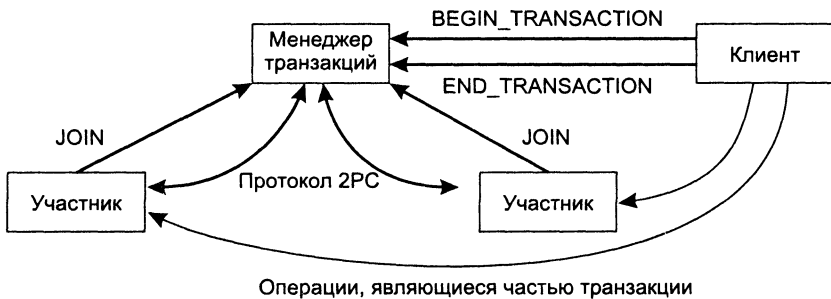


Рис. 12.17. Общая организация транзакций в Jini. Толстые линии обозначают взаимодействие, выполняемое в соответствии с требованиями протокола транзакций Jini

Клиент может требовать от других процессов, чтобы они присоединились к транзакции. Такие процессы должны реализовывать predetermined интерфейс, который позволит менеджеру транзакций управлять транзакцией. Этот интерфейс, содержащий такие операции, как `commit` и `abort`, вызывается менеджером транзакций, чтобы указать участнику транзакции, что ему делать. Задача участника — корректно реализовать эти методы.

Если транзакция заканчивается до истечения срока аренды, клиент указывает менеджеру транзакций, что он должен подтвердить или прервать транзакцию. После этого менеджер транзакций запускает протокол двухфазного подтверждения и сообщает о результатах клиенту. Этот протокол мы детально обсуждали в главе 7 и не будем повторяться.

Jini поддерживает также и вложенные транзакции. В этом случае менеджеру транзакций посылается запрос с требованием начать вложенную транзакцию

в ходе существующей транзакции. И снова клиент контролирует организацию транзакции, то есть определяет, какие процессы примут в ней участие. При вложенной транзакции процессы должны участвовать в конкретных частях общей транзакции.

Пространство JavaSpace также может участвовать в транзакциях. В транзакции могут принимать участие несколько пространств JavaSpace. Когда процесс пересылает пространству JavaSpace сведения об операции, которую оно должно выполнить, процесс может передать также и идентификатор транзакции. Если пространство JavaSpace не было вовлечено в эту транзакцию, это нужно сделать. Серверы JavaSpace, работая вместе с менеджером транзакций, гарантируют транзакции соблюдение свойств ACID. Кроме того, чтобы избежать каскадных прерываний, реализация JavaSpace в Jini и стандартный менеджер транзакций, который, собственно говоря, является частью Jini, поддерживают строгую двухфазную блокировку. Мы говорили о двухфазной блокировке в главе 5.

12.3.6. Кэширование и репликация

Как и в TIB/Rendezvous, в Jini не предусмотрено никаких специальных средств кэширования или репликации. Эти аспекты полностью оставлены приложениям, которые становятся частью системы Jini. Единственная служба, для которой Jini предполагает возможность репликации с целью повышения отказоустойчивости, — это служба поиска.

12.3.7. Отказоустойчивость

Сама система Jini не предоставляет никакой специальной поддержки отказоустойчивости, если не считать менеджера транзакций, реализующего описанный выше протокол транзакций. Jini рассчитывает на то, что компоненты, использующие Jini как базу, при необходимости реализуют собственные механизмы обеспечения отказоустойчивости.

Существует немало работ по отказоустойчивым решениям для исходных пространств кортежей Linda (которые образуют основу пространств JavaSpace). Так, например, в [26] описан подход на основе активной репликации пространств кортежей. Кроме того, в [410] получила дальнейшее развитие программная модель с группировкой нескольких операций в транзакцию.

Что же касается связи, отметим, что фактически все взаимодействие осуществляется посредством обращений RMI языка Java, а эти обращения обычно реализуются при помощи надежного низкоуровневого протокола связи, такого как HTTP или TCP.

12.3.8. Защита

Защита в Jini основана исключительно на возможностях защиты RMI языка Java. Мы уже обсуждали в главе 8 множество вопросов, связанных с обращениями RMI и в основном касающихся предохранения от динамически загружаемого ко-

да путем самоанализа стека. Напомним, что важным свойством схемы самоанализа стека является возможность назначать права доступа классам. Эти права во время исполнения можно проверять с помощью менеджера защиты Java.

В Jini введена специальная *служба аутентификации и авторизации Java (Java Authentication and Authorization Service, JAAS)*, которая осуществляет аутентификацию и авторизацию пользователей в системах на базе Java, таких как Jini. В соответствии с подходом, принятым и в других распределенных системах, JAAS отделяет интерфейс, предоставляемый пользователям для аутентификации и контроля доступа, от истинной реализации этих служб. Это разделение [394] осуществляется при помощи *подключаемого модуля аутентификации (Pluggable Authentication Module, PAM)*. PAM образует, в сущности, промежуточный уровень между приложениями и службами защиты. Как показано на рис. 12.18, он предоставляет этим двум группам процессов стандартные интерфейсы. JAAS — это реализация PAM на языке Java.

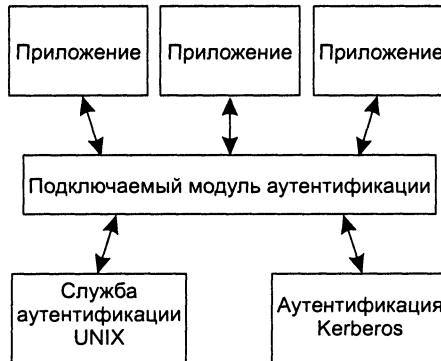


Рис. 12.18. Положение PAM по отношению к службам защиты

JAAS добавляет к существующим механизмам контроля доступа Java средства для контроля доступа уже аутентифицированных пользователей. Напомним, что Java, как рассказывалось в главе 8, имеет средства управления привилегиями на уровне классов, когда привилегии ассоциируются с классами. JAAS может также работать и с пользователями.

Поскольку JAAS, в сущности, похожа на другие службы аутентификации и авторизации, в том числе и те, которые мы уже рассматривали, мы не станем вдаваться в детали. Дополнительную информацию о JAAS можно почерпнуть в [247].

12.4. Сравнение TIB/Rendezvous и Jini

Две системы согласования, которые мы описали в этой главе, являются характерными представителями распределенных систем согласования.

Философия

Как TJB/Rendezvous, так и Jini нацелены на организацию ссылочной несвязности процессов, то есть на предоставление средств, при помощи которых взаимодействующие процессы могли бы при желании оставаться более или менее безмянными. TIB/Rendezvous ссылочную несвязность обеспечивает при помощи механизма публикации/подписки, а Jini — генеративной связью через JavaSpace. Кроме того, Jini обеспечивает еще и временную несвязность процессов.

Другое различие между этими системами состоит в том, что TIB/Rendezvous обслуживает большую часть взаимодействия между процессами. В Jini же главная идея состоит в том, что система должна представить процессы друг другу, но сразу после этого взаимодействие между ними обеспечивается обращениями RMI языка Java.

Связь

Как мы говорили, взаимодействие в TIB/Rendezvous осуществляется в первую очередь при помощи базового механизма публикации/подписки. В результате в этой системе важнейшую роль играет групповая рассылка. Предпринимаются специальные меры, чтобы гарантировать успешность групповой рассылки и в глобальных сетях. Чтобы связь не зависела от приложений, сообщения сделаны самоопределяющимися.

В противоположность этой системе Jini, в сущности, использует групповую рассылку только для того, чтобы сразу найти службы поиска, которые потом помогают клиенту искать другие процессы. Любое другое взаимодействие в основном реализуется обращениями RMI языка Java, в том числе и взаимодействие с серверами JavaSpace. Другими словами, групповая рассылка применяется процессом для поиска другого процесса, с которым он хотел бы взаимодействовать. Далее он может использовать механизмы сквозной связи. Кроме того, формат и содержимое сообщений полностью определяются взаимодействующими сторонами.

В обеих системах события играют важную, но различную роль. В TIB/Rendezvous механизм событий обслуживает все взаимодействия. В принципе входящее сообщение может быть получено только в том случае, если непосредственно у получателя установлен обработчик события для этого сообщения. С точки зрения приложения это означает, что для обработки входящих сообщений не нужны блокирующие операции.

События в Jini имеют другую природу. Каждый процесс может предложить службу событий, благодаря которой появляется возможность зарегистрировать другой процесс для последующей передачи уведомлений. Когда происходит определенное событие, зарегистрированный процесс делает обратный вызов, который может быть соответствующим образом обработан. Механизм событий Jini — это, в сущности, служба обратного вызова, которая организуется между парами процессов.

Процессы

Поскольку TIB/Rendezvous и Jini, в сущности, содержат только средства взаимодействия между процессами, в способах организации самих процессов нет ничего

особенного. Кроме того, хотя обе системы поддерживают множество специфических процессов, которые реализуют, например, транзакции, не предпринимается никаких специальных мер, чтобы сделать эти процессы хотя бы немного отличными от процессов пользовательских приложений.

Именованние

Если рассматривать вопросы именования, между TIB/Rendezvous и Jini имеется три отличия. Имена в TIB/Rendezvous играют важную роль в контексте адресации по теме. Все адреса имеют вид символьных строк, напоминая DNS-имена. Предоставляются простые, но эффективные средства именования, включая символы-заменители и аббревиатуры для групп тем.

По сравнению с TIB/Rendezvous имена в Jini имеют вид строк байтов, в которых каждая строка соответствует объекту после маршалинга. Строки сравниваются только на равенство, хотя можно также использовать символ-заменитель в форме строки без значения (NULL).

Благодаря эффективности парадигмы публикации/подписки в TIB/Rendezvous нет реальной необходимости в выделенной службе именования, которая решала бы имена в адреса процессов. В противоположность ей система Jini вынуждена поддерживать выделенную службу поиска, позволяющую клиенту локализовать процесс по имени на базе атрибутов (это имя опять-таки выражается в виде последовательности строк байтов).

Синхронизация

Системы TIB/Rendezvous и Jini поддерживают механизмы транзакций, но используют разные модели транзакций. В TIB/Rendezvous основная идея транзакций состоит в том, что они объединяют последовательности операций публикации и получения сообщений (а возможно, и операций с базами данных) в единую транзакцию. Транзакции (точнее, транзакционный обмен сообщениями) осуществляются на специальном уровне, на котором расположены расширения обычной библиотеки публикации/подписки. Кроме того, имеется менеджер транзакций. TIB/Rendezvous следит за тем, чтобы транзакции в ходе обмена сообщениями удовлетворяли свойствам ACID.

Jini, в сущности, содержит только протокол транзакций, который позволяет объединить в транзакцию несколько операций, вызванных клиентом. Однако обеспечить соблюдение свойств ACID, то есть атомарности, непротиворечивости, изолированности и долговечности транзакции, — забота иницировавшего ее процесса. Jini содержит специальный менеджер транзакций и гарантирует, что совместно с серверами JavaSpace этот менеджер обеспечит соблюдение свойств ACID для транзакций.

Работа транзакции в TIB/Rendezvous ограничена рамками одного процесса. Если принять участие в транзакции хотят несколько процессов, необходим традиционный менеджер транзакций, который мог бы обслуживать не только обмен сообщениями. В Jini в одной транзакции могут участвовать несколько клиентских процессов, хотя обычно инициативу по организации и завершению транзакции проявляет только один клиент.

Добавок к транзакциям Jini поддерживает также механизм синхронизации на основе блокирующих операций в JavaSpace, который эффективно реализует распределенные блокировки. В TIB/Rendezvous нет никаких других механизмов синхронизации, кроме транзакций.

Кэширование и репликация

Ни TIB/Rendezvous, ни Jini не поддерживают кэширование и репликацию. Этими вопросами должны заниматься приложения.

Отказоустойчивость

Что касается отказоустойчивости, обе системы используют надежную связь, причем TIB/Rendezvous поддерживает сохранную связь — как в форме сертифицированной доставки сообщений, так и в форме транзакционного обмена сообщениями. Надежная связь Jini полностью полагается на надежность, обеспечиваемую обращениями RMI языка Java.

В TIB/Rendezvous отказоустойчивость обеспечивается явной поддержкой групп процессов на промежуточном уровне. Jini не имеет специальных средств поддержания групп процессов. Предполагается, что при необходимости эту поддержку будут реализовывать приложения. Ни одна из систем не имеет явных средств восстановления после отказов, если не считать тех, которые реализованы как часть транзакций.

Защита

И, наконец, TIB/Rendezvous поддерживает защиту в форме отдельного протокола организации защищенного канала между издателем и подписчиком. Для контроля доступа не предпринимается ничего, этот вопрос остается приложением, использующим TIB/Rendezvous в качестве промежуточного уровня взаимодействия.

В Jini защита целиком обеспечивается теми стандартными средствами защиты, которые обычно присутствуют в системах на базе Java. Эти средства подразумевают контроль доступа на уровне классов, который мы обсуждали в главе 8, а также аутентификацию и авторизацию пользователей в форме выделенного сервера защиты JAAS.

Наиболее важные аспекты каждой из систем перечислены в табл. 12.6.

Таблица 12.6. Сравнение систем TJB/Rendezvous и Jini

Аспект	TIB/Rendezvous	Jini
Основная цель разработки	Отсутствие связности процессов	Гибкая интеграция
Модель согласования	Публикация/подписка	Генеративная связь
Сетевое взаимодействие	Групповая рассылка	RMI языка Java
Сообщения	Самоопределяющиеся	Зависят от процесса
Механизм событий	Входящие сообщения	Служба обратного вызова
Процессы	Общего назначения	Общего назначения

Аспект	TIB/Rendezvous	Jini
Имена	Символьные строки	Строки байтов
Службы именования	Отсутствуют	Служба поиска
Транзакции (операции)	Сообщения	Обращения к методам
Транзакции (область видимости)	Единственный процесс	Несколько процессов
Блокировка	Отсутствует	Как у операций в <code>JavaSpace</code>
Кэширование и репликация	Отсутствуют	Отсутствуют
Надежная связь	Имеется	Имеется
Группы процессов	Имеются	Отсутствуют
Механизмы восстановления	Явной поддержки нет	Явной поддержки нет
Защита	Защищенные каналы	Полностью определяется механизмами защиты <code>Java</code>

12.5. Итоги

Распределенные системы согласования начинают играть важную роль в разработке распределенных приложений. Большинство подобных систем отличаются отсутствием ссылочной связности процессов в том смысле, что процессы для связи между собой не нуждаются в явных ссылках друг на друга. Кроме того, может обеспечиваться и отсутствие временной связности, в этом случае для взаимодействия процессов друг с другом им вовсе не обязательно выполняться одновременно.

Одна из важных групп систем согласования — это системы, основанные на парадигме издателя/подписчика, такие как TIB/Rendezvous. В этой модели сообщения доставляются получателям не в соответствии с их адресами, а в соответствии с темой сообщений. Процессы, желающие получать сообщения, должны подписаться на определенную тему. За выбор пути сообщений от издателя к подписчику отвечает промежуточный уровень.

Другая группа систем согласования использует модель генеративной связи, впервые предложенную в системе Linda. Генеративная связь реализуется разделяемыми пространствами кортежей. Кортеж — это типовая структура данных, сходная с записью. Чтобы извлечь из пространства кортежей нужный кортеж, процесс отыскивает его при помощи эталонного кортежа. Кортеж, соответствующий эталону, выбирается и возвращается запросившему его процессу. Если совпадений нет, процесс блокируется.

Системы согласования отличаются от большинства других распределенных систем тем, что в них в основном решается задача предоставления удобного способа связи между процессами, которые ничего не знают друг о друге заранее. Связь может и далее осуществляться с сохранением анонимности. Основное преимущество подобного подхода — его гибкость. Можно дополнять и изменять продолжающую работать систему.

Принципы распределенных систем, которые мы обсуждали в первой половине книги, равно применимы и к системам согласования, хотя кэширование и репликация играют в современных их реализациях не столь важную роль. Кроме того, именование в них сильно связано с поиском по атрибутам. Аналогичный подход реализован и в службах каталогов.

Вопросы и задания

1. К какому типу модели согласования вы отнесете системы очередей сообщений, описанные в главе 2?
2. Отсутствие ссылочной связности в TIB/Rendezvous компенсируется адресацией по теме. Какими еще средствами компенсируется отсутствие ссылочной связности в этой системе?
3. Опишите реализацию системы публикации/подписки на основе системы очередей сообщений, такой как MQSeries компании IBM.
4. Во что на самом деле разрешается имя темы в TIB/Rendezvous и как протекает это разрешение?
5. Опишите простую реализацию полностью упорядоченной доставки сообщений в системе TIB/Rendezvous.
6. Когда в ходе транзакции TIB/Rendezvous сообщение доставляется процессу *P*, процесс подтверждает доставку. Можно ли на уровне транзакций сделать так, чтобы подтверждение отсылалось демону транзакций автоматически?
7. Предположим, что процесс в системе TIB/Rendezvous реплицируется. Приведите два решения, позволяющие исключить случаи публикации сообщений реплицированными процессами более одного раза.
8. Насколько необходима полностью упорядоченная групповая рассылка при репликации процессов в системе TIB/Rendezvous?
9. Опишите простую схему для PGM, которая позволяла бы получателям обнаруживать пропавшие сообщения, даже если пропадет первое сообщение в последовательности.
10. Могут ли журналы регистрации в TIB/Rendezvous, реализованные в виде файлов, гарантировать, что сообщение, даже при наличии отказавших процессов, не будет потеряно?
11. Как следовало бы реализовать в TIB/Rendezvous модель согласования на основе генеративной связи?
12. Не считая отсутствия временной связности в Jini, в чем, по вашему мнению, состоит наиболее существенная разница между Jini и TIB/Rendezvous с точки зрения моделей согласования?
13. Срок аренды в Jini всегда определяется его продолжительностью и никогда — в виде абсолютного значения времени истечения аренды. Почему?
14. В чем состоят наиболее серьезные проблемы масштабирования в Jini?

15. Пусть в распределенной реализации JavaSpace кортежи реплицируются на нескольких машинах. Приведите протокол удаления кортежа, в котором при попытке двух процессов удалить один и тот же кортеж не возникает условий гонок.
16. Представьте себе, что транзакция T в Jini требует блокировки объекта, который уже заблокирован другой транзакцией T' . Опишите, что произойдет.
17. Представьте себе, что клиент Jini кэширует полученный из JavaSpace кортеж, чтобы иметь возможность в следующий раз не обращаться к JavaSpace. Имеет ли такое кэширование смысл?
18. Ответьте на предыдущий вопрос для случая, когда клиент кэширует результаты, полученные службой поиска.
19. Опишите простую реализацию отказоустойчивого пространства JavaSpace.

Глава 13

Библиография

13.1. Литература для дополнительного чтения

13.2. Список литературы

В предыдущих главах мы изучили самые разные темы. Эта глава призвана помочь читателям, заинтересовавшимся каким-либо конкретным вопросом в области распределенных систем и желающим расширить свои знания. Раздел 13.1 представляет собой описание литературы, рекомендуемой для дополнительного чтения, а в разделе 13.2 в алфавитном порядке перечислены книги и статьи, на которые есть ссылки в предыдущих главах.

13.1. Литература для дополнительного чтения

13.1.1. Введение и общие вопросы

Coulouris et al., Distributed Systems — Concepts and Design

Хорошая книга по распределенным системам. Темы, которые в ней освещаются, в основном соответствуют темам, рассмотренным в данной книге, но материал организован совершенно иначе. В ней значительно больше сведений по распределенным транзакциям и меньше — по именованию и защите. В число учебных примеров входят CORBA и распределенная операционная система Mach.

Chow and Johnson, Distributed Operating Systems and Algorithms

В первой части книги распределенные системы рассматриваются с позиций операционных систем, включая распределенную разделяемую память и планирование распределенных процессов. Во второй части исследуется алгоритмический

подход к синхронизации, распределенному согласованию, репликации и отказоустойчивости.

Geihs, «Middleware Challenges Ahead»

В этой статье автор дает краткий обзор промежуточного уровня распределенных систем, в основном делая упор на том, что может потребоваться для систем промежуточного уровня в будущем. В статье обсуждаются модели программирования, пользовательские настройки и вопросы, связанные с мобильными вычислениями.

Mullender, Distributed Systems, 2nd ed.

Труды летней школы, содержащие 21 статью ведущих разработчиков распределенных систем и посвященные моделированию, специфицированию, отказоустойчивости, проблемам реального времени, связи, именованию, файловым системам, планированию и, наконец, защите. Хотя в книге не упоминаются последние разработки, она все еще охватывает множество вопросов, относящихся к распределенным системам в целом.

Neuman, «Scale in Distributed Systems»

Одна из немногих статей, содержащих систематическое изложение материалов по вопросам масштабирования в распределенных системах. Рассматривается применение для этой цели кэширования, репликации и распределенного выполнения; приводится множество практических рекомендаций по применению этих методик при разработке крупномасштабных систем.

Silberschatz et al., Applied Operating System Concepts

Основной учебник по операционным системам, содержащий материалы с упором на файловые системы и распределенные системы согласования.

Umar, Object-Oriented Client/Server Internet Environments

Не слишком формальная книга, посвященная исключительно системам клиент-сервер, которая охватывает традиционные темы, такие как обработка транзакций и удаленный доступ к базам данных. Позволяет получить представление о том, какое количество людей реально интересуется распределенными системами.

Verissimo and Rodrigues, Distributed Systems for Systems Architects

Требующая значительной подготовки книга по распределенным системам, охватывающая в основном те же материалы, что и книга, которую вы держите в руках. Повышенное внимание уделяется вопросам отказоустойчивости и распределенным системам реального времени. Также особое влияние уделяется управлению распределенными системами.

13.1.2. Связь

Birrell and Nelson, «Implementing Remote Procedure Calls»

Классический труд по проектированию и реализации одной из первых систем удаленного вызова процедур.

Crowcroft et al., Internetworking Multimedia

Всеобъемлющая книга по передаче мультимедиа через Интернет. В ней описываются транспортные протоколы передачи мультимедиа, организация сеансов, управление конференциями, получение данных по запросу, а также вопросы защиты при обмене мультимедийной информацией.

Halsall, Multimedia Communications—Applications, Networks, Protocols, and Standards

Обширный труд, в котором содержится описание множества сетевых протоколов передачи мультимедиа, охватывающий также подавляющее большинство традиционных материалов по сетям. К темам, связанным с мультимедиа, относятся сети мультимедиа, представление информации и методы сжатия.

Kurose and Ross, Computer Networking, A Top-Down Approach Featuring the Internet

Хорошее общее руководство, содержащее полную информацию по компьютерным сетям. Основы описываются исключительно полно и детально. Изучив их, можно получить исчерпывающее представление о процессах взаимодействия в распределенных системах.

Oram, Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology

Следующее поколение распределенных систем, возможно, будет создаваться в виде эпизодически объединяемых через Интернет компьютеров конечных пользователей. Эта книга содержит несколько статей, посвященных одноранговым сетям. Кроме того, в ней обсуждаются вопросы защиты, доверия и ответственности.

Waldo, «Remote Procedure Calls and Java Remote Method Invocation»

Хорошо написанное и легкое для чтения обсуждение сходств и различий между технологиями RPC и RMI, написанное одним из основных разработчиков распределенных технологий Java.

13.1.3. Процессы

Andrews, Foundations of Multithreaded, Parallel, and Distributed Programming

Если вам когда-нибудь придется серьезно заняться программированием параллельных и распределенных систем, эта книга вам пригодится.

Lewis and Berg, Multithreaded Programming with Pthreads

Р-поток выполнения (Pthreads) — это стандарт POSIX для реализации потоков выполнения в операционных системах, который поддерживается операционными системами на базе UNIX. Хотя авторы сконцентрировали свое внимание на Р-потоках, книга представляет собой хорошее введение в программирование потоков выполнения вообще. Она дает прочный фундамент для разработки многопоточных приложений серверов и клиентов.

Milojicic et al., «Process Migration»

Великолепное и всестороннее исследование по переносу процессов, включая мобильный код и мобильных агентов. Перенос процессов всегда манил разработчиков распределенных операционных систем, но так никогда и не был реализован. Эта статья дополнит материалы по переносу кода из главы 3.

Nwana and Ndumu, «A Perspective on Software Agents Research»

Критический обзор о том, что обещали разработчики систем на базе агентов и что у них на самом деле получилось, написанное двумя авторитетами в этой области. Они не слишком оптимистично настроены в вопросах практического применения агентов и утверждают, что основные проблемы начнутся при попытке разработать более мощные системы.

Schmidt et al., Pattern-Oriented Software Architecture — Patterns for Concurrent and Networked Objects

Лишь недавно разработчики начали присматриваться к общим шаблонам проектирования распределенных систем. Эти шаблоны способны упростить разработку распределенных систем, так как позволяют программистам сосредоточиться на специфических вопросах конкретной системы. В книге рассматриваются шаблоны приложений для доступа к службам, обработки событий, синхронизации и параллельной обработки.

13.1.4. Именованное

Albitz and Liu, DNS and BIND

BIND — это открытая и широко используемая реализация DNS-сервера. В книге рассматриваются все подробности организации домена DNS на базе BIND. Кроме того, в ней содержится множество практических сведений по гигантским распределенным службам именования, используемым в настоящее время.

Hudson et al., Garbage Collecting the World: One Car at a Time

Одним из наиболее сложных вопросов, связанных со сборкой мусора в крупных распределенных системах, является трассировка. В этой книге рассматривается

альтернативный подход к сборке мусора, отчасти основанный на решениях, которые мы обсуждали в главе 3.

Loshin, Big Book of Lightweight Directory Access Protocol (LDAP) RFCs

Системы на основе LDAP мгновенно завоевали популярность в распределенных системах. Полный источник данных по службам LDAP — это рекомендации RFC, которые публикует комитет IETF. В данном случае все рекомендации RFC, относящиеся к этой теме, собраны в одной книге, поэтому она является исчерпывающим источником сведений по проектированию и реализации служб LDAP.

Needham, «Names»

Великолепная легко читаемая статья о роли имен в распределенных системах. Упор делается на системы именования, которые мы рассматривали в разделе 4.1, в качестве примера рассматривается система GNS компании DEC.

Pitoura and Samaras, «Locating Objects in Mobile Computing»

Эта статья — прекрасное введение в вопросы локализации. Авторы обсуждают различные типы служб локализации, включая соответствующие службы в системах телекоммуникаций. Имеется обширный список ссылок, который может быть использован в качестве исходной точки для подборки материалов для изучения.

Saltzer, «Naming and Binding Objects»

Хотя книга была написана в 1978 году и упор в ней делается на нераспределенные объекты, она может стать хорошей отправной точкой для дальнейших исследований в области именования. Автор проделал потрясающую работу по отношениям между именами и объектами и, в частности, по разрешению имени в объект, на который оно указывает. Отдельное внимание уделяется концепции механизмов свертывания.

13.1.5. Синхронизация

Anceaume and Puaut, «Performance Evaluation of Clock Synchronization Algorithms»

В этом отчете содержится разбор нескольких общеизвестных алгоритмов синхронизации часов. Кроме того, в нем приведен модельный анализ производительности нескольких алгоритмов с учетом вопросов отказоустойчивости. Также приводится обзор свойств разных алгоритмов.

Babaoglu and Marzullo, «Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms»

Великолепное, хотя излишне теоретизированное описание глобального состояния в распределенных системах. Статья содержит материалы по логическим

и векторным часам, распределенным снимкам состояния и глобальным предикатам.

Gray and Reuter, Transaction Processing: Concepts and Techniques

Исчерпывающее исследование по транзакциям, как распределенным, так и нераспределенным, для распределенных и нераспределенных систем. В книге последовательно рассматриваются подходы к обработке транзакций с описанием множества деталей и возможностей. Если вы ищете книгу, в которой умещается все, что на сегодня известно по транзакциям, это то, что вам нужно.

Lynch, Distributed Algorithms

В книге по единому шаблону описываются разные типы распределенных алгоритмов. Обсуждаются три модели синхронизации — простые синхронные модели, асинхронные модели и частично синхронные модели, лежащие ближе всего к реальным системам. Если вам требуется теоретическое описание, вы найдете в этой книге множество полезных алгоритмов.

Raynal and Singhal, «Logical Time: Capturing Causality in Distributed Systems»

В этой работе в относительно простых терминах описываются три вида логических часов: скалярное время (то есть отметки времени Лампорта), векторное время и матричное время. Кроме того, в ней представлены различные реализации времени, которые использовались во множестве практических и экспериментальных системах.

Wu, Distributed System Design

Для практиков подобный кошмарный заголовок означает, что книга посвящена теории процесса. Однако в ней представлены разнообразные распределенные алгоритмы, включая алгоритмы маршрутизации и распределения загрузки. Таким образом, книга является хорошим дополнением к главе 5.

13.1.6. Непротиворечивость и репликация

Ahamad and Kordale, «Scalable Consistency Protocols for Distributed Services»

В этой работе описываются такие протоколы поддержания непротиворечивости, в которых изменения не нужно немедленно передавать всем копиям. Вместо этого копия может находиться в предыдущем глобально непротиворечивом состоянии и не переходить в другое состояние до тех пор, пока ее текущее состояние соответствует последнему глобальному состоянию. Допустимость отставания между текущим глобальным состоянием и локально непротиворечивым состоянием является ключом к решению проблемы масштабируемости.

Gray et al., «The Dangers of Replication and a Solution»

В работе обсуждается разница между репликацией, реализующей модель последовательной непротиворечивости (так называемая «энергичная» репликация), и «ленивой» репликацией. Оба вида репликации сформулированы в понятиях транзакций. Проблема энергичной репликации — в ее плохой масштабируемости, в то время как ленивая репликация легко может привести к тяжелым или вообще неразрешимым конфликтам. Авторы предлагают гибридную схему.

Protic et al., «Distributed Shared Memory: Concepts and Systems»

Модели непротиворечивости памяти играют важную роль в системах с распределенной разделяемой памятью, и большинство решений в этой области касаются разработки подобных систем. В работе приводится хороший обзор систем подобного типа с кратким описанием различных моделей непротиворечивости.

Saito, «Consistency Management in Optimistic Replication Algorithms»

В отчете представлена таксономия алгоритмов оптимистической репликации, используемых в моделях слабой непротиворечивости. Описан еще один взгляд на репликацию и соответствующие протоколы непротиворечивости. Интересно обсуждение масштабируемости различных решений. Отчет содержит исчерпывающий список ссылок.

Yu and Vahdat, «Design and Evaluation of a Continuous Consistency Model for Replicated Services»

В работе обсуждается система, которая могла бы поддерживать прикладные модели непротиворечивости. Необходимость дифференцирования моделей выводится из соотношения между производительностью, доступностью и непротиворечивостью. Непротиворечивость определяется при помощи трех непрерывных параметров: числа записей, которые реплика может разрешить произвести бесконтрольно, числа записей, которые можно выполнить локально до начала распространения изменений, и времени возможной задержки распространения изменений.

Wiesmann et al., «Understanding Replication in Databases and Distributed Systems»

Существуют традиционные различия между репликацией в распределенных базах данных и распределенных системах общего назначения. В базах данных основной причиной репликации является увеличение производительности. В распределенных системах общего назначения репликация обычно производится для повышения отказоустойчивости. В работе предложен удобный способ сравнения этих двух подходов.

13.1.7. Отказоустойчивость

Cristian and Fetzer, The Timed Asynchronous Distributed System Model»

В работе рассматривается более реалистичная модель распределенных систем, чем чисто синхронный или чисто асинхронный подходы. Введены два важных предположения — о том, что службы требуют для работы определенного временного интервала и о том, что связь ненадежна и допускает провалы пропускной способности. В работе показана применимость этой модели для воспроизведения основных свойств реальных распределенных систем.

Guerraoui and Schiper, «Software-Based Replication for Fault Tolerance»

Краткий и ясный обзор применимости репликации для повышения отказоустойчивости в распределенных системах. Обсуждается репликация на основе первичной копии и активная репликация, а также связь репликации с взаимодействием в группе.

Guerraoui and Schiper, «A Generic Consensus Service»

Одна из проблем, связанных с отказоустойчивостью в распределенных системах, состоит в слишком большом числе различных решений и протоколов, применимость которых в определенных ситуациях трудно оценить. В этом документе описывается каркас, который позволяет собрать различные отказоустойчивые протоколы в единую модульную структуру и реализовать в форме выделенной службы.

Jalote, Fault Tolerance in Distributed Systems

Один из немногих учебников, посвященных собственно отказоустойчивости в распределенных системах. Книга охватывает надежную ширококвещательную рассылку, репликацию и устойчивость процессов. Имеется отдельная глава по отказам программного обеспечения.

Obraczka, «Multicast Transport Protocols: A Survey and Taxonomy»

Существует большое количество протоколов массовой рассылки, созданных с целью обеспечить надежность в глобальных системах. Комбинацию масштабируемости и надежности на практике реализовать нелегко. Этот документ представляет собой обзор нескольких последних попыток реализации масштабируемой надежной массовой рассылки.

13.1.8. Защита

Anderson, Security Engineering: A Guide to Building Dependable Distributed Systems

Одна из немногих удачных попыток охватить все вопросы защиты. В книге обсуждаются базовые понятия — пароли, контроль доступа и криптография. Защита

в значительной степени зависит от прикладной области, поэтому вопросы защиты обсуждаются в нескольких предметных областях, включая военное дело, банки, медицинские системы. И наконец, рассматриваются социальные, организационные и политические аспекты защиты. Отличное начало для дальнейшего чтения и исследований.

Blaze et al, «The Role of Trust Management in Distributed Systems Security»

В этой работе утверждается, что крупномасштабные распределенные системы будущего должны быть в состоянии предоставлять доступ к ресурсам более простыми способами, чем имеющиеся сейчас системы. В частности, если известно, что запрос соответствует локальным правилам защиты, он должен удовлетворяться. Другими словами, авторизация должна происходить без отдельных этапов аутентификации и контроля доступа. Приводится соответствующая модель и описываются способы ее реализации.

Ellison and Schneier, «Ten Risks of PKI: What You're Not Being Told about Public Key Infrastructure»

Чтобы использовать открытые ключи, необходима инфраструктура для распространения и поддержания этих ключей. Построение подобной инфраструктуры — дело непростое, и в этой статье объясняется, почему.

Kaufman et al., Network Security

Эта авторитетная и местами остроумная книга является лучшим введением в вопросы сетевой защиты. В ней рассматриваются алгоритмы и протоколы секретных и открытых ключей, хэширование сообщений, идентификация, Kerberos и электронная почта. Наиболее хороши дискуссии между авторами (а иногда и споры автора с самим собой).

Menezes et al., Handbook of Applied Cryptography

Название говорит само за себя. В книге описывается математический аппарат, необходимый для понимания множества различных криптографических решений для шифрования, хэширования и т. п. Отдельные главы посвящены аутентификации, цифровым подписям, созданию и поддержанию ключей.

Schneier, Applied Cryptography

Великолепная книга, охватывающая все аспекты криптографии, а также описания и реализации множества протоколов защиты. Книга написана в первую очередь для ученых-компьютерщиков. Это легко заметить, если сравнить ее с книгами для математиков. Имеется исчерпывающий список ссылок на литературу для дополнительного чтения.

Schneier, Secrets and Lies

Книга написана тем же автором, что и предыдущая. Она позволяет донести вопросы защиты до людей, не имеющих к технике никакого отношения. Важно понять, что защита — это не только технологическая проблема. Прочтя эту книгу,

человек действительно понимает, что, возможно, большая часть рисков, связанных с защитой, исходят от людей и организации их действий. Это серьезное дополнение к тем материалам, которые мы излагали в главе 8.

Sherif and Sechrouchni, Protocols for Secure Electronic Commerce

Электронная коммерция — это новая область, для которой защита играет важнейшую роль. Эта книга охватывает множество протоколов организации защищенных транзакций в сетях, которым мы не доверяем. В круг обсуждаемых тем входят платежные системы, коммерческие операции между организациями, удаленные платежи по банковским картам, микроплатежи и многое другое.

13.1.9. Распределенные системы объектов

Eddon and Eddon, Inside Distributed COM

Один из лучших способов лучше понять модель Distributed COM — узнать, что происходит за кулисами. В этой книге содержится множество технических деталей, имеющих отношение к DCOM, с примерами кода. Она охватывает компоненты, библиотеки типов, потоки выполнения, мониторы, маршалинг, защиту и еще многое другое.

Emmerich, Engineering Distributed Objects

Великолепная книга, посвященная непосредственно технологии удаленных объектов и уделяющая особое внимание CORBA, DCOM и RMI для Java. Таким образом, она предоставляет хорошую базу для сравнения этих трех популярных объектных моделей. Кроме того, представлен материал по проектированию систем удаленных объектов, использованию различных способов связи, локализации объектов, сохранности, транзакциям и защите.

Grimshaw et al., «Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems

Legion — это глобальная распределенная система объектов, которая, подобно Globe, предназначена для поддержания глобальных приложений, но с упором на приложения, требующие интенсивных вычислений. Legion предлагает модель удаленных объектов. В рассматриваемом отчете содержится множество важных подробностей по структуре Legion, что позволяет сравнить эту систему с другими, о которых мы говорили в главе 9.

Henning and Vinoski, Advanced CORBA Programming with C++

Если вам необходима информация о программировании CORBA, вашим выбором должна стать эта книга. Написанная двумя людьми, которые занимались описанием и разработкой систем CORBA, книга наполнена практическими и тех-

ническими подробностями, которые не ограничиваются одной реализацией CORBA.

Siegel, «CORBA and the OMA in Enterprise Computing»

Эта статья представляет собой хорошее несложное введение в технологию CORBA без той массы деталей, которые имеются в главе 9. Статья в основном посвящена вопросам архитектуры CORBA.

13.1.10. Распределенные файловые системы

Groenvall et al., «The Design of a Multicast-based Distributed File System»

Существует множество распределенных файловых систем, часто отличающихся лишь минимальными особенностями архитектуры. В этом документе описывается абсолютно иной подход к файловым системам, названный JetFile и предполагающий крупномасштабное использование масштабируемой надежной групповой рассылки, которую мы обсуждали в главе 7. JetFile позволяет клиентам вести себя подобно серверам.

Lee et al., «Operation-based Update Propagation in a Mobile File System»

Одна из проблем, возникающих в файловых системах с поддержкой мобильных агентов при операциях в автономном режиме, состоит в том, что обновление крупных файлов — довольно дорогостоящая операция. В подобных случаях при очередном подключении необходимо передать с клиента на сервер весь файл целиком, используя для этого низкоскоростное соединение типа беспроводной сети. В этом документе описывается решение, состоящее в переносе операции обновления на суррогатного клиента, имеющего хорошую связь с сервером.

Rao and Peterson, «Accessing Files in an Internet: The Jade File System»

В статье описана одна из новых файловых систем, которая позволяет пользователю Интернета комбинировать различные файловые системы. Модель именования Jade похожа на граф именования, который мы рассматривали в главе 4. Каждый пользователь строит собственное локальное пространство имен и монтирует удаленные пространства имен, задавая контактный адрес удаленного файлового сервера и требуемый протокол доступа.

Spasojevic and Satyanarayanan, «An Empirical Study of a Wide-Area Distributed File System»

В статье содержится описание рабочих характеристик файловой системы AFS, послужившей базой для создания Coda. Интересной статью делает тот факт, что AFS поддерживают более 1000 серверов и 20 000 клиентов, разбросанных по всему Интернету. Таким образом, это, может быть, самая большая из разработанных

и используемых распределенных файловых систем. В статье показано, что AFS действительно прекрасно поддается масштабированию, поскольку первоначально она была установлена в сети кампуса CMU.

Thekkath et al., «Frangipani: A Scalable Distributed File System»

Frangipani — распределенная файловая система, спроектированная для кластеров рабочих станций. Она создавалась как двухуровневая система, в которой нижний уровень образуют системы хранения, предоставляющие виртуальные диски. Виртуальный диск имеет очень большое, разделенное адресное пространство, блоки в котором выделяются только по запросу. Файловые серверы с легкостью могут совместно использовать файлы, хранящиеся на одном виртуальном диске. Такой подход делает общую структуру Frangipani относительно простой.

Triantafillou and Neilson, «Achieving Strong Consistency in a Distributed File System»

Для простоты большинство распределенных файловых систем используют протоколы репликации с первичной копией, которые позволяют добиться производительности при высокой степени доступности. В этой работе обсуждается протокол реплицируемой записи, эффективно реализующий семантику разделения файлов UNIX. Протокол интересен тем, что показывает, сколько вопросов нужно уточнить и прояснить при использовании реплицируемой записи.

13.1.11. Распределенные системы документов

Dourish et al., «Extending Document Management Systems with User-Specific Active Properties»

В этой работе описывается распределенная система управления документами под названием Placeless, в которой документы систематизируются в соответствии с их свойствами, а не местоположением. Документы содержат присоединенный код, например, для протоколирования информации о доступе или составления рефератов. Эти активные свойства являются той особенностью, которая отличает Placeless от систем, подобных Lotus Notes.

Pierre et al., «Self-Replicating Web Documents»

В этом отчете описывается результат расширения концепции web-документов, с тем, чтобы каждый документ мог иметь собственную стратегию репликации. Стратегия может динамически адаптироваться, изменяя сценарий работы. Само-реплицирующиеся web-документы позволяют поддерживать строгую непротиворечивость с минимизацией потребности в пропускной способности сети и времени доступа для клиента, что невозможно при использовании единой глобальной стратегии репликации всех документов.

Qiu et al., «On the Placement of Web Server Replicas»

В статье изложены некоторые алгоритмы размещения M web-серверов в сети из N хостов. Используя понятие стоимости размещения, можно создать несложный алгоритм, который поместит первую реплику на узел с минимальной стоимостью, вторую реплику — на следующий по дешевизне узел, и т. д., давая результат, близкий к оптимальному. Обсуждаются различные алгоритмы расчета стоимости размещения, в том числе и основанные на измерении расстояния и загрузке запросов.

Rodriguez and Sibal, «SPREAD: Scalable Platform for Reliable and Efficient Automated Distribution»

В статье описывается один из подходов к использованию сетей CDN. В SPREAD сеть заместителей динамически конфигурируется для доставки наиболее важного содержимого. Отличительная особенность этой системы состоит в том, что заместители динамически, для каждого web-документа в отдельности решают, распространять ли его изменения методом продвижения или извлечения и следует ли посылать сообщения о недействительности документа или полные обновления. Для передачи трафика с одного заместителя на другой используют информацию о маршрутизации низкого уровня.

Wang, «A Survey of Web Caching Schemes for the Internet»

Приведен обзор разных схем кэширования статических web-документов. В статье помимо других тем обсуждается архитектура, кооперативное кэширование, предварительная доставка документов, алгоритмы замещения кэша и согласованность кэша. Имеется обширный список ссылок.

13.1.12. Распределенные системы согласования**Banavar et al., «A Case for Message-Oriented Middleware»**

Авторы обсуждают, как построить слабо связанную распределенную систему для обмена событиями и сообщениями с особым упором на архитектуру публикации/подписки.

Carzaniga et al., «Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service»

Создание в глобальных сетях систем на основе сообщений — серьезная задача. В этой работе описывается система, в которой издатели отождествляются с подписчиками, с относительно простой моделью данных, позволяющей находить эффективные способы маршрутизации, отталкиваясь от содержимого. В ней используется комбинация технологий подписки и рассылки предложений на основе групповой рассылки. Для предотвращения рассылки информации туда, где она не нужна, задействуются технологии фильтрации.

Eugster et al., «The Many Faces of Publish/Subscribe»

Хороший легкий для понимания обзор различных схем публикации/подписки, которые могут использоваться в распределенных системах. Рассматриваются отношение к существующим схемам, таким как RPC, очереди сообщений и пространства кортежей, которые также могут иметь свои варианты схем публикации/подписки (например, на базе тем, содержимого и типов).

Papadopoulos and Arbab, «Coordination Models and Languages»

Исчерпывающее руководство по моделям и языкам согласования. Эта книга не предназначена конкретно для изучения распределенных систем, но содержит описания различных моделей, разработанных для распределенных систем, например Linda. Книга основывается на простой таксономии моделей, отделяющей модели, управляемые данными, от моделей с управляющими элементами.

Wyckhoff et al., «T Spaces»

Вводится важное расширение модели пространств кортежей, которую мы описывали в главе 12, — возможность сохранения кортежей в базах данных. Этот подход, примененный в T-пространствах, позволяет получить мощные процедуры поиска и сопоставления кортежей, имеющиеся обычно в базах данных. Кроме того, использование баз данных означает, что пространства кортежей добавляются к модели данных, что делает их более структурированными, а это, в свою очередь, помогает управляться с проблемами больших наборов кортежей.

13.2. Список литературы

1. Abadi, M. and Needham, R.: «Prudent Engineering Practice for Cryptographic Protocols.» IEEE Trans. Softw. Eng., vol. 22, no. 1, pp. 6–15, Jan. 1996.
2. Abdullahi, S. and Ringwood, G.: «Garbage Collecting the Internet: A Survey of Distributed Garbage Collection.» ACM Comput. Surv., vol. 30, no. 3, pp. 330–373, Sept. 1998.
3. Adler, R.: «Distributed Coordination Models for Client/Server Computing.» IEEE Computer, vol. 28, no. 4, pp. 14–22, Apr. 1995.
4. Adve, S. and Gharachorloo, K.: «Shared Memory Consistency Models: A Tutorial.» IEEE Computer, vol. 29, no. 12, pp. 66–76, Dec. 1996.
5. Ahamad, M., Bazzi, R., John, R., Kohli, P., and Neiger, G.: «The Power of Processor Consistency.» Technical Report GIT-CC-92/34, College of Computing, Georgia Institute of Technology, Dec. 1992.
6. Ahamad, M. and Kordale, R.: «Scalable Consistency Protocols for Distributed Services.» IEEE Trans. Par. Distr. Syst., vol. 10, no. 9, pp. 888–903, Sept. 1999.
7. Ahuja, S., Carriero, N., Gelernter, D., and Krishnaswamy, V.: «Matching Languages and Hardware for Parallel Computation in the Linda Machine.» IEEE Trans. Comp., vol. 37, no. 8, pp. 921–929, Aug. 1988.

8. Albitz, P. and Liu, C.: DNS and BIND. Sebastopol, CA: O'Reilly & Associates, 3rd ed., 1998.
9. Alvestrand, H.: «Mapping between X.400 and RFC-822/MIME Message Bodies.» RFC 2157, Jan. 1998.
10. Alvisi, L. and Marzullo, K.: «Message Logging: Pessimistic, Optimistic, Causal, and Optimal.» IEEE Trans. Softw. Eng., vol. 24, no. 2, pp. 149–159, Feb. 1998.
11. Amir, V., Peterson, A., and Shaw, D.: «Seamlessly Selecting the Best Copy from Internet-Wide Replicated Web Servers.» Proc. Int'l Symp. Distributed Computing (DISC), 1998. pp. 22–33.
12. Anceaume, E. and Puaut, I.: «Performance Evaluation of Clock Synchronization Algorithms.» Technical Report RR-3526, INRIA, Rennes, France, Oct. 1998.
13. Anderson, R.: Security Engineering — A Guide to Building Dependable Distributed Systems. New York: John Wiley, 2001.
14. Anderson, T., Bershad, B., Lazowska, E., and Levy, H.: «Scheduler Activations: Efficient Kernel Support for the User-Level Management of Parallelism.» Proc. 13th Symp. Operating System Principles. ACM, 1991. pp. 95–109.
15. Anderson, T., Dahlin, M., Neefe, J., Roselli, D., Patterson, D., and Wang, R.: «Serverless Network File Systems.» ACM Trans. Comp. Syst., vol. 14, no. 1, pp. 41–79, Feb. 1996.
16. Anderson, T., Culler, D., Patterson, D., and The Now Team: «A Case for NOW.» IEEE Micro, vol. 15, no. 2, pp. 54–64, Feb. 1995.
17. Andrews, G.: Foundations of Multithreaded, Parallel, and Distributed Programming. Reading, MA: Addison-Wesley, 2000.
18. Aron, M., Sanders, D., Druschel, P., and Zwaenepoel, W.: «Scalable Content-aware Request Distribution in Cluster-based Network Servers.» Proc. Ann. Techn. Conf USENIX, 2000. pp. 323–336.
19. Asokan, N., Janson, P., Steiner, M., and Waidner, M.: «The State of the Art in Electronic Payment Systems.» IEEE Computer, vol. 30, no. 9, pp. 28–35, Sept. 1997.
20. Attiya, H. and Welch, J.: «Sequential Consistency versus Linearizability.» ACM Trans. Comp. Syst., vol. 12, no. 2, pp. 91–122, May 1994.
21. Babaoglu, O. and Marzullo, K.: «Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms.» In Mullender, S. (ed.), Distributed Systems, pp. 55–96. Wokingham: Addison-Wesley, 2nd ed., 1993.
22. Babaoglu, O. and Toueg, S.: «Non-Blocking Atomic Commitment.» In Mullender, S. (ed.), Distributed Systems, pp. 147–168. Wokingham: Addison-Wesley, 2nd ed., 1993.
23. Baggio, A., Ballintijn, G., and Van Steen, M.: «Mechanisms for Effective Caching in the Globe Location Service.» Proc. Ninth SIGOPS European Workshop. ACM, 2000. pp. 55–60.
24. Baggio, A., Ballintijn, G., Van Steen, M., and Tanenbaum, A.: «Efficient Tracking of Mobile Objects in Globe.» Comp. J., vol. 44, no. 5, 2001.

25. Baker, S.: *Corba Distributed Objects Using Orbix*. Reading, MA: Addison-Wesley, 1997.
26. Bakken, D. and Schlichting, R.: «Supporting Fault-Tolerant Parallel Programming in Linda.» *IEEE Trans. Par. Distr. Syst.*, vol. 6, no. 3, pp. 287–302, Mar. 1995.
27. Bal, H.: *The Shared Data-Object Model as a Paradigm for Programming Distributed Systems*. PhD. Thesis, Vrije Universiteit, Amsterdam, 1989.
28. Bal, H., Bhoedjang, R., Hofman, R., Jacobs, C., Langendoen, K., Ruhl, T., and KAASHOEK, M.: «Performance Evaluation of the Orca Shared Object System.» *ACM Trans. Comp. Syst.*, vol. 16, no. 1, pp. 1–40, Feb. 1998.
29. Bal, H. and Kaashoek, M.: «Object Distribution in Orca using Compile-Time and Run-Time Techniques.» *Proc. Eighth Ann. Conf Object-Oriented Programming Systems and Languages (OOPSLA)*. ACM, 1993. pp. 162–177.
30. Bal, H., Kaashoek, M., and Tanenbaum, A.: «Orca: A Language for Parallel Programming of Distributed Systems.» *IEEE Trans. Softw. Eng.*, vol. 18, no. 3, pp. 190–205, Mar. 1992.
31. Ballintijn, G., Van Steen, M., and Tanenbaum, A.: «Exploiting Location Awareness for Scalable Location-Independent Object IDs.» *Proc. Fifth ASCI Ann. Conf. ASCI*, 1999. pp. 321–328.
32. Ballintijn, G., Van Steen, M., and Tanenbaum, A.: «A Scalable Implementation for Human-Friendly URIs.» *IEEE Internet Comput.*, vol. 5, no. 5, Sept. 2001.
33. Ballintijn, G., Verkaik, P., Crawl, D., Baggio, A., and Van Steen, M.: «The Globe Location Server.» *Technical Report*, Vrije Universiteit, Department of Mathematics and Computer Science, 2001.
34. Banavar, G., Chandra, T., Strom, R., and Sturman, D.: «A Case for Message-Oriented Middleware.» In *Proc. DISC*, vol. 1693 of *Lect. Notes Comp. Sc.*, pp. 1–18. Berlin: Springer-Verlag, Sept. 1999.
35. Banavar, G., Chandra, T., Mukherjee, B., Nagarajao, J., Strom, R., and Sturman, D.: «An Efficient Multicast Protocol for Content-based Publish-Subscribe Systems.» *Proc. 19th Int'l Conf on Distributed Computing Systems*. IEEE, 1999.
36. Barber, S.: «Common NNTP Extensions.» *RFC 2980*, Oct. 2000.
37. Barborak, M., Malek, M., and Dahbura, A.: «The Consensus Problem in Fault-Tolerant Computing.» *ACM Comput. Surv.*, vol. 25, no. 2, pp. 171–220, June 1993.
38. Barford, P., Bestavros, A., Bradley, A., and Crovella, M. E.: «Changes in Web Client Access Patterns: Characteristics and Caching Implications.» *World Wide Web*, vol. 2, no. 1–2, pp. 15–28, Aug. 1999.
39. Barish, G. and Obraczka, K.: «World Wide Web Caching: Trends and Techniques.» *IEEE Commun. Mag.*, vol. 38, no. 5, pp. 178–184, May 2000.
40. Barron, D.: *Pascal — The Language and its Implementation*. New York: John Wiley, 1981.

41. Bass, L., Clements, P., and Kazman, R.: *Software Architecture in Practice*. Reading, MA: Addison-Wesley, 1998.
42. Bell Labs Computing Science Research Center: *Plan 9 Programmer's Manual*, Vol. 1. Bell Laboratories, Lucent Technologies, Murray Hill, NJ, 3rd ed., 2000.
43. Berners-lee, T., Cailliau, R., Nielson, H. F., and Secret, A.: «The World-Wide Web.» *Commun. ACM*, vol. 37, no. 8, pp. 76–82, Aug. 1994.
44. Berners-lee, T., Fielding, R., and Masinter, L.: «Uniform Resource Identifiers (URI): Generic Syntax.» RFC 2396, Aug. 1998.
45. Bernstein, P.: «Middleware: A Model for Distributed System Services.» *Commun. ACM*, vol. 39, no. 2, pp. 87–98, Feb. 1996.
46. Bernstein, P. and Goodman, N.: «Concurrency Control in Distributed Database Systems.» *ACM Comput. Surv.*, vol. 13, no. 2, pp. 185–221, June 1981.
47. Bernstein, P., Hadzilacos, V., and Goodman, N.: *Concurrency Control and Recovery in Database Systems*. Reading, MA: Addison-Wesley, 1987.
48. Bershad, B., Zekauskas, M., and Sawdon, W.: «The Midway Distributed Shared Memory System.» *Proc. COMPCON. IEEE*, 1993. pp. 528–537.
49. Bershad, B., Anderson, T., Lazowska, E., and Levy, H.: «Lightweight Remote Procedure Call.» *ACM Trans. Comp. Syst.*, vol. 8, no. 1, pp. 37–55, Feb. 1990.
50. Bershad, B. and Zekauskas, M.: «Midway: Shared Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors.» Technical Report CMU-CS-91-170, Carnegie Mellon University, Sept. 1991.
51. Bhoedjang, R., Ruhl, T., Hofman, R., Langendoen, K., Bal, H., and Kaashoek, F.: «Panda: A Portable Platform to Support Parallel Programming Languages.» *Proc. Symp. on Experiences with Distributed and Multiprocessor Systems IV*, 1993. pp. 213–226.
52. Bhoedjang, R., Ruhl, T., and Bal, H.: «User-Level Network Interface Protocols.» *IEEE Computer*, vol. 31, no. 11, pp. 53–60, Nov. 1998.
53. Bhoedjang, R.: *Communication Architectures for Parallel Programming Systems*. Ph.D. Thesis, Vrije Universiteit, Department of Mathematics and Computer Science, Amsterdam, June 2000.
54. Birman, K.: «A Response to Cheriton and Skeen's Criticism of Causal and Totally Ordered Communication.» *Oper. Syst. Rev.*, vol. 28, no. 1, pp. 11–21, Jan. 1994.
55. Birman, K.: *Building Secure and Reliable Network Applications*. Englewood Cliffs, NJ: Prentice Hall, 1996.
56. Birman, K. and Joseph, T.: «Exploiting Virtual Synchrony in Distributed Systems.» *Proc. 11th Symp. Operating System Principles. ACM*, 1987. pp. 123–138.
57. Birman, K. and Joseph, T.: «Reliable Communication in the Presence of Failures.» *ACM Trans. Comp. Syst.*, vol. 5, no. 1, pp. 47–76, Feb. 1987.
58. Birman, K., Schiper, A., and Stephenson, P.: «Lightweight Causal and Atomic Group Multicast.» *ACM Trans. Comp. Syst.*, vol. 9, no. 3, pp. 272–314, Aug. 1991.

59. Birman, K. and Van Renesse, R. (eds.): *Reliable Distributed Computing with the Isis Toolkit*. Los Alamitos, CA: IEEE Computer Society Press, 1994.
60. Birrell, A., Evers, D., Nelson, G., Owicki, S., and Wobber, E.: «Distributed Garbage Collection for Network Objects.» Technical Report SRC-116, Digital Systems Research Center, Palo Alto, CA, Dec. 1993.
61. Birrell, A., Levin, R., Needham, R., and Schroeder, M.: «Grapevine: An Exercise in Distributed Computing.» *Commun. ACM*, vol. 25, no. 4, pp. 260–274, Apr. 1982.
62. Birrell, A. and Nelson, B.: «Implementing Remote Procedure Calls.» *ACM Trans. Comp. Syst.*, vol. 2, no. 1, pp. 39–59, Feb. 1984.
63. Bjornson, R.: *Linda on Distributed Memory Multicomputers*. Ph.D. Thesis, Yale University, Department of Computer Science, 1993.
64. Black, A. and Artsy, V.: «Implementing Location Independent Invocation.» *IEEE Trans. Par. Distr. Syst.*, vol. 1, no. 1, pp. 107–119, Jan. 1990.
65. Blair, G. and Stefani, J.-B.: *Open Distributed Processing and Multimedia*. Reading, MA: Addison-Wesley, 1998.
66. Blakley, B.: *CORBA Security*. Reading, MA: Addison-Wesley, 2000.
67. Blaze, M., Feigenbaum, J., Ioannidis, J., and Keromytis, A.: «The Role of Trust Management in Distributed Systems Security.» In Vitek, J. and Jensen, C. (eds.), *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, vol. 1603 of *Lect. Notes Comp. Sc.*, pp. 185–210. Berlin: Springer-Verlag, 1999.
68. Blaze, M.: *Caching in Large-Scale Distributed File Systems*. PhD thesis, Department of Computer Science, Princeton University, Jan. 1993.
69. Bloomer, J.: *Power Programming with RPC*. Sebastopol, CA: O'Reilly & Associates, 1992.
70. Boden, N., Cohen, D., Felderman, R., Kulawik, A., Seitz, C., Seizovic, J., and Su, W.: «Myrinet — A Gigabit-per-Second Local-Area Network.» *IEEE Micro*, vol. 15, no. 2, pp. 29–36, Feb. 1995.
71. Bracha, G. and Toueg, S.: «Distributed Deadlock Detection.» *Distributed Computing*, vol. 2, pp. 127–138, 1987.
72. Braden, R., Zhang, L., Berson, S., Herzog, S., and Jamin, S.: «Resource Reservation Protocol (RSVP) — Version 1 Functional Requirements.» RFC 2205, Sept. 1997.
73. Bray, T., Paoli, J., Sperberg-McQueen, C., and Maler, E.: «Extensible Markup Language (XML) 1.0 (Second Edition).» W3C Recommendation, Oct. 2000.
74. Brewington, B., Gray, R., Moizumi, K., Kotz, D., Cybenko, G., and Rus, D.: «Mobile Agents for Distributed Information Retrieval.» In Klusch, M. (ed.), *Intelligent Information Agents*, pp. 355–395. Berlin: Springer-Verlag, 1999.
75. Briot, J.-P., Guerraoui, R., and Lohr, K.-P.: «Concurrency, Distribution and Parallelism in Object-Oriented Programming.» *ACM Comput. Surv.*, vol. 30, no. 3, pp. 291–329, Sept. 1998.

76. Budhijara, N., Marzullo, K., Schneider, F., and Toueg, S.: «The Primary-Backup Approach.» In Mullender, S. (ed.), *Distributed Systems*, pp. 199–216. Wokingham: Addison-Wesley, 2nd ed., 1993.
77. Budhiraja, N. and Marzullo, K.: «Tradeoffs in Implementing Primary-Backup Protocols.» Technical Report TR 92–1307, Department of Computer Science, Cornell University, 1992.
78. Buretta, M.: *Data Replication: Tools and Techniques for Managing Distributed Information*. New York: John Wiley, 1997.
79. Cabri, G., Leonardi, L., and Zambonelli, F.: «Mobile-Agent Coordination Models for Internet Applications.» *IEEE Computer*, vol. 33, no. 2, pp. 82–89, Feb. 2000.
80. Callaghan, B.: *NFS Illustrated*. Reading, MA: Addison-Wesley, 2000.
81. Camp, L.: *Privacy and Reliability in Internet Commerce*. Ph.D. Thesis, Carnegie Mellon University, Aug. 1996.
82. Cao, P., Zhang, J., and Beach, K.: «Active Cache: Caching Dynamic Contents on the Web.» *Proc. Middleware '98. IFIP*, 1998. pp. 373–388.
83. Cao, P. and Liu, C.: «Maintaining Strong Cache Consistency in the World Wide Web.» *IEEE Trans. Comp.*, vol. 47, no. 4, pp. 445–457, Apr. 1998.
84. Carriero, N. and Gelernter, D.: «The S/Net's Linda Kernel.» *ACM Trans. Comp. Syst.*, vol. 32, no. 2, pp. 110–129, May 1986.
85. Carriero, N. and Gelernter, D.: «How to Write Parallel Programs: A Guide to the Perplexed.» *ACM Comput. Surv.*, vol. 21, no. 3, pp. 323–358, 1989.
86. Carzaniga, A., Rosenblum, D. S., and Wolf, A. L.: «Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service.» *Proc. 19th Symp. on Principles of Distributed Computing. ACM*, 2000. pp. 219–227.
87. Cate, V.: «Alex — A Global File System.» *Proc. File Systems Workshop. USENIX*, 1992. pp. 1–11.
88. Chadwick, D.: *Understanding X.500. The Directory*. London: Chapman & Hall, 1994.
89. Chandy, K. and Lamport, L.: «Distributed Snapshots: Determining Global States of Distributed Systems.» *ACM Trans. Comp. Syst.*, vol. 3, no. 1, pp. 63–75, Feb. 1985.
90. Chang, J. and Maxemchuk, N.: «Reliable Broadcast Protocols.» *ACM Trans. Comp. Syst.*, vol. 2, no. 3, pp. 251–273, Aug. 1984.
91. Chankhunthod, A., Danzig, P., Neerdaels, C., Schwartz, M., and Worrell, K.: «A Hierarchical Internet Object Cache.» *Proc. Ann. Techn. Conf. USENIX*, 1996. pp. 153–163.
92. Chappell, D.: *Understanding Windows 2000 Distributed Services*. Redmond, WA: Microsoft Press, 2000.
93. Chaum, D.: «Blind Signatures for Untraceable Payments.» *Proc. Crypto '82*, 1982. pp. 199–203.
94. Chaum, D.: «Security without Identification: Transaction Systems to make Big Brother Obsolete.» *Commun. ACM*, vol. 28, no. 10, pp. 1030–1044, Oct. 1985.

95. Chaum, D.: «Achieving Electronic Privacy.» *Scientific American*, vol. 267, no. 2, pp. 96–101, Aug. 1992.
96. Chawathe, V. and Brewer, E.: «System Support for Scalable and Fault Tolerant Internet Services.» *Proc. Middleware '98. IFIP*, 1998. pp. 71–88.
97. Chen, P., Lee, E., Gibson, G., Katz, R., and Patterson, D.: «RAID: High-Performance, Reliable Secondary Storage.» *ACM Comput. Surv.*, vol. 26, no. 2, pp. 145–186, June 1994.
98. Cheriton, D. and Mann, T.: «Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance.» *ACM Trans. Comp. Syst.*, vol. 7, no. 2, pp. 147–183, May 1989.
99. Cheriton, D. and Skeen, D.: «Understanding the Limitations of Causally and Totally Ordered Communication.» *Proc. 14th Symp. Operating System Principles. ACM*, 1993. pp. 44–57.
100. Chervenak, A., Foster, I., Kesselman, C., Salisbury, C., and Tuecke, S.: «The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets.» *J. Netw. Comp. App.*, vol. 23, no. 3, pp. 187–200, July 2000.
101. Cheswick, W. and Bellovin, S.: *Firewalls and Internet Security*. Reading, MA: Addison-Wesley, 2nd ed., 2000.
102. Chockler, G. V., Dolev, D., Friedman, R., and Vitenberg, R.: «Implementing a Caching Service for Distributed CORBA Objects.» In *Proc. Middleware 2000*, vol. 1795 of *Lect. Notes Comp. Sc.*, pp. 1–23. Berlin: Springer-Verlag, 2000.
103. Chow, R. and Johnson, T.: *Distributed Operating Systems and Algorithms*. Reading, MA: Addison-Wesley, 1997.
104. Chun, B., Mainwaring, A., and Culler, D.: «Virtual Network Transport Protocols for Myrinet.» *IEEE Micro*, vol. 18, no. 1, pp. 53–63, Jan. 1998.
105. Chung, P. E., Huang, Y., Yajnik, S., Liang, D., Smith, J. C., Wang, C.-Y., and Wang, Y.-M.: «DCOM and CORBA Side by Side, Step by Step, and Layer by Layer.» *C++ Report*, vol. 10, no. 1, pp. 18–29, Jan. 1998.
106. Ciancarini, P., Omicini, A., and Zambonelli, F.: «Coordination for Internet Agents.» *Nordic. J. Comput.*, vol. 6, no. 3, pp. 215–240, 1999.
107. Ciancarini, P., Tolksdorf, R., Vitali, F., and Knoche, A.: «Coordinating Multiagent Applications on the WWW: A Reference Architecture.» *IEEE Trans. Softw. Eng.*, vol. 24, no. 5, pp. 362–375, May 1998.
108. Cohen, D.: «On Holy Wars and a Plea for Peace.» *IEEE Computer*, vol. 14, no. 10, pp. 48–54, Oct. 1981.
109. Comer, D.: *Internetworking with TCP/IP, Volume I: Principles, Protocols, and Architecture*. Upper Saddle River, NJ: Prentice Hall, 4th ed., 2000.
110. Comer, D.: *The Internet Book*. Upper Saddle River, NJ: Prentice Hall, 3rd ed., 2000.
111. Coppersmith, D.: «The Data Encryption Standard (DES) and its Strength Against Attacks.» *IBM J. Research and Development*, vol. 38, no. 3, pp. 243–250, May 1994.

112. Coulouris, G., Dollimore, J., and Kindberg, T.: *Distributed Systems, Concepts and Design*. Wokingham: Addison-Wesley, 3rd ed., 2001.
113. Cristian, F.: «Probabilistic Clock Synchronization.» *Distributed Computing*, vol. 3, pp. 146–158, 1989.
114. Cristian, F.: «Understanding Fault-Tolerant Distributed Systems.» *Commun. ACM*, vol. 34, no.2, pp. 56–78, Feb. 1991.
115. Cristian, F. and Fetzer, C.: «The Timed Asynchronous Distributed System Model.» *IEEE Trans. Par. Distr. Syst.*, vol. 10, no. 6, pp. 642–657, June 1999.
116. Crowcroft, J., Handley, M., and Wakeman, I.: *Internetworking Multimedia*. London: UCL Press, 1999.
117. Crowley, C.: *Operating Systems, A Design-Oriented Approach*. Chicago: Irwin, 1997.
118. Davidson, S., Garcia-Molina, H., and Skeen, D.: «Consistency in Partitioned Networks.» *ACM Comput. Surv.*, vol. 17, no. 3, pp. 341–370, Sept. 1985.
119. Davies, D. and Price, W.: *Security for Computer Networks: An Introduction to Data Security in Teleprocessing and Electronic Funds Transfer*. Chichester: John Wiley, 2nd ed., 1989.
120. Day, J. and Zimmerman, H.: «The OSI Reference Model.» *Proceedings of the IEEE*, vol. 71, no. 12, pp. 1334–1340, Dec. 1983.
121. Deering, S., Estrin, D., Farinacci, D., Jacobson, V., Liu, C.-G., and Wei, L.: «The PIM Architecture for Wide-Area Multicast Routing.» *IEEE/ACM Trans. Netw.*, vol. 4, no. 2, pp. 153–162, Apr. 1996.
122. Deering, S. and Cheriton, D.: «Multicast Routing in Datagram Internetworks and Extended LANs.» *ACM Trans. Comp. Syst.*, vol. 8, no. 2, pp. 85–110, May 1990.
123. Deitel, H., Deitel, P., and Nieto, T.: *Internet and World Wide Web – How to Program*. Upper Saddle River, NJ: Prentice Hall, 2000.
124. Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., and Terry, D.: «Epidemic Algorithms for Replicated Data Management.» *Proc. Sixth Symp. on Principles of Distributed Computing*. ACM, 1987. pp. 1–12.
125. Dierks, T. and Allen, C.: «The Transport Layer Security Protocol.» *RFC 2246*, Jan. 1996.
126. Diffie, W. and Hellman, M.: «New Directions in Cryptography.» *IEEE Trans. Information Theory*, vol. IT-22, no. 6, pp. 644–654, Nov. 1976.
127. Dimitrov, B. and Rego, V.: «Arachne: A Portable Threads System Supporting Migrant Threads on Heterogeneous Network Farms.» *IEEE Trans. Par. Distr. Syst.*, vol. 9, no. 5, pp. 459–469, May 1998.
128. Dourish, P., Edwards, K., Lamarca, A., Lamping, J., Petersen, K., Salisbury, M., Terry, D., and Thornton, J.: «Extending Document Management Systems with User-Specific Active Properties.» *ACM Trans. Inf Syst.*, vol. 18, no. 2, pp. 140–170, Apr. 2000.

129. Drummond, R. and Babaoglu, O.: «Low-Cost Clock Synchronization.» *Distributed Computing*, vol. 6, pp. 193–203, 1993.
130. Dubois, M., Scheurich, C., and Briggs, F.: «Synchronization, Coherence, and Event Ordering in Multiprocessors.» *IEEE Computer*, vol. 21, no. 2, pp. 9–21, Feb. 1988.
131. Duvvuri, V., Shenoy, P., and Tewari, R.: «Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web.» *Proc. 19th INFOCOM Conf. IEEE*, 2000. pp. 834–843.
132. Eddon, G. and Eddon, H.: *Inside Distributed COM*. Redmond, WA: Microsoft Press, 1998.
133. Eisler, M.: «Lipkey — A Low Infrastructure Public Key Mechanism Using SPKM.» *RFC 2847*, June 2000.
134. Eisler, M., Chiu, A., and Ling, L.: «RPCSEC_GSS Protocol Specification.» *RFC 2203*, Sept. 1997.
135. Ellison, C. and Schneier, B.: «Ten Risks of PKI: What You're Not Being Told about Public Key Infrastructure.» *Computer Security Journal*, vol. 16, no. 1, pp. 1–7, Jan. 2000.
136. Elnozahy, E., Johnson, D., and Zwaenepoel, W.: «The Performance of Consistent Checkpointing.» *Proc. 12th Symp. on Reliable Distributed Systems. IEEE*, 1992. pp. 39–47.
137. Elnozahy, E., Johnson, D., and Wang, V.: «A Survey of Rollback-Recovery Protocols in Message-Passing Systems.» *Technical Report CMU-CS-96-181*, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, Oct. 1996.
138. Emmerich, W.: *Engineering Distributed Objects*. New York: John Wiley, 2000.
139. Eswaran, K., Gray, J., Lorie, R., and Traiger, I.: «The Notions of Consistency and Predicate Locks in a Database System.» *Commun. ACM*, vol. 19, no. 11, pp. 624–633, Nov. 1976.
140. Eugster, P., Felber, P., Guerraoui, R., and Kermarrec, A.-M.: «The Many Faces of Publish/Subscribe.» *Technical Report MSR-TR-2001-104*, Microsoft Research Laboratories, Cambridge, UK, Jan. 2001.
141. Farmer, W. M., Guttman, J. D., and Swarup, V.: «Security for Mobile Agents: Issues and Requirements.» *Proc. 19th National Information Systems Security Conf*, 1996. pp. 591–597.
142. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T.: «Hypertext Transfer Protocol — HTTP/1.1.» *RFC 2616*, June 1999.
143. FIPA: *FIPA 97 Specification, Version 2.0 — Agent Communication Language*. Foundation for Intelligent Physical Agents, Geneva, Oct. 1998.
144. FIPA: *FIPA 98 Specification, Version 1.0 — Agent Management*. Foundation for Intelligent Physical Agents, Geneva, Oct. 1998.
145. Fischer, M., Lynch, N., and Patterson, M.: «Impossibility of Distributed Consensus with one Faulty Processor.» *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.

146. Floyd, S., Jacobson, V., Mccanne, S., Liu, C.-G., and Zhang, L.: «A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing.» IEEE/ACM Trans. Netw., vol. 5, no. 6, pp. 784–803, Dec. 1997.
147. Foster, I. and Kesselman, C. (eds.): Computational Grids: The Future of High Performance Distributed Computing. San Mateo, CA: Morgan Kaufman, 1998.
148. Foster, I., Kesselman, C., Tsudik, G., and Tuecke, S.: «A Security Architecture for Computational Grids.» Proc. Fifth Conf Computer and Communications Security. ACM, 1998. pp. 83–92.
149. Fowler, R.: Decentralized Object Finding Using Forwarding Addresses. Ph.D. Thesis, University of Washington, Seattle, 1985.
150. Fox, A., Gribble, S. D., Chawathe, V., Brewer, E. A., and Gauthier, P.: «Cluster-Based Scalable Network Services.» Proc. 16th Symp. Operating System Principles. ACM, 1997. pp. 78–91.
151. Franklin, M. J., Carey, M. J., and Livny, M.: «Transactional Client-Server Cache Consistency: Alternatives and Performance.» ACM Trans. Database Syst., vol. 22, no. 3, pp. 315–363, Sept. 1997.
152. Franklin, S. and Graesser, A.: «Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents.» Proc. Third Int'l Workshop on Agent Theories, Architectures, and Languages. 1996. pp. 21–35.
153. Fredrickson, N. and Lynch, N.: «Electing a Leader in a Synchronous Ring.» J. ACM, vol. 34, no. 1, pp. 98–115, Jan. 1987.
154. Freed, N. and Borenstein, N.: «Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types.» RFC 2046, Nov. 1996.
155. Freeman, E., Hupfer, S., and Arnold, K.: JavaSpaces, Principles, Patterns and Practice. Reading, MA: Addison-Wesley, 1999.
156. Fuggetta, A., Picco, G. P., and Vigna, G.: «Understanding Code Mobility.» IEEE Trans. Softw. Eng., vol. 24, no. 5, pp. 342–361, May 1998.
157. Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: Design Patterns, Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1994.
158. Garbinato, B., Guerraoui, R., and Mazouni, K.: «Distributed Programming in GARF.» In Guerraoui, R., Nierstrasz, O., and Riveill, M. (eds.), Object-Based Distributed Programming, vol. 791 of Lect. Notes Comp. Sc., pp. 225–239. Berlin: Springer-Verlag, 1994.
159. Garcia-Molina, H.: «Elections in a Distributed Computing System.» IEEE Trans. Comp., vol. 31, no. 1, pp. 48–59, Jan. 1982.
160. Garcia-Molina, H., Ullman, J. D., and Widom, I.: Database System Implementation. Upper Saddle River, NJ: Prentice Hall, 2000.
161. Garlan, D.: «Software Architecture: A Roadmap.» Proc. 22nd Int'l Conf Future of Software Engineering. ACM, 2000. pp. 93–101.
162. Geihs, K.: «Middleware Challenges Ahead.» IEEE Computer, vol. 34, no. 6, pp. 24–31, June 2001.

163. Gelernter, D.: «Generative Communication in Linda.» ACM Trans. Prog. Lang. Syst., vol. 7, no. 1, pp. 80–112, 1985.
164. Gelernter, D. and Carriero, N.: «Coordination Languages and their Significance.» Commun. ACM, vol. 35, no. 2, pp. 96–107, Feb. 1992.
165. Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J.: «Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors.» Proc. 17th Ann. Int'l Symp. on Computer Architecture. ACM, 1990. pp. 15–26.
166. Gibson, G. A., Nagle, D. F., Khalil Amiri, Butler, J., Chang, F. W., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D., and Zelenka, J.: «A Cost-Effective, High-Bandwidth Storage Architecture.» Proc. Eighth Int'l Conf Architectural Support for Programming Languages and Operating Systems. ACM, 1998. pp. 1–12.
167. Gifford, D.: «Weighted Voting for Replicated Data.» Proc. Seventh Symp. Operating System Principles. ACM, 1979. pp. 150–162.
168. Gifford, D.: «Cryptographic Sealing.» Commun. ACM, vol. 25, no. 4, pp. 274–286, Apr. 1982.
169. Gilman, L. and SCHREIBER, R.: Distributed Computing with IBM MQSeries. New York: John Wiley, 1996.
170. Gladney, H.: «Access Control for Large Collections.» ACM Trans. Inf Syst., vol. 15, no. 2, pp. 154–194, Apr. 1997.
171. Goland, Y., Whitehead, E., Faizi, A., Carter, S., and Jensen, D.: «HTTP Extensions for Distributed Authoring – WEBDAV.» RFC 2518, Feb. 1999.
172. Gollmann, D.: Computer Security. New York: John Wiley, 1999.
173. Gong, L. and SCHEMERS, R.: «Implementing Protection Domains in the Java Development Kit 1.2.» Proc. Symp. Network and Distributed System Security. Internet Society, 1998. pp. 125–134.
174. Goodman, J.: «Cache Consistency and Sequential Consistency.» Technical Report 61, IEEE Scalable Coherent Interface Working Group, 1989.
175. Gray, C. and Cheriton, D.: «Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency.» Proc. 12th Symp. Operating System Principles. ACM, 1989. pp. 202–210.
176. Gray, J., Helland, P., O'Neil, P., and Sashna, D.: «The Dangers of Replication and a Solution.» Proc. SIGMOD Int'l Conf on Management Of Data. ACM, 1996. pp. 173–182.
177. Gray, J. and Reuter, A.: Transaction Processing: Concepts and Techniques. San Mateo, CA: Morgan Kaufman, 1993.
178. Gray, J.: «Notes on Database Operating Systems.» In Bayer, R., Graham, R., and Seegmuller, G. (eds.), Operating Systems: An Advanced Course, vol. 60 of Lect. Notes Comp. Sc., pp. 393–481. Berlin: Springer-Verlag, 1978.
179. Gray, R.: «Agent Tcl: Alpha Release 1.1.» Technical Report, Dartmouth College, Hanover, NH, Dec. 1995.

180. Gray, R.: «Agent Tcl: A Flexible and Secure Mobile-Agent System.» Proc. Fourth Tcl/Tk Workshop. USENIX, 1996. pp. 9–23.
181. Grimshaw, A., Ferrari, A., Knabe, F., and Humphrey, M.: «Wide-Area Computing: Resource Sharing on a Large Scale.» IEEE Computer, vol. 32, no. 5, pp. 29–37, May 1999.
182. Grimshaw, A., Lewis, M., Ferrari, A., and Karpovich, I.: «Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems.» Technical Report CS-98-12, University of Virginia, Department of Computer Science, June 1998.
183. Groenvall, B., Westerlund, A., and Pink, S.: «The Design of a Multicast-based Distributed File System.» Proc. Third Symp. on Operating System Design and Implementation. USENIX, 1999. pp. 251–264.
184. Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., and Snir, M.: MPI: The Complete Reference — The MPI-2 Extensions. Cambridge, MA: MIT Press, 1998.
185. Gropp, W., Lusk, E., and Skjellum, A.: Using MPI, Portable Parallel Programming with the Message-Passing Interface. Cambridge, MA: MIT Press, 2nd ed., 1998.
186. Guerraoui, R. and Schiper, A.: «The Generic Consensus Service.» IEEE Trans. Softw. Eng., vol. 27, no. 1, pp. 29–41, Jan. 2001.
187. Guerraoui, R. and Schiper, A.: «Software-Based Replication for Fault Tolerance.» IEEE Computer, vol. 30, no. 4, pp. 68–74, Apr. 1997.
188. Gusella, R. and Zatti, S.: «The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3BSD.» IEEE Trans. Softw. Eng., vol. 15, no. 7, pp. 847–853, July 1989.
189. Guyton, J. and Schwartz, M.: «Locating Nearby Copies of Replicated Internet Services.» Proc. SIGCOMM. ACM, 1995. pp. 288–298.
190. Gwertzman, J. and Seltzer, M.: «The Case for Geographical Push-Caching.» Proc. Fifth Workshop Hot Topics in Operating Systems. IEEE, 1996. pp. 51–55.
191. Hadzilacos, V. and Toueg, S.: «Fault-Tolerant Broadcasts and Related Problems.» In Mullender, S. (ed.), Distributed Systems, pp. 97–145. Wokingham: Addison-Wesley, 2nd ed., 1993.
192. Halsall, F.: Multimedia Communications: Applications, Networks, Protocols and Standards. Reading, MA: Addison-Wesley, 2001.
193. Hamilton, G. and Kougiouris, P.: «The Spring Nucleus: A Microkernel for Objects.» Proc. Summer Techn. Conf USENIX, 1993. pp. 147–160.
194. Handel, R., Huber, M., and Schroder, S.: ATM Networks. Wokingham: Addison-Wesley, 2nd ed., 1994.
195. Hartman, J. and Ousterhout, J.: «The Zebra Striped Network File System.» ACM Trans. Comp. Syst., vol. 13, no. 3, pp. 274–310, Aug. 1995.
196. Hayes, C. C.: «Agents in a Nutshell — A Very Brief Introduction.» IEEE Trans. Know. Data Eng., vol. 11, no. 1, pp. 127–132, Jan. 1999.

197. Helary, J.: «Observing Global States of Asynchronous Distributed Applications.» In Proc. Int'l Workshop on Distributed Algorithms, vol. 392 of Lect. Notes Comp. Sc., pp. 124–135. Berlin: Springer-Verlag, 1989.
198. Henning, M. and Vinoski, S.: Advanced CORBA Programming with C++. Reading, MA: Addison-Wesley, 1999.
199. Herlihy, M. and Wing, J.: «Linearizability: A Correctness Condition for Concurrent Objects.» ACM Trans. Prog. Lang. Syst., vol. 12, no. 3, pp. 463–492, July 1991.
200. Hoare, C.: «Monitors: An Operating System Structuring Concept.» Commun. ACM, vol. 17, no. 10, pp. 549–557, Oct. 1974.
201. Iiofmann, M.: «A Generic Concept for Large-Scale Multicast.» In Plattner, B. (ed.), Broadband Communications — Networks, Services, Applications, Future Directions, vol. 1044 of Lect. Notes Comp. Sc., pp. 95–106. Berlin: Springer-Verlag, 1996.
202. Homburg, P. C.: The Architecture of a Worldwide Distributed System. Ph.D. Thesis, Vrije University Amsterdam, Department of Mathematics and Computer Science, 2001.
203. Horowitz, M. and Lunt, S.: «FTP Security Extensions.» RFC 2228, Oct. 1997.
204. Houttuin, J.: «A Tutorial on Gatewaying between X.400 and Internet Mail.» RFC 1506, Sept. 1993.
205. Howard, J., Kazar, M., Meenes, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., and West, M.: «Scale and Performance in a Distributed File System.» ACM Trans. Comp. Syst., vol. 6, no. 2, pp. 55–81, Feb. 1988.
206. Howes, T.: «The String Representation of LDAP Search Filters.» RFC 2254, Dec. 1997.
207. Hudson, R. L., Morrison, R., Moss, E. B., and Munro, D. S.: «Garbage Collecting the World: One Car at a Time.» SIGPLAN Notices, vol. 32, no. 10, pp. 162–175, Oct. 1997.
208. Hutto, P. and Ahamad, M.: «Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories.» Proc. Tenth Int'l Conf on Distributed Computing Systems. IEEE, 1990. pp. 302–311.
209. Islam, N.: Distributed Objects, Methodologies for Customizing Systems Software. Los Alamitos, CA: IEEE Computer Society Press, 1996.
210. ISO: «Open Distributed Processing Reference Model.» International Standard ISO/IEC IS 10746, 1995.
211. Jaeger, T., Prakash, A., Liedtke, J., and Islam, N.: «Flexible Control of Downloaded Executable Content.» ACM Trans. Inf Syst. Sec., vol. 2, no. 2, pp. 177–228, May 1999.
212. Jain, R.: «Reducing Traffic Impacts of PCS using Hierarchical User Location Databases.» Proc. Int'l Conf Communications. IEEE, 1996.
213. Jalote, P.: Fault Tolerance in Distributed Systems. Englewood Cliffs, NJ: Prentice Hall, 1994.

214. Jansen, M., Klaver, E., Verkaik, P., Van Steen, M., and Tanenbaum, A.: «Encapsulating Distribution in Remote Objects.» *Information and Software Technology*, vol. 43, no. 6, pp. 353–363, May 2001.
215. Jennings, N. and Wooldridge, M.: «Applications of Intelligent Agents.» In Jennings, N. and Wooldridge, M. (eds.), *Agent Technology Foundations, Applications, and Markets*, pp. 3–26. Berlin: Springer-Verlag, 1998.
216. Jing, J., Helal, A., and Elmagarmid, A.: «Client-Server Computing in Mobile Environments.» *ACM Comput. Surv.*, vol. 31, no. 2, pp. 117–157, June 1999.
217. Johner, H., Brown, L., Hinner, F.-S., Reis, W., and Westman, J.: «Understanding LDAP.» Technical Report SG24-4986-00, International Technical Support Organization, IBM, Austin, TX, May 1998.
218. Johnson, B.: «An Introduction to the Design and Analysis of Fault-Tolerant Systems.» In Pradhan, D.K. (ed.), *Fault-Tolerant Computer System Design*, pp. 1–87. Upper Saddle River, NJ: Prentice Hall, 1995.
219. Johnson, K., Kaashoek, M., and Wallach, D.: «CRL: High-Performance All-Software Distributed Shared Memory.» *Proc. 15th Symp. Operating System Principles. ACM*, 1995. pp. 1–16.
220. Jul, E., Levy, H., Hutchinson, N., and Black, A.: «Fine-Grained Mobility in the Emerald System.» *ACM Trans. Comp. Syst.*, vol. 6, no. 1, pp. 109–133, Feb. 1988.
221. Juszczak, C.: «Improving the Performance and Correctness of an NFS Server.» *Proc. Summer Techn. Conf. USENIX*, 1990. pp. 53–63.
222. Kahn, D.: *The Codebreakers*. New York: Macmillan, 1967.
223. Kantor, B. and Lapsley, P.: «Network News Transfer Protocol: A Proposed Standard for the Stream-Based Transmission of News.» RFC 977, Feb. 1986.
224. Kappe, F.: *Hyperwave Information Server*. Hyperwave Research & Development, June 1999.
225. Karnik, N. and Tripathi, A.: «Security in the Ajanta Mobile Agent System.» *Software Practice & Experience*, vol. 31, no. 4, pp. 301–329, Apr. 2001.
226. Kasera, S., Kurose, J., and TOWSLEY, D.: «Scalable Reliable Multicast Using Multiple Multicast Groups.» *Proc. Int'l Conf. Measurements and Modeling of Computer Systems. ACM*, 1997. pp. 64–74.
227. Katz, E., Butler, M., and Mcgrath, R.: «A Scalable HTTP Server: The NCSA Prototype.» *Comp. Netw. & ISDN Syst.*, vol. 27, no. 2, pp. 155–164, Sept. 1994.
228. Kaufman, C., Perlman, R., and Speciner, M.: *Network Security: Private Communication in a Public World*. Englewood Cliffs, NJ: Prentice Hall, 1995.
229. Kawell, L., Beckhardt, S., Halvorsen, T., Ozzie, R., and Greif, I.: «Replicated Document Management in a Group Communication System.» *Proc. Second Conf. Computer-Supported Cooperative Work*, 1988. pp. 226–235.
230. Keith, E. W.: *Core Jini*. Upper Saddle River, NJ: Prentice Hall, 2nd ed., 2000.
231. Keleher, P., Cox, A., and Zwaenepoel, W.: «Lazy Release Consistency.» *Proc. 19th Ann. Int'l Symp. on Computer Architecture. ACM*, 1992. pp. 13–21.

232. Kent, S.: «Internet Privacy Enhanced Mail.» *Commun. ACM*, vol. 36, no. 8, pp. 48–60, Aug. 1993.
233. Kermarrec, A., Kuz, I., Van Steen, M., and Tanenbaum, A.: «A Framework for Consistent, Replicated Web Objects.» *Proc. 18th Int'l Conf on Distributed Computing Systems*. IEEE, 1998. pp. 276–284.
234. Khoshafian, S. and Buckiewicz, M.: *Introduction to Groupware, Workflow, and Workgroup Computing*. New York: John Wiley, 1995.
235. Kistler, J.: *Disconnected Operation in a Distributed File System*, vol. 1002 of *Lect. Notes Comp. Sc.* Berlin: Springer-Verlag, 1996.
236. Kistler, J. and Satyanaryanan, M.: «Disconnected Operation in the Coda File System.» *ACM Trans. Comp. Syst.*, vol. 10, no. 1, pp. 3–25, Feb. 1992.
237. Kleiman, S.: «Vnodes: an Architecture for Multiple File System Types in UNIX.» *Proc. Summer Techn. Conf. USENIX*, 1986. pp. 238–247.
238. Klessig, R. and Tesink, K.: *Smds, Wide-Area Data Networking with Switched Multimegabit Data Service*. Englewood Cliffs, NJ: Prentice Hall, 1995.
239. Kohl, J., Neuman, B., and T'SO, T.: «The Evolution of the Kerberos Authentication System.» In Brazier, F. and Johansen, D. (eds.), *Distributed Open Systems*, pp. 78–94. Los Alamitos, CA: IEEE Computer Society Press, 1994.
240. Kopetz, H. and Ochsenreiter, W.: «Clock Synchronization in Distributed Real-Time Systems.» *IEEE Trans. Comp.*, vol. C-87, no. 8, pp. 933–940, Aug. 1987.
241. Kopetz, H. and Verissimo, P.: «Real Time and Dependability Concepts.» In Mullender, S. (ed.), *Distributed Systems*, pp. 411–446. Wokingham: Addison-Wesley, 2nd ed., 1993.
242. Kotz, D., Gray, R., Nog, S., Rus, D., Chawla, S., and Cybenko, G.: «Agent Tcl: Targeting the Needs of Mobile Computers.» *IEEE Internet Comput.*, vol. 1, no. 4, pp. 58–67, July 1997.
243. Kumar, P. and Satyanarayanan, M.: «Flexible and Safe Resolution of File Conflicts.» *Proc. Winter Techn. Conf USENIX*, 1995. pp. 95–106.
244. Kung, H. and Robinson, J.: «On Optimistic Methods for Concurrency Control.» *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, June 1981.
245. Kurose, J. F. and Ross, K. W.: *Computer Networking*. Reading, MA: Addison-Wesley, 2001.
246. Ladin, R., Liskov, B., Shira, L., and Ghemawat, S.: «Providing Availability Using Lazy Replication.» *ACM Trans. Comp. Syst.*, vol. 10, no. 4, pp. 360–391, Nov. 1992.
247. Lai, C., Gong, L., Koved, L., Nadalin, A., and Schemers, R.: «User Authentication and Authorization in the Java Platform.» *Proc. 15th Ann. Computer Security Applications Conf. IEEE*, 1999. pp. 285–290.
248. Lamacchia, B. and Odlyzko, A.: «Computation of Discrete Logarithms in Prime Fields.» *Designs, Codes, and Cryptography*, vol. 1, no. 1, pp. 47–62, May 1991.
249. Lamport, L.: «Time, Clocks, and the Ordering of Events in a Distributed System.» *Commun. ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

250. Lamport, L.: «How to Make a Multiprocessor Computer that Correctly Executes Multiprocessor Programs. IEEE Trans. Comp., vol. C-29, no. 9, pp. 690–69 1, Sept. 1979.
251. Lamport, L.: «Concurrent Reading and Writing of Clocks.» ACM Trans. Comp. Syst., vol. 8, no. 4, pp. 305–3 10, Nov. 1990.
252. Lamport, L., Shostak, R., and Paese, M.: «Byzantine Generals Problem.» ACM Trans. Prog. Lang. Syst., vol. 4, no. 3, pp. 382–401, July 1982.
253. Lampson, B.: «Designing a Global Name Service.» Proc. Fourth Symp. on Principles of Distributed Computing. ACM, 1986. pp. 1–10.
254. Lampson, B., Abadi, M., Burrows, M., and Wobber, E.: «Authentication in Distributed Systems: Theory and Practice.» ACM Trans. Comp. Syst., vol. 10, no. 4, pp. 265–310, Nov. 1992.
255. Lang, B., Queinnec, C., and Piquer, J.: «Garbage Collecting the World.» Proc. Symp. on Principles of Programming Languages. ACM, 1992. pp. 39–50.
256. Laprie, J.-C.: «Dependability — Its Attributes, Impairments and Means.» In Randell, B., Laprie, J.-C., Kopetz, H., and Littlewood, B. (eds.), Predictably Dependable Computing Systems, pp. 3–24. Berlin: Springer-Verlag, 1995.
257. Laurie, B. and Laurie, P.: Apache: The Definitive Guide. Sebastopol, CA: O'Reilly & Associates, 2nd ed., 1999.
258. Lee, Y., Leung, K., and Satyanarayanan, M.: «Operation-based Update Propagation in a Mobile File System.» Proc. Ann. Techn. Conf USENIX, 1999. pp. 43–56.
259. Le Hors, A., Le Hegaret, P., Wood, L., Nicol, G., Robie, J., Champion, M., and Byrne, S.: «Document Object Model (DOM) Level 2 Core Specification.» W3C Recommendation, Nov. 2000.
260. Leighton, F. and Lewin, D.: «Global Hosting System.» United States Patent, Number 6,108,703, Aug. 2000.
261. Leiwo, J., Haenle, C., Homburg, P., Gamage, C., and Tanenbaum, A.: «A Security Design for a Wide-Area Distributed System.» In Proc. Second Int'l. Conf Information Security and Cryptology, vol. 1787 of Lect. Notes Comp. Sc., pp. 236–256. Berlin: Springer-Verlag, 1999.
262. Levine, B. and Garcia-Luna-Aceves, J.: «A Comparison of Reliable Multicast Protocols.» ACM Multimedia Systems Journal, vol. 6, no. 5, pp. 334–348, 1998.
263. Lewis, B. and Berg, D. J.: Multithreaded Programming with Pthreads. Englewood Cliffs, NJ: Prentice Hall, 2nd ed., 1998.
264. Li, K. and Hudak, P.: «Memory Coherence in Shared Virtual Memory Systems.» ACM Trans. Comp. Syst., vol. 7, no. 3, pp. 321–359, Nov. 1989.
265. Lilja, D.: «Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons.» ACM Comput. Surv., vol. 25, no. 3, pp. 303–338, Sept. 1993.
266. Lin, M.-J. and Marzullo, K.: «Directional Gossip: Gossip in a Wide-Area Network.» In Proc. Third European Dependable Computing Conf, vol. 1667 of Lect. Notes Comp. Sc., pp. 364–379. Berlin: Springer-Verlag, Sept. 1999.

267. Linn, J.: «Generic Security Service Application Program Interface, version 2.» RFC 2078, Jan. 1997.
268. Lipton, R. and Sandberg, J.: «PRAM: A Scalable Shared Memory.» Technical Report CS-TR-180-88, Princeton University, Sept. 1988.
269. Liskov, B.: «Practical Uses of Synchronized Clocks in Distributed Systems.» *Distributed Computing*, vol. 6, pp. 211–219, 1993.
270. Liu, C.-G., Estrin, D., Shenker, S., and Zhang, L.: «Local Error Recovery in SRM: Comparison of Two Approaches.» *IEEE/ACM Trans. Netw.*, vol. 6, no. 6, pp. 686–699, Dec. 1998.
271. Loshin, P. (ed.): *Big Book of Lightweight Directory Access Protocol (LDAP) RFCs*. San Mateo, CA: Morgan Kaufman, 2000.
272. Lotus Development Corp.: *Inside Notes: The Architecture of Notes and the Domino Server*. Lotus Development Corporation, Cambridge, MA, Oct. 2000.
273. Lowe-Norris, A.: *Windows 2000 Active Directory*. Sebastopol, CA: O'Reilly & Associates, 2000.
274. Lundelius-Welch, J. and Lynch, N.: «A New Fault-Tolerant Algorithm for Clock Synchronization.» *Information and Computation*, vol. 77, no. 1, pp. 1–36, Jan. 1988.
275. Luotonen, A. and Altis, K.: «World-Wide Web Proxies.» *Comp. Netw. & ISDN Syst.*, vol. 27, no. 2, pp. 1845–1855, 1994.
276. Lynch, C., Preston, C., and Daniel, R.: «Using Existing Bibliographic Identifiers as Uniform Resource Names.» RFC 2288, Feb. 1998.
277. Lynch, N.: *Distributed Algorithms*. San Mateo, CA: Morgan Kaufman, 1996.
278. Macgregor, R., Durbin, D., Owlett, J., and Yeomans, A.: *Java Network Security*. Upper Saddle River, NJ: Prentice Hall, 1998.
279. Maekawa, M.: «A Square-root(N) Algorithm for Mutual Exclusion in Decentralized Systems.» *ACM Trans. Comp. Syst.*, vol. 3, no. 4, pp. 145–159, May 1985.
280. Maes, P.: «Agents that Reduce Work and Information Overload.» *Commun. ACM*, vol. 37, no. 7, pp. 31–40, July 1994.
281. Maffei, S.: «Piranha: A CORBA Tool For High Availability.» *IEEE Computer*, vol. 30, no. 4, pp. 59–66, Apr. 1997.
282. Makpangou, M., Gourhant, Y., Le Narzul, J.-P., and Shapiro, M.: «Fragmented Objects for Distributed Abstractions.» In Casavant, T. and Singhal, M. (eds.), *Readings in Distributed Computing Systems*, pp. 170–186. Los Alamitos, CA: IEEE Computer Society Press, 1994.
283. Malkhi, D. and Reiter, M.: «Secure Execution of Java Applets using a Remote Playground.» *IEEE Trans. Softw. Eng.*, vol. 26, no. 12, pp. 1197–1209, Dec. 2000.
284. Masinter, L.: «The Data URL Scheme.» RFC 2397, Aug. 1998.
285. Mattern, F.: «Algorithms for Distributed Termination Detection.» *Distributed Computing*, vol. 2, pp. 161–175, 1987.
286. Mazieres, D.: *Self-Certifying File System*. Ph.D. Thesis, Massachusetts Institute of Technology, May 2000.

287. Mazieres, D., Kaminsky, M., Kaashoek, M., and Witchel, E.: «Separating Key Management from File System Security.» Proc. 17th Symp. Operating System Principles. ACM, 1999. pp. 124–139.
288. Mazouni, K., Garbinato, B., and Guerraoui, R.: «Building Reliable Client-Server Software Using Actively Replicated Objects.» In Graham, I., Magnusson, B., Meyer, B., and Nerson, J.-M (eds.), Technology of Object Oriented Languages and Systems, pp. 37–53. Englewood Cliffs, NJ: Prentice Hall, 1995.
289. Medvidovic, N. and Taylor, R.: «A Classification and Comparison Framework for Software Architecture Description Languages.» IEEE Trans. Softw. Eng., vol. 26, no. 1, pp. 70–93, Jan. 2000.
290. Menezes, A. J., Van Oorschot, P. C., and Vanstone, S. A.: Handbook of Applied Cryptography. Boca Raton: CRC Press, 3rd ed., 1996.
291. Meyer, B.: Object-Oriented Software Construction. Englewood Cliffs, NJ: Prentice Hall, 2nd ed., 1997.
292. Microsoft Corporation: The Component Object Model Specification, Version 0.9. Redmond, WA, Oct. 1995.
293. Mills, D.: «Network Time Protocol (version 3): Specification, Implementation, and Analysis.» RFC 1305, July 1992.
294. Mills, D.: «Improved Algorithms for Synchronizing Computer Network Clocks.» IEEE/ACM Trans. Netw., vol. 3, no. 3, pp. 245–254, June 1995.
295. Milojicic, D., Douglass, F., Paindaveine, Y., Wheeler, R., and Zhou, S.: «Process Migration.» ACM Comput. Surv., vol. 32, no. 3, pp. 241–299, Sept. 2000.
296. Min, S. L. and Baer, J. L.: «Design and Analysis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps.» IEEE Trans. Par. Distr. Syst., vol. 3, no. 1, pp. 25–44, Jan. 1992.
297. Mitchell, J., Gibbons, J., Hamilton, G., Kessler, P., Khalidi, Y., Kougiouris, P., Madany, P., Nelson, M., Powell, M., and Radia, S. R.: «An Overview of the Spring System.» Proc. 39th Int'l. Computer Conf. IEEE, 1994. pp. 122–131.
298. Mizuno, M., Raynal, M., and Zhou, J. Z.: «Sequential Consistency in Distributed Systems.» In Birman, K., Mattern, F., and Schiper, A. (eds.), Theory and Practice in Distributed Systems, vol. 938 of Lect. Notes Comp. Sc., pp. 224–241. Berlin: Springer-Verlag, 1995.
299. Moats, R.: «URN Syntax.» RFC 2141, May 1997.
300. Moats, R.: «A URN Namespace for IETF Documents.» RFC 2648, Aug. 1999.
301. Mockapetris, P.: «Domain Names – Concepts and Facilities.» RFC 1034, Nov. 1987.
302. Mohan, S. and Jain, R.: «Two User Location Strategies for Personal Communication Services.» IEEE Pers. Commun., vol. 1, no. 1, pp. 42–50, Jan. 1994.
303. Mosberger, D.: «Memory Consistency Models.» Oper. Syst. Rev., vol. 27, no. 1, pp. 18–26, Jan. 1993.

304. Moser, L., Melliar-Smith, P., Agarwal, D., Budhia, R., and Lingley-Papadopoulos, C.: «Totem: A Fault-Tolerant Multicast Group Communication System.» *Commun. ACM*, vol. 39, no. 4, pp. 54–63, Apr. 1996.
305. Moser, L., Mellior-Smith, P., and Narasimhan, P.: «Consistent Object Replication in the Eternal System.» *Theory and Practice of Object Systems*, vol. 4, no. 2, pp. 81–92, 1998.
306. Mullender, S.: *Distributed Systems*. Wokingham: Addison-Wesley, 2nd ed., 1993.
307. Mullender, S. and Tanenbaum, A.: «Immediate Files.» *Software — Practice & Experience*, vol. 14, no. 3, pp. 365–368, 1984.
308. Muntz, D. and Honeyman, P.: «Multi-level Caching in Distributed File Systems.» *Proc. Winter Techn. Conf USENIX*, 1992. pp. 305–313.
309. Narasimham, P., Moser, L., and Mellior-Smith, P.: «Using Interceptors to Enhance CORBA.» *IEEE Computer*, vol. 32, no. 7, pp. 62–68, July 1999.
310. Narasimhan, P., Moser, L., and Melliar-Smith, P.: «The Eternal System.» In Urban, J. and Dasgupta, P. (eds.), *Encyclopedia of Distributed Computing*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 2000.
311. Needham, R. and Schroeder, M.: «Using Encryption for Authentication in Large Networks of Computers.» *Commun. ACM*, vol. 21, no. 12, pp. 993–999, Dec. 1978.
312. Needham, R.: «Names.» In Mullender, S. (ed.), *Distributed Systems*, pp. 315–327. Wokingham: Addison-Wesley, 2nd ed., 1993.
313. Nelson, B.: *Remote Procedure Call*. Ph.D. Thesis, Carnegie-Mellon University, 1981.
314. Neuman, B.: «Scale in Distributed Systems.» In Casavant, T. and Singhal, M. (eds.), *Readings in Distributed Computing Systems*, pp. 463–489. Los Alamitos, CA: IEEE Computer Society Press, 1994.
315. Neuman, B.: «Proxy-Based Authorization and Accounting for Distributed Systems.» *Proc. 13th Int'l Conf on Distributed Computing Systems*. IEEE, 1993. pp. 283–291.
316. Neumann, P.: «Architectures and Formal Representations for Secure Systems.» *Technical Report*, Computer Science Laboratory, SRI International, Menlo Park, CA, Oct. 1995.
317. Nielsen, S., Dahm, F., Luscher, M., Yamamoto, H., Collins, F., Denholm, B., Kumar, S., and Softley, J.: «Lotus Notes and Domino R5.0 Security Infrastructure Revealed.» *Technical Report SG24-5341-00*, International Technical Support Organization, IBM, Austin, TX, May 1999.
318. Noble, B., Fleis, B., and Kim, M.: «A Case for Fluid Replication.» *Proc. NetStore '99*, 1999.
319. Nutt, G.: *Operating Systems, A Modern Perspective*. Reading, MA: Addison-Wesley, 2nd ed., 2000.
320. Nwana, H.: «Software Agents: An Overview.» *Know. Eng. Rev.*, vol. 11, no. 3, pp. 205–244, Oct. 1996.

321. Nwana, H. and Ndumu, D.: «A Perspective on Software Agents Research.» *Know. Eng. Rev.*, vol. 14, no. 2, pp. 1–18, Jan. 1999.
322. Obraska, K.: «Multicast Transport Protocols: A Survey and Taxonomy.» *IEEE Commun. Mag.*, vol. 36, no. 1, pp. 94–102, Jan. 1998.
323. Oki, B., Pfluegl, M., Siegel, A., and Skeen, D.: «The Information Bus — An Architecture for Extensible Distributed Systems.» *Proc. 14th Symp. Operating System Principles. ACM*, 1993. pp. 58–68.
324. OMG: «Object Management Architecture Guide, Revision 3.0.» *OMG Document ab/97-05-05*, Object Management Group, Framingham, MA, Jan. 1997.
325. OMG: «CORBAServices: Notification Service Specification.» *OMG Document formal/00-06-20*, Object Management Group, Framingham, MA, June 2000.
326. OMG: «CORBAServices: Trading Service Specification.» *OMG Document formal/00-06-27*, Object Management Group, Framingham, MA, May 2000.
327. OMG: «CORBAServices: Concurrency Service Specification.» *OMG Document formal/00-06-14*, Object Management Group, Framingham, MA, Apr. 2000.
328. OMG: «Fault Tolerant CORBA.» *OMG Document ptc/00-04-04*, Object Management Group, Framingham, MA, Apr. 2000.
329. OMG: «Mobile Agent Facility Specification.» *OMG Document formal/00-01-02*, Object Management Group, Framingham, MA, Jan. 2000.
330. OMG: «CORBAServices: Naming Service Specification.» *OMG Document formal/01-02-65*, Object Management Group, Framingham, MA, Feb. 2001.
331. OMG: «The Common Object Request Broker: Architecture and Specification, revision 2.4.2.» *OMG Document formal/00-02-33*, Object Management Group, Framingham, MA, Feb. 2001.
332. Oram, A. (ed.): *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. Sebastopol, CA: O'Reilly & Associates, 2001.
333. Orfali, R., Harkey, D., and Edwards, J.: *The Essential Distributed Objects Survival Guide*. New York: John Wiley, 1996.
334. Organick, E.: *The Multics System: An Examination of Its Structure*. Cambridge, MA: MIT Press, 1972.
335. OSF: *OSF DCE 1.2.2 Application Development Guide — Core Components*. The Open Group, 1997.
336. Ousterhout, J.: *Tcl and the Tk Toolkit*. Reading, MA: Addison-Wesley, 1994.
337. Ozsu, T. and Valduriez, P.: *Principles of Distributed Database Systems*. Upper Saddle River, NJ: Prentice Hall, 2nd ed., 1999.
338. Page, T., Guy, R., Heidemann, J., Ratner, R., Reiher, P., Goel, A., Kuenning, G., and Popek, G.: «Perspectives on Optimistically Replicated, Peer-to-Peer Filing.» *Software — Practice & Experience*, vol. 28, no. 2, pp. 155–180, Feb. 1998.
339. Pal, V., Aron, M., Banga, G., Svendsen, M., Druschel, P., Zwaenepoel, W., and Nahum, E.: «Locality-Aware Request Distribution in Cluster-Based Network

- Servers.» Proc. Eighth Int'l Conf Architectural Support for Programming Languages and Operating Systems. ACM, 1998. pp. 205–216.
340. Panzieri, F. and Shrivastava, S.: «Rajdoot: A Remote Procedure Call Mechanism with Orphan Detection and Killing.» IEEE Trans. Softw. Eng., vol. 14, no. 1, pp. 30–37, Jan. 1988.
 341. Papadopoulos, G. and Arbab, F.: «Coordination Models and Languages.» In Zelikowitz, M. (ed.), *Advances in Computers*, vol. 46, pp. 329–400. New York, NY: Academic Press, Sept. 1998.
 342. Parker, T. and Pikas, D.: «SESAME V4 Overview.» Technical Report, SESAME Consortium, Dec. 1995.
 343. Partridge, C.: «A Proposed Flow Specification.» RFC 1363, Sept. 1992.
 344. Partridge, C.: *Gigabit Networking*. Reading, MA: Addison-Wesley, 1994.
 345. Partridge, C., Mendez, T., and Milliken, W.: «Host Anycasting Service.» RFC 1546, Nov. 1993.
 346. Pawlowski, B., Juszczak, C., Staubach, P., Smith, C., Hebel, D., and Hitz, D.: «NFS Version 3 Design and Implementation.» Proc. Summer Techn. Conf. USENIX, 1994. pp. 137–152.
 347. Pease, M., Shostak, R., and Lamport, L.: «Reaching Agreement in the Presence of Faults.» J. ACM, vol. 27, no. 2, pp. 228–234, Apr. 1980.
 348. Perkins, C.: *Mobile IP. Design Principles and Practice*. Reading, MA: Addison-Wesley, 1997.
 349. Petersen, K., Spreitzer, M., Terry, D., Theimer, M., and Demers, A.: «Flexible Update Propagation for Weakly Consistent Replication.» Proc. 16th Symp. Operating System Principles. ACM, 1997. pp. 288–301.
 350. Pfitzmann, B. and Wadner, M.: «Properties of Payment Systems: General Definition Sketch and Classification.» Technical Report RZ 2823, IBM Research Division, Zurich Research Laboratory, June 1996.
 351. Pfleeger, C.: *Security in Computing*. Upper Saddle River, NJ: Prentice Hall, 2nd ed., 1997.
 352. Pierre, G., Van Steen, M., and Tanenbaum, A.: «Self-Replicating Web Documents.» Technical Report IR-486, Vrije Universiteit, Department of Mathematics and Computer Science, Feb. 2001.
 353. Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., and Winterbottom, P.: «Plan 9 from Bell Labs.» *Computing Systems*, vol. 8, no. 3, pp. 221–254, Summer 1995.
 354. Pike, R.: «8^{1/2}, the Plan 9 Window System.» Proc. Summer Techn. Conf. USENIX, 1991. pp. 257–265.
 355. Pitoura, E. and Samaras, G.: «Locating Objects in Mobile Computing.» IEEE Trans. Know. Data Eng., vol. 13, 2001.
 356. Platt, D.: *The Essence of COM and ActiveX: A Programmers Workbook*. Englewood Cliffs, NJ: Prentice Hall, 1998.

357. Plainfosse, D. and Shapiro, M.: «A Survey of Distributed Garbage Collection Techniques.» In Proc. Int'l Workshop on Memory Management, vol. 986 of Lect. Notes Comp. Sc., pp. 211–249. Berlin: Springer-Verlag, Sept. 1995.
358. Plummer, D.: «Ethernet Address Resolution Protocol.» RFC 826, Nov. 1982.
359. Pope, A.: The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture. Englewood Cliffs, NJ: Prentice Hall, 1998.
360. Postel, J.: «Simple Mail Transfer Protocol.» RFC 821, Aug. 1982.
361. Postel, J. and Reynolds, J.: «File Transfer Protocol.» RFC 995, Oct. 1985.
362. Protic, J., Tomasevic, M., and Milutinovic, V.: «Distributed Shared Memory: Concepts and Systems.» IEEE Par. Distr. Techn., vol. 4, no. 2, pp. 63–79, Summer 1996.
363. Protic, J., Tomasevic, M., and Milutinovic, V.: Distributed Shared Memory, Concepts and Systems. Los Alamitos, CA: IEEE Computer Society Press, 1998.
364. Qiu, L., Padmanabhan, V., and Voelker, G.: «On the Placement of Web Server Replicas.» Proc. 20th INFOCOM. IEEE, 2001.
365. Rabinovich, M., Rabinovich, I., Rajaraman, R., and Aggarwal, A.: «A Dynamic Object Replication and Migration Protocol for an Internet Hosting Service.» Proc. 19th Int'l Conf on Distributed Computing Systems. ACM, 1999. pp. 101–113.
366. Rabinovich, M. and Aggarwal, A.: «Radar: A Scalable Architecture for a Global Web Hosting Service.» Proc. Eighth Int'l WWW Conf, 1999.
367. Radia, S.: Names, Contexts, and Closure Mechanisms in Distributed Computing Environments. Ph.D. Thesis, University of Waterloo, Ontario, 1989.
368. Radicati, S.: X.500 Directory Services: Technology and Deployment. London: International Thomson Computer Press, 1994.
369. Ramanathan, P., Kandlur, D., and SHIN, K.: «Hardware-Assisted Software Clock Synchronization for Homogeneous Distributed Systems.» IEEE Trans. Comp., vol. C-89, no. 4, pp. 514–524, Apr. 1989.
370. Ramanathan, P., Shin, K., and Butler, R.: «Fault-Tolerant Clock Synchronization in Distributed Systems.» IEEE Computer, vol. 23, no. 10, pp. 33–42, Oct. 1990.
371. Rao, H. and Peterson, L.: «Accessing Files in an Internet: The Jade File System.» IEEE Trans. Softw. Eng., vol. 19, no. 6, pp. 613–624, June 1993.
372. Raynal, M.: Distributed Algorithms and Protocols. New York: John Wiley, 1988.
373. Raynal, M.: «A Simple Taxonomy for Distributed Mutual Exclusion Algorithms. Oper. Syst. Rev., vol. 25, pp. 47–50, Apr. 1991.
374. Raynal, M. and Singhal, M.: «Logical Time: Capturing Causality in Distributed Systems.» IEEE Computer, vol. 29, no. 2, pp. 49–56, Feb. 1996.
375. Rees, J. and Clinger, W.: «Revised Report on the Algorithmic Language Scheme.» SIGPLAN Notices, vol. 21, no. 12, pp. 37–43, Dec. 1986.
376. Reiter, M.: «How to Securely Replicate Services.» ACM Trans. Prog. Lang. Syst., vol. 16, no. 3, pp. 986–1009, May 1994.

377. Reiter, M., Birman, K., and Van Renesse, R.: «A Security Architecture for Fault-Tolerant Systems.» *ACM Trans. Comp. Syst.*, vol. 12, no. 4, pp. 340–371, Nov. 1994.
378. Rescorla, E. and Schiffman, A.: «The Secure HyperText Transfer Protocol.» *RFC 2660*, Aug. 1999.
379. Reynolds, J. and Postel, J.: «Assigned Numbers.» *RFC 1700*, Oct. 1994.
380. Ricart, G. and Agrawala, A.: «An Optimal Algorithm for Mutual Exclusion in Computer Networks.» *Commun. ACM*, vol. 24, no. 1, pp. 9–17, Jan. 1981.
381. Rivest, R.: «The MD5 Message Digest Algorithm.» *RFC 1321*, Apr. 1992.
382. Rivest, R., Shamir, A., and Adleman, L.: «A Method for Obtaining Digital Signatures and Public-key Cryptosystems.» *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.
383. Rizzo, L.: «Effective Erasure Codes for Reliable Computer Communication Protocols.» *ACM Comp. Commun. Rev.*, vol. 27, no. 2, pp. 24–36, Apr. 1997.
384. Rodrigues, L., Fonseca, H., and Verissimo, P.: «Totally Ordered Multicast in Large-Scale Systems.» *Proc. 16th Int'l Conf on Distributed Computing Systems*. IEEE, 1996. pp. 503–510.
385. Rodriguez, P. and Sibal, S.: «SPREAD: Scalable Platform for Reliable and Efficient Automated Distribution.» *Comp. Netw. & ISDN Syst.*, vol. 33, no. 1–6, pp. 33–46, 2000.
386. Rogerson, D.: *Inside COM*. Redmond, WA: Microsoft Press, 1997.
387. Rosenblum, M. and Oosterhout, J.: «The Design and Implementation of a Log-Structured File System.» *ACM Trans. Comp. Syst.*, vol. 10, no. 1, pp. 26–52, Feb. 1992.
388. Rowstron, A.: «Run-time Systems for Coordination.» In Omicini, A., Zambonelli, F., Klusch, M., and Tolksdorf, R. (eds.), *Coordination of Internet Agents: Models, Technologies and Applications*, pp. 78–96. Berlin: Springer-Verlag, 2001.
389. Rowstron, A. and Wray, S.: «A Run-Time System for WCL.» In Bal, H., Belkhouche, B., and Cardelli, L. (eds.), *Internet Programming Languages*, vol. 1686 of *Lect. Notes Comp. Sc.*, pp. 78–96. Berlin: Springer-Verlag, 1998.
390. Saito, V.: «Consistency Management in Optimistic Replication Algorithms.» *Technical Report*, University of Washington, June 2001.
391. Saltzer, J. and Schroeder, M.: «The Protection of Information in Computer Systems.» *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, Sept. 1975.
392. Saltzer, J.: «Naming and Binding Objects.» In Bayer, R., Graham, R., and Seegmuller, G. (eds.), *Operating Systems: An Advanced Course*, vol. 60 of *Lect. Notes Comp. Sc.*, pp. 99–208. Berlin: Springer-Verlag, 1978.
393. Saltzer, J., Reed, D., and Clark, D.: «End-to-End Arguments in System Design.» *ACM Trans. Comp. Syst.*, vol. 2, no. 4, pp. 277–288, Nov. 1984.
394. Samar, V. and Lai, C.: «Making Login Services Independent from Authentication Technologies.» *Proc. SunSoft Developer's Conf*, 1996.

395. Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B.: «Design and Implementation of the Sun Network File System.» Proc. Summer Techn. Conf. USENIX, 1985. pp. 119–130.
396. Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E.: «Role-Based Access Control Models.» IEEE Computer, vol. 29, no. 2, pp. 38–47, Feb. 1996.
397. Satyanarayanan, M.: «Scalable, Secure, and Highly Available Distributed File Access.» IEEE Computer, vol. 23, no. 5, pp. 9–21, May 1990.
398. Satyanarayanan, M.: «The Influence of Scale on Distributed File System Design.» IEEE Trans. Softw. Eng., vol. 18, no. 1, pp. 1–8, Jan. 1992.
399. Satyanarayanan, M., Kistler, J., Kumar, P., Okaskai, M., Siegel, E., and Steere, D.: «Coda: A Highly Available File System for a Distributed Workstation Environment.» IEEE Trans. Comp., vol. 39, no. 4, pp. 447–459, Apr. 1990.
400. Satyanarayanan, M., Mashburn, H., Kumar, P., Steere, D., and Kistler, J.: «Light-weight Recoverable Virtual Memory.» ACM Trans. Comp. Syst., vol. 12, no. 1, pp. 33–57, Feb. 1994.
401. Satyanarayanan, M. and Siegel, E.: «Parallel Communication in a Large Distributed System.» IEEE Trans. Comp., vol. 39, no. 3, pp. 328–348, Mar. 1990.
402. Schmidt, D., Stal, M., Rohnert, H., and Buschmann, F.: Pattern-Oriented Software Architecture – Patterns for Concurrent and Networked Objects. New York: John Wiley, 2000.
403. Schneider, F.: «Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial.» ACM Comput. Surv., vol. 22, no. 4, pp. 299–320, Dec. 1990.
404. Schneier, B.: Applied Cryptography. New York: John Wiley, 2nd ed., 1996.
405. Schneier, B.: Secrets and Lies. New York: John Wiley, 2000.
406. Schulzrinne, H., Casner, S., Frederick, R., and Jacobson, V.: «RTP: A Transport Protocol for Real-Time Applications.» RFC 1889, Jan. 1996.
407. SET: «Secure Electronic Transactions Specification. Book 3: Formal Protocol Definition, May 1997.
408. Shapiro, M., Dickman, P., and Plainfosse, D.: «SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection.» Technical Report 1799, INRIA, Rocquencourt, France, Nov. 1992.
409. Shapiro, M., Gourhant, V., Habert, S., Mosseri, L., Ruffin, M., and Valot, C.: «SOS: An Object-Oriented Operating System – Assessment and Perspectives.» Computing Systems, vol. 2, no. 4, pp. 287–337, Fall 1989.
410. Shasha, D. and Jeong, K.: «PLinda 2.0: A Transactional/checkpointing Approach to Fault Tolerant Linda.» Proc. 13th Symp. on Reliable Distributed Systems. IEEE, 1994. pp. 96–105.
411. Shaw, M. and Clements, P.: «A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems.» Proc. 21st Int'l Comp. Softw. & Appl. Conf, 1997. pp. 6–13.

412. Shepler, S.: «NFS Version 4 Design Considerations.» RFC 2624, June 1999.
413. Shepler, S., Beame, C., Callaghan, B., Eisler, M., Noveck, D., Robinson, D., and Thurlow, R.: «NFS version 4 Protocol.» RFC 3010, Dec. 2000.
414. Sheresh, B. and Sheresh, D.: Understanding Directory Services. Indianapolis, IN: New Riders, 2000.
415. Sherif, M. H. and Sechrouchni, A.: Protocols for Secure Electronic Commerce. Boca Raton: CRC Press, 2000.
416. Sheth, A. P. and Larson, J. A.: «Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases.» ACM Comput. Surv., vol. 22, no. 3, pp. 183–236, Sept. 1990.
417. Siegel, J.: «OMG Overview: CORBA and the OMA in Enterprise Computing.» Commun. ACM, vol. 41, no. 10, pp. 37–43, Oct. 1998.
418. Silberschatz, A., Galvin, P., and Gagne, G.: Applied Operating System Concepts. New York: John Wiley, 2000.
419. Singh, S. and Kurose, J.: «Electing ‘Good’ Leaders.» J. Par. Distr. Comput., vol. 21, pp. 184–201, May 1994.
420. Singhal, M.: «A Taxonomy of Distributed Mutual Exclusion.» J. Par. Distr. Comput., vol. 18, no. 1, pp. 94–101, May 1993.
421. Singhal, M. and Shivaratri, N.: Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems. New York: McGraw-Hill, 1994.
422. Skeen, D.: «Nonblocking Commit Protocols.» Proc. SIGMOD Int’l Conf on Management Of Data. ACM, 1981. pp. 133–142.
423. Skeen, D.: «An Information Bus Architecture for Large-Scale Decision Support Environments.» Proc. Winter Techn. Conf. USENIX, 1992. pp. 183–195.
424. Skeen, D. and Stonebraker, M.: «A Formal Model of Crash Recovery in a Distributed System.» IEEE Trans. Softw. Eng., vol. SE-9, no. 3, pp. 219–228, Mar. 1983.
425. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J.: MPI: The Complete Reference — The MPI Core. Cambridge, MA: MIT Press, 1998.
426. Spasojevic, M. and Satyanarayanan, M.: «An Empirical Study of a Wide-Area Distributed File System.» ACM Trans. Comp. Syst., vol. 14, no. 2, pp. 200–222, May 1996.
427. Speakman, T., Bhaskar, N., Crowcroft, J., Edmonstone, R., Farinacci, D., Gemmell, J., Montgomery, T., Leschchiner, D., Lin, S., Luby, M., Rizzo, L., Sumanasekera, R., Vicisano, L., and Tweedly, A.: «PGM Reliable Transport Protocol Specification.» Internet Draft (work in progress), Feb. 2001.
428. Spector, A.: «Performing Remote Operations Efficiently on a Local Computer Network.» Commun. ACM, vol. 25, no. 4, pp. 246–260, Apr. 1982.
429. Srikanth, T. and Toueg, S.: «Optimal Clock Synchronization.» J. ACM, vol. 34, no. 3, pp. 626–645, July 1987.

- 430. Srinivasan, R.: «RPC: Remote Procedure Call Protocol Specification Version 2.» RFC 1831, Aug. 1995.
- 431. Srinivasan, R.: «XDR: External Data Representation Standard.» RFC 1832, Aug. 1995.
- 432. Stein, L.: Web Security, A Step-by-Step Reference Guide. Reading, MA: Addison-Wesley, 1998.
- 433. Steiner, J., Neuman, C., and Schiller, J.: «Kerberos: An Authentication Service for Open Network Systems.» Proc. Winter Techn. Conf USENIX, 1988. pp. 191–202.
- 434. Steinmetz, R. and Nahrstedt, K.: Multimedia: Computing, Communications and Applications. Upper Saddle River, N.J.: Prentice Hall, 1995.
- 435. Steinmetz, R.: «Human Perception of Jitter and Media Synchronization.» IEEE J. Selected Areas Commun., vol. 14, no. 1, pp. 61–72, Jan. 1996.
- 436. Stevens, W.: Advanced Programing in the UNIX Environment. Reading, MA: Addison-Wesley, 1992.
- 437. Stevens, W.: TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols. Reading, MA: Addison-Wesley, 1996.
- 438. Stevens, W.: UNIX Network Programming — Networking APIs: Sockets and XTI. Englewood Cliffs, NJ: Prentice Hall, 2nd ed., 1998.
- 439. Stevens, W.: UNIX Network Programming — Interprocess Communication. Englewood Cliffs, NJ: Prentice Hall, 2nd ed., 1999.
- 440. Stumm, M. and Zhou, S.: «Algorithms Implementing Distributed Shared Memory.» Computer, vol. 23, no. 5, pp. 54–64, 1990.
- 441. Sun Microsystems: Java Remote Method Invocation Specification, JDK 1.2. Sun Microsystems, Mountain View, Calif., Oct. 1998.
- 442. Sun Microsystems: JavaSpaces Service Specification, Version 1.1. Sun Microsystems, Palo Alto, CA, Oct. 2000.
- 443. Sun Microsystems: Jini Architecture Specification, Version 1.1. Palo Alto, CA, Oct. 2000.
- 444. Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., and Peck, G.: «Scalability in the XFS File System.» Proc. Ann. Techn. Conf. USENIX, 1996. pp. 1–14.
- 445. Tai, S. and Rouvellou, I.: «Strategies for Integrating Messaging and Distributed Object Transactions.» In Proc. Middleware 2000, vol. 1795 of Lect. Notes Comp. Sc., pp. 308–330. Berlin: Springer-Verlag, 2000.
- 446. Tanenbaum, A.: Computer Networks. Englewood Cliffs, NJ: Prentice Hall, 3rd ed., 1996.
- 447. Tanenbaum, A.: Modern Operating Systems. Upper Saddle River, NJ: Prentice Hall, 2nd ed., 2001.
- 448. Tanenbaum, A., Mullender, S., and Van Renesse, R.: «Using Sparse Capabilities in a Distributed Operating System.» Proc. Sixth Int'l Conf on Distributed Computing Systems. IEEE, 1986. pp. 558–563.

449. Tanenbaum, A., Van Renesse, R., Van Staveren, H., Sharp, G., Mullender, S., Jansen, J., and Van Rossum, G.: «Experiences with the Amoeba Distributed Operating System.» *Commun. ACM*, vol. 33, no. 12, pp. 46–63, Dec. 1990.
450. Tanenbaum, A. and Woodhull, A.: *Operating Systems, Design and Implementation*. Englewood Cliffs, NJ: Prentice Hall, 2nd ed., 1997.
451. Tanisch, P.: «Atomic Commit in Concurrent Computing.» *IEEE Concurrency*, vol. 8, no. 4, pp. 34–41, Oct. 2000.
452. Tartalja, I. and Milutinovic, V.: «Classifying Software-Based Cache Coherence Solutions.» *IEEE Softw.*, vol. 14, no. 3, pp. 90–101, May 1997.
453. Terry, D., Demers, A., Petersen, K., Spreitzer, M., Theimer, M., and Welsh, B.: «Session Guarantees for Weakly Consistent Replicated Data.» *Proc. Third Int'l Conf on Parallel and Distributed Information Systems*. IEEE, 1994. pp. 140–149.
454. Terry, D., Petersen, K., Spreitzer, M., and Theimer, M.: «The Case for Non-transparent Replication: Examples from Bayou.» *IEEE Data Engineering*, vol. 21, no. 4, pp. 12–20, Dec. 1998.
455. Tewari, R., Dahlin, M., Vin, H., and Kay, J.: «Design Considerations for Distributed Caching on the Internet.» *Proc. 19th Int'l Conf on Distributed Computing Systems*. IEEE, 1999. pp. 273–284.
456. Thekkath, C., Mann, T., , and Lee, E.: «Frangipani: A Scalable Distributed File System.» *Proc. 16th Symp. Operating System Principles*. ACM, 1997. pp. 224–237.
457. Thomas, R.: «A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases.» *ACM Trans. Database Syst.*, vol. 4, no. 2, pp. 180–209, June 1979.
458. TIBCO: TIB/Rendezvous TX Concepts, Release 1.1. TIBCO Software Inc., Palo Alto, CA, Nov. 2000.
459. TIBCO: TIB/Rendezvous C Reference, Release 6.4. TIBCO Software Inc., Palo Alto, CA, Oct. 2000.
460. TIBCO: TIB/Rendezvous Concepts, Release 6.4. TIBCO Software Inc., Palo Alto, CA, Oct. 2000.
461. Tolksdorf, R.: «A Machine for Uncoupled Coordination and its Concurrent Behaviour.» In *Object-Based Models and Languages for Concurrent Systems*, vol. 924 of *Lect. Notes Comp. Sc.*, pp. 176–193. Berlin: Springer-Verlag, 1995.
462. Towsley, D., Kurose, J., and Pingali, S.: «A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols.» *IEEE J. Selected Areas Commun.*, vol. 15, no. 3, pp. 398–407, Apr. 1997.
463. Triantafillou, P. and Neilson, C.: «Achieving Strong Consistency in a Distributed File System.» *IEEE Trans. Softw. Eng.*, vol. 23, no. 1, pp. 35–55, Jan. 1997.
464. Tripathi, A., Karnik, N., Vora, M., Ahmed, T., and Singh, R.: «Mobile Agent Programming in Ajanta.» *Proc. 19th Int'l Conf on Distributed Computing Systems*. IEEE, 1999. pp. 190–197.

465. Turek, J. and Shasha, S.: «The Many Faces of Consensus in Distributed Systems.» *IEEE Computer*, vol. 25, no. 6, pp. 8–17, June 1992.
466. Ullman, J.: *Principles of Database and Knowledge-Base Systems*, vol. 1. Rockville, MA: Computer Science Press, 1988.
467. Umar, A.: *Object-Oriented Client/Server Internet Environments*. Upper Saddle River, NJ: Prentice Hall, 1997.
468. Vaha-Sipila, A.: «URLs for Telephone Calls.» RFC 2806, Apr. 2000.
469. Van Steen, M., Hauck, F., Ballintijn, G., and Tanenbaum, A.: «Algorithmic Design of the Globe Wide-Area Location Service.» *Comp. J.*, vol. 41, no. 5, pp. 297–310, 1998.
470. Van Steen, M., Hauck, F., Homburg, P., and Tanenbaum, A.: «Locating Objects in Wide-Area Systems.» *IEEE Commun. Mag.*, vol. 36, no. 1, pp. 104–109, Jan. 1998.
471. Van Steen, M., Homburg, P., and Tanenbaum, A.: «Globe: A Wide-Area Distributed System.» *IEEE Concurrency*, vol. 7, no. 1, pp. 70–78, Jan. 1999.
472. Van Steen, M., Tanenbaum, A., Kuz, I., and Sips, H.: «A Scalable Middleware Solution for Advanced Wide-Area Web Services.» *Distributed Systems Engineering*, vol. 6, no. 1, pp. 34–42, Mar. 1999.
473. Verissimo, P. and Rodrigues, L.: *Distributed Systems for Systems Architects*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 2001.
474. Vetter, R., Spell, C., and Ward, C.: «Mosaic and the World-Wide Web.» *IEEE Computer*, vol. 27, no. 10, pp. 49–57, Oct. 1994.
475. Vinoski, S.: «CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments.» *IEEE Commun. Mag.*, vol. 35, no. 2, pp. 46–55, Feb. 1997.
476. Viveney, B.: «DCE and Object Programming.» In Rosenberry, W. (ed.), *DCE Today*, pp. 251–264. Upper Saddle River, NJ: Prentice Hall, 1998.
477. Vixie, P.: «A DNS RR for Specifying the Location of Services (DNS SRV).» RFC 2052, Oct. 1996.
478. Von Eicken, T., Culler, D., Goldstein, S., and Schauser, K.: «Active Messages: a Mechanism for Integrated Communication and Computation.» *Proc. 19th Int'l Symp. on Computer Architecture*, 1992. pp. 256–266.
479. Voydock, V. and Kent, S.: «Security Mechanisms in High-Level Network Protocols.» *ACM Comput. Surv.*, vol. 15, no. 2, pp. 135–171, June 1983.
480. Wahbe, R., Lucco, S., Anderson, T., and Graham, S.: «Efficient Software-based Fault Isolation.» *Proc. 14th Symp. Operating System Principles*. ACM, 1993. pp. 203–216.
481. Wahl, M., Howes, T., and Kille, S.: «Lightweight Directory Access Protocol (v3).» RFC 2251, Dec. 1997.
482. Waldo, J.: «Remote Procedure Calls and Java Remote Method Invocation.» *IEEE Concurrency*, vol. 6, no. 3, pp. 5–7, July 1998.

483. Waldo, J.: *The Jini Specifications*. Upper Saddle River, NJ: Prentice Hall, 2nd ed., 2000.
484. Wallach, D., Balfanz, D., Dean, D., and Felten, E.: «Extensible Security Architectures for Java.» *Proc. 16th Symp. Operating System Principles*. ACM, 1997. pp. 116–128.
485. Wang, H., LO, M. K., and Wang, C.: «Consumer Privacy Concerns about Internet Marketing.» *Commun. ACM*, vol. 41, no. 3, pp. 63–70, Mar. 1998.
486. Wang, J.: «A Survey of Web Caching Schemes for the Internet.» *ACM Comp. Commun. Rev.*, vol. 29, no. 5, pp. 36–46, Oct. 1999.
487. Wang, J.: «A Fully Distributed Location Registration Strategy for Universal Personal Communication Systems.» *IEEE J. Selected Areas Commun.*, vol. 11, no. 6, pp. 850–860, Aug. 1993.
488. Wang, R., Anderson, T., and Dahlin, M.: «Experiences with a Distributed File System Implementation.» *Technical Report CSD-98-986*, Department of Computer Science, University of California, Berkeley, 1998.
489. Whitehead, J. and Goland, Y.: «WebDAV: A Network Protocol for Remote Collaborative Authoring on the Web.» *Proc. Sixth European Conf. on Computer Supported Cooperative Work*, 1999. pp. 291–310.
490. Whitehead, J. and Wiggins, M.: «WebDAV: IETF Standard for Collaborative Authoring on the Web.» *IEEE Internet Comput.*, vol. 2, no. 5, pp. 34–40, Sept. 1998.
491. Wieringa, R. and DE Jonge, W.: «Object Identifiers, Keys, and Surrogates—Object Identifiers Revisited.» *Theory and Practice of Object Systems*, vol. 1, no. 2, pp. 101–114, 1995.
492. Wiesmann, M., Pedone, F., Schiper, A., Kemme, B., and Alonso, G.: «Understanding Replication in Databases and Distributed Systems.» *Proc. 20th Int'l Conf on Distributed Computing Systems*. IEEE, 2000. pp. 264–274.
493. Wilson, P.: «Uniprocessor Garbage Collection Techniques.» *Technical Report*, University of Texas at Austin, 1994.
494. Wollrath, A., Riggs, R., and Waldo, J.: «A Distributed Object Model for the Java System.» *Computing Systems*, vol. 9, no. 4, pp. 265–290, Fall 1996.
495. Wolman, A., Voelker, G., Sharma, N., Cardwell, N., Karlin, A., and Levy, H.: «On the Scale and Performance of Cooperative Web Proxy Caching.» *Proc. 17th Symp. Operating System Principles*. ACM, 1999. pp. 16–31.
496. Wooldridge, M.: «Agent-Based Computing.» *Interoperable Communication Networks*, vol. 1, no. 1, pp. 71–97, Jan. 1998.
497. Wu, J.: *Distributed System Design*. Boca Raton: CRC Press, 1998.
498. Wyckoff J., P., McLaughry, S., Lehman, T., and Ford, D.: «T Spaces.» *IBM Systems*, pp. 454–474, Aug. 1998.
499. Yang, C. and Luo, M.: «A Content Placement and Management System for Distributed Web-Server Systems.» *Proc. 20th Int'l Conf on Distributed Computing Systems*. IEEE, 2000.

- 500. Yeadon, N., Garcia, F., Hutchison, D., and Shephard, S.: «Filters: QoS Support Mechanisms for Multipeer Communications.» IEEE J. Selected Areas Commun., vol. 14, no. 7, pp. 1245–1262, Sept. 1996.
- 501. Yeong, W., Howes, T., and Kille, S.: «Lightweight Directory Access Protocol.» RFC 1777, Mar. 1995.
- 502. Yu, H. and Vahdat, A.: «Design and Evaluation of a Continuous Consistency Model for Replicated Services.» Proc. Fourth Symp. on Operating System Design and Implementation. USENIX, 2000.
- 503. Zhang, L., Deering, S., Estrin, D., Shenker, S., and Zappala, D.: «RSVP: A New Resource Reservation Protocol.» IEEE Network Magazine, vol. 7, no. 5, pp. 8–18, Sept. 1993.
- 504. Zwicky, E., Cooper, S., Chapman, D., and Russell, D.: Building Internet Firewalls. Sebastopol, CA: O'Reilly & Associates, 2nd ed., 2000.

Список терминов

Термин	Оригинальное название
JIT-активизация, или активизация «на лету»	JIT («just-in-time») activation
<i>M,n</i> -пороговая схема	<i>M,n</i> -threshold scheme
RPC для открытых сетевых вычислений	Open Network Computing RPC, ONC RPC
Web-браузер	Web browser
Web-заместитель	Web proxy
X-терминал	X terminal
X-ядро	X kernel
Абсолютное имя пути	Absolute path name
Автоматическая кассовая машина, или банкомат	Automatic Teller Machines, ATM
Автоматическая транзакция	Automatic transaction
Автоматический монтировщик (файловых систем)	Automounter
Авторизация	Authorization
Агент базы	Home agent
Агент канала сообщений	Message Channel Agent, MCA
Агент пользователей каталога	Directory User Agent, DUA
Агент службы каталогов	Directory Service Agent, DSA
Адаптер объектов	Object adapter
Административный уровень	Administrational layer
Адрес	Address
Адрес протокола	Protocol address
Адрес экземпляра	Instance address
Адресация по теме	Subject-based addressing
Активизация «на лету», или JIT-активизация	JIT («just-in-time») activation
Активная репликация	Active replication
Активная цель	Active goal
Активное сообщение	Active message
Активный кэш	Active cache
Алгоритм забияки	Bully algorithm
Алгоритм корзины элементарных пакетов	Token bucket algorithm
Антиэнтропия	Anti-entropy

Термин	Оригинальное название
Апплет	Applet
Аренда	Lease
Архитектура без разделения	Shared-nothing architecture
Асимметричная криптосистема	Asymmetric cryptosystem
Асинхронная связь	Asynchronous communication
Асинхронный вызов RPC	Asynchronous RPC
Асинхронный режим передачи	Asynchronous transmission mode
Ассоциация защиты	Security association
Атака на отражении	Reflection attack
Атомарная (операция, транзакция)	Atomic
Атомарная групповая рассылка	Atomic multicasting
Атомарность, Непротиворечивость, Изолированность и Долговечность (транзакции)	Atomicity-Consistency-Isolation-Durability, ACID
Аудит	Auditing
Аутентификация	Authentication
База данных накопления	Hoard database
База данных размещения томов	Volume location database
База данных репликации томов	Volume replication database
Базовая точка	Home location
Банкомат, или автоматическая кассовая машина	Automatic Teller Machines, ATM
Барьер	Barrier
Безопасная ошибка	Fail-safe failure
Безопасность	Safety
Безотказность	Reliability
Библиотека типов	Type library
Бинарный интерфейс	Binary interface
Блокировка	Locking
Болтовня	Gossiping
Брандмауэр	Firewall
Брокер объектных запросов	Object Request Broker, ORB
Брокер сообщений	Message broker
Вектор версий Coda	Coda version vector
Векторная отметка времени	Vector timestamp
Верификатор байтов кода	Byte code verifier
Версия	Version
Вертикальное разбиение	Vertical fragmentation
Вертикальное распределение	Vertical distribution
Вертикальные средства	Vertical facilities
Взаимная блокировка, или тупик	Deadlock

Термин	Оригинальное название
Взаимодействие, или связь	Communication
Взаимозаменяемые (службы защиты)	Replaceable
Взвешенный подсчет ссылок	Weighted reference counting
Византийская ошибка	Byzantine failure
Виртуальная Java-машина	Java Virtual Machine, JVM
Виртуальная машина	Virtual machine
Виртуальная синхронность	Virtual synchrony
Виртуальная файловая система	Virtual File System, VFS
Виртуально синхронная групповая рассылка	Virtually synchronous multicast
Виртуальный канал	Virtual channel
Виртуальный образ	Virtual ghost
Виртуальный путь	Virtual path
Виртуальный узел	Vnode
Вложенная транзакция	Nested transaction
Вложенный поток данных	Substream
Внучатая сирота	Grandorphan
Восприимчивый (сервер)	Susceptible
Восстановимая виртуальная память	Recoverable Virtual Memory, RVM
Восстановимый объект	Recoverable object
Входное имя	Login
Выборочный показ	Selective revealing
Выдвинутый кэш	Push cache
Вызов через копирование/восстановление	Call-by-copy/restore
Генеративная связь	Generative communication
Гетерогенная (система)	Heterogeneous
Гиперкуб	Hypercube
Гиперссылка	Hyperlink
Глобальная объектная среда	Global Object-Based Environment, Globe
Глобальная сеть	Wide-Area Network, WAN
Глобальная служба имен	Global Name Service, GNS
Глобальное время по атомным часам	International Atomic Time, TAI
Глобальное имя	Global name
Глобальное состояние	Global state
Глобальный каталог	Global catalog
Глобальный уровень	Global layer
Голосование	Voting
Гомогенная (система)	Homogeneous
Горизонтальное распределение	Horizontal distribution
Горизонтальные средства	Horizontal facilities
Граница восстановления	Recovery line

Термин	Оригинальное название
Группа	Group
Группа доступных хранилищ тома	Accessible Volume Storage Group, AVSG
Группа нарезки	Stripe group
Группа объектов	Object group
Группа очередей	Queue group
Группа управления объектами	Object Management Group, OMG
Группа хранилищ тома	Volume Storage Group, VSG
Групповая рассылка	Multicasting
Дайджест сообщений	Message digest
Двойная подпись	Dual signature
Двойное извлечение	Pull-pull
Двухфазная блокировка	Two-Phase Locking, 2PL
Делегирование	Delegation
Делегирование открытия	Open delegation
Демаршалинг	Unmarshaling
Демон контактов	Rendezvous daemon
Демон транзакций	Transaction daemon
Дерево доменов	Domain tree
Дескриптор объекта	Object handle
Дескриптор привязки	Binding handle
Дескриптор реализации	Implementation handle
Дескриптор файла	File handle
Динамический распределенный объект	Distributed dynamic object
Динамический язык HTML	Dynamic HTML
Динамическое обращение	Dynamic invocation
Дискретная среда представления	Discrete representation media
Диспетчеризация	Dispatching
Доверенная вычислительная база	Trusted Computing Base, TSB
Долговечная (транзакция, память)	Durable
Домен	Domain
Домен защиты	Protection domain
Доменное имя	Domain name
Доменный сервер кэширования	Domain Caching Server, DCS
Доставка «с максимальными усилиями»	Best effort delivery
Доставка сертифицированных сообщений	Certified message delivery
Доступность	Availability
Европейская система защиты приложений, работающих в разнородной среде	Secure European System for Application in a Multi-vendor Environment, SESAME
Жесткая ссылка	Hard link
Журнал регистрации	Ledger
Журнал с упреждающей записью	Write-ahead log

Термин	Оригинальное название
Журнал только для записи	Append-only log
Заглушка удаления	Deletion stub
Заголовок	Header
Загрузчик классов	Class loader
Задание сервера	Server task
Заместитель	Proxy
Заместитель пользователя	User proxy
Заместитель ресурса	Resource proxy
Заметка	Note
Заметка с данными	Data note
Запись о ресурсе	Resource record
Захват	Acquire
Защита	Security
Защита транспортного уровня	Transport Layer Security, TLS
Защищенная файловая система	Secure File System, SFS
Защищенная электронная транзакция	Secure Electronic Transactions, SET
Защищенные (данные)	Protected
Защищенный канал	Secure channel
Зеркало, или зеркальный сайт	Mirror site
Зона	Zone
Идемпотентный (запрос, операция)	Idempotent
Идентификатор адреса	Address identifier
Идентификатор базы данных	Database ID
Идентификатор заметки	Note ID
Идентификатор инициатора	Originator ID, OID
Идентификатор интерфейса	Interface Identifier, IID
Идентификатор класса	Class Identifier, CLSID
Идентификатор протокола	Protocol identifier
Идентификатор реплики	Replica ID
Идентификатор реплицированного тома	Replicated Volume Identifier, RVID
Идентификатор службы	Service identifier
Идентификатор тома	Volume Identifier, VID
Идентификатор транзакции	Transaction Identifier, XID
Идентификатор хранения	Repository identifier
Избыточность	Redundancy
Извлечение	Pull
Извлечение-продвижение	Pull-push
Изменение представления	View change
Изолированная (транзакция)	Isolated
Изохронный режим передачи	Isochronous transmission mode

Термин	Оригинальное название
Именованное	Naming
Именованный распределенный объект	Distributed named object
Имя входа	Inbox name
Имя пути	Path name
Имя, удобное для восприятия	Human-friendly name
Инвентаризация накопления	Hoard walk
Инкрементный снимок состояния	Incremental snapshot
Интернет-протокол обмена между ORB	Internet Inter-ORB Protocol, IIOP
Интерфейс	Interface
Интерфейс динамического обращения	Dynamic Invocation Interface, DII
Интерфейс очередей сообщений	Message Queue Interface, MQI
Интерфейс передачи сообщений	Message-Passing Interface, MPI
Интерфейс транспортного уровня	Transport Layer Interface, TLI
Интерфейсный агент	Interface agent
Инфицированный (сервер)	Infective
Информационная база каталога	Directory Information Base, DIB
Информационная база управления защитой	Security Management Information Base, SMIB
Информационная шина	Information bus
Информационное дерево каталогов	Directory Information Tree, DIT
Информационный агент	Information agent
Искатель (агентов)	Finder
Исключение	Exception
Истечение срока	Expiration
Истребление сирот	Extermination
Исходящая очередь	Source queue
Итеративное разрешение имени	Iterative name resolution
Итеративный сервер	Iterative server
Кадр	Frame
Канал связи между агентами	Agent Communication Channel, ACC
Канал событий	Event channel
Канал сообщений	Message channel
Каноническое имя	Canonical name
Карта активных объектов	Active object map
Карта индексов	Imap
Карта менеджеров	Manager map
Каскадные прерывания	Cascaded aborts
Каталог Domino	Domino directory
Качество обслуживания	Quality of Service, QoS
Кворум записи	Write quorum
Кворум чтения	Read quorum
Кэш дублированных запросов	Duplicate-request cache

Термин	Оригинальное название
Кэш обратной записи	Write-back cache
Кэш сквозной записи	Write-through cache
Кэширование	Caching
Кэширование указателей	Pointer caching
Кэш-память	Cache memory
Кэш-попадание	Cache hit
Кэш-промах	Cache miss
Класс событий	Event class
Кластер	Cluster
Кластер рабочих станций	Cluster Of Workstations, COW
Клиент	Client
Клиент доступа к аутентификации и привилегиям	The Authentication and Privilege Access Client, APA Client
Клиентская заглушка	Client stub
Клиентский протокол	Client-based protocol
Коммутируемая (архитектура)	Switched
Коммутируемая мультимегабитная служба данных	Switched Multi-megabit Data Service, SMDS
Коммутирующая решетка	Crossbar switch
Комплексный поток данных	Complex stream
Компонент очереди	Queued Component, QC
Конвейеризация	Pipelining
Конец связи	Out-of-band
Конечная точка	Endpoint
Контактный адрес	Contact address
Контекст	Context
Контекст защиты	Security context
Контекст именования	Naming context
Контекст потока выполнения	Thread context
Контроллер домена	Domain controller
Контроль доступа	Access control
Контрольная сумма	Checksum
Контрольная точка	Checkpoint
Контрольный адрес	Care-of address
Конфиденциальная почта	Privacy Enhanced Mail, PEM
Конфиденциальность	Confidentiality
Конфликт двойной записи	Write-write conflict
Конфликт чтения-записи	Read-write conflict
Конфликтующая операция	Conflicting operation
Кооперативное кэширование	Collaborative caching, или cooperative caching

Термин	Оригинальное название
Кооперативный агент	Collaborative agent
Координатор распределенных транзакций	Distributed Transaction Coordinator, DTC
Координированное создание контрольных точек	Coordinated checkpointing
Корень, или корневой узел	Root node
Корневой дескриптор файлов	Root file handle
Корневой набор	Root set
Корневой направляющий узел	Root directory node
Корневой узел, или корень	Root node
Кортеж	Tuple
Коэффициент кэш-попаданий	Hit rate
Криптографическая пломба	Cryptographic seal
Кусочно-детерминированная модель	Piecewise deterministic model
Ленивая свободная непротиворечивость	Lazy release consistency
Лес доменов	Domain forest
Линеаризуемость	Linearizability
Листовой домен	Leaf domain
Листовой узел, или лист	Leaf node
Логические часы	Logical clocks
Локализующая запись	Location record
Локальная сеть	Local-Area Network, LAN
Локально независимое (имя)	Local independent
Локальное имя	Local name
Локальное представление	Local representative
Локальный объект	Local object
Локальный псевдоним	Local alias
Максимальная скорость дрейфа	Maximum drift rate
Мандат	Capability
Мандат владельца	Owner capability
Маркер аутентификации	Authentication token
Маркер, или токен	Token
Маркерное кольцо	Token Ring
Маршалинг	Marshaling
Маршалинг параметров	Parameter marshaling
Маршрутизация	Routing
Маршрутизирующий демон контактов	Rendezvous router daemon
Масштабируемая надежная групповая рассылка	Scalable Reliable Multicasting, SRM
Матрица контроля доступа	Access control matrix
Международная организация по стандартам	International Standards Organization, ISO
Межоперационная ссылка на группу объектов	Interoperable Object Group Reference, IOGR

Термин	Оригинальное название
Межоперационная ссылка на объект	Interoperable Object Reference, IOR
Межпроцессное взаимодействие	InterProcess Communication, IPC
Менеджер данных	Data manager
Менеджер защиты	Security manager
Менеджер метаданных	Metadata manager
Менеджер окон	Window manager
Менеджер очередей	Queue manager
Менеджер репликации	Replication manager
Менеджер ресурсов	Resource manager
Менеджер транзакций	Transaction manager
Менеджер управления службами	Service Control Manager, SCM
Место (в системе агентов)	Place
Метод	Method
Метод модификации	Modify method
Метод чтения	Read method
Механизм защиты	Security mechanism
Механизм свертывания	Closure mechanism
Микроданные	Microdata
Микроядро	Microkernel
Многоцелевые расширения почты Интернета	Multipurpose Internet Mail Extensions, MIME
Мобильный агент	Mobile agent
Модель доверия	Trust model
Модель загрузки-выгрузки	Upload/download model
Модель извлечения	Pull model
Модель компонентных объектов	Component Object Model, COM
Модель непротиворечивости	Consistency model
Модель обратного вызова	Callback model
Модель опроса	Polling model
Модель продвижения	Push model
Модель удаленного доступа	Remote access model
Модификация	Modification
Модуль расширения	Plug-in
Моникер	Moniker
Монитор	Monitor
Монитор ссылок	Reference monitor
Монтажная точка	Mount point
Мультикомпьютер	Multicomputer
Мультикомпьютерная операционная система	Multicomputer operating system
Мультипроцессор	Multiprocessor

Термин	Оригинальное название
Мультипроцессорная операционная система	Multiprocessor operating system
Мьютекс	Mutex
Мягкая реинкарнация	Gentle reincarnation
Надежная групповая рассылка	Reliable multicasting
Надежная групповая рассылка в порядке FIFO	Reliable FIFO-ordered multicast
Надежная неупорядоченная групповая рассылка	Reliable unordered multicast
Надежная причинно упорядоченная групповая рассылка	Reliable causally-ordered multicast
Надежная система	Dependable system
Надежность	Dependability
Название темы	Subject name
Накопление (файлов)	Hoarding
Направленный ациклический граф	Directed acyclic graph
Направляющая таблица	Directory table
Направляющий узел	Directory node
Нарезка	Striping
Независимое создание контрольных точек	Independent checkpointing
Непрерывная среда представления	Continuous representation media
Неприсоединенный ресурс	Unattached resource
Непротиворечивая (транзакция)	Consistent
Непротиворечивость	Consistency
Непротиворечивость записи за чтением	Writes-follow-reads consistency
Непротиворечивость монотонного чтения	Monotonic-read consistency
Непротиворечивость монотонной записи	Monotonic-write consistency
Непротиворечивость чтения собственных записей	Read-your-writes consistency
Непротиворечивость, ориентированная на клиента	Client-centric consistency
Непрямая привязка	Indirect binding
Нерезидентная связь	Transient communication
Нерезидентный локальный объект	Transient local object
Нерезидентный объект	Transient object
Несохранное (соединение)	Nonpersistent
Неунифицированный доступ к памяти	NonUniform Memory Access, NUMA
Неустойчивое сообщение	Unstable message
Неявная привязка	Implicit binding
Обещание обратного вызова	Callback promise
Облегченный процесс	LightWeight Process, LWP
Обмен ключами по Диффи—Хеллману	Diffie—Hellman key exchange

Термин	Оригинальное название
Обобщенная архитектура брокера объектных запросов	Common Object Request Broker Architecture, CORBA
Обобщенный интерфейс программ службы защиты	Generic Security Service Application Program Interface, GSS-API
Обобщенный интерфейс шлюзов	Common Gateway Interface, CGI
Обобщенный протокол обмена между ORB	General Inter-ORB Protocol, GIOP
Оболочка	Wrapper
Обработчик	Handler
Обратное исправление	Backward recovery
Обратный удар	Feedback implosion
Объединенный каталог	Union directory
Объект	Object
Объект класса	Class object
Объект отмены	Cancel object
Объект поддержки	Principal object
Объект правил	Policy object
Объект события	Event object
Объектная модель документа	Document Object Model, DOM
Объект-слушатель	Listener object
Объект-фабрика	Factory object
Оверлейная сеть	Overlay network
Ограниченный интерфейс к защищенным системным компонентам	Reduced Interfaces for Secure System Components, RISSC
Однократная запись с множественным чтением	Write-Once Read-Many, WORM
Однопроцессорная система	Single-processor system
Одноразовая операция	At-most-once operation
Одноранговое распределение	Peer-to-peer distribution
Односторонний вызов RPC	One-way RPC
Односторонний запрос	One-way request
Односторонняя функция	One-way function
Омега-сеть	Omega network
Онтология	Ontology
Организация регистрации правил в Интернете	Internet Policy Registration Authority, IPRA
Организация сертификации атрибутов	Attribute certification authority
Организация сертификации правил	Policy Certification Authority, PCA
Ориентированный на сообщения промежуточный уровень	Message-Oriented Middleware, MOM
Освобождение	Release
Осиротевший процесс	Orphan process
Остроконечный (формат)	Little endian

Термин	Оригинальное название
Отзыв (сертификата)	Revoke
Отказ	Fault
Отказоустойчивость	Fault tolerance
Откат (транзакции)	Rollback
Открытая распределенная система	Open distributed system
Отложенный синхронный вызов RPC	Deferred synchronous RPC
Отложенный синхронный запрос	Deferred synchronous request
Отмена обратного вызова	Callback break
Относительно различимое имя	Relative Distinguished Name, RDN
Относительное имя пути	Relative path name
Отсоединенный (клиент)	Disconnected
Очереди сообщений Microsoft	Microsoft Message Queuing, MSMQ
Очередь назначения	Destination queue
Очищенный (сервер)	Removed
Ошибка	Error
Ошибка аварийной остановки	Fail-stop failure
Ошибка значения	Value failure
Ошибка отклика	Response failure
Ошибка передачи состояния	State transition failure
Ошибка производительности	Performance failure
Ошибка синхронизации	Timing failure
Ошибочное разделение	False sharing
Пакет	Packet
Параллельные (события, операции)	Concurrent
Параллельный сервер	Concurrent server
Пассивный (объект)	Passive
Первичная двухфазная блокировка	Primary two-phase locking
Передача сообщений	Message passing
Перекрестный сертификат	Cross certificate
Перебегающий отказ	Intermittent fault
Переменная синхронизации	Synchronization variable
Перенос зон	Zone transfer
Перенос процессов	Process migration
Переносимость	Portability
Переносимый адаптер объектов	Portable Object Adapter, POA
Перетаскивание	Drag-and-drop
Перехват	Interception
Перехватчик	Interceptor
Перехватчик защищенных обращений	Secure invocation interceptor
Перехватчик контроля доступа	Access control interceptor
Перехватчик уровня запросов	Request-level interceptor

Термин	Оригинальное название
Перехватчик уровня сообщений	Message-level interceptor
Период амнистии	Grace period
Персональный идентификационный номер	Personal Identification Number, PIN
План	Schedule
Планировщик	Scheduler
Плоская транзакция	Flat transaction
Побочный эффект	Side effect
Подавление откликов	Feedback suppression
Подделка	Fabrication
Подкачка	Swapping
Подключаемый модуль аутентификации	Pluggable Authentication Module, PAM
Подобъект защиты	Security subobject
Подобъект репликации	Replication subobject
Подобъект связи	Communication subobject
Подобъект семантики	Semantics subobject
Подобъект управления	Control subobject
Подписание кода	Code-signing
Подсчет поколений ссылок	Generation reference counting
Полигон	Playground
Политика активизации	Activation policies
Полностью упорядоченная групповая рассылка	Totally-ordered multicasting
Поломка	Crash failure
Пользовательский агент	User agent
Пользовательский режим	User mode
Пометка о захвате (элемента)	Check out
Пометка об освобождении (элемента)	Check in
Поочередная сериализация копий	One-copy serializability
Порт	Port
Порт сервера	Server port
Последовательная непротиворечивость	Sequential consistency
Поставщик	Supplier
Постоянный отказ	Permanent fault
Потенциальная непротиворечивость	Eventual consistency
Потерянная секунда	Leap second
Поток выполнения	Thread
Поток данных	Data stream, или stream
Потребитель	Consumer
Поэлементная непротиворечивость	Entry consistency
Правило защиты	Security policy

Термин	Оригинальное название
Правильный идентификатор	True identifier
Право доступа	Access right
Прагматическая общая групповая рассылка	Pragmatic General Multicast, PGM
Представление группы	Group view
Прерывание	Interruption
Привязка	Binding
Привязка по значению	Binding by value
Привязка по идентификатору	Binding by identifier
Привязка по типу	Binding by type
Приложение-помощник	Helper application
Примитивный локальный объект	Primitive local object
Причинная непротиворечивость	Causal consistency
Проблема атомарной групповой рассылки	Atomic multicast problem
Проблема византийских генералов	Byzantine generals problem
Проблема двух армий	Two-army problem
Программный агент	Software agent
Программный поток данных	Program stream
Продвижение	Push
Прозрачная (система)	Transparent
Прозрачность доступа	Access transparency
Прозрачность местоположения	Location transparency
Прозрачность отказов	Failure transparency
Прозрачность параллельного доступа	Concurrency transparency
Прозрачность переноса	Migration transparency
Прозрачность репликации	Replication transparency
Прозрачность смены местоположения	Relocation transparency
Прозрачность сохранности	Persistence transparency
Произвольная ошибка	Arbitrary failure
Прокси-шлюз	Proxy gateway
Пропуск данных	Omission failure
Пропуск передачи	Send omission
Пропуск приема	Receive omission
Простой поток данных	Simple stream
Простой текст	Plaintext
Пространство имен	Name space
Протокол	Protocol
Протокол аутентификации Нидхема—Шредера	Needham—Schroeder authentication protocol
Протокол без установления соединения	Connectionless protocol
Протокол блокирующего подтверждения	Blocking commit protocol
Протокол двухфазного подтверждения	Two-Phase Commit Protocol, 2PC

Термин	Оригинальное название
Протокол запрос-ответ	Challenge-response protocol
Протокол извлечения	Pull-based protocol
Протокол Интернета	Internet protocol, IP
Протокол непротиворечивости	Consistency protocol
Протокол о несостоятельности	Invalidation protocol
Протокол однофазного подтверждения	One-phase commit protocol
Протокол оптимистического протоколирования	Optimistic logging protocol
Протокол первичного архивирования	Primary-backup protocol
Протокол передачи гипертекста	Hypertext Transfer Protocol, HTTP
Протокол передачи файлов	File Transfer Protocol, FTP
Протокол пессимистического протоколирования	Pessimistic logging protocol
Протокол продвижения	Push-based protocol
Протокол разрешения адресов	Address Resolution Protocol, ARP
Протокол резервирования ресурсов	Resource reSerVation Protocol, RSVP
Протокол с установлением соединения	Connection-oriented protocol
Протокол сетевого времени	Network Time Protocol, NTP
Протокол трехфазного подтверждения	Three-Phase Commit Protocol, 3PC
Протокол управления передачей	Transmission Control Protocol, TCP
Протоколирование отправителем	Sender-based logging
Протоколирование получателем	Receiver-based logging
Протоколирование сообщений	Message logging
Преходной отказ	Transient fault
Процедура разрешения имен	Name resolver
Процесс	Process
Процессоры с массовым параллелизмом	Massively Parallel Processors, MPP
Прямая привязка	Direct binding
Прямое исправление	Forward recovery
Прямое согласование	Direct coordination
Псевдоним	Alias
Публикация	Publishing
Путь	Path
Равновесие	Equilibrium
Раздел	Partition
Разделение секрета	Secret sharing
Различимое имя	Distinguished name
Разрешение имени	Name resolution
Распределение	Distribution
Распределение запросов с известным содержимым	Content-aware request distribution

Термин	Оригинальное название
Распределенная двухфазная блокировка	Distributed two-phase locking
Распределенная модель COM	Distributed COM, DCOM
Распределенная операционная система	Distributed Operating System, DOS
Распределенная подготовка документов в Web с контролем версий	Web Distributed Authoring and Versioning (WebDAV)
Распределенная разделяемая память	Distributed Shared Memory, DSM
Распределенная система	Distributed system
Распределенная транзакция	Distributed transaction
Распределенная файловая система	Distributed file system
Распределенное подтверждение (транзакции)	Distributed commit
Распределенный документ	Distributed document
Распределенный объект	Distributed object
Распределенный разделяемый объект	Distributed shared object
Распределенный сборщик мусора	Distributed garbage collector
Распределенный снимок состояния	Distributed snapshot
Распространение слухов	Rumor spreading
Рассинхронизация часов	Clock skew
Расширенный самоанализ стека	Extended stack introspection
Расширяемый язык разметки	eXtensible Markup Language, XML
Расширяемый язык стилей	eXtensible Style Language, XSL
Регион	Region
Регистр хранения	Holding register
Редактирование по месту	In-place editing
Режим работы запрос-ответ	Request-reply behavior
Режим ядра	Kernel mode
Реинкарнация	Reincarnation
Рекурсивное разрешение имени	Recursive name resolution
Ремонтопригодность	Maintainability
Репликация	Replication
Роль	Role
Самосертифицирующееся имя пути	Self-certifying pathname
Свидетельство о смерти	Death certificate
Свободная непротиворечивость	Release consistency
Свободный маркер	Clear token
Связанный ресурс	Fastened resource
Связность данных	Data coherence
Связность памяти	Memory coherence
Связующий документ	Connection document
Связующий объект	Connectable object
Связывание и внедрение объектов	Object Linking and Embedding, OLE

Термин	Оригинальное название
Связь, или взаимодействие	Communication
Сеансовый ключ	Session key
Сейфовый объект	Vault object
Секвенсор	Sequencer
Секретный маркер	Secret token
Семантика «максимум однажды»	At most once semantics
Семантика «минимум однажды»	At least once semantics
Семантика «только однажды»	Exactly once semantics
Семантика UNIX	UNIX semantics
Семантика сеансов	Session semantics
Семафор	Semaphore
Сервер	Server
Сервер Domino	Domino server
Сервер атрибутов привилегий	Privilege Attribute Server, PAS
Сервер аутентификации	Authentication Server, AS
Сервер без фиксации состояния	Stateless server
Сервер времени	Time server
Сервер защиты домена	Domain Security Server, DSS
Сервер локализации	Location server
Сервер объектов	Object server
Сервер распространения ключей	Key Distribution Server, KDS
Сервер с фиксацией состояния	Stateful server
Сервер хранения	Storage server
Серверная заглушка	Server stub
Серверный протокол	Server-based protocol
Серверный сценарий	Server-side script
Сервлет	Servlet
Сериализуемый (тип)	Serializable
Сертификат	Certificate
Сертификат атрибута	Attribute certificate
Сертификат атрибутов привилегий	Privilege Attribute Certificate, PAC
Сертификат открытого ключа	Public key certificate
Сертифицирующая организация	Certification authority
Сетевая операционная система	Network Operating System, NOS
Сетевая файловая система	Network File System, NFS
Сеть доставки содержимого	Content Delivery Network, CDN
Сеть распределения содержимого	Content distribution network
Сильная мобильность	Strong mobility
Сильная устойчивость к коллизиям	Strong collision resistance
Символическая ссылка	Symbolic link

Термин	Оригинальное название
Симметричная криптосистема	Symmetric cryptosystem
Синхронизируемый (метод)	Synchronized
Синхронная связь	Synchronous communication
Синхронный режим передачи	Synchronous transmission mode
Сирота	Orphan
Система агентов	Agent system
Система групповой работы	Groupware
Система остановки без уведомления	Fail-silent system
Система очередей сообщений	Message-queuing system
Система проверки PAC	PAC Validation Facility, PVF
Система промежуточного уровня	Middleware
Система публикации/подписки	Publish/subscribe system
Система с открытым ключом	Public-key system
Системная сеть	System Area Network, SAN
Сито	Sandbox
Сквозной аргумент	End-to-end argument
Скелетон	Skeleton
Слабая мобильность	Weak mobility
Слабая непротиворечивость	Weak consistency
Слабая устойчивость к коллизиям	Weak collision resistance
Слепая подпись	Blind signature
Слуга	Servant
Служба аутентификации и авторизации Java	Java Authentication and Authorization Service, JAAS
Служба внешних связей	Externalization service
Служба времени	Time service
Служба жизненного цикла	Life cycle service
Служба запросов	Query service
Служба защиты	Security service
Служба именования	Naming service
Служба каталогов	Directory service
Служба коллекций	Collection service
Служба лицензирования	Licensing service
Служба локализации	Location service
Служба обмена	Trading service
Служба объектов Notes	Notes Object Service, NOS
Служба отношений	Relationship service
Служба параллельного доступа	Concurrency service
Служба поиска	Lookup service
Служба предоставления талонов	Ticket Granting Service, TGS
Служба распределенного времени	Distributed time service

Термин	Оригинальное название
Служба распределенных файлов	Distributed file service
Служба свойств	Property service
Служба событий	Event service
Служба сообщений	Messaging service
Служба сохранности	Persistence service
Служба транзакций	Transaction service
Служба уведомлений	Notification service
Слушатель событий	Listener event
Совместное резервирование	Share reservation
Согласование на встрече	Meeting-oriented coordination
Согласование через почтовый ящик	Mailbox coordination
Согласованная (память)	Coherent
Согласующее сообщение	Flush message
Создание зеркал	Mirroring
Сокет	Socket
Солнечная секунда	Solar second
Солнечный день	Solar day
Солнечный переход	Transit of sun
Сообщение о получении	Capture message
Составная процедура	Compound procedure
Составной документ	Compound document
Составной локальный объект	Composite local object
Состояние	State
Состояние только для чтения	Read-only state
Сохранная связь	Persistent communication
Сохранное (соединение)	Persistent
Сохранность	Persistence
Сохранный локальный объект	Persistent local object
Сохранный объект	Persistent object
Спецификация передачи	Flow specification
Список контроля доступа	Access Control List, ACL
Список контроля исполнения	Execution Control List, ECL
Список отозванных сертификатов	Certificate Revocation List, CRL
Список ссылок	Reference list
Спонсор пользователя	User sponsor
Способность к взаимодействию	Interoperability
Среда распределенных вычислений	Distributed Computing Environment, DCE
Среднее время по Гринвичу	Greenwich mean time
Средняя солнечная секунда	Mean solar second
Средства CORBA	CORBA facilities

Термин	Оригинальное название
Средства связи	Communication facilities
Срез	Cut
Срок жизни (сертификатов)	Lifetime
Ссылка на заметку	notelink
Стандарт шифрования данных	Data Encryption Standard, DES
Статическое обращение	Static invocation
Стек команд	Command stack
Стек переноса	Migration stack
Страж	Guard
Стратегии установления согласованности	Coherence enforcement strategy
Стратегия обнаружения согласованности	Coherence detection strategy
Строгая двухфазная блокировка	Strict two-phase locking
Строгая непротиворечивость	Strict consistency
Строка запроса	Request line
Строка инициализации	Initialization string
Строка состояния	Status line
Субъект	Subject
Суперсервер	Superserver
Схема	Scheme
Счетчик	Counter
Таблица запущенных объектов	Running Object Table, ROT
Таблица процессов	Process table
Таймер	Timer
Талон	Ticket
Теговый профиль	Tagged profile
Теневой блок	Shadow block
Тень	Shadow
Тик таймера	Clock tick
Тип MIME	MIME type
Токен, или маркер	Token
Том	Volume
Точка доступа	Access point
Точка монтирования	Mounting point
Транзакционная очередь	Transactional queue
Транзакционный обмен сообщениями	Transactional messaging
Транзакционный объект	Transactional object
Транспорт	Transport
Транспортный протокол реального времени	Real-time Transport Protocol, RTP
Трасси	g-based garbage collection
Тройное модульное резервирование	Triple Modular Redundancy, TMR
Тупик, или взаимная блокировка	Deadlock

Термин	Оригинальное название
Тупоконечный (формат)	Big endian
Угроза защите	Security threat
Удаленная файловая служба	Remote file service
Удаленное обращение к методам	Remote Method Invocations, RMI
Удаленный вызов процедур	Remote Procedure Calls, RPC
Удостоверение	Identity
Узловой коммутатор	Crosspoint switch
Универсальное согласованное время	Universal Coordinated Time, UTC
Универсальный идентификатор	Universal ID, UNID
Универсальный протокол дейтаграмм	Universal Datagram Protocol, UDP
Унифицированное имя ресурса	Uniform Resource Name, URN
Унифицированный идентификатор ресурса	Uniform Resource Identifier, URI
Унифицированный указатель ресурса	Uniform Resource Locator, URL
Упакованный адрес	Stacked address
Упаковщик объектов	Object wrapper
Управление контекстом защиты	Secure Association Context Management, SACM
Управление пространством имен	Name space management
Управленческий уровень	Managerial layer
Упрощенный вызов RPC	Lightweight RPC
Упрощенный протокол доступа к каталогам	Lightweight Directory Access Protocol, LDAP
Уровень аутентификации	Authentication level
Уровень защищенных сокетов	Secure Socket Layer, SSL
Уровень обезличивания	Impersonation level
Уровень протокола записей TLS	TLS record protocol layer
Условная анонимность	Conditional anonymity
Условная переменная	Condition variable
Устойчивое сообщение	Stable message
Устойчивое хранилище	Stable storage
Фаза подъема	Growing phase
Фаза спада	Shrinking phase
Файловая система Andrew	Andrew File System, AFS
Файловая система со структурой журнала	Log-Structured File System, LFS
Файловый моникер	File moniker
Файловый сервер	File server
Физически двухзвенная архитектура	Physically two-tiered architecture
Физически трехзвенная архитектура	Physically three-tiered architecture
Фиксированный ресурс	Fixed resource
Фрагмент паритета	Parity fragment
Функция обратного вызова	Callback function

Термин	Оригинальное название
Хэш-функция	Hash function
Хранилище данных	Data store
Хранилище интерфейсов	Interface repository
Хранилище реализаций	Implementation repository
Целевая группа инженерной поддержки Интернета	Internet Engineering Task Force, IETF
Целевая рассылка	Unicasting
Целостность	Integrity
Центр распространения ключей	Key Distribution Center, KDC
Централизованная двухфазная блокировка	Centralized two-phase locking
Централизованная система	Centralized system
Цифровая подпись	Digital signature
Цифровые деньги	Digital money
Читаем раз, пишем все	Read-One — Write-All, ROWA
Шина	Bus
Широковещательная рассылка	Broadcasting
Шифрование	Encryption
Шифрованный текст	Ciphertext
Шлюз прикладного уровня	Application-level gateway
Шлюз фильтрации пакетов	Packet-filtering gateway
Экземпляр кортежа	Tuple instance
Экспорт	Exporting
Элемент	Item
Элемент службы	Service item
Энергичная свободная непротиворечивость	Eager release consistency
Эпидемический протокол	Epidemic protocol
Эталон	Template
Эталонная модель взаимодействия открытых систем	Open systems interconnection reference model
Эффект домино	Domino effect
Явная привязка	Explicit binding
Язык IDL от Microsoft	Microsoft IDL, MIDL
Язык взаимодействия агентов	Agent Communication Language, ACL
Язык определения интерфейсов	Interface Definition Language, IDL
Язык разметки гипертекста	HyperText Markup Language, HTML

Алфавитный указатель

A

Access Control List, ACL, 494, 744
Accessible Volume Storage Group,
AVSG, 666
Active Directory, 586
ActiveX, 573
Address Resolution Protocol, ARP, 244
Agent Communication Channel,
ACC, 208
Agent Communication Language,
ACL, 208
Amoeba, 392
Andrew File System, AFS, 623, 654
Apache, 716
Atomicity-Consistency-Isolation-
Durability, ACID, 309
Authentication Server, AS, 518, 521
Automatic Teller Machines, ATM, 528

C

Carnegie Mellon University, CMU, 654
Class Identifier, CLSID, 575
Clusters Of Workstations, COW, 43
Coda, 654
Common Gateway Interface, CGI, 706
Common Object Request Broker
Architecture, CORBA, 540
Component Object Model, COM, 572
Content Delivery Network, CDN, 727
cookie, 183

D

D'Agents, 204
Data Encryption Standard, DES, 473
Directory Information Base,
DIB, 238
Directory Information Tree, DIT, 238
Directory Service Agent, DSA, 240
Directory User Agent, DUA, 240
Distributed COM, DCOM, 572
Distributed Computing Environment,
DCE, 106
Distributed Operating System,
DOS, 45
Distributed Shared Memory,
DSM, 54
Distributed Transaction Coordinator,
DTC, 589
Document Object Model, DOM, 703
Domain Caching Server, DCS, 565
Domain Name System, DNS, 232
Domain Security Server, DSS, 521
Dynamic Invocation Interface,
DII, 543

E

Execution Control Lists, ECL, 744
EXtensible Markup Language,
XML, 703
eXtensible Style Language, XSL, 704

F

File Transfer Protocol, FTP, 91
Foundation for Intelligent Physical
Agents, FIPA, 207

G

General Inter-ORB Protocol,
GIOP, 553
Global Name Service, GNS, 224
GLObal Object-Based Environment,
Globe, 592

H

HyperText Markup Language,
HTML, 701
Hypertext Transfer Protocol,
HTTP, 91, 709

I

Interface Definition Language,
IDL, 30, 102, 108, 542
Interface Identifier, IID, 574
International Atomic Time, TAI, 278
International Standards Organization,
ISO, 82
Internet Engineering Task Force,
IETF, 723
Internet Inter-ORB Protocol,
IIOP, 553
Internet Link, IL, 675
Internet Policy Registration Authority,
IPRA, 510
Internet protocol, IP, 87
Interoperable Object Group Reference,
IOGR, 567
Interoperable Object Reference,
IOR, 561
InterProcess Communication, IPC, 103
ISO OSI, 83

J

Java Authentication and Authorization
Service, JAAS, 783
Java Virtual Machine, JVM, 502
JavaScript, 702
JavaSpace, 770
JIT-активизация, 583

K

Key Distribution Center, KDC, 482
Key Distribution Server, KDS, 522

L

Lightweight Directory Access Protocol,
LDAP, 240
LightWeight Processes, LWP, 169
LIPKEY, 651
Local-Area Network, LAN, 22
Log-Structured File System,
LFS, 682

M

m,n-пороговая схема, 493
Massively Parallel Processors,
MPP, 43
Message Channel Agent, MCA, 143
Message Queue Interface, MQI, 146
Message-Oriented Middleware,
MOM, 136
Message-Passing Interface, MPI, 134
Microsoft IDL, MIDL, 574
Microsoft Message Queuing,
MSMQ, 580
Motion Picture Experts Group,
MPEG, 158
Multipurpose Internet Mail Extensions,
MIME, 705
MultiRPC, 658

N

Network File System, NFS, 223
Network Operating Systems, NOS, 45
Network Time Protocol, NTP, 284
нонсе, случайное число, 484
NonUniform Memory Access,
NUMA, 42

O

Object Linking and Embedding,
OLE, 573
Object Management Group,
OMG, 540
Object Request Broker, ORB, 541
Open Network Computing RPC,
ONC RPC, 629
Orca, 388
Originator ID, OID, 737

P

PAC Validation Facility, PVF, 523
Personal Identification Number,
PIN, 528
Plan 9, 674
Pluggable Authentication Module,
PAM, 783
Policy Certification Authorities,
PCA, 510
Pragmatic General Multicast,
PGM, 766
Privacy Enhanced Mail, PEM, 510
Privilege Attribute Certificate,
PAC, 522
Privilege Attribute Server, PAS, 522

Q

Quality of Service, QoS, 151
Queued Components, QC, 580

R

Read-One — Write-All, ROWA, 666
Real-time Transport Protocol,
RTP, 88
Recoverable Virtual Memory,
RVM, 670
Reduced Interfaces for Secure System
Components, RISSC, 469
Relative Distinguished Name,
RDN, 238
Remote Method Invocation,
RMI, 117, 418
Remote Procedure Call,
RPC, 62, 94, 418
Replicated Volume Identifier,
RVID, 660
Resource reSerVation Protocol,
RSVP, 154
ROWA, 385
RPCSEC_GSS, 651
Running Object Table, ROT, 585

S

Scalable Reliable Multicasting,
SRM, 427
Secure Association Context
Management, SACM, 523
Secure Electronic Transactions,
SET, 533
Secure File System, SFS, 686
Secure Socket Layer, SSL, 468, 729
Security Management Information Base,
SMIB, 521
Service Control Manager,
SCM, 576, 583
Sun Network File System, NFS, 624
Switched Multi-megabit Data Service,
SMDS, 467
System Area Network, SAN, 42
S-бюкс, 474

T

The Authentication and Privilege Access Client, APA Client, 522
Ticket Granting Service, TGS, 518
Token Ring, 303
Transmission Control Protocol, TCP, 88
Transport Layer Interface, TLI, 132
Transport Layer Security, TLS, 729
Triple Modular Redundancy, TMR, 409
Trusted Computing Base, TSB, 468
Two-Phase Locking, 2PL, 319

U

Uniform Resource Identifier, URI, 606, 717, 721
Uniform Resource Locator, URL, 25, 700, 721
Uniform Resource Name, URN, 721
Universal Coordinated Time, UTC, 279
Universal Datagram Protocol, UDP, 88
Universal ID, UNID, 737

V

Virtual File System, VFS, 625
Volume Storage Group, VSG, 666

W

Web Distributed Authoring and Versioning, WebDAV, 723
web-браузер, 715
web-заместитель, 715
Wide-Area Network, WAN, 23
World Wide Web, WWW, 25, 699
Write-Once Read-Many, WORM, 677

X

X.500, 238
xFS, 680
X-Windows, 175
X-протокол, 176
X-терминал, 177
X-ядро, 176

A

абсолютное имя пути, 218
автоматическая кассовая машина, 528
автоматическая транзакция, 588
автоматический монтировщик, 636
автоматическое монтирование, 635
авторизация, 460, 493
агент, 200
 базы, 247
 интерфейсный, 206
 информационный, 206
 канала сообщений, 143
 кооперативный, 205
 мобильный, 205
 пользователей каталога, 240
 почтовый, 206
 программный, 205
 службы каталогов, 240
адаптер объектов, 114, 185
административный уровень, 226
адрес, 215
 временный, 248
 контрольный, 248
 протокола, 606
 упакованный, 605
 экземпляра, 606
адресация по теме, 756
активизация
 на лету, 583
 планировщика, 171
активная репликация, 371, 382
активная цель, 768
активное сообщение, 681

активный кэш, 726
алгоритм
 Беркли, 282
 голосования, 296
 забияки, 296
 кольцевой, 298
динамической репликации, 367
корзины элементарных пакетов, 152
Кристиана, 281
Лампорта, 286
маркерного кольца, 303
распределенный, 300
синхронизации часов, 280
усредняющий, 283
централизованный, 299
амнистия, 649
антиэнтропийная модель, 375
апплет, 708
аренда, 264, 373, 631
архитектура
 без разделения, 366
 двухзвенная, 75
 коммутируемая, 38
 трехзвенная, 76
 шинная, 38
асимметричная криптосистема, 472
асинхронная связь, 35, 128
асинхронный вызов RPC, 104
асинхронный режим передачи, 149
ассоциация защиты, 571
атака
 воспроизведения сообщений, 489
 на отражении, 480
 посредника, 481
атомарная групповая
 рассылка, 430, 434
атомарная операция, 49
атомарность, 92
атомарность транзакции, 309
аудит, 460
аутентификация, 460

Б

база данных
 накопления, 669
 размещения томов, 660
 репликации томов, 660
базовая точка, 247
банкомат, 528
барьер, 349
безопасная ошибка, 408
безопасность, 404
безотказность, 404
библиотека типов, 575
бинарный интерфейс, 542, 574
блок теневой, 313
блокировка, 318
 взаимная, 320
 двухфазная, 319
блокирующее подтверждение, 442
болтовня, 376
брандмауэр, 498
брокер
 объектных запросов, 541
 сообщений, 141

В

вектор версий Coda, 667
векторная отметка времени, 290
верификатор байтов кода, 502
версия файла, 679
вертикальное разбиение, 77
вертикальное распределение, 77
вертикальные средства CORBA, 542
взаимная блокировка, 320
взаимное исключение, 299, 391
 алгоритм маркерного
 кольца, 303, 304
 распределенный алгоритм, 300
 централизованный алгоритм, 299
взаимодействие, 81
взаимозаменяемые службы
 защиты, 570

взвешенный подсчет ссылок, 260
византийская ошибка, 408
виртуальная Java-машина, 502
виртуальная машина, 46
виртуальная синхронность, 433
виртуальная файловая система, 625
виртуально синхронная групповая рассылка, 432
виртуальный канал, 87
виртуальный образ, 727
виртуальный путь, 87
виртуальный узел, 660
вложенная транзакция, 310
вложенный поток данных, 150
внучатая сирота, 423
восприимчивый сервер, 375
восстановимая виртуальная память, 670
восстановимый объект, 564
восстановление, 449
временная избыточность, 409
временный адрес, 248
время
 глобальное по атомным часам, 278
 относительное, 277
 среднее по Гринвичу, 279
 универсальное согласованное, 279
входное имя, 519
выборочный показ, 501
выдвинутый кэш, 367
вызов
 по значению, 94
 по ссылке, 94
 через копирование/
 восстановление, 95

Г

генеративная связь, 754
гетерогенная система, 39
гиперкуб, 43
гиперссылка, 701
гипертекст, 701

глобальная объектная среда, 592
глобальная сеть, 23
глобальная служба имен, 224
глобально уникальное имя, 238
глобальное время по атомным часам, 278
глобальное имя, 218
глобальное состояние, 292
глобальный каталог, 587
глобальный уровень, 226
голосование, 384
гомогенная система, 39
горизонтальное распределение, 77
горизонтальные средства CORBA, 541
граница восстановления, 449
граф
 именования, 217
 направленный ациклический, 219
группа, 496
 доступных хранилищ тома, 666
 иерархическая, 412
 нарезки, 682
 объектов, 567
 одноранговая, 412
 очереди, 761
 процессов, 411
 управления объектами, 540
 хранилищ тома, 666
групповая рассылка, 140, 374
 атомарная, 430, 434
 виртуально синхронная, 432
 надежная, 424
 в порядке FIFO, 433
 неупорядоченная, 433
 причинно упорядоченная, 434
 ненадежная, 425
 полностью
 упорядоченная, 431, 434, 569

Д

дайджест сообщений, 476
двойная подпись, 534

двухзвенная архитектура, 75
двухфазная блокировка, 319
 первичная, 321
 распределенная, 321
 строгая, 320
 централизованная, 320
двухфазное подтверждение, 437
делегирование, 515, 645
демаршализинг, 108, 542
демон
 DCE, 110
 времени, 282
 контактов, 756
 транзакций, 764
дерево доменов, 587
дескриптор
 объекта, 597, 605
 привязки, 122
 реализации, 117, 606
 файла, 124, 627, 634
динамический распределенный
 объект, 120
динамический язык HTML, 703
динамическое обращение, 117
дискретная среда представления, 149
диспетчер, 173
диспетчеризация, 759
доверенная вычислительная база, 468
документ
 связующий, 738
 составной, 177
долговечность транзакции, 309
домен, 36, 233, 249, 586
доменное имя, 233
доменный сервер кэширования, 565
доставка
 с максимальными усилиями, 548
 сертифицированных
 сообщений, 767
доступность, 404

Ж

жесткая ссылка, 221
журнал
 регистрации, 767
 с упреждающей записью, 314, 670
 только для записи, 500

З

заглушка, 245
 клиентская, 96, 109
 серверная, 96
 удаления, 739
заголовок сообщения, 83
загрузчик классов, 502
задание сервера, 734
закрытый ключ, 472
заместитель, 113, 245, 516
 пользователя, 463
 ресурса, 463
заметка, 731
 администрирования, 732
 конструкции, 732
 определение, 732
 с данными, 732
запись
 за чтением, 362
 локализующая, 249
 монотонная, 360
 о ресурсе, 233
 реплицируемая, 382
запрос
 идемпотентный, 422
 односторонний, 548
 отложенный синхронный, 548
захват, 595
защита, 64
 авторизация, 460
 аудит, 460
 аутентификация, 460
 домена, 521
 механизм, 460

защита (*продолжение*)

- модификация, 460
 - перехват, 460
 - подделка, 460
 - правила, 460
 - прерывание, 460
 - транспортного уровня, 729
 - угроза, 460
 - управление, 506
 - шифрование, 460
- защищенная файловая система, 686
- защищенная электронная транзакция, 533
- защищенные данные, 349
- защищенный домен, 495
- защищенный канал, 478
- зеркало, 366, 727
- зона, 36, 226

И

- идемпотентная операция, 263
- идемпотентная процедура, 111
- идемпотентный запрос, 422
- идентификатор, 216
- адреса, 605
 - базы данных, 737
 - заметки, 737
 - инициатора, 737
 - интерфейса, 574
 - класса, 575
 - протокола, 606
 - реплики, 737
 - реплицируемого тома, 660
 - ресурса унифицированный, 606
 - службы, 778
 - тома, 660
 - транзакции, 648
 - универсальный, 737
 - файла, 660
 - хранения, 544
- иерархическая группа, 412
- иерархический подход, 249

- избыточность, 408
- временная, 409
 - информационная, 408
 - программная, 409
 - физическая, 409
- изменение представления, 431
- изолированность транзакции, 309
- изохронный поток данных, 407
- изохронный режим передачи, 149
- именование, 63, 214
- именованная сущность, 215
- именованный распределенный объект, 121
- имя
- входа, 759
 - входное, 519
 - глобально уникальное, 238
 - глобальное, 218
 - доменное, 233
 - каноническое, 234
 - локально независимое, 216
 - локальное, 218
 - относительно различимое, 238
 - очереди, 138
- пути
- абсолютное, 218
 - относительное, 218
 - удобное для восприятия, 217
- инвентаризация накопления, 670
- индивидуальная рассылка, 272
- инкрементный снимок состояния, 451
- Интернет-протокол обмена между ORB, 553
- интерфейс, 30, 112
- бинарный, 542, 574
 - динамического обращения, 543
 - очереди сообщений, 146
 - передачи сообщений, 134
 - транспортного уровня, 132
- интерфейсный агент, 206
- инфицированный сервер, 375
- информационная база каталога, 238
- управления защитой, 521

информационная избыточность, 408
информационная шина, 755
информационное дерево
каталогов, 238
информационный агент
мобильный, 206
стационарный, 206
искатель, 560
исключение, 419
исправление
обратное, 445
прямое, 445
истечение срока, 424
истребление сирот, 423
исходящая очередь, 138
итеративное разрешение имен, 229
итеративный сервер, 180

К

кадр, 85
канал
защищенный, 478
связи между агентами, 208
событий, 549
сообщений, 143
канальный уровень, 85
каноническое имя, 234
карта
активных объектов, 557
индексов, 682
менеджеров, 683
каскадные прерывания, 320
каталог
Domino, 736
глобальный, 587
объединенный, 678
качество обслуживания, 151
кворум
записи, 385
чтения, 385
класс, 113, 575
класс событий, 579
кластер, 735
кластер рабочих станций, 43, 366
клиент, 33, 58, 67, 175, 681
NFS, 626
доступа к аутентификации
и привилегиям, 522
многопоточный, 171
отсоединенный, 666
клиентская заглушка, 109
клиентский кэш, 369
клиентский протокол, 372
клонирование, 194
ключ
закрытый, 472
открытый, 472
сеансовый, 479, 489
кольцевой алгоритм, 298
коммутируемая архитектура, 38
коммутируемая мультимегабитная
служба данных, 467
коммутирующая решетка, 41
комплексный поток данных, 150
комплект протоколов, 85
компонент очереди, 580
конвейеризация, 710
конечная точка, 110, 180
контекст
защиты, 571
именования, 564
потока выполнения, 166
процессора, 166
контроллер домена, 586
контроль доступа, 493
контрольная сумма, 85
контрольная точка, 445
координированное создание, 451
независимое создание, 450
контрольный адрес, 248
конфиденциальная почта, 510
конфиденциальность, 459
конфликт
двойной записи, 318
чтения-записи, 318

конфликтующая операция, 318
кооперативное кэширование, 685, 726
кооперативный агент, 205
координатор распределенных транзакций, 589
координированное создание контрольных точек, 451
корневой дескриптор файлов, 635
корневой набор, 258
корневой узел, 218
кортеж, 754, 770
коэффициент кэш-попаданий, 40
криптографическая пломба, 522
криптосистема
 асимметричная, 472
 симметричная, 472
кусочно детерминированная модель, 452
кэш
 активный, 726
 выдвинутый, 367
 дублированных запросов, 647
 клиентский, 369
 обратной записи, 388
 попадание, 369
 промах, 372
 сквозной записи, 387
кэширование, 37
 кооперативное, 685
 указателей, 253
кэш-память, 40

Л

ленивая свободная
 непротиворечивость, 350
лес доменов, 587
линеаризуемость, 338, 339
листовой домен, 249
листовой узел, 217
логические часы, 285
логический том, 660
локализирующая запись, 249

локальная сеть, 22
локальное имя, 218
локальное представление, 593
локальное состояние, 292
локальный объект, 593
 нерезидентный, 604
 примитивный, 593
 составной, 594
 сохранный, 604
локальный псевдоним, 146

М

максимальная скорость дрейфа, 280
мандат, 494, 504, 513
мандат владельца, 514
маркер, 304
 аутентификации, 672
 свободный, 672
 секретный, 672
маркерное кольцо, 303
маршалинг, 108, 542
маршалинг параметров, 98
маршрутизатор, 139, 498
маршрутизация, 87
маршрутизирующий демон
 контактов, 757
масштабируемая надежная групповая рассылка, 427
матрица контроля доступа, 494
Международная организация по стандартам, 82
межоперационная ссылка
 на группу объектов, 567
 на объект, 561
межпроцессное взаимодействие, 103
межсетевой протокол, 87
менеджер
 данных, 315
 защиты, 503
 метаданных, 680
 окон, 176
 очередей, 139, 143

менеджер (*продолжение*)
 репликации, 568
 ресурсов, 45
 синхронизаций, 350
 транзакций, 316, 589
 управления службами, 576, 583
место, 559
метод, 112
 модификации, 608
 синхронизируемый, 123
 чтения, 608
механизм
 защиты, 460
 свертывания, 220
микроданные, 529
микроядро, 47
многопоточный клиент, 171
многопоточный сервер, 173
многоцелевые расширения почты
 Интернета, 705
мобильность
 сильная, 192
 слабая, 192
мобильный агент, 205
модель
 взаимодействия открытых
 систем, 82, 83
 доверия, 510
 загрузки-выгрузки, 625
 извлечения, 549
 компонентных объектов, 572
 кусочно детерминированная, 452
 непротиворечивости, 335
 обратного вызова, 550
 опроса, 551
 продвижения, 549
 распределенная COM, 572
 удаленного доступа, 625
модификация, 460
модуль расширения, 715
монитор, 584
монитор, 49
монитор ссылок, 494

монотонная запись, 360
монотонное чтение, 358
монтажная точка, 222
монтирование, 221, 635
мультикомпьютер, 38
мультикомпьютерная операционная
 система, 46
мультипроцессор, 38
мультипроцессорная операционная
 система, 46
мьютекс, 51
мягкая реинкарнация, 424

Н

надежная групповая рассылка, 424
надежная групповая рассылка
 в порядке FIFO, 433
надежная неупорядоченная групповая
 рассылка, 433
надежная причинно упорядоченная
 групповая рассылка, 434
надежная система, 404
надежность, 404, 459
название темы, 756
накопление файлов, 669
направленный ациклический
 граф, 219
направляющая таблица, 217
направляющий узел, 217, 564
нарезка, 682
независимое создание контрольных
 точек, 450
ненадежная групповая рассылка, 425
непрерывная среда представления, 148
неприсоединенный ресурс, 195
непротиворечивость, 37, 328, 329, 388
 FIFO, 344
 PRAM, 344
 записи за чтением, 362
 монотонного чтения, 358
 монотонной записи, 360
 ориентированная на данные, 335

непротиворечивость (*продолжение*)
ориентированная на клиента, 358
последовательная, 338
потенциальная, 357
поэлементная, 351
причинная, 343
свободная, 349
слабая, 346
строгая, 336
транзакции, 309
чтения собственных
записей, 361, 566
непрямая привязка, 563
нерезидентная связь, 128
нерезидентный локальный объект, 604
нерезидентный объект, 114
несохранное соединение, 710
неунифицированный доступ
к памяти, 42
неустойчивое сообщение, 435
неявная привязка, 115
низкоуровневый протокол, 85

О

обещание обратного вызова, 665
облегченный процесс, 169
обмен ключами по
Диффи—Хеллману, 507
обобщенная архитектура брокера
объектных запросов, 540
обобщенный интерфейс шлюзов, 706
обобщенный протокол обмена между
ORB, 553
оболочка, 556
обработчик, 717
обратное исправление, 445
обратный вызов
обещание, 665
отмена, 665
обратный удар, 426
обращение
динамическое, 117
статическое, 117

объединенный каталог, 678
объект, 493
Огса, 392
без фиксации состояния, 584
восстановимый, 564
класса, 575
локальный, 593
нерезидентный, 114
отмены, 578
пассивный, 600
поддержки, 612
правил, 570
распределенный, 62, 112
связующий, 578
сейфовый, 571
слушатель, 773
события, 760
сохранный, 114
транзакционный, 564
удаленный, 113
фабрика, 546
объектная модель документа, 703
оверлейная сеть, 139, 757
однократная запись с множественным
чтением, 677
однопроцессорная система, 23
одноразовая операция, 111
одноранговая группа, 412
одноранговое распределение, 77
односторонний вызов RPC, 106
односторонний запрос, 548
односторонняя функция, 473
однофазное подтверждение, 437
омега-сеть, 41
онтология, 209
операционная система
мультикомпьютерная, 46
мультипроцессорная, 46
распределенная, 45
сетевая, 45
операция
записи, 318, 397
захвата, 349

операция (*продолжение*)
идемпотентная, 263
конфликтующая, 318
освобождения, 349
чтения, 396
оптимистическое
протоколирование, 454
организация
регистрации правил
в Интернете, 510
сертификации атрибутов, 515
сертификации правил, 510
освобождение, 595
осиротевший процесс, 453, 454
остроконечный формат, 99
отзыв сертификата, 510
отказ
перемежающийся, 406
постоянный, 406
проходной, 405
откат транзакции, 315
открытая распределенная система, 30
открытый ключ, 472
отложенный вызов RPC, 105
отложенный синхронный запрос, 548
отмена обратного вызова, 665
отметка времени
векторная, 290
Лампорта, 286
относительно различимое имя, 238
относительное время, 277
относительное имя пути, 218
отсоединенный клиент, 666
очередь
исходящая, 138
назначения, 138
сообщений Microsoft, 580
транзакционная, 581
очищенный сервер, 375
ошибка
аварийной остановки, 408
безопасная, 408
византийская, 408

ошибка (*продолжение*)
значения, 407
отклика, 407
передачи состояния, 407
производительности, 407
произвольная, 408
синхронизации, 407
ошибочное разделение, 56

П

пакет, 87
память
кэш, 40
согласованная, 40
параллельный сервер, 180
паритет, 682
пассивный объект, 600
первичная двухфазная
блокировка, 321
передача сообщений, 51
перекрестный сертификат, 743
перемежающийся отказ, 406
перенос
зон, 235
кода
инициированный
отправителем, 192
инициированный
получателем, 193
клонирование, 194
процессов, 190
переносимость, 30
переносимый адаптер объектов, 556
перетаскивание, 177
перехват, 460
перехватчик, 555
защищенных обращений, 570
контроля доступа, 570
уровня
запросов, 555
сообщений, 555
период амнистии, 649

- персональная система связи, 249
- персональный идентификационный номер, 528
- пессимистическое
 - протоколирование, 454
- план, 317
- планировщик, 171, 315
- плоская транзакция, 309
- побег, 245
- побочный эффект, 656
- подавление откликов, 427
- подделка, 460
- подкачка, 166
- подключаемый модуль аутентификации, 783
- подобъект
 - защиты, 611
 - репликации, 596
 - связи, 595
 - семантики, 594
 - управления, 596
- подписание кода, 504
- подпись
 - двойная, 534
 - слепая, 532
- подслушивание, 490
- подсчет ссылок
 - взвешенный, 260
 - поколений, 262
 - простой, 259
- подтверждение
 - блокирующее, 442
 - двухфазное, 437
 - однофазное, 437
 - трехфазное, 443
- полигон, 503
- политика активизации, 185
- полностью упорядоченная групповая рассылка, 289, 431, 434, 569
- поломка, 407
- пользовательский агент, 686
- пользовательский режим, 47
- поочередная сериализация копий, 663
- порт, 110, 180
- порт сервера, 513
- последовательная
 - непротиворечивость, 338
- поставщик, 549
- постоянная реплика, 366
- постоянный отказ, 406
- потенциальная
 - непротиворечивость, 357
- потерянная секунда, 279
- поток
 - выполнения, 165, 658
 - данных, 149
 - вложенный, 150
 - изохронный, 407
 - комплексный, 150
 - простой, 150
- потребитель, 549
- почтовый агент, 206
- поэлементная
 - непротиворечивость, 351
- правило защиты, 460
- правильный идентификатор, 216
- право доступа, 493
- прагматическая общая групповая рассылка, 766
- представление группы, 431
- прерывание, 460
- привязка, 107, 562
 - непрямая, 563
 - неявная, 115
 - по значению, 194
 - по идентификатору, 194
 - по типу, 195
 - прямая, 563
 - явная, 115
- примитивный локальный объект, 593
- причинная непротиворечивость, 343
- проблема
 - атомарной групповой рассылки, 414, 430
 - византийских генералов, 415
 - двух армий, 414

- программная избыточность, 409
- программный агент, 205
- программный поток данных, 159
- прозрачность, 26
 - доступа, 27
 - местоположения, 27
 - отказов, 28
 - параллельного доступа, 28
 - переноса, 27
 - репликации, 28
 - смены местоположения, 27
 - сохранности, 29
- произвольная ошибка, 408
- прокси-шлюз, 499
- пропуск
 - данных, 407
 - передачи, 407
 - приема, 407
- простой подсчет ссылок, 259
- простой поток данных, 150
- простой текст, 470
- пространство имен, 217, 506
- протокол, 83
 - RPC для открытых сетевых вычислений, 629
 - аутентификации Нидхема-Шредера, 483
 - без установления соединения, 83
 - блокирующего подтверждения, 442
 - двухфазного подтверждения, 437
 - запрос-ответ, 479
 - защиты транспортного уровня, 729
 - защищенных электронных транзакций, 533
 - Интернета, 87
 - кворума, 385
 - клиентский, 372
 - локальной записи, 380
 - масштабируемой надежной групповой рассылки, 427
 - межсетевой, 87
 - на базе первичной копии, 378
 - непротиворечивости, 378
 - протокол (*продолжение*)
 - низкоуровневый, 85
 - о несостоятельности, 370
 - обмена ключами по Диффи-Хеллману, 507
 - однофазного подтверждения, 437
 - оптимистического протоколирования, 454
 - первичного архивирования, 379, 413
 - передачи
 - гипертекста, 91, 709
 - файлов, 91
 - пессимистического протоколирования, 454
 - прагматической общей групповой рассылки, 766
 - разрешения адресов, 244
 - резервирования ресурсов, 154
 - реплицируемой записи, 382
 - с установлением соединения, 83
 - серверный, 371
 - сетевого времени, 284
 - согласования кэша, 386
 - транспортный, 26
 - транспортный реального времени, 88
 - трехфазного подтверждения, 443
 - удаленной записи, 378
 - управления передачей, 88
 - упрощенный доступа
 - к каталогам, 240
 - эпидемический, 375
- протоколирование
 - оптимистическое, 454
 - пессимистическое, 454
 - сообщений, 452
 - отправителем, 446
 - получателем, 447
- профиль теговый, 561
- проходной отказ, 405
- процедура
 - разрешения имен, 229
 - составная, 630

процесс, 165
 клиентский, 175
 облегченный, 169
 осиротевший, 453, 454
 сервер, 181

процессор с массовым
 параллелизмом, 43
прямая привязка, 563
прямое исправление, 445
прямое согласование, 753
псевдоним, 146, 221
публикация, 756
путь, 679

Р

рабочий поток выполнения, 173
равновесие, 670
раздел, 662
разделение секрета, 491
различимое имя, 736
разрешение имени, 220
 итеративное, 229
 рекурсивное, 230
распределение, 36
 вертикальное, 77
 горизонтальное, 77
 запросов с известным
 содержимым, 719
 одноранговое, 77
распределенная двухфазная
 блокировка, 321
распределенная модель СОМ, 572
распределенная операционная
 система, 45
распределенная разделяемая
 память, 54
распределенная система, 23, 81, 388
 непротиворечивость, 328
 открытая, 30
 репликация, 328
 синхронизация, 274
распределенная транзакция, 63, 311

распределенная файловая
 система, 62, 623
распределенное подтверждение, 437
распределенный алгоритм, 300
распределенный документ, 62
распределенный объект, 62, 112
 динамический, 120
 именованный, 121
 разделяемый, 593
распределенный сборщик мусора, 257
распределенный снимок
 состояния, 293, 449
распространение слухов, 376
рассинхронизация часов, 277
рассылка
 групповая, 374
 индивидуальная, 272
 целевая, 374
 широковещательная, 374
расширенный самоанализ стека, 505
расширяемый язык
 разметки, 703
 стилей, 704
регион, 559
регистр
 счетчик, 276
 хранения, 276
редактирование по месту, 177
режим
 передачи
 асинхронный, 149
 изохронный, 149
 синхронный, 149
 пользовательский, 47
 работы запрос-ответ, 67
 ядра, 47
резервирование, 643
реинкарнация, 423
рекурсивное разрешение имени, 230
ремонтопригодность, 404
реплика
 инициируемая клиентом, 369
 инициируемая сервером, 367
 постоянная, 366

репликация, 37, 328, 388
 активная, 371, 382
 слабая причинно
 непротиворечивая, 394
реплицированный том, 660
реплицируемая запись, 382
ресурс
 неприсоединенный, 195
 связанный, 195
 фиксированный, 195
ретранслятор, 139
роль, 466, 497

С

самосертифицирующееся имя
 пути, 687
сборка мусора, 450
свертывание, 220
свидетельство о смерти, 377
свободная непротиворечивость, 349
 ленивая, 350
 обычная, 350
 энергичная, 351
свободный маркер, 672
связанный ресурс, 195
связность
 данных, 342
 памяти, 342
связующий документ, 738
связующий объект, 578
связывание и внедрение объектов, 573
связь
 асинхронная, 35, 128
 без установления соединения, 83
 генеративная, 754
 нерезидентная, 128
 с установлением соединения, 83
 синхронная, 33, 128
 сохранная, 127
сеансовый ключ, 479, 489
сегмент
 исполнения, 192
 кода, 192
 ресурсов, 192

сейфовый объект, 571
секвенсор, 382, 393
секрет, 491
секретный маркер, 672
секунда
 потерянная, 279
 солнечная, 277
 средняя солнечная, 278
семантика
 UNIX, 639
 максимум однажды, 420, 548, 578
 минимум однажды, 420
 сеансов, 640
 только однажды, 421
семафор, 49
сервер, 67, 181
 Domino, 731, 734
 атрибутов привилегий, 522
 аутентификации, 518, 521
 без фиксации состояния, 182, 631
 восприимчивый, 375
 времени, 281
 групп, 412
 защиты домена, 521
 инфицированный, 375
 итеративный, 180
 локализации, 116
 многопоточный, 173
 объектов, 183
 очищенный, 375
 параллельный, 180
 распространения ключей, 522
 с фиксацией состояния, 182, 631
 файловый, 58
 хранения, 680
серверный протокол, 371
серверный сценарий, 707
сервлет, 708
сериализуемость, 124
сериализуемость транзакции, 309
сертификат, 497
 атрибута, 515
 атрибутов привилегий, 522

сертификат (*продолжение*)

- отзыв, 510
- открытого ключа, 508
- перекрестный, 743
- срок жизни, 510

сертифицированное сообщение, 767

сертифицирующая организация, 508

сетевая операционная система, 45

сетевая файловая система, 223, 624

сетевой уровень, 87

сетевые новости, 290

сеть

- глобальная, 23
- доставки содержимого, 727
- локальная, 22
- оверлейная, 139, 757
- распределения содержимого, 727
- системная, 42

сильная мобильность, 192

сильная устойчивость

- к коллизиям, 473

символическая ссылка, 221

симметричная криптосистема, 472

синхронизация, 274

- взаимного исключения, 391
- условная, 391
- часов, 275, 280

синхронизированные часы, 284

синхронизируемый метод, 123

синхронная связь, 33, 128

синхронный режим передачи, 149

сирота, 423, 453

система

- Аноеба, 392
- Огса, 388
- агентов, 558
- гетерогенная, 39
- гомогенная, 39
- групповой работы, 26
- доменных имен, 233
- надежная, 404
- однопроцессорная, 23
- остановки без уведомления, 408

система (*продолжение*)

- очереди сообщений, 136
- проверки РАС, 523
- промежуточного уровня, 24, 45
- публикации/подписки, 579, 754
- распределенная, 23, 388
- с открытым ключом, 472
- управления контекстом защиты, 523
- централизованная, 23

системная сеть, 42

сито, 502

сквозная передача, 418

скелетон, 113, 245

слабая мобильность, 192

слабая непротиворечивость, 346

слабая причинно непротиворечивая
репликация, 394

слабая устойчивость к коллизиям, 473

слепая подпись, 532

слуга, 556

служба

- аутентификации и авторизации
Java, 783

блокировок, 565

внешних связей, 546, 599

времени, 547

жизненного цикла, 546, 599

запросов, 546

защиты, 107, 547

именования, 237, 547, 600

Интернета, 237

каталогов, 107, 237, 560

коллекций, 546

кэширования, 565

лицензирования, 546

локализации, 243, 597, 605

маршалинга, 599

обмена, 547

объектов Notes, 731

отношений, 547

параллельного доступа, 546, 564

поиска, 778

предоставления талонов, 518

различимых имен, 736

служба (*продолжение*)

- распределенного времени, 107
- распределенных файлов, 107
- свойств, 547
- событий, 546, 549
- сообщений, 550
- сохранности, 547, 600
- транзакций, 546, 564
- уведомлений, 546, 550

случайное число попсо, 484

слушатель событий, 759

снимок состояния, 293, 451

совместное резервирование, 643

согласование

- на встрече, 754
- прямое, 753
- через почтовый ящик, 753

согласованная память, 40

согласующее сообщение, 436

соединение

- несохранное, 710
- сохранное, 710

сокет, 132

солнечная секунда, 277

солнечный день, 277

солнечный переход, 277

сообщение

- активное, 681
- неустойчивое, 435
- о получении, 535
- сертифицированное, 767
- согласующее, 436
- устойчивое, 435, 453

составная процедура, 630

составной документ, 177, 572

составной локальный объект, 594

состояние, 112

- глобальное, 292
- локальное, 292
- только для чтения, 500

сохранная связь, 127

сохранное соединение, 710

сохранность, 63, 74

сохранный локальный объект, 604

сохранный объект, 114

спецификация

- передачи, 152
- синхронизации, 158

список

- контроля

 - доступа, 494, 744
 - исполнения, 744

- мандатов, 494
- отозванных сертификатов, 510
- ссылки, 263

спонсор пользователя, 522

способность к взаимодействию, 30

среда

- представления

 - дискретная, 149
 - непрерывная, 148

- распределенных вычислений, 106

среднее время по Гринвичу, 279

средняя солнечная секунда, 278

средство связи, 63

срез, 293

срок жизни, 510

ссылка

- жесткая, 221
- на заметки, 733
- символическая, 221

стандарт шифрования данных, 473

статическое обращение, 117

стек

- записей активизации, 204
- команд, 204
- переноса, 198
- протоколов, 85
- расширенный самоанализ, 505

страж, 389

стратегия

- обнаружения согласованности, 386
- установления согласованности, 387

строгая двухфазная

- блокировка, 320

строгая непротиворечивость, 336

строка
 запроса, 712
 инициализации, 606
 состояния, 713
субъект, 493
суперсервер, 180
сущность, 215
схема, 721
 m,n-пороговая, 493
 двойного извлечения, 739
 извлечения, 739
 извлечения-продвижения, 738
 продвижения, 739
счетчик, 276
счетчик ссылок, 258

Т

таблица
 запущенных объектов, 585
 направляющая, 217
 процессов, 165
таймер, 276
талон, 483, 519
теговый профиль, 561
текст
 простой, 470
 шифрованный, 470
теневого блок, 313
тик таймера, 276
тип MIME, 705
токен, 304
том, 659
 логический, 660
 реплицированный, 660
 физический, 660
точка
 базовая, 247
 доступа, 215
 контрольная, 445
 монтажная, 222
 монтирования, 222
транзакционная очередь, 581

транзакционный обмен
 сообщениями, 764
транзакционный объект, 564
транзакция
 автоматическая, 588
 атомарность, 309
 вложенная, 310
 долговечность, 309
 изолированность, 309
 непротиворечивость, 309
 откат, 315
 плоская, 309
 распределенная, 63, 311
 реализация, 312
 сериализуемость, 309
транспорт, 759
транспортный протокол, 26
транспортный протокол реального
 времени, 88
трассировка в группах, 266
трассировочная сборка мусора, 264
трехзвенная архитектура, 76
трехфазное подтверждение, 443
тройное модульное
 резервирование, 409
тупик, 320
тупоконечный формат, 99

У

угроза защите, 460
удаленная файловая служба, 625
удаленное обращение
 к методам, 117, 418
удаленный вызов процедур, 62, 94, 418
удаленный объект, 113
удостоверение, 742
узел
 виртуальный, 660
 графа именования, 564
 корневой, 218
 корневой направляющий, 249
 листовой, 217
 направляющий, 217, 564

узловой коммутатор, 41
универсальное согласованное время, 279
универсальный идентификатор, 737
универсальный протокол датаграмм, 88
унифицированное имя ресурса, 721
унифицированный идентификатор ресурса, 606, 717, 721
унифицированный указатель ресурса, 25, 700, 721
упакованный адрес, 605
упаковщик объектов, 185
управление защитой, 506 пространством имен, 506
управленческий уровень, 226
упрощенный вызов RPC, 103
упрощенный протокол доступа к каталогам, 240
уровень административный, 226 аутентификации, 590 глобальный, 226 защищенных сокетов, 468, 729 канальный, 85 обезличивания, 590 протокола записей TLS, 729 протоколов, 83 сетевой, 87 управленческий, 226 физический, 85
условная анонимность, 529
условная переменная, 50
условная синхронизация, 391
усредняющий алгоритм, 283
устойчивое сообщение, 435, 453
устойчивое хранилище, 447, 629
устойчивость к коллизиям сильная, 473 слабая, 473

Ф

фаза подъема, 319 спада, 319
файловая система Andrew, 654 виртуальная, 625 защищенная, 686 распределенная, 62 сетевая, 624 со структурой журнала, 682
файловая служба, 625
файловый моникер, 584
файловый сервер, 58
физическая избыточность, 409
физические часы, 276
физический том, 660
физический уровень, 85
фиксированный ресурс, 195
фильтрация, 151
формат MPEG, 158
фрагмент паритета, 682
фрейм вызова, 204
функция обратного вызова, 138 односторонняя, 473 хэш-функция, 472

Х

хэш-функция, 472
хранилище данных, 335 интерфейсов, 544 реализаций, 544 устойчивое, 447, 629

Ц

целевая группа инженерной поддержки Интернета, 723
целевая рассылка, 374

целостность, 459
центр распространения ключей, 482
централизованная двухфазная
 блокировка, 320
централизованная система, 23
централизованный алгоритм, 299
цепочка SSP, 245
цифровая подпись, 487
цифровые деньги, 528

Ч

часы
 логические, 285
 синхронизированные, 284
число поппе, 484
чтение
 монотонное, 358
 собственных записей, 361

Ш

шинная архитектура, 38
широковещательная рассылка, 374
шифрование, 460
шифрованный текст, 470

шлюз
 прикладного уровня, 499
 прокси-шлюз, 499
 фильтрации пакетов, 498

Э

экземпляр кортежа, 770
экспорт каталога, 632
элемент, 732
элемент службы, 778
энергичная свободная
 непротиворечивость, 351
эпидемический протокол, 375
эффект
 домино, 449
 побочный, 656

Я

явная привязка, 115
язык
 Огса, 389
 взаимодействия агентов, 208
 определения
 интерфейсов, 30, 102, 108, 542
 разметки гипертекста, 701

Эндрю Таненбаум, Маартен ван Стеен
Распределенные системы. Принципы и парадигмы

Перевел с английского В. Горбунков

Главный редактор	<i>Е. Строганова</i>
Заведующий редакцией	<i>И. Корнеев</i>
Руководитель проекта	<i>А. Васильев</i>
Научный редактор	<i>С. Орлов</i>
Литературный редактор	<i>А. Жданов</i>
Художник	<i>Н. Биржаков</i>
Иллюстрации	<i>М. Жданова</i>
Корректор	<i>В. Листова</i>
Верстка	<i>Л. Панич</i>

Лицензия ИД № 05784 от 07.09.01.

Подписано в печать 05.05.03. Формат 70×100/16. Усл. п. л. 70,95.

Тираж 5000 экз. Заказ № 2885.

ООО «Питер Принт». 196105, Санкт-Петербург, ул. Благодатная, д. 67в.

Налоговая льгота – общероссийский классификатор продукции

ОК 005-93, том 2; 953005 – литература учебная.

Отпечатано с готовых диапозитивов в ФГУП «Печатный двор» им. А. М. Горького
Министерства РФ по делам печати, телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.