В 3-м издании классического труда профессоров Амстердмского университета Эндрю Таненбаума и Мартена ван Стина обсуждаются принципы и парадигмы распределенных систем.

Кроме обширного теоретического материала в книге приведен код на языке Python (размещен на сайте dmkpress.com), демонстрирующий использование полученных знаний на практике.

В числе рассматриваемых тем:

- основные характеристики распределенных систем;
- архитектуры программных компонентов, входящих в систему;
- процессы и коммуникации;
- присваивание имен;
- согласованность и репликация;
- отказоустойчивость и безопасность.

Для опытных разработчиков, занимающихся распределенными системами, а также студентов профильных вузов.

Эндрю Стюарт Таненбаум – признанный эксперт в области компьютерных технологий, преподаватель Амстердамского свободного университета. Автор ряда книг, которые в настоящее время считаются классическими трудами в отрасли: «Компьютерные сети», «Современные операционные системы», «Архитектура компьютера» и др. Разработчик MINIX – микроядерной Unix-подобной операционной системы, используемой в учебных целях и наглядно иллюстрирующей положения, описываемые в его книгах.

Интернет-магазин:
www.dmkpress.com

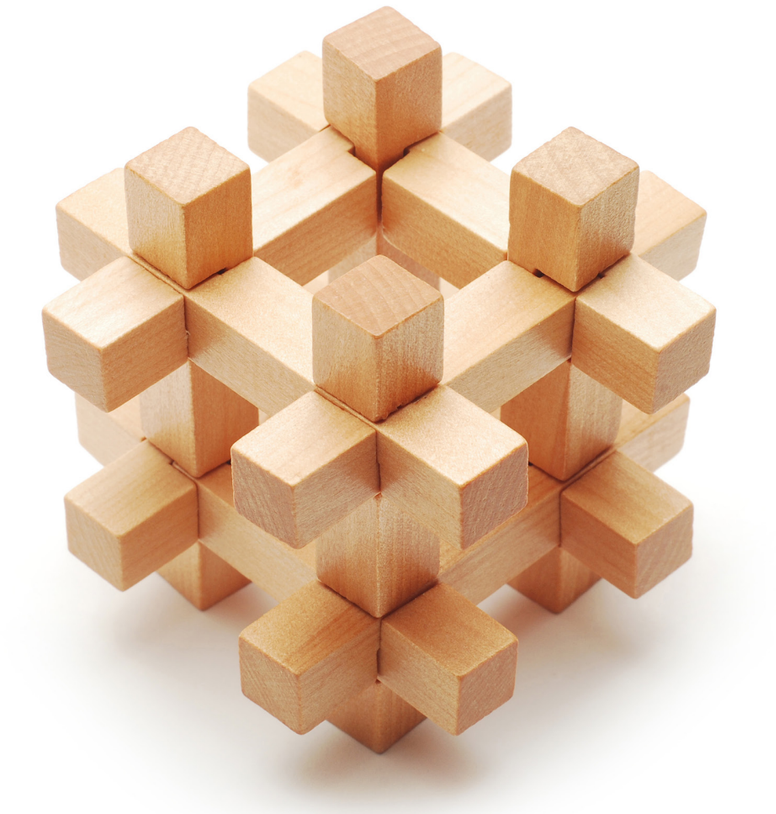
Оптовая продажа:
КТК «Галактика»
books@alians-kniga.ru

www.дмк.рф

ISBN 978-5-97060-708-4

Распределенные системы

Распределенные системы



Эндрю С. Таненбаум
Мартен ван Стин

Distributed Systems

Third edition

Maarten van Steen
Andrew S. Tanenbaum

Распределенные системы

Мартен ван Стин
Эндрю С. Таненбаум

ДМК Пресс
издательство
Москва, 2021

УДК 004.45
ББК 32.973
С80

Стин ван М., Таненбаум Э. С.
С80 Распределенные системы / пер. с англ. В. А. Яроцкого. – М.: ДМК Пресс, 2021. – 584 с.: ил.

ISBN 978-5-97060-708-4

В третьем издании классического труда професоров Амстердамского университета Эндрю Таненбаума и Мартена ван Стина обсуждаются принципы и парадигмы распределенных систем.

Кроме обширного теоретического материала в книге приведен код на языке Python (размещен на сайте dmkpress.com), демонстрирующий использование полученных знаний на практике.

В числе рассматриваемых тем: основные характеристики распределенных систем; архитектуры программных компонентов, входящих в систему; процессы и коммуникации; присваивание имен; согласованность и репликация; отказоустойчивость и безопасность.

Для опытных разработчиков, занимающихся распределенными системами, а также студентов профильных вузов.

УДК 004.45
ББК 32.973

ISBN 978-15-430573-8-6 (англ.)

ISBN 978-5-97060-708-4 (рус.)

Мариэлле, Максу и Элке.
– Мартен ван Стин

Сюзанне, Барбаре, Мэрвину, Арону, Натану, Оливии и Мирте.
– Эндрю С. Таненбаум

Содержание

Предисловие

Это третье издание книги «Распределенные системы». Во многих отношениях она сильно отличается от предыдущих выпусков, и, возможно, самое важное состоит в том, что мы полностью обобщили «принципы» и «парадигмы», включив последние в соответствующие главы, где обсуждаются принципы распределенных систем.

Материал был существенно переработан и дополнен, и в то же время мы были заинтересованы в ограничении общего объема книги. Поэтому он был сокращен более чем на 10 % по сравнению со вторым изданием, в основном за счет удаления материала по парадигмам. Для лучшего понимания материала книги широким кругом читателей мы перенесли конкретные материалы в отдельные выделенные разделы. Эти разделы могут быть пропущены при первом чтении.

Еще одним важным отличием является использование кодов примеров, написанных на языке программирования Python с поддержкой коммуникаций посредством пакета Redis. Примеры в книге опускают много деталей для удобства чтения, но полные примеры доступны на веб-сайте www.distributed-systems.net. Рядом с кодами для запуска, тестирования и расширений алгоритмов сайт предоставляет доступ к слайдам, всем рисункам и упражнениям.

Новый материал был проверен в учебном процессе, за что мы выражаем особую благодарность Тилю Кильманну (Thilo Kielmann) из университета Амстердама. Его конструктивные и критические замечания помогли нам значительно улучшить книгу.

Наш издатель Pearson Education любезно вернул нам авторские права, и мы должны сказать большое спасибо Трейси Джонсон (Tracy Johnson) за возможность осуществить этот плавный переход. Возврат авторских прав позволил нам начать то, что мы оба хотели сделать: запустить эксперимент. Он заключался в том, чтобы найти способ, обеспечивающий доступность материала, сделать его относительно недорогим и упростить процедуру обновления.

Книга теперь может быть (свободно) загружена, что делает намного более простым использование гиперссылок, где это уместно. В то же время предлагается и печатная версия, доступная через Amazon.com по минимальной цене.

Книга полностью оцифрована, что позволяет нам включать обновления, когда это необходимо. Мы планируем выпускать обновления ежегодно, оставляя доступными предыдущие цифровые версии, а также с некоторой периодичностью публиковать печатные версии книги. Часто выпускать обновления не всегда правильно с точки зрения перспектив обучения, но ежегодные обновления и поддержка предыдущих версий кажется нам хорошим компромиссом.

Мартен ван Стин
Эндрю С. Таненбаум

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Springer очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Глава 1

Введение[1]

Темпы изменения компьютерных систем были, есть и продолжают оставаться большими. С 1945 года, когда началась современная компьютерная эра, до 1985 года компьютеры были большими и дорогими. Более того, из-за ограниченных возможностей их соединения они работали независимо друг от друга.

Однако начиная с середины 1980-х годов два технологических достижения начали изменять ситуацию. Первым достижением была разработка мощных микропроцессоров. Первоначально это были 8-битные машины, но вскоре общедоступными стали 16-, 32- и 64-битные процессоры. С многоядерными процессорами мы сейчас переходим к решению проблемы адаптации и разработки программ для использования параллелизма. Цена компьютеров нынешнего поколения в тысячу раз меньше, чем цена мейнфреймов 30- или 40-летней давности, обладающих такой же вычислительной мощностью.

Вторым достижением было изобретение и разработка высокоскоростных компьютерных сетей. **Локальные сети** (Local-area networks, LAN) позволяют тысячам компьютеров в пределах одного здания быть связанными таким образом, что небольшие объемы информации могут быть переданы в течение нескольких микросекунд. Большие объемы данных могут перемещаться между машинами со скоростью миллиардов *бит в секунду* (бит/с, bps). **Глобальные сети** (Wide-area networks, WAN) позволяют сотням миллионов компьютеров по всему миру быть подключенными друг к другу и обмениваться информацией на скоростях от десятков тысяч до сотен миллионов бит/с.

Параллельно с развитием все более мощных и объединенных сетью машин мы стали свидетелями миниатюризации компьютерных систем, и, возможно, смартфон – здесь самый впечатляющий результат. Снабженные датчиками, большой памятью и мощным процессором, эти устройства стали полноценными компьютерами, и они, конечно, обладают и сетевыми возможностями. Находят свой путь к рынку и так называемые **подключаемые компьютеры** (plug computers). Эти небольшие компьютеры, часто размером с адаптер питания, могут быть подключены непосредственно ко входу устройства и выполнять роль настольного компьютера.

В результате появления этих технологий теперь несложно собрать компьютерную систему, состоящую из большого числа сетевых компьютеров, будь

---

[1] Версия данной главы была опубликована как «Краткое введение в распределенные системы». Computing, vol. 98 (10): 967–1009, 2016.

они большие или маленькие. Компьютеры такой системы, как правило, географически разбросаны, и потому обычно говорят, что они образуют **распределенную систему** (distributed system). Распределенная система может состоять как из нескольких, так и из миллионов компьютеров. Соединяющая их сеть может быть проводной, беспроводной или сочетанием того и другого. Кроме того, распределенные системы часто очень динамичны, в том смысле, что компьютеры могут присоединяться и отсоединяться, так что топология базовой сети и ее производительность почти непрерывно меняются.

В этой главе мы представим начальное исследование распределенных систем и целей их построения, а затем обсудим некоторые известные типы систем.

1.1. Что такое распределенная система?

В литературе приводятся различные определения распределенных систем, ни одно из них не является удовлетворительным и плохо согласуется с другими. Для наших целей достаточно дать такое достаточно широкое определение:

> *Распределенная система представляет собой совокупность автономных вычислительных элементов и является для его пользователей единой связанной системой.*

В этом определении отмечается две характерные особенности распределенных систем. Во-первых, распределенная система представляет собой совокупность вычислительных элементов, каждый из которых в состоянии работать независимо от других. Вычислительный элемент, который мы обычно будем называть узлом, может быть аппаратным устройством или программным процессом. Вторая особенность заключается в том, что пользователи (будь то люди или приложения) считают, что они имеют дело с единой системой. Это означает, что в той или иной степени автономные узлы должны сотрудничать. Существо такого сотрудничества лежит в основе разработки распределенных систем. Обратите внимание, что мы не делаем никаких предположений относительно типа узлов. В принципе, даже в пределах одной системы они могут варьироваться от высокопроизводительных мейнфреймов до небольших устройств в сенсорных сетях. Кроме того, мы не делаем никаких предположений относительно того, как узлы взаимосвязаны.

Характеристика 1: совокупность автономных вычислительных элементов

Современные распределенные системы могут и часто состоят из всех видов узлов, начиная от очень больших высокопроизводительных компьютеров и заканчивая маленькими компьютерами или даже еще меньшими устройствами. Основополагающий принцип заключается в том, что узлы могут

действовать независимо друг от друга, хотя очевидно, что если они игнорируют друг друга, то нет смысла помещать их в одну и ту же распределенную систему. На практике узлы запрограммированы для достижения общих целей, которые реализуются путем обмена сообщениями между ними. Узел реагирует на входящие сообщения, которые затем обрабатываются и, в свою очередь, ведут к общению посредством дальнейшей передачи сообщений.

Важным является то, что временна́я последовательность взаимодействия независимых узлов определяется каждым из узлов самостоятельно. Другими словами, не всегда можно предположить, что есть что-то вроде **глобальных часов** (global clock). Этот недостаток приводит к необходимости решения фундаментальных вопросов, касающихся синхронизации и координации внутри распределенной системы, которые мы обсудим подробно в главе 6. Тот факт, что мы имеем дело с *совокупностью* узлов, подразумевает, что нам может понадобиться организация и управление этой совокупностью. Другими словами, нам может понадобиться регистрация узлов, входящих и не входящих в систему, а также обеспечение всех членов системы списком узлов, с которыми они могут общаться напрямую.

Управление **членством в группе** (group membership) может быть чрезвычайно сложным, хотя бы по причине контроля допуска. Проще говоря, мы различаем открытые и закрытые группы. В **открытой группе** любой узел может присоединиться к распределенной системе, что означает, что она может отправлять сообщения любому другом узлу в системе. В **закрытой группе** общаться друг с другом могут только члены этой группы. Необходим также отдельный механизм, позволяющий узлу присоединиться к группе или покинуть ее.

Нетрудно видеть, что контроль допуска может быть сложным. Во-первых, необходим механизм аутентификации узла, и, как мы увидим в главе 9, если он неправильно разработан, управление аутентификацией может легко создать узкое место масштабируемости. Во-вторых, каждый узел должен, в принципе, проверить, является ли его общение действительно общением с другим членом группы, а не, например, со злоумышленником, стремящимся создать хаос. Наконец, учитывая, что член группы может легко начать общаться в системе с участниками, не являющимися членами его группы, то при общении внутри распределенной системы должна обеспечиваться конфиденциальность, иначе можно столкнуться с проблемами доверия.

Что касается организации совокупности узлов, практика показывает, что распределенная система часто организуется в виде **оверлейной сети** (overlay network) [Tarkoma, 2010]. В этом случае узел обычно представляет собой программный процесс, снабженный списком других процессов, которым он может напрямую отправлять сообщения. Может также оказаться, что соседу нужно будет начать общение первым. Передача сообщений осуществляется через TCP/IP- или UDP-каналы, но, как мы увидим в главе 4, могут быть доступны также средства более высокого уровня. Существует два типа оверлейных сетей.

1. **Структурное наложение:** в этом случае каждый узел имеет четко определенный набор соседей, с которыми он может общаться. Например, узлы организованы в виде дерева или логического кольца.

2. **Неструктурированное наложение:** в этом случае каждый узел имеет несколько ссылок на случайно выбранные другие узлы.

В любом случае, оверлейная сеть должна быть в принципе всегда **соединена**, то есть между любыми двумя узлами всегда должен существовать канал связи, позволяющий этим узлам отправлять сообщения друг другу. Хорошо известный класс оверлейных сетей формируется **одноранговыми** (peer-to-peer, P2P) сетями. Примеры оверлейных сетей будут подробно обсуждаться в главе 2 и последующих главах. Важно понимать, что организация узлов требует особых усилий и что это иногда одна из самых сложных задач управления распределенными системами.

Характеристика 2: единая связанная система

Как уже упоминалось, распределенная система должна выглядеть как единая связанная согласованная система. В некоторых случаях исследователи даже зашли так далеко, что говорят, что должно быть единое системное представление, означающее, что конечные пользователи не должны даже замечать, что они имеют дело с процессами, данными и контролем, рассредоточенными по всей компьютерной сети. Достижение единства системы часто требует слишком многого, и по этой причине мы выбрали более слабое определение распределенной системы как *кажущейся* пользователям связанной. Грубо говоря, распределенная система является связанной, если она ведет себя в соответствии с ожиданиями ее пользователей. Более конкретно, в единой связанной системе совокупность узлов в целом работает одинаково, независимо от того, где, когда и как происходит взаимодействие между пользователем и системой.

Часто достаточно сложно представлять распределенную систему как единую связанную систему. Например, это требует, чтобы конечный пользователь не мог точно сказать, на каком компьютере в настоящее время выполняется процесс, или, возможно, эта часть задачи была вызвана другим процессом, выполняющимся где-то еще. Место, где хранятся данные, также не должно быть проблемой; не должно иметь значения и то, что система может реплицировать данные для повышения производительности. Эта так называемая **прозрачность распределения** (distributing transparancy), о которой мы поговорим подробнее в разделе 1.2, является важной целью проектирования распределенных систем. В некотором смысле это сродни подходу, принятому во многих Unix-подобных операционных системах, в которых доступ к ресурсам осуществляется через единый интерфейс файловой системы, эффективно скрывающий различия между файлами, устройствами хранения и основной памятью.

Однако стремление к единой согласованной связанной системе является важным компромиссом. Поскольку мы не можем игнорировать тот факт, что распределенная система состоит из нескольких сетевых узлов, неизбежно, что в любое время какая-то часть системы может отказать. Это означает, что неожиданное поведение системы, при котором, например, некоторые приложения могут продолжать успешно выполняться, в то время как другие

полностью останавливаются, – это реальность, с которой приходится иметь дело. Хотя частичные отказы присущи любой сложной системе, в распределенных системах с ними особенно трудно справляться. Это позволило обладателю премии Тьюринга Лесли Лэмпорт (Leslie Lamport) описать распределенную систему как «[...] такую, в которой сбой какого-то компьютера, о существовании которого вы даже не подозревали, может сделать ваш компьютер бесполезным».

Промежуточное программное обеспечение и распределенные системы

Чтобы помочь разработке распределенных приложений, распределенные системы часто организованы так, чтобы иметь отдельный слой программного обеспечения, который логически размещен поверх соответствующих операционных систем компьютеров, являющихся частью системы. Эта организация показана на рис. 1.1 и известна как **промежуточное программное обеспечение** (middleware) [Bernstein, 1996].

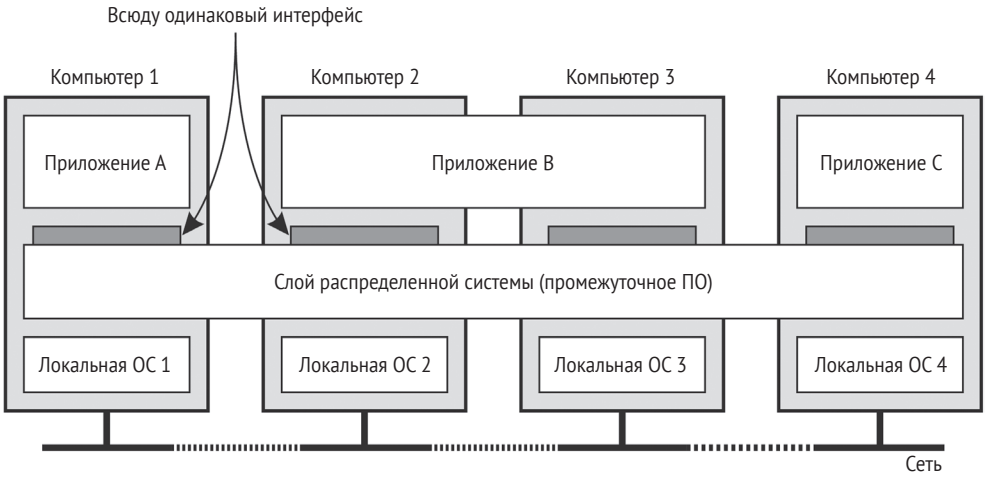


Рис. 1.1 ❖ Распределенная система, организованная на уровне промежуточного программного обеспечения, которая распространяется на несколько машин, предлагая каждому приложению один и тот же интерфейс

На рис. 1.1 показаны четыре сетевых компьютера и три приложения, причем приложение B распределено между компьютерами 2 и 3. Каждому приложению предлагается один и тот же интерфейс. Распределенная система предоставляет средства для связи компонентов одного распределенного приложения с каждым другим и позволяет различным приложениям общаться друг с другом. В то же время она скрывает, насколько это возможно и разумно, различия в оборудовании и операционных системах каждого приложения. В некотором смысле промежуточное ПО распределенной системы

подобно операционной системе компьютера – это менеджер ресурсов, предлагающий свои приложения для эффективного обмена и размещения этих ресурсов в сети. Помимо управления ресурсами, оно предлагает услуги, которые также можно найти в большинстве операционных систем, в том числе:

- средства для взаимодействия приложений;
- услуги безопасности;
- бухгалтерские услуги;
- маскировка и восстановление после сбоев.

Основное отличие от эквивалентов операционной системы заключается в том, что услуги промежуточного программного обеспечения предлагаются в сетевом окружении. Обратите внимание, что большинство услуг полезно для многих приложений. В этом смысле промежуточное программное обеспечение может также рассматриваться как контейнер часто используемых компонентов и функций, позволяющий реализовывать приложения отдельно. Чтобы проиллюстрировать это, давайте кратко рассмотрим несколько примеров типичных услуг промежуточного программного обеспечения.

- **Связь.** Общепринятой услугой связи является так называемый **удаленный вызов процедуры** (Remote Procedure Call, RPC). Сервис RPC, к которому мы вернемся в главе 4, позволяет приложению вызывать функцию, которая применяется и выполняется на удаленном компьютере, как если бы она была доступна локально. В конце разработчик должен просто указать заголовок функции, выраженный на специальном языке программирования, который позволяет подсистеме RPC генерировать затем необходимый код, который и устанавливает удаленные вызовы.
- **Транзакции.** Многие приложения используют несколько распределенных сервисов среди нескольких компьютеров. Промежуточное обеспечение обычно предлагает специальную поддержку для выполнения таких услуг, как «все или ничего», и обычно упоминается как атомарная **транзакция** (atomic transaction). В этом случае разработчик приложения должен только указать задействованные удаленные сервисы, и, следуя стандартизированному протоколу, промежуточное программное обеспечение гарантирует, что каждая служба вызывается или не вызывается вообще.
- **Состав услуг.** Становится все более распространенной разработка новых приложений путем выбора из существующих программ и склеивания их. Это особенно касается многих веб-приложений, в частности тех, которые известны как **веб-сервисы** [Alonso et al., 2004]. Промежуточное программное обеспечение на основе интернета может помочь стандартизировать способ доступа к веб-сервисам и обеспечить средства для генерации своих функций в определенном порядке. Простой пример того, как размещается состав службы, формируется с помощью гибридных веб-приложений **машапов** (mashup): веб-страницы, которые объединяют и группируют данные из разных источников. Известными машапами являются машапы на картах Google, которые дополнены такой информацией, как планировщики поездок или прогноз погоды в режиме реального времени.

- **Надежность.** В качестве последнего примера: было проведено множество исследований по расширенным функциям для создания надежных распределенных приложений. Инструментарий Horus [van Renesse et al., 1994] позволяет разработчику создавать приложение как группу процессов, такую что любое сообщение, отправленное одним процессом, гарантированно будет получено всеми или никакими другими процессами. Как оказалось, такие гарантии могут значительно упростить разработку распределенных приложений и, как правило, реализуются как часть промежуточного программного обеспечения.

Примечание 1.1 (историческая справка: термин «промежуточное программное обеспечение»)

Хотя термин «промежуточное программное обеспечение» стал популярным в середине 1990-х годов, он впервые, скорее всего, упоминается в сборнике докладов о разработке программного обеспечения конференции НАТО под редакцией Питера Наура (Peter Naur) и Брайана Рэнделла (Brian Randell) в октябре 1968 года [Peter Naur and Brian Randell, 1968]. В этих докладах промежуточное ПО действительно было размещено как раз между приложениями и подпрограммами обслуживания (эквивалент операционных систем).

1.2. Цели дизайна

Только то, что существует возможность создания распределенных систем, вовсе не обязательно означает, что это хорошая идея. В этом разделе мы обсудим четыре важные цели, которые должны иметь место, чтобы сделать построение распределенной системы стоящим усилий. Распределенная система должна сделать ресурсы легкодоступными; она должна скрывать тот факт, что ресурсы распределены по сети; она должна быть открытой; и она должна быть масштабируемой.

Поддержка совместного использования ресурсов

Важной целью распределенной системы является облегчение пользователям (и приложениям) доступа и совместного использования удаленных ресурсов. Ресурсы могут быть виртуально чем угодно, но типичные примеры включают в себя периферийные устройства, хранилища, данные, файлы, сервисы и сети, и это лишь некоторые из них. Есть много причин для желания поделиться ресурсами. Одна очевидная причина – это экономика. Например, дешевле иметь одно высококлассное надежное общее хранилище, чем покупать и поддерживать хранилище для каждого пользователя в отдельности.

Подключение пользователей и ресурсов также облегчает сотрудничество и обмен информацией, о чем свидетельствует успех интернета с его простыми протоколами для обмена файлами, почтой, документами, аудио и видео.

Подключение к интернету позволило географически широко рассредоточенным группам людей работать совместно с помощью всех видов **групповых программ** (groupware), то есть программного обеспечения для совместного редактирования, телеконференций и т. д., как показали многонациональные компании-разработчики программного обеспечения, которые передали многие работы по разработке кодов в Азию.

Однако совместное использование ресурсов в распределенных системах, возможно, лучше всего иллюстрируется успешным обменом файлами между одноранговыми сетями, такими как BitTorrent. Эти распределенные системы упрощают обмен файлами между пользователями интернета. Одноранговые сети часто связаны с распределением медиафайлов, таких как аудио и видео. В других случаях технология используется для распространения больших объемов данных, как в случае обновлений программного обеспечения, резервного копирования услуг и синхронизации данных на нескольких серверах.

Примечание 1.2 (дополнительная информация: общий доступ к папкам по всему миру)

Чтобы проиллюстрировать, где мы находимся в настоящее время, когда дело доходит до полной интеграции совместного использования ресурсов в сетевой среде, достаточно сказать, что сейчас развернуты веб-службы, которые позволяют группе пользователей размещать файлы в специальной общей папке, которая загружается и поддерживается третьей стороной где-то в интернете. Используя специальное программное обеспечение, общая папка почти не отличается от других папок на компьютере пользователя. По сути, эти службы заменяют использование общего каталога в локальной распределенной файловой системе, делая данные доступными для пользователей независимо от организации, которой они принадлежат, и вне зависимости от того, где они находятся. Эта услуга предлагается для разных операционных систем. Где именно хранятся данные, полностью скрыто от конечного пользователя.

Создание прозрачных распределений

Важной целью распределенной системы является скрытие того факта, что ее процессы и ресурсы физически распределены по нескольким компьютерам и, возможно, разделены большими расстояниями. Другими словами, она пытается сделать распределение процессов и ресурсов прозрачным, то есть невидимым для конечных пользователей и приложений.

Типы прозрачности распределений

Концепция прозрачности, то есть незаметности внутренней структуры для пользователя, может быть применена к нескольким аспектам распределенной системы, из которых наиболее важные перечислены на рис. 1.2. Мы используем термин «*объект*», имея в виду или процесс, или ресурс.

Прозрачность	Описание
Доступ	Скрыть различия в представлении данных и то, как получен доступ к объекту
Местоположение	Скрыть, где находится объект
Перемещение	Скрыть, что объект может быть перемещен в другое место, когда используется
Миграция	Скрыть, что объект может переместиться в другое место
Репликация	Скрыть, что объект тиражируется или дублируется
Параллельность	Скрыть, что объект может быть разделен между несколькими независимыми пользователями
Отказ	Скрыть сбой и восстановление объекта

Рис. 1.2 ❖ Различные формы прозрачности в распределенной системе (см. ISO[1995]). Объект может быть ресурсом или процессом

Прозрачность доступа (access transparency) связана с сокрытием различий в представлении данных и способа, который позволяет сделать объекты доступными. На базовом уровне мы хотим скрыть различия в архитектуре машин, но, что более важно, мы достигаем соглашения о том, как данные должны быть представлены различными машинами и операционными системами. Например, распределенная система может иметь в составе компьютеры, которые работают в разных операционных системах и каждая из которых имеет свои соглашения об именах файлов. Различия в соглашениях об именах, различия в файловых операциях или различия в том, какова низкоуровневая связь с другими процессами, являются примерами проблем доступа, которые желательно скрывать от пользователей и приложений.

Важной группой типов прозрачности являются местоположения процесса или ресурса. **Прозрачность местоположения** (location transparency) означает, что пользователи не могут сказать, где физически объект находится в системе. Наименование играет важную роль в достижении прозрачности местоположения. В частности, прозрачность местоположения часто может быть достигнута путем назначения ресурсам только логических имен, то есть имена, в которых местоположение ресурса не является тайным, не закодированы. Примером такого имени является **унифицированный указатель ресурса** (uniform resource locator, URL) http://www.prenhall.com/index.html, который не дает никакой информации о фактическом расположении основного веб-сервера Prentice Hall. URL также не дает подсказки о том, был ли файл index.html всегда в его текущем местоположении или недавно был перемещен туда. Например, весь сайт, возможно, был перемещен из одного центра данных в другой, но пользователи не должны этого замечать. **Прозрачность перемещения** (relocation transparency) становится все более важной в контексте облачных вычислений, к чему мы вернемся позже в этой главе.

Если прозрачность перемещения относится к перемещению самой распределенной системы, то **прозрачность миграции** (migration transparency) обеспечивается распределенной системой, когда она поддерживает мобильность процессов и ресурсов, инициируемых пользователями, без влияния на текущую связь и операции. Типичным примером этого является связь между

мобильными телефонами: независимо от того, как на самом деле движутся два человека, их мобильные телефоны позволят им продолжать разговор. Другие примеры, которые приходят на ум, включают онлайн-отслеживание местоположения и отслеживание товаров при их транспортировке из одного места в другое, а также телеконференции (частично) с использованием устройств, которые оснащены мобильным интернетом.

Как мы увидим, репликация (дублирование, тиражирование, replication) играет важную роль в распределенных системах. Например, ресурсы могут быть дублированы для повышения доступности или улучшения производительности путем размещения копии рядом с местом, где она доступна. **Прозрачность репликации** (replication transparency) связана с сокрытием того факта, что существует несколько копий ресурса или что несколько процессов работают в том или ином режиме перехвата, чтобы один мог работать, когда другой терпит неудачу. Чтобы скрыть репликацию от пользователей, необходимо, чтобы все реплики имели одинаковое имя. Следовательно, система, которая поддерживает прозрачность, как правило, должна поддерживать прозрачность местоположения, потому что иначе было бы невозможно обратиться к репликам в разных местах.

Мы уже упоминали, что важной целью распределенных систем является разрешение совместного использования ресурсов. Во многих случаях совместное использование ресурсов осуществляется путем кооперации, как и в случае каналов связи. Тем не менее также есть много примеров конкурентного совместного использования ресурсов. Например, каждый из двух независимых пользователей может хранить свои файлы на одном файловом сервере или иметь доступ к тем же таблицам в общей базе данных. В таких случаях важно, чтобы каждый пользователь не заметил, что другой использует тот же ресурс. Это явление называется **прозрачностью параллелизма** (concurrency transparency). Важной проблемой является то, что одновременный доступ к общему ресурсу оставляет ресурс в согласованном состоянии. Последовательность использования может быть достигнута с помощью блокировки механизма, который предоставляет пользователям эксклюзивный доступ к ресурсу по очереди. Более совершенный механизм заключается в использовании транзакций, но их, возможно, будет сложно реализовать в распределенной системе, особенно когда масштабируемость является проблемой.

Наконец, не менее важно, чтобы распределенная система обеспечивала **прозрачность отказов** (failure transparency). Это означает, что пользователь или приложение не замечает, что какая-то часть системы должным образом не работает, а система впоследствии (и автоматически) восстанавливается после этого сбоя. Маскировка отказов – это одна из самых сложных проблем в распределенных системах и даже невозможна, когда будут сделаны некоторые, по-видимому, реалистичные предположения, которые мы обсудим в главе 8. Основная трудность в маскировке и прозрачном восстановлении заключается в неспособности отличить полностью не работающий процесс и тот, который медленно, с трудом, но работает. Например, при контакте с перегруженным интернет-сервером браузер в конечном итоге берет таймаут и сообщает, что веб-страница недоступна. В этот момент пользователь

не может сказать, является сервер перегруженным на самом деле или сильно перегружена сеть.

Степень прозрачности распределения

Хотя прозрачность распределения обычно считается предпочтительной для любой распределенной системы, существуют ситуации, в которых пытаются слепо скрыть все аспекты распространения от пользователей, и это не очень хорошая идея. Простой пример – попросить вашу электронную газету появиться в вашем почтовом ящике до 7 утра по местному времени, в то время когда вы находитесь на другом конце света и в другом часовом поясе. Ваша утренняя газета не будет той утренней газетой, к которой вы привыкли.

Аналогично от глобальной распределенной системы, которая связывает процесс в Сан-Франциско, нельзя ожидать, что удастся скрыть факт связи с процессом в Амстердаме, что мать-природа не позволит ей отправить сообщение от одного процесса другому менее чем за 35 миллисекунд. Практика показывает, что это при использовании компьютерной сети передача на самом деле занимает несколько сотен миллисекунд. Передача сигнала ограничена не только скоростью света, но и ограниченной вычислительной мощностью и задержками в промежуточных переключениях. Существует также компромисс между высокой степенью прозрачности и производительностью системы. Например, многие приложения в интернете неоднократно попробуют связаться с сервером, прежде чем окончательно сдаться. Следовательно, пытаясь замаскировать временный сбой сервера, можно замедлить систему в целом. В таком случае, возможно, было бы лучше отказаться раньше или, по крайней мере, позволить пользователю отменить попытки установить контакт.

Другой пример – когда мы должны гарантировать, что несколько копий, расположенных на разных континентах, должны быть все время согласованными. Иными словами, если одна копия изменена, это изменение должно быть распространено на все копии до разрешения на любую другую операцию. Понятно, что одна только операция обновления может занять до нескольких секунд, что не может быть скрыто от пользователей.

Наконец, есть ситуации, в которых совершенно не очевидно, что скрытое распределение – это хорошая идея. Когда распределенные системы распространяются на устройства, которые люди носят с собой и где само понятие места в контексте осведомленности становится все более важным, может быть, лучше *раскрыть* распределение, а не пытаться скрыть его. Очевидный пример использования услуг на основе определения местоположения, которые часто можно найти на мобильных телефонах, – как найти ближайший китайский ресторан или проверить, далеко ли друзья.

Есть и другие аргументы против прозрачности распространения. Признаем также, что полная прозрачность распределения просто невозможна, и мы должны спросить самих себя, разумно ли делать вид, что мы можем этого достичь. Может быть, гораздо лучше сделать распределение явным, чтобы

пользователь и разработчик приложения никогда не обманывались, полагая, что есть такая вещь, как прозрачность. В результате пользователи будут намного лучше понимать (иногда неожиданно) поведение распределенной системы и поэтому лучше подготовятся к такому ее поведению.

Примечание 1.3 (обсуждение: против прозрачности распределения)

Некоторые исследователи утверждают, что скрытое распространение приведет только к дальнейшему усложнению разработки распределенных систем, именно по той причине, что полная прозрачность распределения никогда не может быть достигнута. Популярная техника для достижения прозрачности доступа заключается в расширении вызовов процедур для удаленных серверов. Однако Уолдо и др. [Waldo et al., 1997] уже показали, что попытка скрыть распространение с помощью процедур удаленных вызовов может привести к плохо понимаемой семантике, по той простой причине, что вызов процедуры *изменяется* при неисправности канала связи. В качестве альтернативы различные исследователи и практики в настоящее время выступают за меньшую прозрачность, например благодаря более явному использованию коммуникации в стиле сообщений или более явному размещению запросов и получению результатов от удаленной машины, как это делается в интернете при загрузке страниц. Такие решения будут подробно обсуждаться в следующей главе.

Несколько радикальной точки зрения придерживается Уэмс [Wams, 2011], утверждая, что частичные сбои не позволяют полагаться на успешное выполнение удаленного сервиса. Если такая надежность не может быть гарантирована, то всегда лучше положиться только на локальное исполнение, ведущее к принципу **копирования перед использованием** (copy-before-use). Согласно этому принципу, доступ к данным возможен только после их передачи на компьютер процесса, желающего получить эти данные. Кроме того, не должен быть измененен элемент данных. Вместо этого он может быть обновлен только до новой версии. Несложно представить, что появятся и другие проблемы. Тем не менее Уэмс показывает, что многие существующие приложения могут быть адаптированы к этому альтернативному подходу без ущерба для функциональности.

Вывод заключается в том, что стремление к прозрачности распределения может быть хорошей целью при разработке и реализации распределенных систем, но ее следует рассматривать совместно с другими вопросами, такими как производительность и понятность. Цена за достижение полной прозрачности может быть на удивление высокой.

Открытость

Другой важной целью распределенных систем является открытость. Открытая распределенная система, по сути, представляет собой систему, предлагающую компоненты, которые могут легко использоваться или интегрироваться в другие системы. В то же время сама открытая распределенная система часто состоит из компонентов, которые созданы в другом месте.

Функциональная совместимость, компоновка и расширяемость

Быть открытым означает, что компоненты должны придерживаться стандартных правил, которые описывают синтаксис и семантику того, что могут предложить эти компоненты (т. е. какую услугу они предоставляют). Общий подход заключается в определении услуг через интерфейсы, использующие **язык определения интерфейса** (Interface Definition Language, IDL). Определения интерфейса, записанные в IDL, почти всегда фиксируют только синтаксис сервисов. Другими словами, они точно определяют имена функций, которые доступны вместе с типами параметров, возвращаемыми значениями, возможными исключениями, которые можно применять, и т. д. Эта жесткая часть точно определяет, что эти сервисы делают, то есть семантику интерфейсов. На практике такие спецификации предоставляются в неформальной форме с помощью естественного языка.

При правильном указании определения интерфейса допускается произвольный процесс, который нуждается в определенном интерфейсе, чтобы обменяться с другим процессом, который обеспечивает этот интерфейс. Это также позволяет двум независимым сторонам строить совершенно разные реализации этих интерфейсов, что приводит к двум отдельным компонентам, которые действуют совершенно одинаково.

Правильные характеристики всегда полны и нейтральны. Полнота означает, что все, что необходимо для реализации, действительно указано. Однако многие определения интерфейсов не являются полными потому, что разработчику необходимо добавить детали реализации. Не менее важным является и то, что спецификации не предписывают, как реализация должна выглядеть; они должны быть нейтральными.

Как показывают Блэр и Стефани [Blair and Stefani, 1998], полнота и нейтральность важны для совместимости и мобильности. **Функциональная совместимость** (interoperability) характеризует степень, в которой две реализации систем или компонентов разных производителей могут сосуществовать и работать совместно, просто полагаясь на взаимные услуги в соответствии с общим стандартом. **Портативность** (portability) характеризует то, в какой степени приложение, разработанное для распределенной системы A, может работать без изменений в другой распределенной системе, в которой реализуются те же интерфейсы, что и в A.

Другой важной целью для открытой распределенной системы является то, что она должна легко конфигурировать систему из разных компонентов (возможно, и разных разработчиков). Кроме того, должна быть обеспечена простота добавления новых компонентов или замены существующих, не затрагивая остающихся компонентов. Иными словами, открытая распределенная система также должна быть **расширяемой** (extensible). Например, в расширяемую систему, видимо, относительно несложно добавить компоненты из другой операционной системы или даже заменить всю файловую систему.

Примечание 1.4 (обсуждение: открытые системы на практике)

Конечно, то, что мы только что описали, является идеальной ситуацией. Практика показывает, что многие распределенные системы не так открыты, как хотелось бы, и необходимо приложить еще много усилий и собрать разные кусочки, чтобы создать распределенную систему. Одним из решений вопроса открытости является раскрытие всех деталей компонента и предоставление разработчикам действительного исходного кода. Этот подход становится все более популярным и приводит к проектам с так называемым открытым исходным кодом, в которые многие вносят свой вклад по улучшению и отладке системы. По общему признанию, это наибольшая открытость, которую система может получить, но является ли это лучшим способом получить открытость, все еще под вопросом.

Отделение политики от механизма

Для достижения гибкости в открытых распределенных системах крайне важно, чтобы система была организована как набор относительно небольших и легко заменяемых или адаптируемых компонентов. Это подразумевает, что мы должны обеспечить определения не только интерфейсов самого высокого уровня, то есть тех, которые видят пользователи и приложения, но и определения интерфейсов для внутренних компонентов системы и описания, как эти компоненты взаимодействуют. Этот подход является относительно новым. Многие более старые и даже современные системы построены с использованием монолитного подхода, в котором компоненты только логически разделены, но реализованы как одна огромная программа. Такой подход затрудняет замену или адаптацию компонента, не оказывая влияния на всю систему. Таким образом, монолитные системы имеют тенденцию быть замкнутыми, а не открытыми.

Необходимость изменений в распределенной системе часто вызывается компонентом, который не обеспечивает оптимальную политику для конкретного пользователя или приложения. В качестве примера рассмотрим кеширование в веб-браузерах. Есть много разных параметров, которые необходимо учитывать.

- **Хранение:** где находятся данные для кеширования? Как правило, наряду с хранилищем на диске будет встроенная память – кеш. В этом случае должно быть рассмотрено ее точное положение в локальной файловой системе.
- **Исключение:** когда кеш заполняется, какие данные нужно удалить, чтобы вновь загруженные страницы могли быть сохранены?
- **Совместное использование:** каждый браузер использует отдельный кеш, или кеш делится между браузерами разных пользователей?
- **Обновление:** когда браузер проверяет актуальность кешированных данных? Кеши наиболее эффективны, когда браузер может возвращать страницы при отсутствии необходимости связи с оригинальным веб-сайтом. Это, однако, несет риск возвращения устаревших данных. Обратите внимание, что частота обновления сильно зависит от того, относительно каких данных фактически осуществляется кеширование:

в то время как расписание поездов вряд ли изменится, это совсем не так с веб-страницами, показывающими, скажем, погодные условия на трассе или, что еще хуже, цены на акции.

Необходимо разделение политики и механизма. В случае веб-кеширования, например, браузер в идеале должен обеспечить возможности только для хранения документов и в то же время позволять пользователям решать, какие документы хранятся и как долго. На практике это может быть реализовано обеспечением богатого набора параметров, которые пользователь может установить (динамически). Если сделать еще один шаг вперед, браузер даже может предложить возможности для подключения политики, которую пользователь реализовал как отдельный компонент.

Примечание 1.5 (обсуждение: действительно ли нам нужно строгое разделение?)

Теоретически следует строго отделять политику от механизма. Тем не менее есть важный компромисс, который необходимо учитывать: чем строже разделение, тем в большей степени нам нужно убедиться, что мы предлагаем соответствующий набор механизмов. На практике это означает, что предлагается богатый набор функций, что, в свою очередь, приводит ко многим параметрам конфигурации. Например, популярный браузер Firefox предлагает несколько сотен параметров конфигурации. Просто представьте, как пространство конфигурации буквально взрывается при рассмотрении больших распределенных систем, состоящих из многих компонентов. Другими словами, строгое разделение политики и механизмов может привести к очень сложным проблемам конфигурации.

Одним из вариантов решения этих проблем является предоставление разумных значений по умолчанию, и это то, что часто происходит на практике. Альтернативный подход заключается в том, что система наблюдает за своим использованием и динамически изменяет настройки параметров. Это приводит к системам, известным как самонастраиваемые системы. Тем не менее уже сам факт поддержки широкого ряда политик часто усложняет кодирование распределенных систем. Жесткое кодирование политик в распределенной системе может значительно снизить сложность, но ценой меньшей гибкости.

Поиск правильного баланса в отделении политики от механизмов является одной из причин, по которым проектирование распределенной системы часто является скорее искусством, чем наукой.

Масштабирование

Для многих из нас подключение через интернет столь же обычно, как возможность отправить открытку кому угодно в любую точку мира. Более того, там, где до недавнего времени для офисных приложений и хранения информации мы привыкли использовать настольные компьютеры, сейчас мы являемся свидетелями того, что подобные приложения и услуги размещаются в так называемом «облаке», а это, в свою очередь, приводит к увеличению количества гораздо меньших сетевых устройств, таких как планшетные компьютеры. Ввиду этого масштабируемость стала для разработчиков распределенных систем одной из наиболее важных целей проектирования.

Размерность масштабируемости

Масштабируемость системы может определяться как минимум по трем различным измерениям (см. [Neuman, 1994]):

- **масштабируемость размеров.** Система может быть масштабируема относительно ее размера, что означает, что мы можем легко добавить больше пользователей и ресурсов в систему без заметной потери ее производительности;
- **географическая масштабируемость.** Географически масштабируемая система – это система, в которой пользователи и ресурсы могут находиться далеко друг от друга, но тот факт, что задержки в линии связи могут быть значительными, не должен быть замечен системой;
- **административная масштабируемость.** Административно масштабируемая система – это система, которой все еще можно легко управлять, даже если она охватывает много независимых административных организаций.

Давайте подробнее рассмотрим каждое из этих трех измерений масштабируемости.

Масштабируемость размеров. Когда система нуждается в масштабировании, приходится решать очень разные типы проблем. Давайте сначала рассмотрим масштабирование относительно размера. Если необходимо поддерживать больше пользователей или ресурсов, мы часто сталкиваемся с ограничениями централизованных услуг, хотя и по очень разным причинам. Например, многие услуги централизованы в том смысле, что они реализуются с помощью одного сервера, работающего в распределенной системе на конкретной машине. В более современной обстановке мы можем иметь группу сотрудничающих серверов, расположенных в кластере тесно связанных машин, физически размещенных в одном и том же месте. Проблема с этой схемой очевидна: сервер или группа серверов может просто стать узким местом, когда необходимо обрабатывать все большее количество запросов. Чтобы проиллюстрировать, как это может случиться, предположим, что служба реализована на одной машине. В этом случае, по сути, существует три основные причины стать узким местом:

- вычислительная мощность, ограниченная процессорами;
- емкость памяти, включая скорость передачи ввода/вывода;
- сеть между пользователем и централизованным сервисом.

Давайте сначала рассмотрим вычислительную мощность. Просто представьте сервис для вычисления оптимальных маршрутов с учетом информации о движении в реальном времени. Такой сервис связан с вычислениями, требующими нескольких (десятков) секунд для выполнения запроса. Если есть и доступна только одна машина, то даже современная система высокого класса в конечном итоге столкнется с проблемами, если количество запросов возрастет выше определенного уровня.

Точно так же, но по разным причинам, мы столкнемся с проблемами, когда сервис в основном связан с вводом/выводом. Типичный пример – плохо спроектированный централизованный поисковик. Проблема с поисковыми запросами на основе контента – в том, что нам необходимо сопоставить за-

прос со всем набором данных. Даже с передовыми методами индексации мы все еще можем столкнуться с проблемой обработки огромного количества данных, превышающего емкость основной памяти машины, обслуживающей сервис. Как следствие большая часть времени обработки будет определяться относительно медленным доступом к диску и передачей данных между диском и основной памятью. Увеличение количества дисков или их скорости не даст устойчивого решения, если количество запросов продолжает расти.

Наконец, сеть между пользователем и службой сервиса также может быть причиной плохой масштабируемости. Просто представьте себе услугу «видео по запросу», которая должна выдавать потоковое видео высокого качества для нескольких пользователей. Видеопоток может легко потребовать полосу пропускания от 8 до 10 Мбит/с, что означает, что если служба устанавливает соединения со своими клиентами по принципу точка-точка, то это может скоро превзойти пределы пропускной способности сети ее собственных линий передачи.

Есть несколько решений проблемы размера масштабирования, которые мы обсудим далее, после рассмотрения географической и административной масштабируемостей.

Примечание 1.6 (дополнительно: анализ возможностей обслуживания)

Проблемы масштабируемости размера для централизованных сервисов могут быть формально проанализированы с использованием теории очередей и нескольких упрощающих предположений. На концептуальном уровне централизованный сервис может быть смоделирован как простая система очередей, показанная на рис. 1.3: запросы отправляются в службу, где они находятся в очереди до дальнейшего уведомления. Как только процесс может обработать следующий запрос, он извлекает его из очереди, выполняет обработку и выдает ответ. В объяснении работы централизованной службы мы в значительной степени следуем за [Menasce и Almeida, 2002].

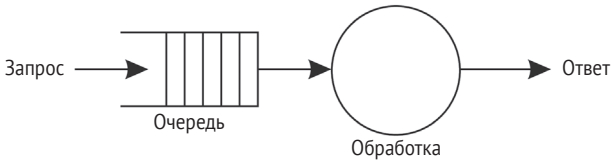


Рис. 1.3 ❖ Простая модель сервиса системы массового обслуживания

Во многих случаях мы можем предположить, что очередь имеет бесконечную емкость, что означает, что нет ограничений на количество запросов, которые могут быть приняты для дальнейшей обработки. Строго говоря, это означает, что скорость поступления запросов не зависит от того, что в данный момент находится в очереди или обрабатывается. Если предположить, что скорость поступления запросов равна λ запросов в секунду и что скорость обработки услуги составляет μ запросов в секунду, можно подсчитать временно́е отношение p_k, когда в системе находится k запросов:

$$p_k = \left(1 - \frac{\lambda}{\mu}\right)\left(\frac{\lambda}{\mu}\right)^k.$$

Если мы определим использование U службы как время, которое она занята, то очевидно, что

$$U = \sum_{k>0} p_k = 1 - p_0 = \frac{\lambda}{\mu} \Rightarrow p_k = (1-U)U^k.$$

Затем мы можем вычислить $\bar{N}$ – среднее количество запросов в системе:

$$\bar{N} = \sum_{k\geq 0} k \cdot p_k = \sum_{k\geq 0} k \cdot (1-U)U^k = (1-U)\sum_{k\geq 0} k \cdot U^k = \frac{(1-U)U}{(1-U)^2} = \frac{U}{1-U}.$$

Что нас действительно интересует, так это время отклика R, то есть сколько времени это займет до того, как сервис обработает запрос, включая время, проведенное в очереди. Для этого нам нужна средняя пропускная способность X. Учитывая, что сервис «занят», когда обрабатывается хотя бы один запрос, и что это происходит с пропускной способностью μ запросов в секунду и в течение доли U от общего времени, получаем:

$$X = \underbrace{U \cdot \mu}_{\text{сервер работает}} + \underbrace{(1-U)\cdot 0}_{\text{сервер не работает}} = \frac{\lambda}{\mu} \cdot \mu = \lambda.$$

Используя формулу Литтла [Trivedi, 2002], мы можем вывести время отклика как

$$R = \frac{\bar{N}}{X} = \frac{S}{1-U} \Rightarrow \frac{R}{S} = \frac{1}{1-U},$$

где $S = \frac{1}{\mu}$ – фактическое время обслуживания. Обратите внимание, что если U очень мало, то отношение времени ответ–обслуживание близко к 1, что означает, что запрос практически мгновенно обработан, и на максимально возможной скорости. Однако как только использование становится ближе к 1, мы видим, что отношение времени ответ–обслуживание быстро увеличивается до очень высоких значений, что фактически означает, что система приближается к остановке. Здесь мы видим проблемы масштабируемости. Из этой простой модели заключаем, что единственным решением является сокращение времени обслуживания S. Мы предоставляем читателю в качестве упражнения определить, как можно уменьшить S.

Географическая масштабируемость. Географическая масштабируемость имеет свои проблемы. Одной из главных причин, почему все еще трудно масштабировать существующие распределенные системы, которые были разработаны для локальных сетей, является то, что многие из них основаны на **синхронной связи** (synchronous communication). В этой форме общения сторона запрашиваемого сервиса, обычно называемая **клиентом**, блокирует до тех пор, пока ответ не отправлен обратно с сервера, реализующего сервис. Более конкретно, мы часто видим подобную картину связи многих клиент-серверных взаимодействий, как это случается с транзакциями базы данных. Этот подход обычно работает нормально в локальных сетях, где связь между двумя машинами часто бывает в худшем случае несколько сотен микросекунд. Однако в глобальной системе необходимо принять во внимание, что межпроцессорное взаимодействие может составлять сотни миллисекунд, то

есть на три порядка медленнее. Создание приложений с использованием синхронной связи в глобальных системах требует большой осторожности (а не просто некоторого терпения), особенно с расширенной моделью взаимодействия между клиентом и сервером.

Другой проблемой, препятствующей географической масштабируемости, является то, что связь в глобальных сетях по своей природе гораздо менее надежна, чем связь в локальных сетях. Кроме того, необходимо также иметь дело с ограниченной пропускной способностью. В результате решения, разработанные для локальных сетей, не всегда легко переносятся на глобальную систему. Типичным примером является потоковое видео. В домашней сети это довольно просто, даже когда есть только беспроводные соединения, обеспечивающие стабильный, быстрый поток качественных видеокадров с медиасервера на дисплей. Но поместите тот же сервер подальше и используйте стандартное TCP-соединение с дисплеем, и связь обязательно ухудшится: немедленно проявятся ограничения пропускной способности, будет затруднено и поддержание того же уровня надежности.

Еще одна проблема, которая возникает, когда компоненты находятся далеко друг от друга, – это тот факт, что глобальные системы, как правило, имеют очень ограниченные возможности для многоточечной связи. Напротив, локальные сети часто поддерживают эффективные механизмы вещания. Такие механизмы оказались очень полезными для обнаружения компонентов и услуг, что очень важно с точки зрения управления. В глобальных системах необходимо разрабатывать отдельные службы, такие как службы имен и каталогов, к которым могут обращаться направляемые запросы. Эти службы поддержки, в свою очередь, также должны быть масштабированы, и во многих случаях для этого не существует очевидных решений, о чем мы поговорим в следующих главах.

Административная масштабируемость. Наконец, сложный и во многих случаях открытый вопрос – как распределить распределенную систему по нескольким независимым административным доменам. Основная проблема, которая должна быть решена, – это конфликт политики в отношении использования (и оплаты) ресурсов, управления и безопасности.

Иллюстрацией этого может служить то, что ученые в течение многих лет искали решения для получения возможности поделиться своим (часто дорогим) оборудованием и создать так называемую **вычислительную сетку** (computational grid). В этих сетках глобальная распределенная система строится как федерация локальных распределенных систем, что позволяет программе работать на компьютере в организации A с прямым доступом к ресурсам в организации B.

Например, многие компоненты распределенной системы, которые находятся в каком-то одном домене, часто могут пользоваться доверием компонентов, работающих в этом же домене. В таких случаях системное администрирование должно протестировать и сертифицировать приложения и, возможно, принять специальные меры для предотвращения появления поддельных компонентов. По сути, пользователи доверяют свои системы администратору. Однако это доверие не распространяется естественно за пределы домена.

Примечание 1.7 (пример: современный радиотелескоп)

В качестве примера рассмотрим разработку современного радиотелескопа, такого как обсерватория Pierre Auger [Abraham et al., 2004]. Полная система может рассматриваться как федеративная распределенная система:

- непосредственно радиотелескоп можно определить как беспроводную распределенную систему, разработанную в виде сетки из нескольких тысяч сенсорных узлов, каждый из которых собирает радиосигналы и взаимодействует с соседними узлами для фильтрации соответствующих событий. Узлы динамически поддерживают дерево приемников, по которому выбранные события отправляются к центральной точке для дальнейшего анализа;
- центральная точка должна быть достаточно мощной системой, способной хранить и обрабатывать события, отправленные ей сенсорными узлами. Эта система обязательно размещается в непосредственной близости от узлов датчика, но, с другой стороны, считается действующей независимо. В зависимости от ее функциональности она может работать как небольшая локальная распределенная система. В частности, она хранит все записанные события и предлагает доступ к удаленным системам, принадлежащим партнерам консорциума;
- большинство партнеров имеют локальные распределенные системы (часто в форме кластера компьютеров), которые они используют для дальнейшей обработки данных, собранных телескопом. В этом случае локальные системы имеют прямой доступ к центральной точке телескопа с использованием стандартного протокола связи. Естественно, многие результаты, полученные в рамках консорциума, предоставляются каждому партнеру.

Таким образом, видно, что вся система пересекает границы нескольких административных доменов и что необходимы специальные меры для обеспечения того, чтобы доступ к данным получали только (конкретные) партнеры по консорциуму и не разглашались неуполномоченным лицам. Как в этих условиях добиться административной масштабируемости, не очевидно.

Если распределенная система расширяется до другого домена, должны быть приняты два типа мер безопасности. Во-первых, распределенная система должна защищаться от вредоносных атак нового домена. Например, пользователи из нового домена могут иметь доступ только для чтения в файловой системе своего конкретного домена. Аналогично такие объекты, как дорогие сеттеры изображений или высокопроизводительные компьютеры, могут быть недоступны для посторонних пользователей. Во-вторых, новый домен должен защищать себя от злонамеренных атак распределенной системы. Типичный пример – загрузка таких программ, как апплеты в веб-браузерах. В принципе, новый домен не знает, чего ожидать от такого чужого кода. Проблема, как мы увидим в главе 9, в том, каким образом осуществлять эти ограничения.

В качестве контрпримеров распределенных систем, охватывающих несколько административных доменов и которые явно не затрагивают проблемы административной масштабируемости, рассмотрим современные файлообменные одноранговые сети. В этих случаях конечные пользователи просто устанавливают программу, реализующую распределенный поиск, загружают функции и в течение нескольких минут могут начать загрузку файлов. Другие примеры включают одноранговые приложения для телефонии

через интернет, такие как Skype [Baset and Schulzrinne, 2006] и потоковая передача таких аудиоприложений, как Spotify [Kreitz and Niemelä, 2010]. Что общее у таких распределенных систем, так это то, что для того, чтобы поддерживать систему в рабочем состоянии, сотрудничают конечные пользователи, а не административные организации. В лучшем случае административные организации, такие как **интернет-провайдеры** (Internet Service Providers, ISPs), могут контролировать сетевой трафик, вызываемый этими одноранговыми системами, но пока подобные усилия не были очень эффективными.

Техника масштабирования

Обсудив некоторые проблемы масштабируемости, мы подходим к вопросу о том, как эти проблемы могут быть решены в целом. В большинстве случаев проблемы масштабируемости в распределенных системах появляются в виде проблем с производительностью, вызванных ограничениями емкости серверов и сети. Просто улучшением своих возможностей (например, путем увеличения памяти, обновления процессоров или замены сетевых модулей) часто находится решение, которое называется **расширением масштаба** (scaling up). Когда дело касается масштабирования, такое расширение распределенной системы путем развертывания большего количества компьютеров мы можем, в принципе, осуществить только тремя методами: сокрытием коммуникационной задержки, распределением работы и репликацией (см. также [Neuman, 1994]).

Сокрытие коммуникационных задержек. Сокрытие коммуникационных задержек применимо в случае географической масштабируемости. Основная идея проста: попробовать в максимально возможной степени избегать ожидания ответов на запросы удаленного обслуживания. Например, когда служба была запрошена на удаленном компьютере, альтернативой ожидания ответа от сервера является выполнение другой полезной работы на стороне запрашивающего. По сути, это означает создание запрашивающего приложения таким образом, что оно использует только **асинхронную связь** (asynchronous communication). Когда приходит ответ, приложение прерывается, и вызывается специальный обработчик завершения ранее оформленного запроса. Асинхронная связь часто может использоваться в системах пакетной обработки и параллельных приложениях, в которых независимые задачи могут быть запланированы для выполнения, в то время как данная задача ожидает завершения связи. В качестве альтернативы может быть начат процесс выполнения нового запроса. Хотя это блокирует ожидание ответа, другие потоки в процессе могут продолжаться.

Однако есть много приложений, которые не могут эффективно использовать асинхронную связь. Например, в интерактивных приложениях, когда пользователь отправляет запрос, он, как правило, не имеет ничего лучшего, чем ожидание ответа.

В таких случаях гораздо лучшим решением является уменьшение общения, например путем перемещения части вычислений, которые обычно выполняются на сервере по клиентскому процессу, запрашивающему сервис. Типичный случай, когда этот подход работает, – это доступ к базам данных

с использованием форм. Заполнение форм может быть сделано путем отправки отдельного сообщения для каждого поля и ожидания подтверждения от сервера, как показано на рис. 1.4а. Например, сервер может проверить наличие синтаксических ошибок, прежде чем принять запись. Гораздо лучшее решение – отправить код для заполнения формы и, возможно, проверки записей клиенту и заставить клиента вернуть заполненную форму, как показано на рис. 1.4б. Такой подход кода доставки широко поддерживается в интернете с помощью Java-апплетов и Javascript.

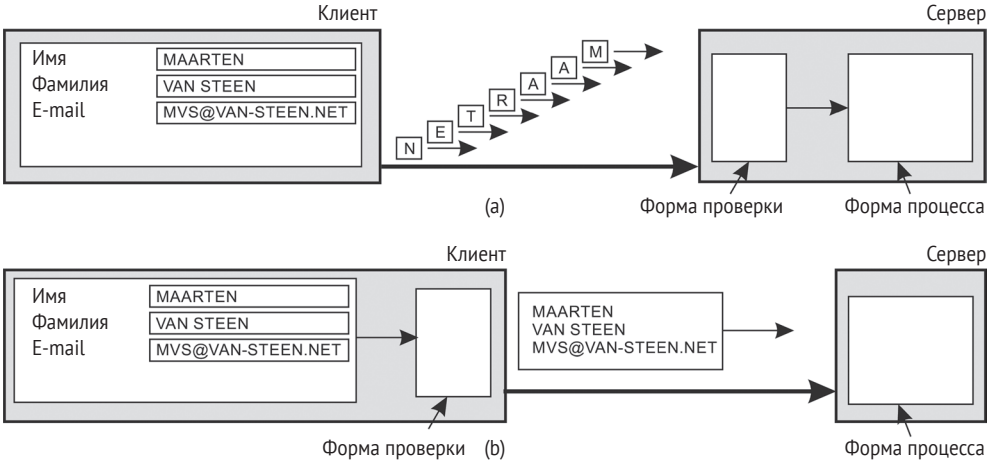


Рис. 1.4 ❖ Разница между разрешением (a) проверки сервера и (b) проверки клиента

Разбиение и распространение. Другой важный метод масштабирования – это **разбиение и распространение** (partitioning and distribution), которое включает в себя получение компонента, разделение его на более мелкие части, а затем распространение этих частей по всей системе. Хорошим примером разбиения и распространения является доменное имя в интернете (DNS). Пространство имен DNS иерархически организовано как дерево **доменов**, которые разделены на непересекающиеся **зоны**, как показано для оригинальной системы DNS на рис. 1.5. Имена в каждой зоне обрабатываются одним сервером имен. Не вдаваясь сейчас в подробности (подробно вернемся к DNS в главе 5), можно представить, что каждое имя пути является именем хоста в интернете и, следовательно, связано с сетевым адресом этого хоста. По сути, раскрытие имени означает возврат сетевого адреса связанному хосту. Рассмотрим, например, имя flits.cs.vu.nl. Чтобы раскрыть это имя, оно сначала передается на сервер зоны Z1 (см. рис. 1.5), который возвращает адрес сервера для зоны Z2, к которой относится остальное имя, flits.cs.vu, может быть переданным. Сервер для Z2 вернет адрес сервера для зоны Z3, который способен обрабатывать последнюю часть имени, и вернет адрес связанного хоста.

Этот пример иллюстрирует, как *сервис наименований* (naming service), обеспечиваемый DNS, распределен по нескольким машинам, избегая, та-

ким образом, того, чтобы один сервер имел дело со всеми запросами на разрешение имен.

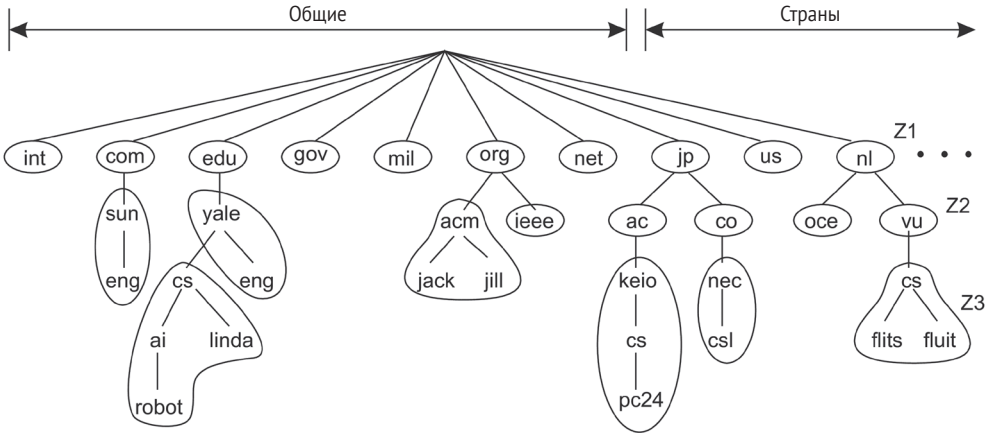


Рис. 1.5 ❖ Пример разделения (оригинального) пространства имен DNS на зоны

В качестве другого примера рассмотрим сеть интернета World Wide Web. Для большинства пользователей эта сеть представляется огромной информационной системой на основе документов, в которой каждый документ имеет свое уникальное имя в виде URL. Концептуально может даже показаться, что существует только один сервер. Тем не менее веб-сеть физически разделена и распределена по *нескольким сотням миллионов* серверов, каждый из которых обрабатывает несколько веб-документов. Имя сервера обработки документов закодировано в URL этого документа. Только благодаря такому распределению документов интернет смог быть масштабирован до существующего размера.

Репликация. Учитывая, что проблемы масштабируемости часто приводят к снижению производительности, как правило, хорошая идея – реплицировать (копировать, дублировать) компоненты в распределенной системе. Репликация не только увеличивает доступность, но и помогает сбалансировать нагрузку между компонентами, что позволяет повысить производительность. Кроме того, в широко географически рассредоточенных системах наличие поблизости копии может устранить большую часть проблем, связанных с задержкой связи, о чем упоминалось ранее.

Кеширование – это особая форма репликации, хотя найти различие между ними часто трудно или даже неестественно. Как и в случае репликации, кеширование (caching) приводит к созданию копии ресурса, как правило, в непосредственной близости от клиента, который обращается к этому ресурсу. Однако, в отличие от репликации, при кешировании решение принимается клиентом ресурса, а не его владельцем.

У кеширования и репликации существует один серьезный недостаток, который может влиять на масштабируемость. Поскольку существует несколько копий ресурса, изменение одной копии делает эту копию отличной от

других. Следовательно, кеширование и репликация приводят к проблемам согласованности. В какой степени такое несоответствие приемлемо, в значительной степени зависит от использования ресурса. Например, многие веб-пользователи считают приемлемым, если их браузер возвращает кешированный документ, когда его срок действия не был проверен в последние несколько минут. Вместе с тем существует также много вариантов, когда должны быть соблюдены строгие гарантии согласованности, например в случае электронных бирж и аукционов. Проблема со строгим соответствием заключается в том, что обновление должно быть немедленно распространено на все остальные копии. Более того, если два обновления происходят одновременно, часто также требуется, чтобы обновления везде обрабатывались в одном и том же порядке, приводя к дополнительной глобальной проблеме с заказом. Проблема усугубляется при необходимости сочетания соответствия с другими желательными свойствами, такими как доступность, и может стать просто нерешаемой, что мы обсудим в главе 8.

Поэтому для репликации часто требуется некоторый глобальный механизм синхронизации. К сожалению, такие механизмы чрезвычайно сложны, или их даже невозможно реализовать при масштабировании, хотя бы потому, что сетевые задержки имеют естественную нижнюю границу. Следовательно, масштабирование путем репликации может привести к другим, не относящимся к масштабируемости проблемам. Мы возвращаемся к репликации и согласованности в главе 7.

Обсуждение. При рассмотрении методов масштабирования можно отметить, что размеры масштабируемости наименее проблематичны с технической точки зрения. Во многих случаях увеличение мощности компьютера спасет положение, хотя, возможно, это и приведет к большим финансовым затратам. Географическая масштабируемость – гораздо более сложная проблема, поскольку сетевые задержки, естественно, имеют границы. Как следствие мы можем быть вынуждены копировать данные в места, близкие к клиентам, что, в свою очередь, приводит к проблемам поддержания копий согласованными. Практика показывает, что сочетание методов распространения, репликации и кеширования с различными формами согласованности обычно приводит к приемлемым решениям. Наконец, административная масштабируемость представляется наиболее сложной проблемой, отчасти потому, что нам нужно заниматься не техническими вопросами, а такими, как политика организаций и человеческое сотрудничество. Введено и в настоящее время широко распространено использование одноранговой технологии, которая успешно продемонстрировала, что может быть достигнуто, если конечные пользователи получают контроль [Lua et al., 2005; Орам, 2001]. Однако одноранговые сети, очевидно, не являются универсальным решением для всех проблем административной масштабируемости.

Ловушки

Теперь уже должно быть ясно, что разработка распределенной системы является трудной задачей. Как мы увидим в этой книге не однажды, вопро-

сов, которые следует одновременно учитывать, так много, что сложность их решения кажется закономерной. Вместе с тем, следуя ряду принципов проектирования, могут быть разработаны распределенные системы, в которых строго выполняются поставленные нами в этой главе цели.

Распределенные системы отличаются от традиционного программного обеспечения тем, что компоненты рассредоточены в сети. Не принимать во внимание при проектировании эту дисперсию – значит делать системы излишне сложными с недостатками, которые приходится потом исправлять. Питер Дойч (Peter Deutsch), работавший тогда в компании Sun Microsystems, перечислил следующие ложные предположения, которые делает каждый разрабатывающий впервые распределенное приложение:

- сеть надежна;
- сеть безопасна;
- сеть однородна;
- топология не меняется;
- задержка равна нулю;
- пропускная способность бесконечна;
- транспортные расходы равны нулю;
- есть один администратор.

Обратите внимание, как эти предположения относятся к свойствам, которые являются уникальными для распределенных систем: надежность, безопасность, неоднородность и топология сети; задержка и пропускная способность; транспортные расходы; и, наконец, административные домены. При разработке нераспределенных приложений многие из этих проблем, скорее всего, не появятся.

Большинство принципов, которые мы обсуждаем в этой книге, имеют непосредственное отношение к этим предположениям. Во всех случаях мы будем обсуждать решения проблем, которые вызваны тем, что одно или несколько предположений являются ложными. Например, надежные сети просто не существуют и приведут к невозможности достижения прозрачности отказа. Мы посвящаем целую главу тому, что сетевое общение по своей природе небезопасно. Мы уже обсуждали вопрос о том, что распределенные системы должны быть открытыми и учитывать гетерогенность. Аналогично при обсуждении репликации для решения проблем масштабируемости мы в основном решаем проблемы с временно́й задержкой и пропускной способностью. В этой книге в различных разделах мы также коснемся вопросов управления.

1.3. Типы распределенных систем

Прежде чем начать обсуждение принципов, посмотрим более внимательно на различные типы распределенных систем и разделим распределенные системы на информационные распределенные системы и распространенные (всеобъемлющие) системы (которые являются распределенными по своей природе).

Высокопроизводительные распределенные вычисления

Важным классом распределенных систем является класс систем, который используется для обеспечения высокой производительности вычислительных задач. Грубо можно различать две подгруппы. В **кластерных вычислениях** (cluster computing) базовое оборудование состоит из набора аналогичных рабочих станций или компьютеров, тесно связанных посредством высокоскоростной локальной сети. Кроме того, все узлы имеют одинаковые операционные системы. Ситуация становится совсем другой в случае **сетевых вычислений** (grid computing). Эта подгруппа состоит из распределенных систем, которые часто строятся как федерация компьютерных систем, где каждая система может подпадать под административный домен, и может сильно отличаться, когда речь идет об оборудовании, программном обеспечении и развернутой сетевой технологии.

С точки зрения сетевых вычислений, следующим логическим шагом будет просто *аутсорсинг* всей инфраструктуры, необходимой для приложений с интенсивными вычислениями. По сути, это и есть **облачные вычисления** (cloud computing): обеспечение средствами для динамичного построения инфраструктуры и объединения того, что необходимо для доступа к услугам. В отличие от сетевых вычислений, которые связаны с высокопроизводительными вычислениями, облачные вычисления гораздо больше, чем просто предоставление большого количества ресурсов. Мы кратко обсудим это здесь, но будем возвращаться к различным аспектам этого вопроса на протяжении всей книги.

Примечание 1.8 (дополнительная информация: параллельная обработка)

Высокопроизводительные вычисления в значительной мере обязаны появлению многопроцессорных машин. В них несколько процессоров организованы таким образом, что все они имеют доступ к одной и той же физической памяти, как показано на рис. 1.6a. Напротив, в мультикомпьютерной системе несколько компьютеров подключены через сеть и нет разделения основной памяти, как показано на рис. 1.6b. Модель общей памяти оказалась очень удобной для повышения производительности программ и относительно легко программировалась.

Суть в том, что одновременно выполняется несколько потоков управления и все потоки имеют доступ к общим данным. Доступ к этим данным контролируется через хорошо понятные механизмы синхронизации, такие как семафоры (для получения дополнительной информации о разработке параллельных программ см. [Ari, 2006] или [Herlihy and Shavit, 2008]). К сожалению, такая модель не столь легко масштабируется: до настоящего времени разрабатывались машины, в которых эффективный доступ к общей памяти имеют всего несколько десятков (иногда сотен) процессоров. Отметим, что в определенной степени те же ограничения характерны и для многоядерных процессоров.

Чтобы преодолеть ограничения систем с разделяемой памятью, высокопроизводительные вычисления стали осуществлять на системах с распределенной памятью. Этот сдвиг также означал, что многие программы должны были использовать передачу сообщений вместо изменения общих данных как средство общения

и синхронизации между потоками. К сожалению, модели передачи сообщений оказались гораздо сложнее и более подвержены ошибкам по сравнению с моделями программирования с общей памятью. По этой причине были проведены значительные исследования в попытке построить так называемую **распределенную общую память мультикомпьютеров** (distributed sharedmemory multicomputers, DSM) [Amza et al., 1996].

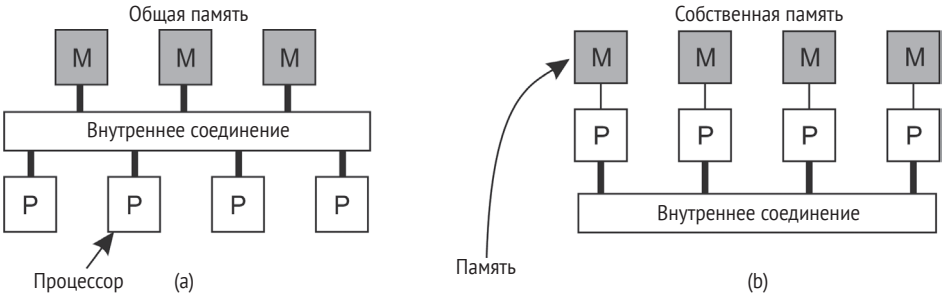


Рис. 1.6 ❖ Сравнение архитектур (a) мультипроцессора и (b) мультикомпьютера

По сути, система DSM позволяет процессору обращаться к области памяти на другом компьютере, как если бы это была локальная память. Это может быть достигнуто с помощью существующих методов, доступных операционной системе, например образованием из всех страниц основной памяти различных процессоров единого виртуального адресного пространства. Если процессор A обращается к странице, расположенной на другом процессоре B, страница с ошибкой, появляющаяся на A, позволяет операционной системе в точке A получать содержимое страницы B так же, как она обычно получает содержимое локально из диска, а процессор B будет проинформирован о том, что эта страница в настоящее время недоступна. От этой элегантной идеи имитации систем с разделяемой памятью и использованием мультикомпьютеров пришлось, в конце концов, отказаться по той простой причине, что производительность не соответствовала ожиданиям программистов, которые предпочли бы более сложные, но предсказуемо выполняемые модели программ передачи сообщений. Важным побочным эффектом изучения аппаратно-программных границ параллельной обработки стало глубокое понимание согласованности моделей, к которой мы вернемся в главе 7.

Кластерные вычисления

Кластерные вычислительные системы стали популярны, когда для персональных компьютеров и рабочих станций улучшилось соотношение цена/производительность. С некоторых пор стало и финансово, и технически привлекательным создание суперкомпьютера с использованием готовой технологии, просто соединив набор относительно простых компьютеров высокоскоростной сетью. Практически кластерные вычисления используются во всех случаях параллельного программирования, в котором одна (интенсивная вычислительная) программа выполняется параллельно на нескольких машинах.

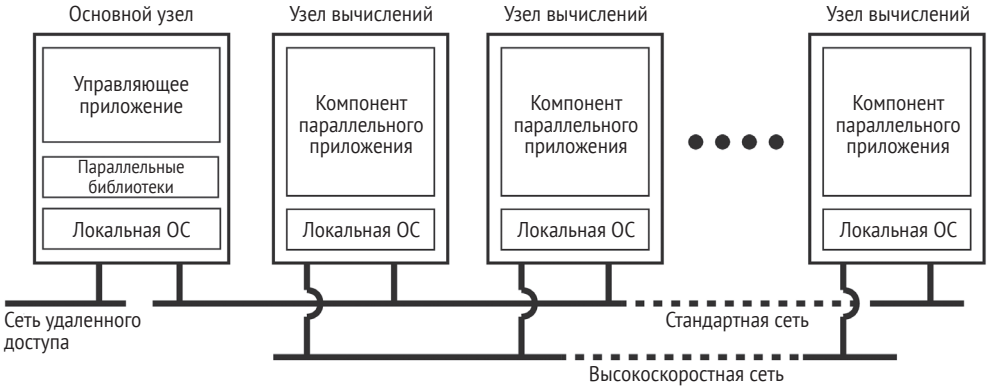


Рис. 1.7 ❖ Пример кластерной вычислительной системы

Примером широко применяемого кластерного компьютера является компьютер, созданный на основе ОС Linux с кластерами Беовульфа, общая конфигурация которых показана на рис. 1.7. Каждый кластер состоит из набора вычислительных узлов, которые управляются и доступны с помощью одного основного узла. Основной узел типично обрабатывает распределение узлов для конкретной параллельной программы, поддерживает пакетную очередь отправленных заданий и предоставляет интерфейс для пользователей системы. Таким образом, основной узел фактически запускает промежуточное программное обеспечение, необходимое для выполнения программ и управления кластером, а вычислительные узлы оснащены стандартной операционной системой с расширенной функцией промежуточного программного обеспечения для связи, хранения, отказоустойчивости и т. д. Таким образом, кроме основного узла, все вычислительные узлы идентичны.

Еще более симметричный подход используется в системе MOSIX [Amar et al., 2004]. Система MOSIX пытается предоставить **односистемный образ** (single-system image) кластера, что означает предоставление кластерным компьютером в высшей степени прозрачного распределения для обработки, представившись одним компьютером. Как мы уже упоминали, предоставление такого образа при любых обстоятельствах невозможно. В случае же MOSIX высокая степень прозрачности обеспечивается за счет разрешения процессов динамически и превентивной миграции между узлами, которые составляют кластер. Процесс миграции позволяет пользователю запускать приложение на любом узле (называемом домашним узлом), после которого он может прозрачно перейти к другим узлам, например с целью эффективного использования ресурсов. Мы вернемся к процессу миграции в главе 3. Аналогичные подходы с попыткой обеспечить односистемные образы сравниваются в [Lottiaux et al., 2005]. Однако в нескольких современных кластерных компьютерах отказались от таких симметричных архитектур в пользу более гибридных решений, в которых промежуточное программное обеспечение функционально разделено между различными узлами [Engelmannt al., 2007]. Преимущество такого разделения очевидно: наличие вычислительных узлов с выделенными, облегченными операционными системами обеспечит

более оптимальную производительность для приложений с интенсивными вычислениями. Точно так же, скорее всего, функциональность хранилища может быть оптимально обработана другими сконфигурированными узлами, такими как файловый сервер и сервер каталогов. То же самое относится и к иным специализированным сервисам промежуточного программного обеспечения, включая управление заданиями, услуги баз данных и, возможно, общий доступ в интернете к внешним услугам.

Сетевые вычисления

Характерной особенностью традиционных кластерных вычислений является их однородность. В большинстве случаев компьютеры в кластере в основном одинаковы, имеют одну и ту же операционную систему и все подключены к одной сети. Однако, как мы только что обсуждали, наметилась тенденция к архитектурам, в которых узлы специально настроены для определенных задач. Такое разнообразие в большой степени присуще сетевым вычислительным системам, в которых не делается никаких предположений относительно одинаковости аппаратных средств, операционных систем, сетей, административных доменов, политики безопасности и т. д.

Ключевой вопрос в сетевых вычислительных системах заключается в том, что собранные вместе ресурсы разных организаций и обеспечивающие сотрудничество группы людей из разных учреждений должны действительно образовывать федерацию систем. Такое сотрудничество осуществляется в форме **виртуальной организации** (virtual organization). Процессы, принадлежащие одной виртуальной организации, имеют право доступа к предоставляемым этой организации ресурсам, которые, как правило, состоят из вычислительных серверов (включая суперкомпьютеры, возможно, реализованные в виде кластера компьютеров), хранилищам и базам данных. Кроме того, в кластере также могут быть предоставлены и специальные сетевые устройства, такие как телескопы, датчики и т. д.

Учитывая существо сетевых вычислений, большая часть программного обеспечения для их реализации сводится к предоставлению доступа к ресурсам из разных административных доменов и только тем пользователям и приложениям, которые входят в конкретную виртуальную организацию. По этой причине основное внимание чаще всего уделяется архитектурным вопросам. Архитектура, первоначально предложенная в [Foster et al., 2001] и которая до сих пор является основой для многих вычислительных систем, показана на рис. 1.8.

Архитектура состоит из четырех слоев. Самый нижний *уровень сборки* обеспечивает интерфейсы к локальным ресурсам на конкретном сайте. Обратите внимание, что эти интерфейсы адаптированы для совместного использования ресурсов внутри виртуальной организации. Как правило, они будут предоставлять функции для запроса состояния и возможностей ресурса наряду с функциями для фактического управления ресурсами (например, блокировки ресурсов).

Уровень связи состоит из протоколов связи для поддержки сетевых транзакций, которые охватывают использование нескольких ресурсов. Например,

необходимы протоколы для обмена данными между ресурсами или просто для доступа ресурса из удаленного места. Кроме того, уровень связи будет содержать протоколы безопасности для аутентификации пользователей и ресурсов. Обратите внимание, что во многих случаях пользователи не проходят проверку подлинности. Вместо этого проходят аутентификацию программы, действующие от имени пользователей. В этом смысле делегирование прав от пользователя программе является важной функцией, которая должна поддерживаться на уровне связи. Мы вернемся к делегированию прав при обсуждении вопросов безопасности в распределенных системах в главе 9.

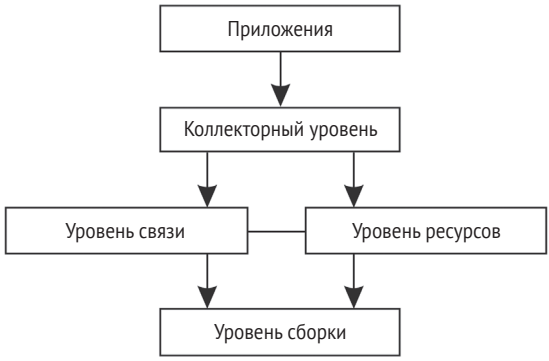


Рис. 1.8 ❖ Многоуровневая архитектура сетевых вычислительных систем

Уровень ресурсов отвечает за управление одним ресурсом. Он использует функции, предоставляемые уровнем связи, и вызывает напрямую интерфейсы, делая их доступными для уровня сборки. Например, этот уровень будет предлагать функции для получения информации о конфигурации конкретного ресурса или, вообще, выполнять определенные операции, такие как создание процесса или чтение данных. Уровень ресурсов, таким образом, считается ответственным за контроль доступа и, следовательно, будет полагаться на аутентификацию, выполняемую как часть уровня связи.

Следующий уровень в иерархии – это *коллекторный уровень*. Он управляет доступом к нескольким ресурсам и обычно состоит из сервисов для обнаружения ресурсов, распределения и планирования задач на нескольких ресурсах, данных тиражирования и т. д. В отличие от уровней связи и ресурсов, состоящих каждый из относительно небольшого, стандартного набора протоколов, коллективный уровень может состоять из множества различных протоколов, отражающих широкий спектр услуг, которые он может предложить виртуальной организации.

Наконец, *уровень приложений* состоит из приложений, которые работают в виртуальной организации и обеспечивают использование сетки вычислительной среды.

Обычно коллекторный уровень, уровень связи и уровень ресурсов формируют то, что можно назвать уровнем сетевого промежуточного программ-

ного обеспечения. Эти уровни совместно обеспечивают доступ к ресурсам, которые потенциально могут быть распределены по нескольким сайтам.

Важным наблюдением с точки зрения промежуточного программного обеспечения является то, что при сетевом вычислении распространенным является понятие сайта (или административной единицы). Эта распространенность подчеркивается постепенным сдвигом в сторону **сервис-ориентированной архитектуры** (service-oriented architecture), в которой сайты предлагают доступ к различным уровням через коллекцию веб-сервисов [Joseph et al., 2004]. Это к настоящему времени привело к определению альтернативной архитектуры, известной как архитектура **открытого сетевого сервиса** (Open Grid Services, OGSA) [Foster et al., 2006]. Архитектура OGSA основана на оригинальных идеях, сформулированных в [Foster et al., 2001], но прохождение ею процесса стандартизации делает ее довольно сложной. Пользователи OGSA обычно следуют стандартам веб-сервиса.

Облачные вычисления

Пока исследователи размышляли о том, как организовать вычислительные сети, которые были бы легкодоступны, организации, отвечающие за эксплуатацию центров обработки данных, столкнулись с проблемой открытия своих ресурсов для клиентов. В конце концов, это привело к концепции утилитарных вычислений, с помощью которых клиент может загружать задачи в центр обработки данных и платить за каждый ресурс. Утилиты вычисления составили основу того, что сейчас называется **облачными вычислениями** (cloud computing). Следуя [Vaquero et al., 2008], облачные вычисления характеризуются легко используемым и доступным пулом *виртуальных ресурсов*, которые, как и используемые ресурсы, могут быть настроены динамически, обеспечивая основу для масштабируемости: если требуется выполнить больше работы, клиент может просто приобрести больше ресурсов. Ссылка на утилиту вычислений формируется тем фактом, что облачные вычисления обычно основаны на модели оплаты за использование, в которой гарантии предлагаются посредством специальных *соглашений об уровне обслуживания* (*service level agreements*, SLA).

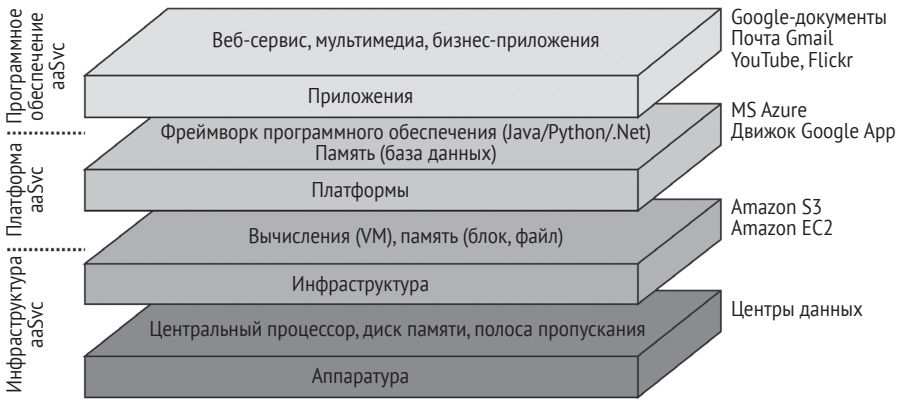


Рис. 1.9 ❖ Организация облаков (адаптировано из [Zhang et al., 2010])

На практике облака организованы в четыре слоя, как показано на рис. 1.9 (см. также [Zhang et al., 2010]):

- **аппаратное обеспечение.** Самый нижний уровень формируется средствами управления необходимым аппаратным обеспечением: процессорами, маршрутизаторами, а также системами питания и охлаждения. Обычно применяется в центрах обработки данных и содержит ресурсы, которые клиенты напрямую никогда не видят;
- **инфраструктура.** Это важный уровень, формирующий основу для большинства платформ облачных вычислений. Он использует методы виртуализации (обсуждается в разделе 3.2), обеспечивающие клиентов инфраструктурой, состоящей из виртуальной памяти и вычислительных ресурсов. На самом деле все это не то, чем кажется: облачные вычисления развиваются вокруг распределения и управления виртуальными устройствами хранения и виртуальными серверами;
- **платформа.** Можно утверждать, что уровень платформы обеспечивает клиенту облачных вычислений то, что операционная система обеспечивает приложению, а именно средства, облегчающие разработку и внедрение разработчиками приложений, которые нужно запустить в облаке. На практике разработчику прилагается специфичный для поставщика интерфейс (API), который включает в себя вызовы для загрузки и выполнения программы в облаке этого поставщика. В некотором смысле это сопоставимо с семейством системных вызовов Unix exec, которые принимают исполняемый файл как параметр и передают его операционной системе для выполнения.

  Как и операционные системы, уровень платформы обеспечивает более высокий уровень абстракции для хранения и других процедур. Например (мы будем обсуждать это более подробно позже), система хранения Amazon S3 [Murty, 2008] предлагается разработчику приложения в виде API, разрешающего (локально созданные) файлы организовать и хранить в пакетах. Пакет можно сравнить с директорией. Сохраненный в пакете файл автоматически загружается в облако Amazon;
- **приложение.** На этом уровне запускаются и предлагаются пользователям для дальнейшей настройки актуальные приложения. Хорошо известные примеры включают приложения, имеющиеся в офисных наборах (текстовые процессоры, электронные таблицы, приложения презентации и т. д.). Важно понимать, что эти приложения выполняются в облаке поставщика. Как и раньше, их можно сравнить с традиционным набором приложений, которые поставляются для установки вместе с операционной системой.

Поставщики облачных вычислений предлагают эти уровни своим клиентам через различные интерфейсы (включая инструменты командной строки, программные интерфейсы и веб-интерфейсы), что приводит к трем различным типам услуг:

- **инфраструктура как услуга** (Infrastructure-as-a-Service IaaS), охватывающая оборудование и уровень инфраструктуры;
- **платформа как услуга** (Platform-as-a-Service, PaaS), охватывающая уровень платформы;

- **программное обеспечение как услуга** (Software-as-a-Service, SaaS), в которое включаются приложения поставщика.

Использовать облака в настоящее время относительно легко, и мы обсудим более конкретные примеры интерфейсов для облачных провайдеров в последующих главах. Как следствие облачные вычисления, будучи средством для инфраструктуры аутсорсинга локальных вычислений, стали хорошим выбором для многих предприятий. Однако есть еще ряд серьезных препятствий, включая блокировку провайдера, безопасность, вопросы конфиденциальности, зависимость от доступности услуг и ряд других (см. также [Armbrust et al., 2010]). Кроме того, в связи с тем, что, как правило, детали того, как фактически выполняются конкретные облачные вычисления, скрыты и даже, возможно, неизвестны или непредсказуемы, выполнение требований по производительности оговорить заранее может стать невозможным. Кроме того, в [Li et al. 2010] показано, что разные провайдеры могут легко указывать очень разные профили производительности. Облачные вычисления больше не ажиотаж и, безусловно, являются серьезной альтернативой для поддержания огромной местной инфраструктуры, но еще многое может быть улучшено.

Примечание 1.9 (дополнительно: облачные вычисления дешевле?)

Одна из важных причин перехода в облачную среду заключается в том, что это намного дешевле по сравнению с обслуживанием локальной вычислительной инфраструктуры. Есть много способов подсчитать экономию, но, как оказалось, реальную оценку можно получить только для простых и очевидных случаев. В работе [Hajjat and et. al, 2010] предлагается более тщательный подход: переносить в облако только часть набора приложений, а другую часть оставлять работающей в локальной инфраструктуре. Суть этого метода заключается в обеспечении правильной модели набора корпоративных приложений.

Основу такого подхода составляет потенциально большой набор *программного обеспечения* компонентов. Предполагается, что каждое корпоративное приложение состоит из компонентов. Кроме того, считается, что каждый компонент C_i работает на N_i-сервере. Простым примером является компонент базы данных, который будет выполняться одним сервером. Более сложным примером является веб-приложение для расчета велосипедных маршрутов, состоящее из интерфейса веб-сервера для рендеринга HTML-страниц и принятия пользовательского ввода компонентов для вычисления кратчайших путей (возможно, при различных ограничениях) и компонентов базы данных, содержащих различные карты.

Каждое приложение моделируется как ориентированный граф, в котором вершина представляет компонент, а дуга $arc\langle i, \overrightarrow{j}\rangle$ – тот факт, что данные передаются от компоненты C_i к компоненте C_j. Каждая дуга имеет два связанных веса: $T_{i,j}$ представляет число транзакций за единицу времени, приводящих к потокам данных от C_i до C_j, а $S_{i,j}$ – среднее значение размера этих транзакций (то есть средний объем данных за транзакцию). Предполагается, что $T_{i,j}$ и $S_{i,j}$ известны и, как правило, получаются простым измерением.

Миграция набора приложений из локальной инфраструктуры в облако затем сводится к поиску оптимального плана миграции M: для каждого компонента C_i выясняется, сколько n_i его серверов N_i следует перенести в облако, так чтобы денежные расходы, получаемые от M, максимально уменьшались на дополнительную стоимость общения по интернету. План M также должен соответствовать следующим ограничениям.

Ограничения политики выполнены. Например, могут быть данные, которые юридически должны быть расположены в локальной инфраструктуре организации.

Поскольку связь в настоящее время частично осуществляется через интернет-каналы большой дальности, определенные транзакции между компонентами могут выполняться намного медленнее. План M может быть приемлем только в том случае, если какие-либо дополнительные задержки не нарушают конкретные ограничения.

Должны соблюдаться уравнения баланса потоков: транзакции продолжают работать правильно, и запросы или данные во время транзакции не теряются.

Экономический эффект. Для каждого плана миграции M можно ожидать экономии денежных средств, выражающейся как экономия (M), поскольку нужно поддерживать меньше машин или сетевых подключений. Во многих организациях такие затраты известны, так что экономия может быть относительно легко подсчитана. С другой стороны, существуют расходы на использование облака. В работе [Hajjat et al., 2010] сделано упрощающее различие между экономией B_c миграции компонента, интенсивно использующего вычислительные ресурсы, и экономией B_s от миграции компонента, интенсивно использующего память. Если обозначить компоненты через M_c с вычислительной интенсивностью и M_s – с интенсивным хранением, у нас есть *экономия*(M) = $B_c M_c + B_s \cdot M_s$. Очевидно, что могут быть развернуты и более сложные модели.

Расходы на интернет. Чтобы рассчитать увеличение затрат на связь, потому что компоненты распределены по облаку и по локальной инфраструктуре, нам нужно принимать во внимание пользовательские запросы. Чтобы упростить дело, мы не делаем различия между внутренними пользователями (то есть членами предприятия) и внешними пользователи (как можно видеть в случае веб-приложений). Трафик от пользователей до миграции может быть выражен как

$$Tr_{local,inet} = \sum_{C_i} (T_{user,i} S_{user,i} + T_{i,user} S_{i,user}),$$

где $T_{user,i}$ обозначает количество транзакций в единицу времени, ведущих к данным, исходящим от пользователей к C_i. У нас есть аналогичные интерпретации для $T_{i,user}$, S_{user_i} и $S_{i,user}$.

Для каждого компонента C_i пусть $C_{i,local}$ обозначает серверы, которые продолжают работать в локальной инфраструктуре, и $C_{i,cloud}$ – размещенные в облаке серверы. Заметим, что $|C_{i,cloud}| = n_i$. Для простоты предположим, что сервер из $C_{i,local}$ раздает трафик в тех же пропорциях, что и сервер от $C_{i,cloud}$. Мы заинтересованы в скорости транзакций между локальными серверами, облачными серверами и между локальными и облачными серверами после миграции. Пусть s_k будет сервером для компонента C_k, и обозначим через f_k дробь n_k/N_k. Тогда мы имеем для скорости транзакций $T^*_{i,j}$ после миграции:

$$T^*_{i,j} = \begin{cases} (1-f_i)\cdot(1-f_j)\cdot T_{i,j} & \text{когда } s_i \in C_{i,local} \text{ и } s_j \in C_{j,local} \\ (1-f_i)\cdot f_j \cdot T_{i,j} & \text{когда } s_i \in C_{i,local} \text{ и } s_j \in C_{j,local} \\ f_i \cdot (1-f_j)\cdot T_{i,j} & \text{когда } s_i \in C_{i,local} \text{ и } s_j \in C_{j,local} \\ f_i \cdot f_j \cdot T_{i,j} & \text{когда } s_i \in C_{i,local} \text{ и } s_j \in C_{j,local} \end{cases}.$$

$S_{i,j}$ – количество данных, связанных с $T^*_{i,j}$. Обратите внимание, что f_k обозначает долю сервера компоненты C_k, которая перемещается в облако. Другими словами, $(1 - f_k)$ – часть, которая остается в местной инфраструктуре. Мы оставляем читателю получение выражения для $T^*_{i,user}$.

Наконец, обозначим через $cost_{local,inet}$ и $cost_{cloud,inet}$ стоимость трафика через интернет в расчете на единицу для в и из локальной инфраструктуры и облака соответственно. Игнорируя несколько тонкостей, объясненных в [Hajjat et al., 2010], мы можем затем вычислить локальный интернет-трафик после миграции как:

$$Tr^*_{local,inet} = \sum_{C_{i,local}, C_{j,local}} (T^*_{i,j} S^*_{i,j} + T^*_{j,i} S^*_{j,i}) + \sum_{C_{j,local}} (T^*_{user,j} S^*_{user,j} + T^*_{j,user} S^*_{j,user}).$$

И таким же образом для трафика облака интернета после миграции:

$$Tr^*_{cloud,inet} = \sum_{C_{i,cloud}, C_{i,cloud}} (T^*_{i,j} S^*_{i,j} + T^*_{j,i} S^*_{j,i}) + \sum_{C_{j,cloud}} (T^*_{user,j} S^*_{user,j} + T^*_{j,user} S^*_{j,user}).$$

Объединив, мы получим модель для увеличения стоимости связи:

$$cost_{local,inet}(Tr^*_{local,inet} - Tr_{local,inet}) + cost_{cloud,inet} Tr^*_{cloud,inet}.$$

Очевидно, что для ответа на вопрос, дешевле ли переход в облако, нужно иметь много подробной информации и провести тщательное планирование того, что именно нужно для миграции. Статья [Hajjat et al., 2010] показывает первый шаг к принятию обоснованного решения. Их модель более подробна, чем мы готовы объяснить ее здесь. Важный аспект, который мы не затронули, – это то, что перенос компонентов требует также внимания к безопасности миграции компонентов. Заинтересованный читатель отсылается к их статье.

Распределенные информационные системы

Другой важный класс распределенных систем приходится на организации, которые сталкиваются с применением многих сетевых приложений и для которых совместимость оказывается болезненным опытом. Многие из существующих решений промежуточного программного обеспечения являются результатом работы с инфраструктурой, в которой проще интегрировать приложения в корпоративную информационную систему [Alonso et al., 2004; Bernstein, 1996; Hohpe and Woolf, 2004].

Мы можем выделить несколько уровней, на которых может происходить интеграция. Во многих случаях сетевое приложение просто состоит из сервера, на котором стоит приложение (и часто включает в себя базу данных) и который делает его доступным для удаленных программ, называемых клиентами. Такие клиенты отправляют запрос на сервер для выполнения конкретной операции, после которой ответ отправляется обратно. Интеграция на самом низком уровне позволяет клиентам упаковать несколько запросов, возможно для разных серверов, в один большой запрос и выполнить его как **распределенную транзакцию** (distributed transaction). Основная идея заключается в том, что выполняются или все, или ни один из запросов. По мере того как приложения становились все более изощренными и постепенно разделялись на независимые компоненты (особенно выделяя компоненты базы данных из компонентов обработки), стало ясно, что, обеспечивая общение напрямую, интеграция приложений также имеет право на существование. Эта привело в настоящее время к огромной отрасли, которая сосредоточена на **корпоративных приложениях** (Enterprise Application Integration, EAI).

Распределенная обработка транзакций

Чтобы конкретизировать наше обсуждение, мы сосредоточимся на приложениях базы данных. На практике операции с базой данных осуществляются

в форме **транзакций**. Программирование с использованием транзакций требует специальных примитивов, которые должны либо предоставляться базовой распределенной системой, либо языковой системой поддержки выполнения программы. Типичные примеры примитивов транзакций показаны на рис. 1.10. Точный список примитивов зависит от того, какие объекты используются в транзакции [Gray and Reuter, 1993; Bernstein and Newcomer, 2009]. В почтовой системе могут быть примитивы для отправки, получения и пересылки почты. В системе бухгалтерского учета они могут быть совершенно разными. Типичными примерами являются `READ` и `WRITE`. Внутри транзакции также допускаются обычные операторы, вызовы процедур и т. д. В частности, **удаленные вызовы процедур** (remote procedure calls, RPC), то есть вызовы процедур к удаленным серверам, также часто инкапсулируются в транзакции, что приводит к тому, что известно как **транзакционный RPC**. Мы обсуждаем **RPC** подробно в разделе 4.2.

Примитив	Описание
`BEGIN_TRANSACTION`	Отметить начало транзакции
`END_TRANSACTION`	Завершить транзакцию и восстановить старые значения
`ABORT_TRANSACTION`	Завершить транзакцию и попытаться зафиксировать
`READ`	Чтение данных из файла, таблицы или иным образом
`WRITE`	Записать данные в файл, таблицу или другое

Рис. 1.10 ❖ Пример примитивов для транзакций

Примитивы `BEGIN_TRANSACTION` и `END_TRANSACTION` используются для разграничения объема транзакции. Операции между ними образуют тело транзакции. Характерной особенностью транзакции является то, что или выполняются все эти операции, или не выполняется никакая. Ими могут быть системные вызовы, библиотечные процедуры или операторы в квадратных скобках на языке, в зависимости от применения.

Это свойство транзакций «все или ничего» является одним из четырех свойств, которые имеют транзакции. В частности, транзакции придерживаются так называемого свойства ACID (Atomic, Consistent, Isolated, Durable):

- **атомарная:** для внешнего мира транзакция осуществляется неделимой;
- **согласованная:** транзакция не нарушает системные инварианты;
- **изолированная:** параллельные транзакции не мешают друг другу;
- **продолжающаяся:** после совершения транзакции изменения являются постоянными.

В распределенных системах транзакции часто строятся как ряд субтранзакций, совместно формирующих **вложенную транзакцию** (nested transaction), как показано на рис. 1.11. Транзакция верхнего уровня может разветвиться на потомков, которые работают параллельно один с другим на разных машинах, для повышения производительности или упрощения программирования. Каждый из этих потомков может также выполнить одну или несколько субтранзакций или разветвлять своих детей.

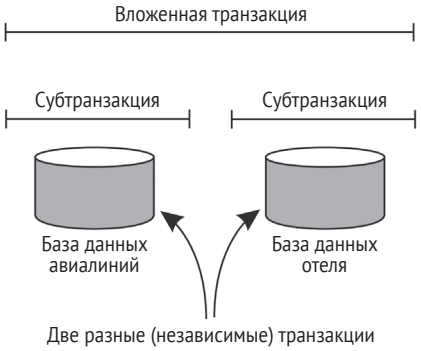


Рис. 1.11 ❖ Вложенная транзакция

Субтранзакции вызывают тонкую и важную проблему. Представьте, что транзакция запускает несколько субтранзакций параллельно, и одна из них осуществляется, делая результаты видимыми для родительской транзакции. После дальнейшего вычисления родитель отменяет работу, восстанавливая всю систему до ее состояния, которое было до начала транзакции верхнего уровня. Следовательно, результаты выполненной субтранзакции тем не менее должны быть отменены. Таким образом, постоянство, упомянутое выше, относится только к транзакциям верхнего уровня.

Поскольку транзакции могут быть вложены сколь угодно глубоко, администрация транзакции требует значительных усилий, нужно, чтобы все было сделано правильно. Однако семантика ясна. Когда любая транзакция или субтранзакция начинается, ей концептуально предоставляется личная копия всех данных во всей системе, чтобы она могла манипулировать ею по своему усмотрению. Если она прерывается, ее личная вселенная просто исчезает, как будто ее никогда не существовало. Если это происходит, ее личная вселенная заменяет родительскую вселенную. Таким образом, если субтранзакция осуществляется, а затем запускается новая субтранзакция, вторая видит результаты, полученные первой. Аналогично, если вложенная (более высокий уровень) транзакция прерывается, все ее субтранзакции должны быть также прерваны. Если несколько транзакций запускаются одновременно, результат таков, как будто они запускались последовательно в неустановленном порядке.

Вложенные транзакции важны в распределенных системах, поскольку они обеспечивают естественный способ распределения транзакций по нескольким машинам. Они следуют *логическому* разделению работы исходной транзакции. Например, транзакция для планирования поездки, по которой должно быть зарезервировано три разных рейса, может быть логически разделена на три субтранзакции. Каждая из этих субтранзакций может управляться отдельно и независимо от других двух.

В начале появления корпоративных систем промежуточного программного обеспечения компонент, который обрабатывал распределенные (или вложенные) транзакции, формировал ядро для интеграции приложения на уровне сервера или базы данных. Этот компонент был назван **монитор обработки транзакций** (transaction processing monitor, TP monitor), или для

краткости TP-монитор. Его главной задачей было разрешить приложению доступ к нескольким серверам/базам данных, предлагая ему модель транзакционного программирования, как показано на рис. 1.12. По сути, TP-монитор координирует осуществление субтранзакций в соответствии со стандартным протоколом, известным как **распределенная фиксация** (distributed commit), который мы обсудим в разделе 8.5.

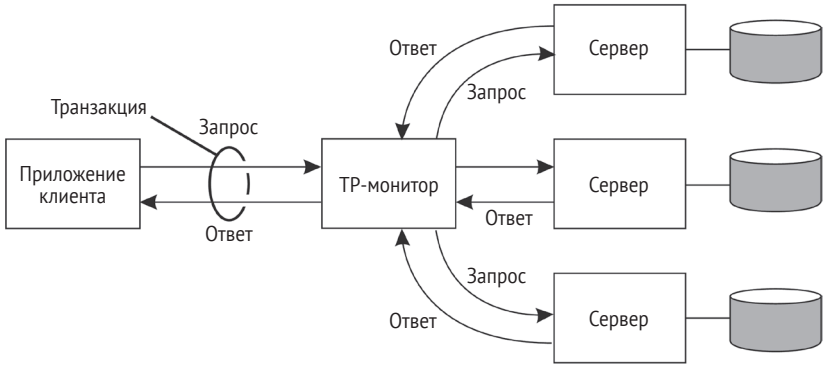


Рис. 1.12 ❖ Роль TP-монитора в распределенной системе

Важным обстоятельством является то, что приложения, желающие координировать несколько субтранзакций в единую транзакцию, не должны реализовывать эту координацию сами. Просто эту координацию для них делает TP-монитор. Именно здесь вступает в игру промежуточное ПО: оно реализует сервисы, которые полезны для многих приложений, и, таким образом, позволяет разработчикам приложений избегать повторения услуги снова и снова.

Интеграция корпоративных приложений

Как уже упоминалось, чем больше приложений отделяется от баз данных, в которых они были построены, тем становится все более очевидным, что необходимы средства для интеграции приложений независимо от их баз данных. В частности, приложения компонентов должны иметь возможность напрямую общаться друг с другом, и не просто с помощью средств реализации запроса/ответа системами обработки транзакций.

Эта потребность во взаимодействии приложений привела ко многим различным моделям коммуникации, основная идея которых заключалась в том, что существующие приложения могут напрямую обмениваться информацией, как показано на рис. 1.13.

Существует несколько типов связующего промежуточного программного обеспечения. Компонент приложения с **удаленной процедурой вызова** (remote procedure calls, RPC) может эффективно отправить запрос другому компоненту приложения, выполняя локальный вызов процедуры, что приводит к упаковке запроса как сообщения и отправке вызываемому абоненту.

Точно так же результат будет отправлен обратно и возвращен в приложение в результате вызова процедуры.

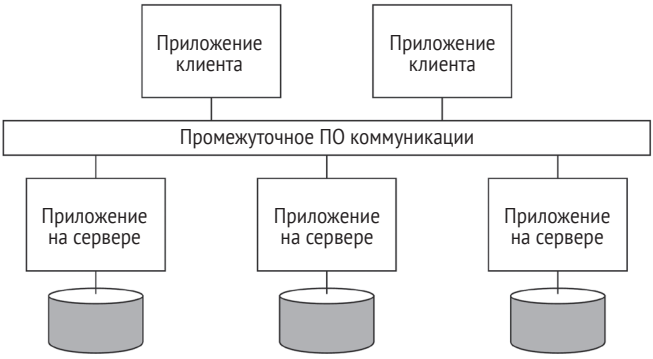


Рис. 1.13 ❖ Промежуточное программное обеспечение как средство коммуникации при объединении корпоративных приложений

Поскольку популярность объектной технологии возросла, были разработаны методы разрешения запросов к удаленным объектам, что привело к тому, что известно как **метод объединения удаленных вызовов** (remote method invocations, RMI). RMI – по сути такой же метод, как RPC, за исключением того, что он воздействует на объекты вместо функций. Методы RPC и RMI имеют тот недостаток, что как вызывающий, так и вызываемый абоненты во время общения должны быть в рабочем состоянии. Кроме того, им нужно точно знать, как обращаться друг к другу. Такое тесное соединение часто является серьезным недостатком и привело к так называемому **промежуточному программному обеспечению, ориентируемому на сообщения** (message-oriented middleware, MOM). В этом случае приложения отправляют сообщения к логическим точкам, часто описываемым средствами субъекта. Точно так же приложения могут указать свой интерес к определенному типу сообщения, после чего ПО коммуникации позаботится о том, чтобы эти сообщения были доставлены в эти приложения. Эти так называемые **системы публикации/подписки** (publish/subscribe systems) формируют важный и расширяющийся класс распределенных систем.

Примечание 1.10 (дополнительная информация: об интеграции приложений)

Поддержка интеграции корпоративных приложений является важной целью для многих промежуточных программ. В общем, есть четыре способа интеграции приложений [Hohpe и Вульф, 2004].

Передача файлов. Суть интеграции с помощью передачи файлов заключается в том, что приложение создает файл, содержащий общие данные, которые впоследствии читаются другими приложениями. Подход технически очень прост, что делает его привлекательным. Недостатком, однако, является то, что есть многое, что должно быть согласовано:

- формат и размещение файла: текст, двоичный файл, его структура и т. д. В настоящее время XML стал популярным, так как его файлы, в принципе, описывают себя сами;

- управление файлами: где они хранятся, как их называют, кто ответствен за удаление файлов;
- обновление распространения: когда приложение создает файл, может быть несколько приложений, которые должны прочитать этот файл, чтобы обеспечить единство согласованной системы, говорилось в разделе 1.1. Как следствие иногда необходимо реализовать отдельные программы, которые уведомляют приложения об обновлениях файлов.

Общая база данных. Многие проблемы, связанные с интеграцией через файлы, облегчаются при использовании общей базы данных. Все приложения будут иметь доступ к тем же данным, и часто с помощью языка высокого уровня, такого как SQL. Кроме того, приложения легко уведомлять о возникновении изменений, поскольку переключатели являются частью современных баз данных. Однако есть два основных недостатка. Во-первых, все еще существует необходимость в разработке общей схемы данных, что может оказаться далеко не тривиальным, если набор приложений, которые необходимо интегрировать, не полностью известен заранее. Во-вторых, когда есть много чтений и обновлений, общая база данных может легко стать узким местом в части производительности.

Удаленный вызов процедуры. Интеграция через файлы или базу данных неявно предполагает, что изменения, внесенные одним приложением, могут легко вызвать участие других приложений. Однако практика показывает, что иногда небольшие изменения действительно должны запускать многие приложения. В таких случаях на самом деле важно не изменение данных, а выполнение серии действий. Последовательность действий лучше всего фиксируется путем выполнения процедуры (что, в свою очередь, может привести к всевозможным изменениям в общих данных). Чтобы предотвратить то, что каждое приложение должно знать все внутренние действия (как это реализовано в другом приложении), могут быть использованы методы стандартной инкапсуляции как развернутые с традиционными вызовами процедур или как обращение к объекту. Для таких ситуаций приложению лучше всего предложить процедуру другим приложениям в форме удаленного вызова процедуры, или RPC. По сути, RPC позволяет приложению A использовать информацию, доступную только приложению B, без предоставления прямого доступа A к этой информации. Есть много преимуществ и недостатков в процедуре удаленных вызовов, которые подробно обсуждаются в главе 4.

Обмен сообщениями. Главный недостаток RPC заключается в том, что вызывающий и вызываемый абоненты для успешного обмена должны работать одновременно. Однако во многих сценариях эту одновременную деятельность часто бывает трудно или невозможно гарантировать. В таких случаях все, что нужно, – чтобы система, предлагающая обмен сообщениями, передающая запросы приложения A, выполнила действие в приложении. Система обмена сообщениями гарантирует, что в конечном итоге запрос будет доставлен, и если необходимо, то ответ также будет возвращен. Очевидно, что обмен сообщениями не является панацеей для интеграции приложений: он также вносит проблемы, касающиеся форматирования данных и размещения, требует, чтобы приложение знало, куда отправить сообщение, должны быть разработаны сценарии поиска потерянных сообщений и т. д. Как и RPC, мы будем обсуждать эти вопросы подробно в главе 4.

Эти четыре подхода говорят нам о том, что интеграция приложений обычно не бывает простой. Промежуточное программное обеспечение (в форме распределенной системы) может, однако, значительно помочь в интеграции, предоставляя необходимые средства, такие как поддержка для RPC или обмена сообщениями. Как уже говорилось, интеграция корпоративных приложений является важным целевым полем приложения сил для разработчиков промежуточного программного обеспечения.

Распространенные системы

Рассмотренные до сих пор распределенные системы в значительной степени характеризуются стабильностью: узлы являются фиксированными и имеют более или менее постоянное и качественное подключение к сети. В определенной степени эта стабильность реализуется различными методами для достижения прозрачности распределения. Например, существует много способов создания иллюзии, что лишь изредка компоненты могут выйти из строя. Кроме того, есть много способов скрыть фактическое сетевое местоположение узла, эффективно позволяющее пользователям и приложениям считать, что узлы остаются на месте.

Однако с появлением мобильных и встроенных вычислительных устройств, которые обычно называют **распространенными системами** (pervasive systems), положение изменилось. Как следует из названия, распространенные системы предназначены для работы в нашей естественной среде. Они также являются распределенными системами и, безусловно, соответствуют характеристикам, которые были приведены в разделе 1.1.

Что делает их уникальными по сравнению с вычислительными и информационными системами, описанными ранее, так это то, что разделение между пользователями и системными компонентами гораздо более размыто. Часто нет единого выделенного интерфейса, такого как комбинация экран/клавиатура. Вместо этого распространенная система нередко оснащена многими **датчиками** (sensors), которые фиксируют различные аспекты поведения пользователя. Кроме того, она может иметь множество **исполнительных механизмов** (actuators), чтобы выявлять и возвращать различные аспекты поведения пользователей и управлять ими.

Многие устройства в распространенных системах – небольшие, с питанием от батареи, мобильные и имеющие только беспроводное соединение, хотя не все эти характеристики применимы ко всем устройствам. Это не обязательно ограничивающие характеристики, как это иллюстрируют смартфоны [Roussos et al., 2005], и их роль в том, что сейчас называют **интернетом вещей** (Internet of Things, IoT) [Маттерн и Floerkemeier, 2010; Stankovic, 2014]. Тот факт, что часто приходится сталкиваться с тонкостями беспроводной и мобильной связи, требует специальных решений, чтобы сделать распространенную систему прозрачной или ненавязчивой, насколько это возможно. В последующем мы отмечаем различие между тремя типами распространенных систем, хотя эти три типа перекрывают друг друга: повсеместно распространенные вычислительные системы (их еще называют вездесущим (ubiquitous) компьютингом), мобильные системы и сенсорные сети. Это различие позволяет нам обратить внимание на разные аспекты распространенных систем.

Повсеместно распространенные вычислительные системы

До сих пор мы говорили о распространяющихся системах, чтобы подчеркнуть, что их элементы распространены во многих сферах окружающей среды. Пойдем на один шаг дальше с повсеместно распространенными вычисли-

тельными системами: система распространена и продолжительно присутствует. Последнее означает, что пользователь постоянно взаимодействует с системой, часто даже не осознавая, что такое взаимодействие происходит. В работе [Poslad, 2009] описываются основные требования для повсеместно распространенных вычислительных систем. Они примерно таковы:

1) (**распространение**) устройства объединены в сеть, распределены и доступны в прозрачном стиле;
2) (**взаимодействие**) взаимодействие между пользователями и устройствами крайне ненавязчиво;
3) (**знание контекста**) система осведомлена о контексте пользователя, чтобы оптимизировать взаимодействие;
4) (**автономность**) устройства работают автономно без вмешательства человека и являются самоуправляемыми;
5) (**интеллект**) система в целом может обрабатывать широкий спектр динамических действий и взаимодействий.

Давайте кратко рассмотрим эти требования с точки зрения распределенных систем.

Добавление 1: распределение. Как уже упоминалось, повсеместно распространенная вычислительная система является примером распределенной системы: устройства и другие компьютеры, образующие узлы системы, просто объединены в сеть и работают вместе, чтобы создать иллюзию единой связной системы. Распределение также происходит естественным образом: устройства, близкие к пользователям (например, датчики и исполнительные механизмы), подключены к компьютерам, скрытым от глаз и, возможно, даже работающим удаленно в облаке. Следует придерживаться большинства, если не всех, требований относительно прозрачности распределения, упомянутых в разделе 1.2.

Добавление 2: взаимодействие. Что касается взаимодействия с пользователями, то повсеместно распространенные вычислительные системы сильно отличаются от систем, которые мы пока обсуждали. Конечные пользователи играют заметную роль в дизайне этих систем, и тому, как происходит взаимодействие между пользователями и основной системой, должно быть уделено особое внимание. Для вездесущих систем вычислений большая часть взаимодействия между людьми будет неявной, с **неявным действием** (implicit action), определяемым как единое действие, «которое в основном не направлено на взаимодействие с компьютеризированной системой, но которую такая система воспринимает как вход» [Schmidt, 2000]. Другими словами, пользователь может не знать о том, что ввод предоставляется компьютерной системе. Можно сказать, что повсеместно распространенные компьютерные вычисления в определенной степени скрывают интерфейсы.

Простой пример – настройка сиденья водителя, рулевого управления и зеркал полностью персонализированы. Если Боб займет место водителя, система поймет, что она имеет дело с Бобом, и сделает соответствующие корректировки. То же самое происходит, когда Алиса использует автомобиль, в то время как неизвестному пользователю система предложит внести необходимые корректировки и запомнит их для последующего управления. Этот пример иллюстрирует важную роль датчиков в повсеместно распространен-

ных вычислениях как устройств ввода, которые используются для идентификации ситуации (конкретный человек, очевидно, желающий управлять автомобилем). Анализ входа приводит к действиям (внесению корректировок). В свою очередь, действия могут привести к естественной реакции, например Боб может немного поменять настройки сидений. Системе придется принимать во внимание все (неявные и явные) действия пользователя и реагировать соответственно.

Добавление 3: понимание контекста. Сделать реакцию на сенсорный ввод от пользователя совсем не просто. От повсеместно распространенной системы вычислений требуется принять во внимание контекст ввода, то есть понять контекст, в котором происходит взаимодействие. Знание контекста отличает системы повсеместных вычислений от более традиционных систем, которые мы обсуждали ранее. Это описано в работе [Dey and Abowd, 2000] следующим образом: «любая информация, которая может быть использована для характеристики ситуации, в которой производится ввод (то есть *кто, где, когда или что*), и считается имеющей отношение ко взаимодействию между пользователем и приложением». На практике контекст часто характеризуется местоположением, идентичностью, временем и деятельностью: где, кто, когда и что. Система должна иметь необходимый (сенсорный) вход, чтобы определить один или несколько такого рода типов контекста.

Что важно с точки зрения распределенных систем, так это то, что необработанные данные, собранные различными датчиками, поднимаются до уровня абстракции, который может быть использован приложениями. Конкретным примером является, например, обнаружение того, где находится человек в соответствии с координатами GPS, а затем отображение этой информации как его фактическое местоположение: угол улицы, или конкретный магазин, или другой известный объект. Вопрос в том, где эта обработка сенсорного ввода производится: путем сбора всех данных в центральном сервере, подключением к базе данных с подробной информацией о городе или сопоставлением на смартфоне пользователя. Очевидно, что существуют компромиссы, которые целесообразно рассмотреть.

В работе [Dey, 2010] обсуждаются более общие подходы к созданию контекстных приложений. Когда необходимы и гибкость, и потенциал распределения, привлекательны так называемые общие пространства данных, в которых процессы разделены во времени и пространстве, но, как мы увидим в следующих главах, при этом возникают проблемы масштабируемости. Обзор вопросов, связанных с осведомленностью о контексте и его влиянии на промежуточное программное обеспечение и распределенные системы в целом, представлен в [Baldauf et al., 2007].

Добавление 4: автономия. Важным аспектом большинства повсеместно распространенных вычислительных систем является то, что явное управление системами сводится к минимуму. В повсеместно распространенной вычислительной среде просто нет места для системного администратора, организующего работу. Как следствие система должна иметь полную возможность действовать автономно и автоматически реагировать на изменения. Это требует множества методов, несколько из которых будут обсуждаться на протяжении всей этой книги. Вот немного простых примеров:

- **распределение адресов.** Чтобы сетевые устройства могли общаться, им нужен IP-адрес. Адреса могут быть распределены автоматически с помощью протоколов, таких как протокол динамической конфигурации хоста (Dynamic Host Configuration Protocol, DHCP) [Droms, 1997] (для чего требуется сервер) или протокол компании Apple Zeroconf [Guttman, 2001];
- **добавление устройств.** Необходимо, чтобы можно было легко добавить устройства в существующую систему. Такая конфигурация реализуется с помощью протоколов автоматического подключения и воспроизведения (Universal Plug and Play Protocol, UPnP) [UPnP Forum, 2008]. Используя протокол UPnP, устройства могут узнавать друг друга и убеждаться, что они могут установить между собой связь;
- **автоматические обновления.** Многие устройства в повсеместной вычислительной системе должны быть в состоянии регулярно проверять через интернет, должно ли быть обновлено их программное обеспечение. Если это необходимо, они могут загрузить новые версии своих компонентов и в идеале продолжать начатый процесс.

Надо признать, что это очень простые примеры, но они показывают, что ручное вмешательство должно быть сведено к минимуму. Мы будем подробно обсуждать многие методики, связанные с самоуправлением.

Дополнение 5: интеллект. Наконец, в работе [Poslad, 2009] отмечается, что повсеместно распространенные вычислительные системы часто используют методы из области искусственного интеллекта. Это означает, что во многих случаях должен быть применен широкий спектр передовых алгоритмов и моделей для обработки недостаточно полного ввода данных, быстрая реакция на изменение среды, обработка неожиданных событий и т. д. Уровень, на котором это может или должно быть сделано в распределенной среде, имеет решающее значение с точки зрения распределенных систем. К сожалению, в области искусственного интеллекта решения для первоочередных задач распределенных систем еще не найдены, и это оказывает влияние на прогресс в развитии сетевых и распределенных устройств.

Мобильные вычислительные системы

Как уже упоминалось, мобильность часто является важной составляющей системы, и многие, если не все, аспекты, которые мы только что обсудили, также относятся и к мобильным вычислениям. Есть несколько задач, которые выделяют мобильные вычисления в распространенных системах в целом (см. также [Adelstein et al., 2005] и [Tarkoma, Kangasharju, 2009]).

Во-первых, устройства, которые образуют часть (распределенной) мобильной системы, могут широко варьироваться. Как правило, мобильные вычисления теперь выполняются такими устройствами, как смартфоны и планшеты. Вместе с тем совершенно разные типы устройств используют для связи интернет-протокол (IP), проводя мобильные вычисления в разных ракурсах. Такие устройства включают в себя пульт, элементы управления, пейджеры, активные знаки, автомобильное оборудование, различные устройства с поддержкой GPS и т. д. Характерной особенностью всех этих устройств является

то, что они используют беспроводные коммуникации. Определение «мобильный» подразумевает, как правило, наличие беспроводной связи (хотя есть исключения).

Во-вторых, предполагается, что в мобильных вычислениях местоположение устройства со временем меняется. Изменение местоположения оказывает влияние на многие задачи. Например, если местоположение устройства регулярно меняется, то, видимо, должны быть локально доступные сервисы. Как следствие может понадобиться особое внимание способу динамического обнаружения сервисов, но также позволяя и сервисам обозначать свое присутствие. Кроме того, также часто надо знать, где устройство находится на самом деле. Это может означать, что нам нужно знать фактические географические координаты устройства, например в приложениях для слежения и сопровождения, или может потребоваться, чтобы положение устройства определялось просто по его положению в сети (как в мобильном IP [Perkins, 2010; Perkins et al., 2011]).

Изменение местоположения также оказывает глубокое влияние на общение. Чтобы проиллюстрировать это, рассмотрим (беспроводную) мобильную специальную сеть, сокращенно **MANET** (Mobile Ad hoc Network). Предположим, что два устройства в MANET обнаружили друг друга в том смысле, что они знают сетевой адрес друг друга. Как мы проложим маршрут сообщений между ними? Статичные маршруты, как правило, не являются устойчивыми, поскольку узлы вдоль пути маршрутизации могут легко выйти за пределы диапазона их соседей, аннулируя путь. Для больших сетей MANET использование априорных путей установки – нежизнеспособный вариант. В нашем случае мы имеем дело с так называемыми **сетями, толерантными к нарушениям** (disruption-tolerant network): сетями, которые могут не гарантировать непосредственную связь между двумя узлами. Получение сообщения от одного узла к другому может тогда стать проблематичным, если не сказать больше.

Хитрость в таких случаях заключается в том, чтобы не пытаться установить канал связи от источника до места назначения, а использовать на два принципа. Первый будет обсуждаться в разделе 4.4 и использует специальные методы на основе **техники волнового распространения** (flooding-based techniques), которые позволяют сообщению постепенно распространяться через часть сети, чтобы в конечном итоге добраться до места назначения. Очевидно, что любой тип волнового распространения будет налагать требование избыточной связи, но это может быть та цена, которую следует заплатить. Второй метод в сети с нарушением толерантности позволит промежуточному узлу хранить полученное сообщение, пока не встретится другой узел, которому он может передать его. Другими словами, узел становится временным носителем сообщения, как показано на рис. 1.14. В конце концов, сообщение достигнет места назначения.

Нетрудно представить, что, выборочно передавая сообщения на встречающиеся узлы, можно обеспечить эффективную доставку. Например, если известно, что узлы принадлежат определенному классу, а источник и пункт назначения принадлежат тому же классу, мы можем решить передавать сообщения только между узлами в этого класса. Аналогично может оказаться

эффективным передавать сообщения только хорошо связанным узлам, то есть узлам, которые были в диапазоне многих других узлов в недавнем прошлом. Обзор предоставлен в [Spyropoulos et al., 2010].

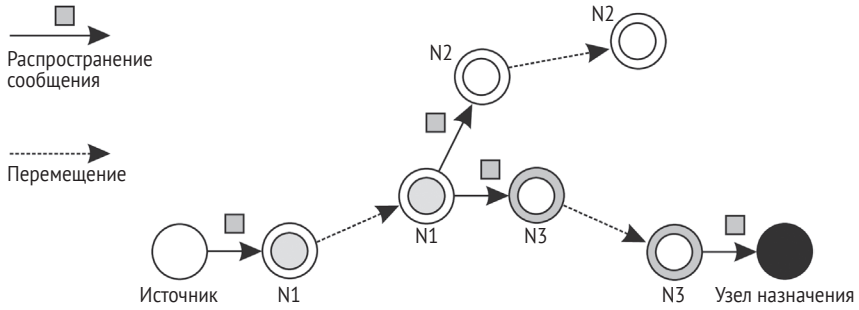


Рис. 1.14 ❖ Передача сообщений в (мобильной) толерантной к нарушениям сети

Примечание 1.11 (дополнительно: социальные сети и модели мобильности)

Неудивительно, что мобильные вычисления тесно связаны с местонахождением людей. С ростом интереса к сложным социальным сетям [Vega-Redondo, 2007; Jackson, 2008] и взрывного роста числа использования смартфонов несколько групп исследователей пытаются объединить анализ социального поведения и распространение информации в так называемых **произвольно подключаемых сетях** (pocket-switched networks) [Hui et al., 2005]. Последние являются сетями, в которых узлы формируются людьми (или фактически их мобильными устройствами) и ссылки образуются, когда два человека пересекаются друг с другом, что позволяет их устройствам обмениваться данными.

Основная идея заключается в том, чтобы позволить информации распространяться с использованием специальных отношений между людьми. При этом становится важно понимать структуру социальной группы. Одной из первых работ, где исследовался вопрос, как социальная осведомленность может быть использована в мобильных сетях, была работа [Miklas et al., 2007]. В их подходе, основанном на информации о встречах между людьми, два человека характеризуются или как друзья, или как незнакомцы. Друзья часто общаются, в то время как число повторяющихся встреч между незнакомцами мало. Цель состоит в том, чтобы убедиться, что сообщение от Алисы Бобу в конечном итоге доставлено.

Оказалось, что когда Алиса принимает стратегию, по которой она раздает сообщение каждому из ее друзей, и каждый из этих друзей передает сообщение Бобу, как только он его встретит, сообщение доходит до Боба с задержкой, превышающей приблизительно 10 % от максимально возможной задержки. Любая другая стратегия, как, например, пересылка сообщения только 1 или 2 друзьям, работает намного хуже. На передачу сообщения незнакомцу количество друзей существенного влияния не оказывает. Другими словами, существует большая разница, принимается во внимание отношение с друзьями или нет, хотя при этом все же необходимо разумно рассмотреть и стратегию пересылки напрямую.

Для больших групп людей требуются более сложные подходы. Во-первых, может случиться так, что сообщения должны быть отправлены между людьми в разные сообщества. Что мы подразумеваем под сообществом? Если мы рассмотрим социальную сеть (где вершину представляет человек и присутствует ссылка на то, что два

человека имеют социальные отношения), то сообщество – грубо говоря, группа вершин, в которых есть много ссылок между его членами и только несколько вершин со ссылками в других группах [Ньюман, 2010]. К сожалению, многие алгоритмы обнаружения сообщества требуют полной информации о социальной структуре, что делает практически невозможной оптимизации связи в мобильных сетях. В работе [Hui et al., 2007] предлагаются алгоритмы обнаружения некоторых децентрализованных сообществ. По сути, эти алгоритмы основаны на том, чтобы позволить узлу i (1) обнаруживать множество узлов, с которым он регулярно пересекался и которое названо знакомым множеством F_i, и (2) постепенно расширять свое локальное сообщество C_i, с $F_i \subseteq C_i$. Изначально C_i и F_i будут пустыми, но постепенно F_i будет расти, а вместе с ним и C_i. В простейшем случае узел j добавил в сообщество C_i следующее:

Узел i добавляет j в C_i, когда $\frac{|F_j \cap C_i|}{|F_j|} > \lambda$ для некоторых $\lambda > 0$.

Другими словами, когда часть знакомого множества j существенно пересекается с сообществом i, узел i должен добавить j к своему сообществу. Кроме того, для объединения сообществ мы имеем:

Объединить два сообщества, когда $|C_i \cap C_j| > \gamma|C_i \cup C_j|$ для некоторого $\gamma > 0$.

Это означает, что два сообщества должны быть объединены, когда они совместно имеют значительное количество членов. (В своих экспериментах Хуэй (Hui) и др. обнаружили, что соотношение $\lambda = \gamma = 0{,}6$ приводит к хорошим результатам.) Как показано в [Hui et al., 2011], знание сообществ в сочетании с подключением узла либо в сообщество, либо глобально может быть использовано впоследствии для эффективных сообщений в толерантной к нарушениям сети. Очевидно, что большая часть производительности мобильной вычислительной системы зависит от того, как движутся узлы. В частности, для того чтобы предварительно оценить эффективность новых протоколов или алгоритмов, важно иметь представление о том, какие модели мобильности на самом деле реалистичны. Долгое время данных о таких закономерностях было немного, но недавние эксперименты изменили это. Различные группы начали собирать статистические данные о мобильности людей, которые используются для построения моделей. Кроме того, для получения более реалистичных моделей мобильности были использованы следы передвижений (см., например, [Kim et al., 2006b]). Однако понимание моделей мобильности человека в целом остается сложной проблемой. В отчете [González et al., 2008] приведены результаты по моделированию на основе данных, собранных по информации о 100 000 пользователей сотовых телефонов в течение шести месяцев. Они отметили, что поведение смещения пользователей может быть представлено следующим, относительно простым распределением:

$$\mathbb{P}[\Delta r] = (\Delta r + \Delta r_0)^{-\beta} \cdot e^{-\Delta r/\kappa},$$

где Δr – фактическое смещение, а $\Delta r_0 = 1{,}5$ км – постоянное начальное смещение. При $\beta = 1{,}75$ км и $\kappa = 400$ это приводит к распределению, показанному на рис. 1.15.

Мы можем сделать вывод, что люди склонны оставаться на месте. На самом деле дальнейший анализ показал, что люди склонны возвращаться в одно и то же место через 24, 48 или 72 часа, ясно показывая, что люди, как правило, ходят в одинаковые места. В последующем исследовании [Song et al., 2010] показано, что человеческая мобильность на самом деле замечательно предсказуема.

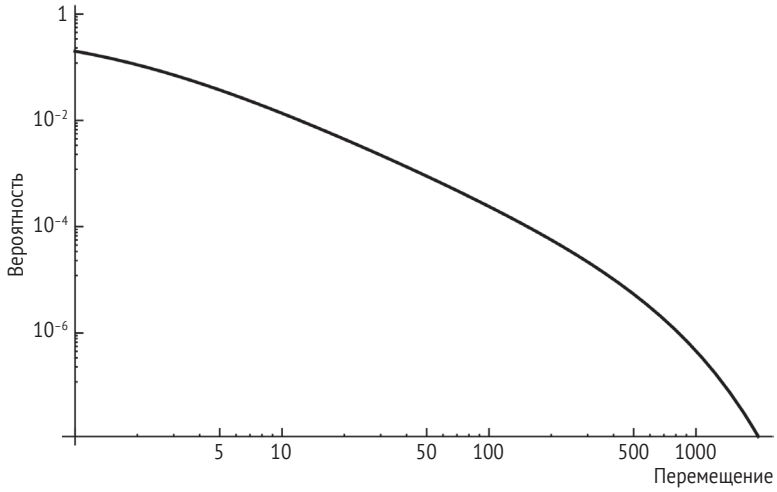


Рис. 1.15 ❖ Распределение перемещений (мобильных) пользователей сотовых телефонов

Сенсорные сети

Наш последний пример распространяющихся систем – сенсорные сети. Эти сети во многих случаях являются частью технологии, способствующей их распространению, и мы видим, что многие решения для сенсорных сетей возвращаются в распространенных приложениях. Сенсорные сети делает интересными с точки зрения распределенной системы то, что они представляют собой нечто большее, чем просто набор устройств ввода. Вместо этого, как мы увидим, сенсорные узлы часто взаимодействуют, чтобы эффективно обрабатывать обнаруженные данные в зависимости от приложения, что делает их очень отличными от, например, традиционных компьютерных сетей. В работах [Akyildiz et al., 2002] и [Akyildizet al., 2005] предоставлен обзор перспектив сетей. Системно-ориентированное введение в сенсорные сети представлено в работах [Zhao and Guibas, 2004], а также [Karl and Willig, 2005].

Сенсорная сеть обычно состоит из десятков, сотен или тысяч сравнительно небольших узлов, каждый из которых оснащен одним или несколькими чувствительными устройствами. Кроме того, узлы часто могут выступать в роли исполнительных механизмов [Akyildiz and Kasimoglu, 2004]. Типичным примером является автоматическая активация спринклеров при обнаружении пожара. Многие сенсорные сети используют беспроводную связь, а узлы часто питаются от батареи. Ограниченные ресурсы батерей уменьшают коммуникационные возможности, и эффективное снижение энергопотребления устройств занимает одно из первых мест в списке критериев их проектирования.

При ближайшем рассмотрении отдельных узлов мы видим, что концептуально их структура не сильно отличается от структуры «обычного» компьютера: над аппаратным уровнем находится уровень программного обеспе-

чения, подобный тем, которые предлагают традиционные операционные системы, обеспечивая низкоуровневый доступ к сети, доступ к датчикам, управление памятью и т. д. Как правило, включена поддержка конкретных услуг, таких как определение местоположения, дополнительная память (например, дополнительные флеш-устройства), а также удобных средств связи для обмена сообщениями и маршрутизации. Как и в других компьютерных сетевых системах, для эффективного применения сенсорных сетевых приложений требуется дополнительная поддержка. В распределенных системах это принимает форму промежуточного программного обеспечения. Для сенсорных сетей вместо рассмотрения промежуточного программного обеспечения целесообразнее рассмотреть программную поддержку, как это сделано в обширном обзоре [Mottola and Picco, 2011].

Одним из типичных аспектов поддержки программирования является объем, предоставляемый примитивами связи. Эта область может варьироваться в зависимости от физической близости узла и предоставления примитивов для всей системы коммуникации. Кроме того, также может быть целесообразным обратиться к конкретной группе узлов. Аналогично вычисления могут быть ограничены отдельным узлом, группой узлов или влиянием всех узлов. Для иллюстрации в работе [Welsh and Mainland, 2004] использованы так называемые абстрактные регионы (abstract regions), позволяющие узлу идентифицировать окрестности, откуда он может, например, собрать информацию:

```
1 регион = k_nearest_region.create (8);
2 чтение = get_sensor_reading ();
3 region.putvar (reading_key, reading);
4 max_id = region.reduce (OP_MAXID, reading_key);
```

В строке 1 узел сначала создает область из восьми ближайших соседей, после чего выбирает значение из своего датчика(ов). Это значение вписывается в ранее определенную область, которая должна быть известна с помощью ключа `reading_key`. В строке 4 узел проверяет, показания какого датчика в определенной области были самыми большими, который возвращается в переменную `max_id`.

В качестве другого примера рассмотрим сенсорную сеть как реализацию распределенной базы данных, которая, согласно [Mottola and Picco, 2011], является одной из четырех возможных способов доступа к данным. Такое представление базы данных является довольно распространенным, и это легко понять, если учесть, как много сенсорных сетей развернуто для измерений и наблюдения [Bonnet et al., 2002]. В таких случаях оператор хотел бы извлечь информацию из (части) сети, просто делая такие запросы, как «Насколько загружен трафик на шоссе 1 в северном направлении на Санта-Круз?». Подобные запросы напоминают традиционные базы данных. В этом случае ответ, вероятно, потребуется предоставить при совместной работе многих датчиков вдоль шоссе 1, не затрагивая другие датчики. Чтобы организовать сенсорную сеть как распределенную базу данных, существуют два крайних случая, как показано на рис. 1.16. В первом случае датчики не взаимодействуют, поэтому просто данные отправляются в централизованную

базу данных, расположенную на сайте оператора. Другой вариант – перенаправление запросов на соответствующие датчики и разрешение каждому вычислить ответ, оставляя оператору агрегировать ответы. Ни одно из этих решений не привлекательно. Первое требует, чтобы датчики отправляли все свои измеренные данные через сеть, что может привести к излишней затрате сетевых ресурсов и энергии. Второе решение также может быть расточительным, поскольку не используются объединяющие возможности датчиков, которые позволили бы отправлять оператору меньше данных. Как видно, необходимы средства для обработки данных в сети, аналогичные предыдущему примеру с абстрактными областями. Обработка в сети может быть выполнена различными способами. Один очевидный – направить запрос на все узлы датчика вдоль дерева, охватывающего все узлы, и впоследствии агрегировать результаты по мере их распространения обратно в корень, где находится инициатор. Агрегация будет происходить, когда два или больше ветвей дерева собираются вместе.

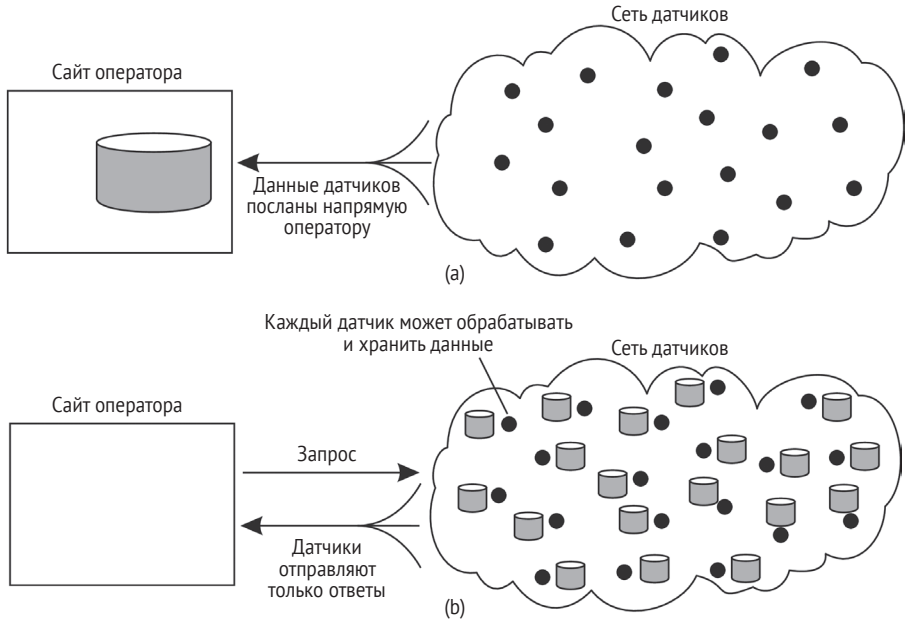


Рис. 1.16 ❖ Организация базы данных сенсорной сети с сохранением и обработкой данных (а) только на месте оператора или (б) только на датчиках

Эта, казалось бы, простая схема требует решения сложных вопросов:

- как мы (динамически) настраиваем эффективное дерево в сенсорной сети?
- как происходит агрегация результатов? Можно ли это контролировать?
- что происходит при сбое сетевых соединений?

Эти вопросы были частично рассмотрены в системе обработки запросов из сети датчиков TinyDB, в которой реализуется декларативный интерфейс

(база данных) для беспроводных сенсорных сетей [Madden et al., 2005]. По сути, система TinyDB может использовать любой алгоритм маршрутизации на основе структуры дерева. Промежуточный узел будет собирать и агрегировать результаты от своих дочерних элементов вместе со своими собственными находками и отправлять к корню. Чтобы это было эффективным, запросы должны охватывать период времени, позволяющий тщательно планировать операции, чтобы сетевые ресурсы и энергия использовались оптимально.

Вместе с тем, когда запросы инициируются из разных точек в сети, использование однокорневых деревьев, таких как в TinyDB, может быть недостаточно эффективным. В качестве альтернативы сенсорные сети могут быть оборудованы специальными узлами, к которым направляются результаты, а также запросы, связанные с этими результатами. Простой пример: запросы и результаты, связанные с показаниями температуры, могут быть собраны в другом месте, чем те, которые связаны с измерениями влажности. Этот подход напрямую связан с понятием публикации/подписки системы.

Примечание 1.12 (дополнительно: когда проблема энергии начинает становиться критической)

Как уже упоминалось, многие сенсорные сети должны работать с экономией энергии, используя батареи или другие источники питания с ограниченным ресурсом. Методом, уменьшающим расход энергии, является разрешение узлам быть активными только часть времени. В частности, предположим, что узел повторно активен в течение единиц времени T_{active}, а между этими активными периодами активность приостановлена на $T_{\text{suspended}}$ единиц. Доля времени, когда узел активен, называется его рабочим циклом τ, то есть

$$\tau = \frac{T_{\text{active}}}{T_{\text{active}} + T_{\text{suspended}}}.$$

Значения τ обычно составляют порядка 10–30 %, но когда сети требуется оставаться в рабочем состоянии в течение периодов, превышающих многие месяцы или даже годы, τ достигают значений 1 %, что становится критическим.

Проблема дежурных сетей состоит в том, что, в принципе, узлы должны быть активными все вместе, иначе общение становится невозможным. Учитывая, что пока узел приостановлен, идут только его локальные часы, а эти часы подвержены дрейфам, пробуждение может оказаться проблематичным. Это особенно верно для сетей с очень малыми рабочими циклами.

Когда группа узлов активна одновременно, говорят, что узлы формируют **синхронизированную группу** (synchronized group). По сути, есть две проблемы, на которые следует обратить внимание. Во-первых, нам нужно убедиться, что узлы в синхронизированной группе остаются активными в одно и то же время. На практике это выполнить относительно просто, если каждый узел передает информацию о своем текущем местном времени. Тогда просто местные настройки часов сделают свое дело. Вторая проблема сложнее. Она состоит в объединении двух разных синхронизированных групп в одну, в которой все узлы синхронизированы. Давайте внимательнее посмотрим на то, с чем мы сталкиваемся в этом случае. Многое в последующем нашем обсуждении основано на работе [Voulgaris et al., 2016].

Чтобы объединить две группы, нам нужно сначала убедиться, что одна группа обнаруживает другую. Действительно, если их соответствующие активные периоды полностью не пересекаются, нет надежды, что любой узел в одной группе сможет

получить сообщение от узла в другой группе. В *активном методе обнаружения* узел отправляет сообщение о присоединении во время приостановленного периода. Другими словами, когда он приостановлен, он на время просыпается, чтобы выявить узлы в других группах, готовых к присоединению. Насколько велика вероятность того, что другой узел подхватит это сообщение? Надо понимать, что учитывать нужно только случай, когда $\tau < 0{,}5$, в другом случае два активных периода всегда будут перекрываться, и две группы могут легко обнаружить присутствие друг друга. Вероятность P_{da} того, что сообщение о присоединении может быть получено в течение активного периода другого узла, равна

$$P_{da} = \frac{T_{\text{active}}}{T_{\text{suspended}}} = \frac{\tau}{1-\tau}.$$

Это означает, что для низких значений τ P_{da} также очень мала.

В *пассивном методе обнаружения* узел пропускает приостановленное состояние с (очень малой) вероятностью P_{dp}, то есть он просто остается активным в течение $T_{suspended}$ единиц времени после своего активного периода. За это время он сможет принять любое сообщение, отправленное его соседями, которые по определению являются синхронизированной группой. Эксперименты показывают, что пассивное обнаружение уступает активному обнаружению.

Просто заявить, что две синхронизированные группы должны объединиться, недостаточно: если A и B обнаружили друг друга, какая группа адаптирует рабочий цикл настройки другой? Простое решение заключается в использовании понятия идентификаторов кластеров. Каждый узел начинается со случайно выбранного идентификатора и, по сути, также идентификатора синхронизированной группы, рассматривая только себя в качестве члена. После обнаружения другой группы B все узлы в группе A соединяются с B, если и только если кластер с идентификатором B больше, чем кластер A.

Синхронизация может быть значительно улучшена с помощью так называемого *целевого сообщения соединения*. Всякий раз, когда узел N получает сообщение о присоединении от группы A с более низким идентификатором кластера, он, очевидно, не должен присоединяться к A. Однако теперь, поскольку N знает активный период A, он может отправить сообщение о присоединении именно в течение этого периода. Очевидно, что вероятность того, что узел из A получит это сообщение, очень высока, можно разрешить узлам из A присоединяться к группе N. Кроме того, когда узел решает присоединиться к другой группе, он может отправить специальное сообщение членам своей группы, давая возможность быстро присоединиться и им.

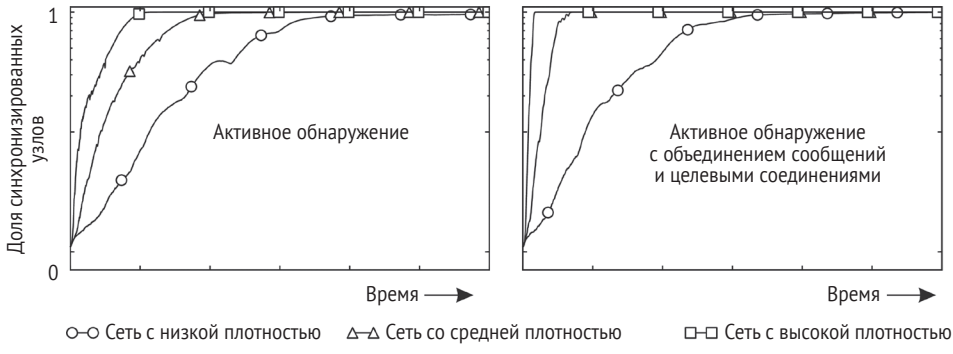


Рис. 1.17 ❖ Скорость, с которой разные синхронизированные группы могут объединиться

На рис. 1.17 показано, как быстро синхронизированные группы могут объединяться с использованием двух разных стратегий. Эксперименты основаны на мобильной сети 4000 узлов с использованием реалистичных моделей мобильности. Узлы имеют рабочий цикл менее 1 %. Эти эксперименты показывают, что привидение даже большой мобильной сети в состояние, в котором все узлы активны одновременно, вполне осуществимо. Для дополнительной информации см. [Voulgaris et al., 2016].

1.4. Резюме

Распределенные системы состоят из автономных компьютеров, которые работают совместно, создавая единую связную систему. Это сочетание независимого и все же согласованного коллективного поведения достигается путем сбора протоколов в промежуточное программное обеспечение: программный уровень, логически размещенный между операционными системами и распределенными приложениями. Протоколы включают все необходимое для связи, транзакции, состава услуг и, возможно самое главное, надежности.

Целями разработки распределенных систем являются совместное использование ресурсов и обеспечение прозрачности. Кроме того, дизайнеры стремятся скрыть многие из сложностей, связанных с распределением процессов, данных и контроля. Однако эта прозрачность распределения достигается не только ценой производительности, и на практике полная прозрачность никогда не может быть достигнута. Тот факт, что для достижения различных форм прозрачности, присущих проекту распределенных систем, должны быть найдены компромиссы, усложняет проектирование и понимание. Одной из трудных конкретных целей дизайна, которая не всегда хорошо сочетается с достижением прозрачности распределения, является масштабируемость. Это особенно верно для географической масштабируемости, поскольку в этом случае могут оказаться трудными скрытие задержек и ограничения пропускной способности. Подобным образом административная масштабируемость системы, которая предназначена для охвата нескольких административных доменов, может легко конфликтовать с целями по достижению прозрачности распределения.

Дело еще более осложняется тем фактом, что многие разработчики изначально делают предположения о базовой сети, что в принципе неправильно. Позже, когда предположения отброшены, может оказаться трудным маскировать нежелательное поведение. Типичный пример – предположение, что латентность сети незначительна. Другие подводные камни включают в себя предположение, что сеть надежна, статична, безопасна и однородна.

Различные типы распределенных систем можно классифицировать как ориентированные на поддержку вычислений, обработку информации и распространенность. Распределенные вычислительные системы обычно используются для высокопроизводительных приложений, часто встречающихся при параллельных вычислениях. Область, возникшая из принципа параллельной обработки, изначально была сеточным вычислением, с упором

на совместное использование ресурсов по всему миру, определив то, что сейчас известно как облачные вычисления. Облачные вычисления, помимо высокопроизводительных вычислений, также поддерживают распределенные системы в традиционных офисных средах, где важную роль играют базы данных. Как правило, системы обработки транзакций развертываются в этих средах. Наконец, появился класс распределенных систем, которые составлены специальным образом из небольших компонентов и чаще всего не управляются через системного администратора. Этот последний класс обычно представлен повсеместно распространяющимися вычислительными средами, включая мобильные вычислительные системы, а также среды с большим количеством датчиков.

Глава 2

Архитектуры

Распределенные системы часто представляют собой сложные программы, компоненты которых по определению распределены по нескольким машинам. Чтобы правильно организовать такие системы, крайне важно понять существо этой сложности. Можно по-разному смотреть на организацию распределенной системы, но очевидным является проведение различия между, с одной стороны, логической организацией сбора компонентов программного обеспечения и, с другой, фактической физической реализацией.

Организация распределенных систем в основном определяется программным обеспечением компонентов, составляющих систему. Архитектуры программного обеспечения (software architectures) говорят нам, как должны быть организованы различные программные компоненты и как они должны взаимодействовать. В этой главе мы сначала обратим внимание на некоторые прикладные архитектурные стили для организации (распределенных) компьютерных систем.

Важной целью распределенных систем является отделение приложений от базовых платформ путем создания промежуточного уровня. Этот уровень является важным решением относительно архитектуры системы, и его основная цель состоит в обеспечении прозрачности распределения. Вместе с тем достижение прозрачности требует компромиссов, которые приводят к различным методам построения промежуточного программного обеспечения в соответствии с потребностями использующих его приложений. Мы обсудим некоторые из наиболее часто применяемых методов, поскольку они влияют на организацию самого промежуточного программного обеспечения.

Реальная реализация распределенной системы требует от нас также размещения программных компонентов на реальных машинах. Есть много различных вариантов достижения этого. Окончательная реализация архитектуры программного обеспечения также определяется как **архитектура системы** (system architecture). В этой главе мы рассмотрим традиционные централизованные архитектуры, в которых один сервер реализует большинство программных компонентов (и, следовательно, функциональность), в то время как удаленные клиенты могут с помощью простых средств связи получить доступ к этому серверу. Кроме того, мы рассматриваем децентрализованные одноранговые архитектуры, в которых все узлы играют более или менее равные роли. Многие реальные распределенные системы часто организованы гибридным способом, объединяя элементы централизованной и децентрализованной архитектур. Мы обсудим несколько типичных примеров.

И заканчиваем эту главу рассмотрением организации двух широко применяемых распределенных систем: системы обмена файлами NFS и интернета.

2.1. Архитектурные стили

Мы начинаем нашу дискуссию об архитектурах с рассмотрения логической организации программных компонентов распределенной системы, также называемой **архитектурой программного обеспечения** (software architectures) [Bass и др., 2003]. Проведены значительные исследования по программным архитектурам, и в настоящее время общепризнано, что проектирование и адаптация архитектуры имеют решающее значение для успешной разработки крупных программных систем.

Для нашего обсуждения важно понятие **архитектурного стиля** (architectural style). Такой стиль формулируется в терминах компонентов: как компоненты соединены друг с другом, как осуществляется обмен данными между компонентами и, наконец, как эти элементы совместно встроены в систему. **Компонент** (component) является модульным устройством с четко определенными необходимыми и предоставляемыми **интерфейсами** (interface), которое *заменяемо* в своем окружении [OMG, 2004]. Важно, что компонент может быть заменен, в частности, и в процессе работы системы. Это связано с тем, что во многих случаях отключение системы для обслуживания невозможно. В лучшем случае только ее часть может временно выйти из строя. Замена компонента может быть выполнена, только если его интерфейсы при этом остаются нетронутыми.

Несколько сложнее понять концепцию **коннектора** (connector, соединитель), который обычно описывается как механизм, который обеспечивает связь, координацию или взаимодействие между компонентами [Mehta et al., 2000; Shaw и Clements, 1997]. Например, соединитель может быть сформирован средствами для (удаленных) процедур вызовов, передачи сообщений или потоковой передачи данных. Другими словами, он обеспечивает поток управления и данных между компонентами. Используя компоненты и коннекторы, мы можем прийти к различным конфигурациям, которые, в свою очередь, были классифицированы в архитектурных стилях. К настоящему времени было определено несколько стилей, из которых наиболее важными для распределенной системы являются следующие:

- многоуровневая архитектура;
- объектно-ориентированные архитектуры;
- ресурсно-централизованные архитектуры;
- архитектуры на основе событий.

Далее мы обсудим каждый из этих стилей отдельно. Мы заранее отмечаем, что в большинстве реальных распределенных систем объединяется много разных стилей. Часто подход, при котором система подразделяется на несколько (логических) уровней, является настолько универсальным принципом, что это, как правило, используется в большинстве архитектурных стилей.

Многоуровневая архитектура

Основная идея для многоуровневого стиля проста: компоненты организованы **многоуровневым образом** (layered fashion), где компонент на уровне L_j может выполнить **обратный вызов** (downcall) компонента на нижнем уровне L_i (при $i < j$) и, как правило, ожидает ответа. Только в исключительных случаях возможен **вызов компоненты более высокого уровня** (upcall). Три распространенных случая показаны на рис. 2.1.

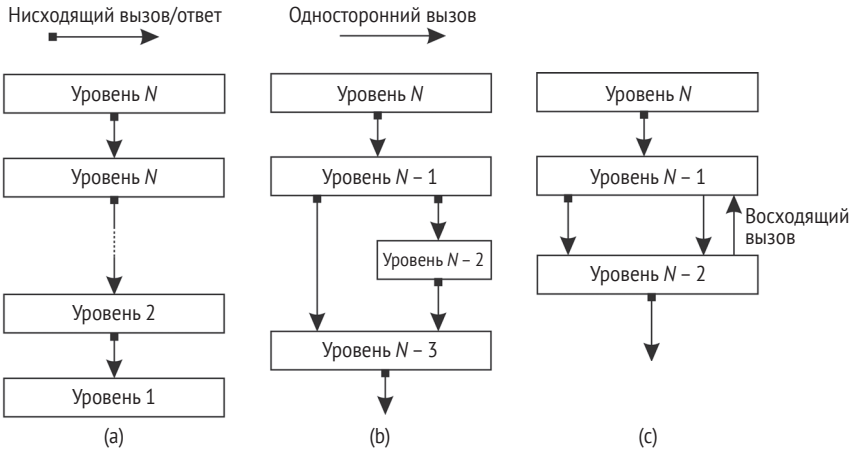


Рис. 2.1 ❖ Многоуровневая организация:
а) чистая; б) смешанная; с) с вызовами ([Krakowiak, 2009])

На рис. 2.1a показана стандартная организация, в которой выполняются только нисходящие вызовы до следующего нижнего уровня. Эта организация обычно развертывается в случае сетевого взаимодействия.

Во многих ситуациях мы также сталкиваемся с организацией, показанной на рис. 2.1б. Рассмотрим, например, приложение A, которое использует библиотеку L_{OS} для взаимодействия с операционной системой. В то же время приложение использует специализированную математическую библиотеку $L_{math,}$ которая также была реализована с применением L_{OS}. В этом случае, ссылаясь на рис. 2.1b, приложение A реализуется на уровне $N – 1$, L_{math} на уровне $N – 2$, и L_{OS}, которая является общей для них обоих, на уровне $N – 3$.

Наконец, особая ситуация показана на рис. 2.1c. В некоторых случаях удобно, чтобы нижний уровень выполнял вызов следующего более высокого уровня. Типичным примером является случай, когда операционная система сообщает о возникновении события, для чего она вызывает пользовательскую операцию, для которой приложение ранее передало ссылку (обычно называемую **дескриптором** (handle)).

Многоуровневые протоколы связи

Хорошо известная и широко распространенная многоуровневая архитектура – это так называемые *стеки протоколов связи*. Мы сосредоточимся здесь только на общей картине и отложим подробное обсуждение до раздела 4.1.

В стеках протоколов связи каждый уровень реализует одну или несколько услуг связи, позволяющих отправлять данные из пункта назначения в одну или несколько целей. Для этого каждый уровень предлагает **интерфейс**, определяющий функции, которые могут быть вызваны. В принципе, интерфейс должен полностью скрывать фактическую реализацию сервиса. Другая важная концепция в случае коммуникации – это **протокол связи** ((communication) protocol), который описывает правила, которым будут следовать стороны для обмена информацией. Важно понимать разницу между услугой, предлагаемой уровнем, интерфейсом, посредством которого эта служба становится доступной, и протоколом, реализуемым уровнем для установления связи. Это различие показано на рис. 2.2.

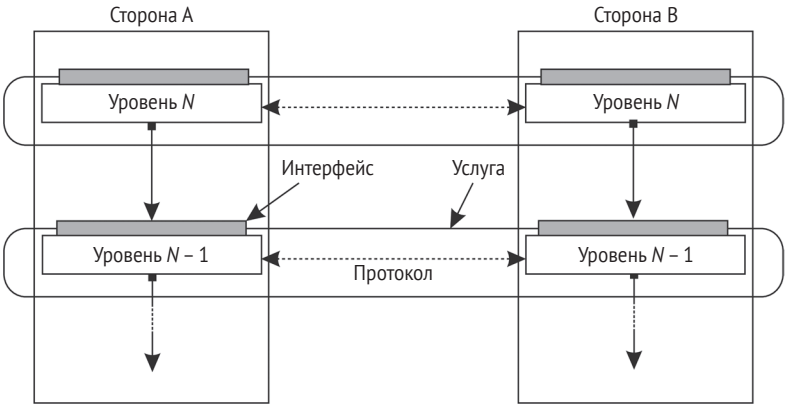


Рис. 2.2 ❖ Многоуровневый стек протоколов связи, показывающий разницу между услугой, ее интерфейсом и протоколом, который она использует

Чтобы прояснить данное различие, рассмотрим надежную, ориентированную на соединение услугу, которая предоставляется многими системами связи. В этом случае передающая сторона должна сначала установить соединение с другой стороной, прежде чем они смогут отправлять и получать сообщения. Надежность обеспечивается тем, что существуют строгие гарантии, что отправленные сообщения действительно будут доставлены другой стороне, даже если существует высокий риск того, что сообщения могут быть потеряны (как, например, может иметь место при использовании беспроводной связи). Кроме того, такие службы обычно также обеспечивают доставку сообщений в том же порядке, в котором они были отправлены.

Этот вид услуг реализуется в интернете посредством **протокола управления передачей данных** (Transmission Control Protocol, TCP). В протоколе указывается, какими сообщениями следует обменяться, чтобы установить или разорвать соединение, что нужно сделать, чтобы сохранить порядок передаваемых данных, и что нужно сделать обеим сторонам для обнаружения и исправления данных, которые были потеряны во время передачи. Служба доступна в виде относительно простого интерфейса программирования, содержащего вызовы для установки соединения, отправки и получения сообщений, а также для разрыва соединения. Фактически существуют различные доступные интерфейсы, часто зависящие от используемой операционной системы или языка программирования. Аналогично существует много разных реализаций протокола и его интерфейсов. (Все подробности можно найти в [Stevens, 1994; Wright and Stevens, 1995].)

Примечание 2.1 (пример: две сообщающиеся стороны)

Чтобы сделать это различие между службой, интерфейсом и протоколом более конкретным, рассмотрим следующие две взаимодействующие стороны, также известные как клиент и сервер соответственно, выраженные на языке Python (обратите внимание, что некоторый код был удален для ясности).

```
from socket import *
s = soket(AF_INET, SOCK_STREAM)
(conn, addr) = s.accept ()   # возвращает новый сокет и addr. client
while True:                  # навсегда
  data = conn.recv(1024)     # получить данные от клиента
  if not data: break         # остановиться, если клиент остановился
  conn.send(str(data) + «*») # вернуть отправленные данные плюс «*»
conn.close()                 # закрыть соединение
```

(а) Простой сервер

```
from socket import *
s = soket(AF_INET, SOCK_STREAM)
s.connect((HOST, PORT)) # подключиться к серверу (блок пока не принят)
s.send('Hello, world')  # отправить некоторые данные
data = s.recv(1024)     # получить ответ
print data              # напечатать результат
s.close()               # закрыть соединение
```

(б) Клиент

Рис. 2.3 ❖ Две связывающиеся стороны

В этом примере создается сервер, который использует сервис, ориентированный на соединение, как это предлагается библиотекой сокетов, доступной в Python. Данная услуга позволяет двум взаимодействующим сторонам надежно отправлять и получать данные по соединению.

Основные функции, доступные в его интерфейсе:

- `socket()`: для создания объекта, представляющего соединение;
- `accept()`: блокирующий вызов для ожидания входящих запросов на соединение, в случае успеха вызов возвращает новый сокет для отдельного соединения;
- `connect()`: установить соединение с указанной стороной;

- `close()`: разорвать соединение;
- `send()`, `recv()`: отправить и получить данные по соединению соответственно.

Комбинация констант `AF_INET` и `SOCK_STREAM` используется для указания того, что при обмене данными между двумя сторонами должен использоваться протокол TCP. Эти две константы можно рассматривать как часть интерфейса, тогда как использование TCP является частью предлагаемой услуги. *Как* реализован TCP или любая часть коммуникационного сервиса, полностью скрыто от приложений.

Наконец, также обратите внимание, что эти две программы неявно придерживаются протокола уровня приложения: очевидно, если клиент отправляет некоторые данные, сервер возвращает их. Действительно, он работает как эхо-сервер: сервер добавляет звездочку к данным, отправленным клиентом.

Уровни приложений

Давайте теперь обратим внимание на логическое разделение приложений. Учитывая, что большой класс распределенных приложений нацелен на поддержку доступа пользователей или приложений к базам данных, многие выступают за различие между тремя логическими уровнями, по существу следуя многоуровневому архитектурному стилю:

- уровень интерфейса приложения;
- уровень обработки;
- уровень данных.

В соответствии с этими уровнями мы видим, что приложения часто могут быть построены из примерно трех разных частей: части, которая обрабатывает взаимодействие с пользователем или каким-либо внешним приложением; части, которая работает с базой данных или файловой системой; и средней части, которая обычно содержит основные функциональные возможности приложения. Эта средняя часть логически размещена на уровне обработки. В отличие от пользовательских интерфейсов и баз данных, существует не так много общих аспектов уровня обработки. Поэтому мы приведем ряд примеров, чтобы прояснить этот уровень.

В качестве первого примера рассмотрим поисковую систему в интернете. Игнорируя все анимированные баннеры, изображения и другие причудливые оформления окна, пользовательский интерфейс поисковой системы может быть очень простым: пользователь вводит строку ключевых слов и впоследствии представляет список заголовков веб-страниц. Результат состоит из огромной базы данных веб-страниц, которые были предварительно выбраны и проиндексированы. Ядром поисковой системы является программа, которая преобразует строку ключевых слов пользователя в один или несколько запросов к базе данных. Впоследствии он ранжирует результаты в список и преобразует этот список в серию HTML-страниц. Эта часть поиска информации обычно размещается на уровне обработки. На рис. 2.4 показана эта организация.

В качестве второго примера рассмотрим систему поддержки принятия решений для биржевого маклера. По аналогии с поисковой системой такая система может быть разделена на следующие три уровня:

- интерфейс, реализующий пользовательский интерфейс или предлагающий интерфейс программирования для внешних приложений;

- интерфейс для доступа к базе данных с финансовыми данными;
- программы анализа между двумя первыми.

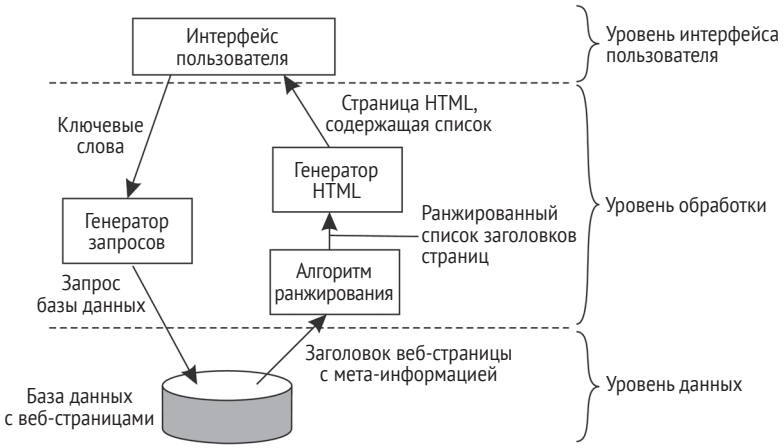


Рис. 2.4 ❖ Упрощенная организация поискового движка интернета на трех различных уровнях

Анализ финансовых данных может потребовать сложных методов и приемов из статистики и искусственного интеллекта. В некоторых случаях ядро системы поддержки принятия финансовых решений может даже требовать выполнения на высокопроизводительных компьютерах, чтобы достичь пропускной способности и скорости реагирования, ожидаемой от пользователей.

В качестве последнего примера рассмотрим типичный настольный пакет, состоящий из текстового процессора, приложения для работы с электронными таблицами, средств связи и т. д. Такие «офисные» комплекты обычно интегрируются через общий пользовательский интерфейс, который поддерживает интегрированное управление документами и работает с файлами из домашнего каталога пользователя. (В офисной среде этот домашний каталог нередко размещается на удаленном файловом сервере.) В этом примере уровень обработки состоит из относительно большой коллекции программ, каждая из которых имеет довольно простые возможности обработки.

Уровень данных содержит программы, которые поддерживают фактические данные, на которых работают приложения. Важным свойством этого уровня является то, что данные часто являются **постоянными** (persistent), то есть даже если ни одно приложение не запущено, данные будут храниться где-то для следующего использования. В простейшем виде уровень данных состоит из файловой системы, но также часто используется полнофункциональная база данных.

Помимо простого хранения данных, уровень данных, как правило, также отвечает за поддержание согласованности данных в разных приложениях. Когда используются базы данных, поддержание согласованности означает, что метаданные, такие как таблица описания, ограничения входа и специфичные для приложения метаданные, также хранятся на этом уровне. На-

пример, в случае банка мы можем захотеть сгенерировать уведомление, когда задолженность клиента по кредитной карте достигает определенного значения. Этот тип информации может поддерживаться через триггер базы данных, который в соответствующий момент активирует обработчик для этого триггера.

Объектно-ориентированные и сервис-ориентированные архитектуры

В **объектно-ориентированных архитектурах** (object-based architectures) используется гораздо более свободная организация, показанная на рис. 2.5. По сути, каждый объект соответствует тому, что мы определили как компонент, и эти компоненты связаны через механизм вызова процедуры. В случае распределенных систем вызов процедуры также может происходить по сети, то есть вызывающий объект не обязательно должен выполняться на той же машине, что и вызываемый объект.

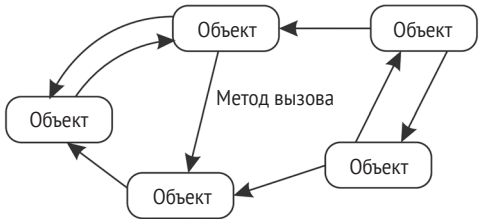


Рис. 2.5 ❖ Объектный архитектурный стиль

Объектно-ориентированные архитектуры привлекательны тем, что обеспечивают естественный способ *инкапсуляции* данных (называемых **состоянием объекта** (object's state)) и операций, которые можно выполнять над этими данными (которые называются **методами объекта** (object's methods)), в единый объект. **Интерфейс**, предлагаемый объектом, скрывает детали реализации, по сути, это означает, что мы, в принципе, можем считать объект полностью независимым от его среды. Как и в случае с компонентами, это также означает, что если интерфейс четко определен и оставлен без изменений, объект должен быть заменен на объект, имеющий точно такой же интерфейс.

Подобное разделение между интерфейсами и объектами, реализующими эти интерфейсы, позволяет нам размещать интерфейс на одной машине, в то время как сам объект находится на другой машине. Эта организация, показанная на рис. 2.6, обычно называется **распределенным объектом** (distributed object).

Когда клиент **связывается** с распределенным объектом, реализация интерфейса объекта, называемая **прокси** (proxy), загружается в адресное пространство клиента.

Прокси-сервер аналогичен «заглушке» клиента в системах с дистанционным вызовом процедур (Remote Procedure Call, RPC). Единственное, что он делает, – это вызывает метод упаковки сообщений и неупакованных ответных сообщений, чтобы вернуть результат вызова метода клиенту. Фактический объект находится на сервере, где он предлагает тот же интерфейс, что и на клиентском компьютере.

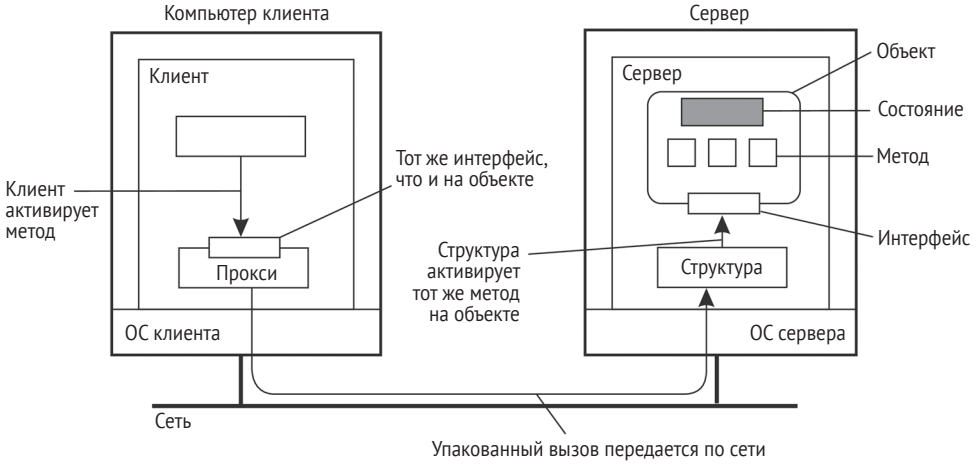


Рис. 2.6 ❖ Общая организация удаленного объекта с клиентским прокси

Входящие запросы на вызов сначала передаются на заглушку сервера, который распаковывает их для вызова методов на интерфейсе объекта на сервере. Заглушка сервера также отвечает за упаковку ответов и пересылку ответных сообщений на прокси на стороне клиента.

Заглушка на стороне сервера часто называется **структурой**, или **каркасом** (skeleton), поскольку она предоставляет простейшие средства, позволяющие промежуточному программному обеспечению сервера получать доступ к определенным пользователем объектам. На практике она часто содержит неполный код в виде специфического языкового класса, который должен быть дополнительно специализирован разработчиком.

Характерной, но несколько противоречивой особенностью большинства распределенных объектов является то, что их состояние не распределено: оно находится на одной машине. Только интерфейсы, реализованные объектом, становятся доступными на других машинах. Такие объекты также называются **удаленными объектами** (remote objects). В общем распределенном объекте само состояние может быть физически распределено по нескольким машинам, но это распределение также скрыто от клиентов за интерфейсами объекта.

Можно утверждать, что объектно-ориентированные архитектуры образуют основу для инкапсуляции сервисов в независимые единицы. Ключевым словом здесь является **инкапсуляция**: сервис в целом реализован как отдельный объект, хотя он может использовать и другие сервисы. Четко раз-

граничивая различные службы, чтобы они могли работать независимо, мы прокладываем дорогу к **сервис-ориентированным архитектурам**, обычно сокращенно обозначаемым как SOA (Service-Oriented Architectures).

В сервис-ориентированной архитектуре распределенное приложение или система по сути строятся как совокупность множества различных сервисов. Не все эти услуги могут принадлежать одной административной организации. Мы уже сталкивались с этим явлением при обсуждении облачных вычислений: вполне возможно, что организация, использующая свое бизнес-приложение, использует службы хранения, предлагаемые облачным провайдером. Эти услуги хранения логически полностью заключены в единое целое, интерфейс которого доступен для клиентов.

Конечно, хранилище является довольно простой услугой, но на ум легко приходят более сложные ситуации. Рассмотрим, например, интернет-магазин по продаже товаров, таких как электронные книги. Простая реализация, следующая за уровнем приложений, который мы обсуждали ранее, может состоять из приложения для обработки заказов, которое, в свою очередь, работает с локальной базой данных, содержащей электронные книги. Обработка заказа обычно включает в себя выбор товаров, регистрацию и проверку канала доставки (возможно, с использованием электронной почты), но также и обеспечение оплаты. Последняя может обрабатываться отдельной службой, управляемой другой организацией, в которую покупатель перенаправляется для оплаты, после чего организация электронной книги уведомляется, чтобы она могла завершить транзакцию.

Таким образом, мы видим, что проблема разработки распределенной системы частично связана с составом сервисов и обеспечением гармоничной работы этих сервисов. Действительно, эта проблема полностью аналогична вопросам интеграции корпоративных приложений, которые обсуждались в разделе 1.3. Важным остается и то, что каждый сервис предлагает четко определенный (программный) интерфейс. На практике это также означает, что каждый сервис предлагает свой собственный интерфейс, что, в свою очередь, делает состав услуг далеко не тривиальным.

Ресурсные архитектуры

По мере того как все большее число сервисов становилось доступным через интернет, а развитие распределенных систем с помощью композиции сервисов становилось все более важным, исследователи начали переосмысливать архитектуру распределенных систем на основе интернета. Одна из проблем с составом сервиса заключается в том, что соединение различных компонентов может легко превратиться в кошмар интеграции.

В качестве альтернативы можно также рассматривать распределенную систему как огромную совокупность ресурсов, которые индивидуально управляются компонентами. Ресурсы могут быть добавлены или удалены приложениями, а также могут быть извлечены или изменены. Этот подход в настоящее время получил широкое распространение в интернете и известен как **передача состояния представления** (Representational State

Transfer, REST) [Fielding, 2000]. Существует четыре ключевые характеристики того, что известно как архитектуры **RESTful** [Pautasso et al., 2008]:

1) ресурсы идентифицируются посредством единой схемы именования;
2) все сервисы предлагают один и тот же интерфейс, состоящий максимум из четырех операций, как показано на рис. 2.7;
3) сообщения, отправляемые в службу или из нее, полностью описывают сами себя;
4) после выполнения операции в службе этот компонент забывает обо всем, что касается вызывающего.

Последнее свойство также называется **выполнением без сохранения состояния** (stateless execution), концепцией, к которой мы вернемся в разделе 3.4.

Операция	Описание
PUT	Создать новый ресурс
GET	Получить состояние ресурса в некотором представлении
DELETE	Удалить ресурс
POST	Изменить ресурс путем передачи нового состояния

Рис. 2.7 ❖ Четыре операции, доступные в архитектурах RESTful

Чтобы проиллюстрировать, как RESTful может работать на практике, рассмотрим службу облачного хранения, такую как **Amazon Simple Storage Service** (**Amazon S3**). Служба Amazon S3, описанная в [Murty, 2008], поддерживает только два ресурса: объекты, которые по существу эквивалентны файлам, и сегменты, эквивалентные директориям. Нет концепции размещения «сегмент в сегменте». Объект с именем ObjectName, содержащийся в BucketName, упоминается с помощью следующего **унифицированного идентификатора ресурса** (Uniform Resource Identifier, URI):

http://BucketName.s3.amazonaws.com/ObjectName.

Чтобы создать сегмент памяти или объект для этого, приложение по сути отправляет запрос PUT с URI сегмента/объекта. В принципе, протокол, который используется с сервисом, является протоколом HTTP. Другими словами, это просто еще один HTTP-запрос, который впоследствии будет правильно интерпретирован S3. Если сегмент или объект уже существует, возвращается сообщение об ошибке HTTP.

Аналогичным образом, чтобы узнать, какие объекты содержатся в сегменте, приложение отправит запрос GET с URI этого сегмента. S3 вернет список имен объектов снова как обычный HTTP-ответ.

Архитектура RESTful стала популярной благодаря своей простоте. Тем не менее ведутся священные войны по поводу того, являются сервисы RESTful лучше, чем те, в которых сервисы определяются через сервис-специфические интерфейсы. В работе [Pautasso et al., 2008] дано сравнение двух подходов, и, как и следовало ожидать, оба они имеют свои преимущества и недостатки. В частности, простота архитектур RESTful может легко запретить простые

решения для сложных схем связи. Одним из примеров является то, в котором необходимы распределенные транзакции, что обычно требует, чтобы службы отслеживали состояние выполнения. С другой стороны, есть много примеров, в которых архитектуры RESTful идеально соответствуют простой схеме интеграции сервисов и в которых множество сервисных интерфейсов усложнит ситуацию.

Примечание 2.2 (дополнительно: по интерфейсам)

Очевидно, что услугу нельзя сделать проще или сложнее только из-за конкретного интерфейса, который она предлагает. Сервис предлагает функциональность, и в лучшем случае способ доступа к сервису определяется интерфейсом. Действительно, можно утверждать, что обсуждение RESTful по сравнению с сервис-специфическими интерфейсами – скорее, о прозрачности доступа. Чтобы лучше понять, почему так много обращается внимания на эту проблему, давайте поближе познакомимся с сервисом Amazon S3, который предлагает интерфейс REST, а также более традиционный интерфейс (называемый интерфейсом SOAP).

Операции с сегментами	Операции с объектами
ListAllMyBuckets	PutObjectInline
CreateBucket	PutObject
DeleteBucket	CopyObject
ListBucket	GetObject
GetBucketAccessControlPolicy	GetObjectExtended
SetBucketAccessControlPo licy	DeleteObject
GetBucketLoggingStatus	GetObjectAccessControlPolicy
SetBucketLoggingStatus	SetOjectAccsessContrrolPolicy

Рис. 2.8 ❖ Операции в интерфейсе Amazon S3 SOAP

Интерфейс SOAP состоит из приблизительно 16 операций, перечисленных на рис. 2.8. Однако если бы мы получили доступ к Amazon S3 с помощью библиотеки boto Python, у нас было бы около 50 доступных операций. Напротив, интерфейс REST предлагает очень мало операций, в основном те, которые перечислены на рис. 2.7. Откуда эти различия? Ответ, конечно, в пространстве параметров. В случае RESTful-архитектур приложение должно предоставлять все, что ему нужно, через параметры, которые оно передает одной из операций. В интерфейсе Amazon SOAP количество параметров для каждой операции обычно ограничено, и это, безусловно, имело бы место, если бы мы использовали библиотеку boto Python.

Придерживаясь принципов (чтобы мы могли избежать сложностей реального кода), предположим, что у нас есть интерфейсный сегмент (`bucket`), который предлагает операцию `create`, требующую входную строку, такую как `mybucket`, для создания сегмента с именем «mybucket». Обычно операция будет вызвана примерно следующим образом:

```
import bucket
bucket.create («mybucket»)
```

Однако в архитектуре RESTful вызов должен быть по существу закодирован в виде одной строки, такой как

```
PUT "http://mybucket.s3.amazonsws.com/"
```

Разница поразительна. Например, в первом случае многие синтаксические ошибки часто могут быть обнаружены уже во время компиляции, тогда как во втором случае проверка должна быть отложена до времени выполнения. Во-вторых, можно утверждать, что указание семантики операции намного проще с конкретными интерфейсами, чем с теми, которые предлагают только общие операции. С другой стороны, с общими операциями изменения гораздо легче приспособить, поскольку они обычно включают изменение структуры строк, которые кодируют то, что фактически и требуется.

Архитектура публикация-подписка

Поскольку системы продолжают расти, а процессы могут легче присоединяться или выходить, становится важным иметь архитектуру, в которой зависимости между процессами становятся максимально свободными. Большой класс распределенных систем принял архитектуру, в которой существует сильное разделение между *обработкой* и *координацией*. Идея состоит в том, чтобы рассматривать систему как совокупность автономно работающих процессов. В этой модели **координация** (coordination) включает в себя взаимодействие и координацию между процессами. Она образует клей, который связывает действия, выполняемые процессами, в единое целое [Gelernter and Carriero, 1992].

В работе [Cabri et al., 2000] авторами предоставлена таксономия координационных моделей, которые могут в равной степени применяться ко многим типам распределенных систем. Слегка адаптируя их терминологию, мы различаем модели по двум разным измерениям, вре́менному и референтному (ссылочному), как показано на рис. 2.9.

	Временно связан	**Временно развязан**
Ссылочный связанный	Прямая	Почтовый ящик
Ссылочный развязанный	Основанная на событиях	Разделенное пространство данных

Рис. 2.9 ❖ Примеры различных форм координации

Когда процессы связаны во времени и по ссылкам, координация происходит прямым образом, называемым **прямой координацией** (direct coordination). Ссылочная связь обычно появляется в форме явной ссылки в сообщении. Например, процесс может взаимодействовать, только если ему известно имя или идентификатор других процессов, с которыми он хочет обмениваться информацией.

Временна́я связь означает, что процессы, которые взаимодействуют, должны быть запущены и работать. В реальной жизни разговор по мобильному телефону (при условии что у мобильного телефона есть только один владелец) является примером прямого общения.

Другой тип координации происходит, когда процессы временно разделены, но связаны друг с другом, что мы называем **координацией почтового ящика** (mailbox coordination). В этом случае, чтобы обеспечить обмен данными, нет необходимости одновременно выполнять два процесса обмена данными. Вместо этого общение происходит путем помещения сообщений в (возможно, общий) почтовый ящик. Поскольку необходимо явно обратиться к почтовому ящику, в котором будут храниться сообщения, подлежащие обмену, существует ссылочная связь.

Комбинация систем со ссылочной развязкой и временной связью образует группу моделей для **координации на основе событий** (event-based coordination). В системах с разделенными ссылками процессы не знают друг друга явно. Единственное, что может сделать процесс, – это **опубликовать уведомление** (publish a notification), описывающее возникновение события (например, что он хочет координировать действия или что он просто дал некоторые интересные результаты). Предполагая, что уведомления приходят во всех видах и формах, процессы могут **подписаться** (**subscribe**) на определенный вид уведомлений (см. также [Mühl et al., 2006]). В идеальной координационной модели, основанной на событиях, публикуемое уведомление будет доставлено именно тем процессам, которые подписались на него. Однако, как правило, требуется, чтобы подписчик работал в момент публикации уведомления.

Хорошо известная координационная модель представляет собой комбинацию процессов, разделенных по ссылкам и по времени, что приводит к так называемому **общему пространству данных** (shared data space). Основная идея заключается в том, что процессы взаимодействуют исключительно через **кортежи** (tuples), которые представляют собой структурированные записи данных, состоящие из нескольких полей, очень похожих на строки в таблице базы данных. Процессы могут помещать любой тип кортежа в общее пространство данных. Чтобы извлечь кортеж, процесс предоставляет шаблон поиска, который сопоставляется с кортежами. Любой соответствующий кортеж возвращается.

Таким образом, общие пространства данных реализуют механизм ассоциативного поиска для кортежей. Когда процесс хочет извлечь кортеж из пространства данных, он указывает (некоторые из) значения полей, в которых заинтересован. Любой кортеж, который соответствует этой спецификации, затем удаляется из пространства данных и передается процессу.

Общие пространства данных часто объединяются с координацией на основе событий: процесс подписывается на определенные кортежи, предоставляя шаблон поиска; когда процесс вставляет кортеж в пространство данных, соответствующие подписчики уведомляются. В обоих случаях мы имеем дело с архитектурой публикации-подписки, и действительно, ключевой характеристикой является то, что процессы не имеют явной ссылки друг на друга. Разница между чисто архитектурным стилем, основанным на событиях, и общим пространством данных показана на рис. 2.10. Мы также продемонстрировали абстракцию механизма сопоставления издателей и подписчиков, известного как **шина событий** (event bus).

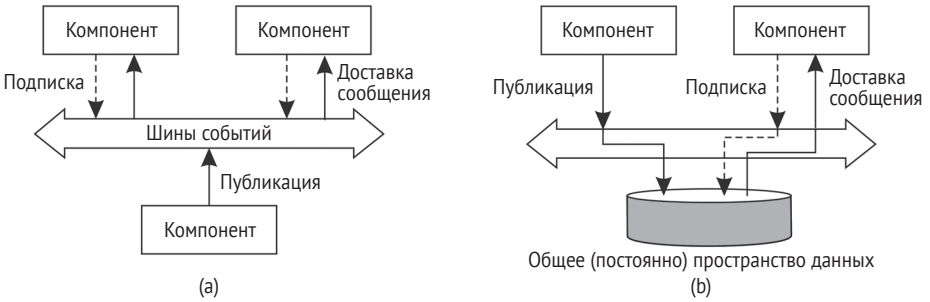


Рис. 2.10 ❖ Стиль пространства данных:
a) основанный на событиях; b) общий архитектурный

Примечание 2.3 (пример: пространства кортежей Linda)

Чтобы конкретизировать ситуацию, мы подробнее рассмотрим модель программирования Linda, разработанную в 1980-х годах [Carriero and Gelernter, 1989]. Общее пространство данных в Linda известно как пространство кортежей, которое, по существу, поддерживает три операции:

- `in(t)`: удалить кортеж, соответствующий шаблону `t`;
- `rd(t)`: получить копию кортежа, соответствующего шаблону `t`;
- `out(t)`: добавить кортеж `t` в пространство кортежей.

Обратите внимание, что если процесс будет вызывать `out(t)` дважды подряд, мы найдем, что две копии кортежа `t` были сохранены. Формально пространство кортежей всегда моделируется как *мультимножество*. И `in`, и `rd` являются операциями блокировки: вызывающий будет *заблокирован*, пока не будет найден соответствующий кортеж или он не станет доступен.

Рассмотрим очень простое приложение для микроблогов, в котором сообщения помечаются именем автора и темой, за которой следует короткая строка. Каждое сообщение моделируется как кортеж `<string, string, string>`, где первая строка содержит название автора, вторая строка представляет тему, а третья – фактический контент. Предполагая, что мы создали общее пространство данных под названием MicroBlog, на рис. 2.11 показано, как Алиса и Боб могут отправлять сообщения в это пространство и как Чак может выбрать (случайно выбранное) сообщение. Мы пропустили некоторый код для ясности. Обратите внимание, что ни Алиса, ни Боб не знают, кто будет читать их сообщения.

В первой строке каждого фрагмента кода процесс ищет пространство кортежей с именем «MicroBlog». Боб публикует три сообщения: два на тему `distsys` и одно на тему `gtcn`. Алиса публикует два сообщения, по одному на каждую тему. Наконец, Чак читает три сообщения: одно от Боба `distsys` и одно `gtcn`, а другое от Алисы `gtcn`.

Очевидно, что есть много возможностей для улучшения. Например, мы должны убедиться, что Алиса не может публиковать сообщения под именем Боба. Однако сейчас важно отметить, что, предоставляя только теги, читатель, такой как Чак, в состоянии забрать сообщения без прямой ссылки на постер. В частности, Чак мог также прочитать случайно выбранное сообщение на тему `disysys` через утверждение

```
t = blog_rd((str, "distsys", str))
```

Мы оставляем читателю в качестве упражнения расширение фрагментов кода таким образом, чтобы вместо случайного было выбрано *следующее* сообщение.

```
blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]

blog._out(("bob","distsys","I am studying chap 2"))
blog._out(("bob","distsys","The linda example's pretty simple"))
blog._out(("bob","gtcn","Cool book!"))
```

(a) Код Боба для создания микроблога и отправки двух сообщений

```
blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]

blog._out(("alice","gtcn","This graph theory stuff is not easy"))
blog._out(("alice","distsys","I like systems more than graphs"))
```

(b) Код Алисы для создания микроблога и отправки двух сообщений

```
1 blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]

t1 = blog._rd(("bob","distsys",str))
t2 = blog._rd(("alice","gtcn",str))
t3 = blog._rd(("bob","gtcn",str))
```

(c) Чак читает сообщение из микроблога Боба и Алисы

Рис. 2.11 ❖ Простой пример использования общего пространства данных

Важным аспектом систем публикации-подписки является то, что общение происходит путем описания событий, в которых заинтересован подписчик. Как следствие именование играет решающую роль. Мы вернемся к именованию позже, но на данный момент важной проблемой является то, что во многих случаях элементы данных отправителями и получателями явно не идентифицируются.

Давайте сначала предположим, что события описываются рядом **атрибутов** (attributes). Считается, что уведомление, описывающее событие, **публикуется** (published), когда оно становится доступным для чтения другими процессами. Для этого необходимо передать **подписку** (subscription) промежуточному программному обеспечению, содержащему описание события, в котором заинтересован подписчик. Такое описание обычно состоит из нескольких пар (атрибут, значение), что является общим для так называемых **систем на основе тематической публикации-подписки** (topic-based publish-subscribe systems).

В качестве альтернативы в системах, основанных на **контенте публикации-подписки** (content-based publish-subscribe systems), подписка может также состоять из пар (*атрибут, диапазон*). В этом случае ожидается, что указанный атрибут будет принимать значения в указанном диапазоне. Иногда описания могут быть даны с использованием всевозможных предикатов, сформулированных по атрибутам, которые очень похожи по своей природе на SQL-подобные запросы в случае реляционных баз данных. Очевидно, что чем сложнее описание, тем сложнее будет проверить, соответствует ли событие описанию.

В настоящее время мы сталкиваемся с ситуацией, в которой подписки должны **сопоставляться** (matched) с уведомлениями, как показано на

рис. 2.12. Во многих случаях событие фактически соответствует доступу данных. В этом случае при успешном сопоставлении возможны два сценария. В первом случае промежуточное программное обеспечение может принять решение переслать опубликованное уведомление вместе со связанными данными своему текущему набору подписчиков, то есть процессам с соответствующей подпиской. В качестве альтернативы промежуточное программное обеспечение также может пересылать только уведомление, когда подписчики могут выполнить операцию чтения, чтобы извлечь связанный элемент данных.

В тех случаях, когда данные, связанные с событием, немедленно передаются подписчикам, промежуточное программное обеспечение обычно не предлагает хранение данных. Хранение либо явно обрабатывается отдельной службой, либо является обязанностью подписчиков. Другими словами, у нас есть ссылочная, но временно связанная система.

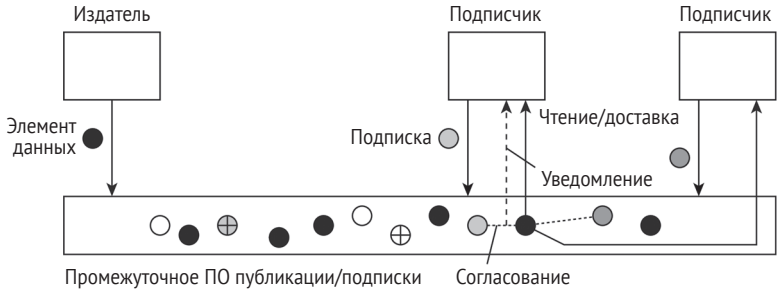


Рис. 2.12 ❖ Принцип обмена элементами данных между издателями и подписчиками

Эта ситуация отличается от той, когда отправляются уведомления, так что подписчикам необходимо явно прочитать связанные данные. Промежуточное ПО должно обязательно хранить элементы данных. В этих ситуациях существуют дополнительные операции для управления данными. Также возможно прикрепить аренду к элементу данных так, чтобы по истечении срока аренды элемент данных автоматически удалялся.

События могут легко усложнить обработку подписок. Чтобы проиллюстрировать это, рассмотрим, например, подписку: «Уведомлять, когда комната ZI.1060 свободна и дверь открыта». Как правило, распределенная система, поддерживающая такие подпрограммы, может быть реализована путем размещения независимых датчиков для контроля занятости помещения (например, датчиков движения) и датчиков для регистрации состояния дверного замка. Следуя изложенному выше подходу, нам нужно будет объединить такие примитивные события в публикуемый элемент данных, на который затем могут подписаться процессы. Составление событий оказывается сложной задачей, особенно когда примитивные события генерируются из источников, разделенных по распределенной системе.

Ясно, что в таких системах, как публикация-подписка, важнейшей проблемой является эффективная и масштабируемая реализация сопоставле-

ния подписок на уведомления. Внешне архитектура «публикация-подписка» имеет большой потенциал для построения крупномасштабных распределенных систем ввиду сильной развязки процессов. С другой стороны, разработка масштабируемых реализаций без потери этой независимости не является тривиальным упражнением, особенно в случае основанных на контенте систем публикации-подписки.

2.2. Организация промежуточного программного обеспечения

В предыдущем разделе мы обсудили ряд архитектурных стилей, которые часто используются в качестве общих рекомендаций для построения и организации распределенных систем. Давайте теперь рассмотрим фактическую организацию промежуточного программного обеспечения, то есть независимую от общей организации распределенной системы или приложения. В частности, существует два важных типа *шаблонов проектирования*, которые часто применяются для организации промежуточного программного обеспечения: оболочки и перехватчики. Каждый из них нацелен на разные проблемы, но при этом они решают одну и ту же задачу для промежуточного программного обеспечения: достижение открытости (как мы обсуждали в разделе 1.2). Можно, однако, утверждать, что максимальная открытость достигается, когда мы можем создавать промежуточное программное обеспечение во время выполнения. Мы кратко обсудим конструкции, основанные на компонентах, как популярное средство для того, что в работе [Parlavantzas and Coulson, 2007] называется модифицированным промежуточным программным обеспечением.

Упаковщики

При построении распределенной системы из существующих компонентов мы сразу же сталкиваемся с фундаментальной проблемой: интерфейсы, предлагаемые устаревшим компонентом, скорее всего, не подходят для всех приложений. В разделе 1.3 мы обсуждали, как можно интегрировать корпоративные приложения через промежуточное ПО в качестве посредника в коммуникации, но там мы все еще неявно предполагали, что, в конце концов, компоненты могут быть доступны через их собственные интерфейсы.

Уппаковщик (wrapper) или **адаптер** (adapter) – это специальные компоненты, которые предлагают приемлемый для клиентского приложения интерфейс, чьи функции преобразуются в функции, доступные этому компоненту. По сути, это решает проблему несовместимых интерфейсов (см. также [Gamma et al., 1994]).

Первоначально узко определяемые в контексте объектно-ориентированного программирования, упаковщики в контексте распределенных систем

представляют собой гораздо больше, чем простые преобразователи интерфейса. Например, объектный адаптер – это компонент, который позволяет приложениям вызывать удаленные объекты, хотя эти объекты могли быть реализованы как комбинация библиотечных функций, работающих с таблицами реляционной базы данных.

В качестве другого примера пересмотрите сервис хранения Amazon S3. Теперь доступно два типа интерфейсов: один придерживается архитектуры RESTful, другой – более традиционного подхода. Для интерфейса RESTful клиенты будут использовать протокол HTTP, по сути, обмениваясь данными с традиционным веб-сервером, который теперь действует как адаптер для фактической службы хранения, путем частичного разбора входящих запросов и последующей передачи их на специализированные серверы, внутренние для S3.

Упаковщики всегда играли важную роль в расширении систем с помощью существующих компонентов. Расширяемость, которая имеет решающее значение для достижения открытости, использовалась для добавления оболочек по мере необходимости. Другими словами, если приложение A управляет данными, которые необходимы приложению B, один из подходов заключается в разработке специальной оболочки для B, чтобы он мог иметь доступ к данным A. Ясно, что этот подход плохо масштабируется: с N приложениями нам теоретически нужно разработать $N(N-1) = \mathcal{O}(N^2)$ оболочек.

Сокращение количества оболочек обычно осуществляется через промежуточное ПО. Один из способов сделать это – реализовать так называемый **брокер** (broker), являющийся логически централизованным компонентом, который обрабатывает все обращения между различными приложениями. Часто используемый тип – это **брокер сообщений** (message broker), технические характеристики которого мы обсуждаем в разделе 4.3. В случае посредника сообщений приложения просто отправляют посреднику запросы, содержащие информацию о том, что им нужно. Брокер, имеющий знания обо всех соответствующих приложениях, связывается с этими приложениями, возможно, объединяет и преобразует ответы и возвращает результат в исходное приложение. В принципе, поскольку брокер предлагает один интерфейс для каждого приложения, нам теперь нужно не более $2N = \mathcal{O}(N)$ упаковщиков вместо $\mathcal{O}(N^2)$. Эта ситуация показана на рис. 2.13.

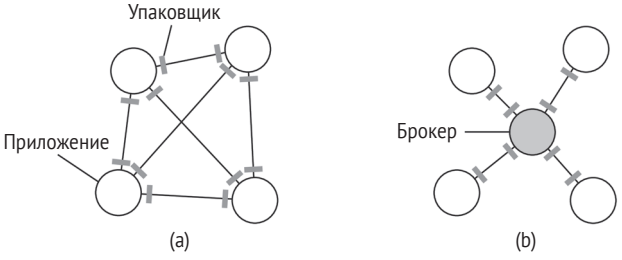


Рис. 2.13 ❖ a) Требование к приложению – иметь упаковщик для каждого брокера; b) уменьшение числа оболочек при использовании брокера

Перехватчики

Концептуально **перехватчик** (interceptor) – это не что иное, как программная конструкция, которая нарушает обычный поток управления и позволяет выполнять другой (специфичный для приложения) код. Перехватчики являются основным средством адаптации промежуточного программного обеспечения к конкретным потребностям приложения. Как таковые они играют важную роль в открытии промежуточного программного обеспечения.

Чтобы сделать перехватчики общими, могут потребоваться значительные усилия при реализации, как показано в [Schmidt et al., 2000], и не ясно, следует ли отдавать предпочтение общности в таких случаях перед ограниченной применимостью и простотой. Кроме того, во многих случаях наличие лишь ограниченных средств перехвата улучшит управление программным обеспечением и распределенной системой в целом.

Чтобы конкретизировать ситуацию, рассмотрим перехват, поддерживаемый во многих объектных распределенных системах. Основная идея проста: объект A может вызывать метод, который принадлежит объекту B, в то время как последний находится на другом компьютере. Как мы подробно объясним позже в этой книге, такой вызов удаленного объекта выполняется в три этапа:

1) объекту A предлагается локальный интерфейс, который точно такой же, как интерфейс, предлагаемый объектом B. Объект A вызывает метод, доступный в этом интерфейсе;
2) вызов A преобразуется в общий вызов объекта, который становится возможным благодаря общему интерфейсу вызова объекта, предлагаемому промежуточным программным обеспечением на машине, где находится A;
3) наконец, общий вызов объекта преобразуется в сообщение, которое отправляется через сетевой интерфейс транспортного уровня, предлагаемый локальной операционной системой A.

Эта схема показана на рис. 2.14. После первого шага вызов `B.doit(val)` преобразуется в общий вызов, такой как `invoke(B, &doit, val)`, со ссылкой на метод B и параметры, которые сопровождают вызов. Теперь представьте, что объект B реплицируется. В этом случае должна быть вызвана каждая реплика. Это именно тот момент, когда перехват может помочь. Что **перехватчик уровня запроса** (request-level interceptor) будет делать, так это просто вызывать `invoke(B, &doit, val)` для каждой из реплик. Прелесть всего этого в том, что объект A не должен знать о репликации B, а также промежуточному программному обеспечению объекта не нужны специальные компоненты, которые имеют дело с этим реплицированным вызовом. Только перехватчик уровня запросов, который может быть добавлен к промежуточному программному обеспечению, должен знать о репликации B.

В конечном итоге вызов к удаленному объекту должен быть отправлен по сети. На практике это означает, что должен быть вызван интерфейс обмена сообщениями, предлагаемый локальной операционной системой. На этом уровне **перехватчик уровня сообщения** (message-level interceptor) может

помочь в передаче вызова целевому объекту. Например, представьте, что параметр `val` на самом деле соответствует огромному массиву данных. В этом случае может быть целесообразно разбить данные на более мелкие части, чтобы они снова были собраны в месте назначения. Такая фрагментация может улучшить производительность или надежность. Опять же, промежуточному программному обеспечению не нужно знать об этой фрагментации; перехватчик нижнего уровня будет прозрачно обрабатывать остальную часть связи с локальной операционной системой.

Рис. 2.14 ❖ Использование перехватчиков для обработки вызовов удаленных объектов

Модифицируемое промежуточное ПО

То, что предлагают упаковщики и перехватчики, – это средство для расширения и адаптации промежуточного программного обеспечения. Необходимость адаптации обусловлена тем фактом, что среда, в которой выполняются распределенные приложения, постоянно меняется. Изменения включают в себя, среди прочего, мобильность, сильное расхождение в качестве обслуживания сетей, отказ оборудования и разрядку аккумулятора.

Вместо того чтобы назначать приложения ответственными за реагирование на изменения, эта задача помещается в промежуточное ПО. Более того, по мере увеличения размера распределенной системы изменение ее частей редко может быть выполнено путем ее временного отключения. Что нужно, так это иметь возможность вносить изменения на лету.

Эти сильные влияния окружающей среды привели многих разработчиков промежуточного программного обеспечения к разработке *адаптивного программного обеспечения*. Промежуточное программное обеспечение может быть не только адаптивным, но и *модифицируемым*, и мы должны быть в состоянии преднамеренно изменять его, не снижая его эффективности [Parlavantzas and Coulson, 2007]. В этом контексте можно предположить, что перехватчики предлагают средства для адаптации стандартного потока управления. Замена компонентов программного обеспечения во время выполнения – пример изменения системы. И действительно, возможно, один из самых популярных подходов к изменяемому промежуточному программному обеспечению – это динамическое построение промежуточного программного обеспечения, исходя из компонентов.

Компонентный дизайн ориентирован на поддержку изменчивости посредством композиции. Система может быть настроена статически во время разработки или динамически во время выполнения. Последнее требует поддержки позднего связывания, метода, который был успешно применен в средах языка программирования, но также и для операционных систем, где модули могут быть загружены и выгружены по желанию. В настоящее время ведутся исследования для автоматического выбора наилучшей реализации компонента во время выполнения [Yellin, 2003], но, опять же, процесс остается сложным для распределенных систем, особенно если учесть, что замена одного компонента требует точно знать, какое влияние эта замена будет оказывать на другие компоненты. Во многих случаях компоненты менее независимы, чем можно было бы подумать.

Дело в том, что для обеспечения динамических изменений в программном обеспечении, составляющем промежуточное программное обеспечение, нам необходима как минимум базовая поддержка для загрузки и выгрузки компонентов во время выполнения. Кроме того, для каждого компонента необходимы явные спецификации интерфейсов, которые он предлагает, а также требуемых интерфейсов. Если состояние поддерживается между вызовами к компоненту, то необходимы дополнительные специальные меры. В целом должно быть ясно, что организация промежуточного программного обеспечения, которое должно быть модифицируемым, требует особого внимания.

2.3. Системная архитектура

Теперь, когда мы кратко обсудили некоторые общепринятые архитектурные стили, давайте посмотрим, как много распределенных систем фактически организовано, с учетом того, где расположены компоненты программного обеспечения. Выбор программных компонентов, их взаимодействия и их размещения приводит к примеру программной архитектуры, также известной как **системная архитектура** (system architecture) [Bass et al., 2003]. Мы обсудим централизованные и децентрализованные организации, а также различные гибридные формы.

Централизованные организации

Несмотря на отсутствие консенсуса по многим вопросам распределенных систем, есть одна проблема, с которой согласны многие исследователи и практики: взгляд на проблему с точки зрения клиентов, запрашивающих услуги у серверов, помогает понять сложность распределенных систем и управлять ими [Saltzer and Kaashoek, 2009]. Далее мы сначала рассмотрим простую уровневую организацию, а затем многоуровневые организации.

Простая архитектура клиент-сервер

В базовой модели клиент-сервер процессы в распределенной системе делятся на две (возможно, перекрывающиеся) группы. **Сервер** – это процесс, реализующий конкретную службу, например службу файловой системы или службу базы данных. **Клиент** – это процесс, который запрашивает услугу у сервера, отправляя ему запрос и впоследствии ожидая ответа сервера. Это взаимодействие клиент–сервер, также известное как **поведение «запрос-ответ»** (request-reply behavior), показано на рис. 2.15 в форме диаграммы последовательности сообщений.

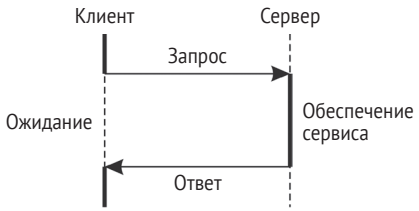


Рис. 2.15 ❖ Общее взаимодействие между клиентом и сервером

Когда базовая сеть достаточно надежна, как во многих локальных сетях, связь между клиентом и сервером может быть реализована с помощью простого бесконтактного протокола. В этих случаях, когда клиент запрашивает услугу, он просто упаковывает сообщение для сервера, идентифицируя нужную ему услугу, вместе с необходимыми входными данными. Затем сообщение отправляется на сервер. Последний, в свою очередь, всегда будет ждать входящего запроса, обработает его и упакует результаты в ответное сообщение, которое затем отправит клиенту.

Использование бесконтактного протокола имеет очевидное преимущество, заключающееся в его эффективности. До тех пор, пока сообщения не будут потеряны или повреждены, протокол запрос-ответ, только что созданный, работает отлично. К сожалению, сделать протокол устойчивым к случайным сбоям передачи нетривиально. Единственное, что мы можем сделать, – это, возможно, позволить клиенту повторно отправить запрос, когда не поступит ответное сообщение. Однако проблема заключается в том, что клиент не может определить, было ли потеряно исходное сообщение с запросом или про-

изошла ошибка при передаче ответа. Если ответ был потерян, то повторная отправка запроса может привести к выполнению операции дважды. Если бы операция была чем-то вроде «перечисления 10 000 долларов с моего банковского счета», тогда было бы лучше, если бы мне просто сообщали об ошибке.

С другой стороны, если операция «скажи мне, сколько денег у меня осталось», было бы вполне приемлемо отправить запрос повторно. Когда операция может повторяться несколько раз без вреда, она называется **идемпотентной** (idempotent). Поскольку некоторые запросы являются идемпотентными, а другие – нет, должно быть ясно, что не существует единого решения для работы с потерянными сообщениями. Мы отложим подробное обсуждение обработки ошибок при передаче до раздела 8.3.

В качестве альтернативы многие клиент-серверные системы используют надежный протокол, ориентированный на соединение. Хотя это решение не совсем подходит в локальной сети из-за относительно низкой производительности, оно прекрасно работает в глобальных системах, в которых связь по своей природе ненадежна. Например, практически все протоколы интернет-приложений основаны на надежных соединениях TCP/IP. В этом случае, когда клиент запрашивает услугу, он сначала устанавливает соединение с сервером перед отправкой запроса. Сервер обычно использует это же соединение для отправки ответного сообщения, после чего соединение разрывается. Проблема может заключаться в том, что установка и разрыв соединения являются относительно дорогостоящими, особенно когда сообщения запроса и ответа невелики.

Модель клиент-сервер была предметом многочисленных дискуссий и споров на протяжении многих лет. Одним из основных вопросов было то, как провести четкое различие между клиентом и сервером. Неудивительно, что зачастую нет четкого разграничения. Например, сервер для распределенной базы данных может непрерывно действовать как клиент, поскольку он пересылает запросы на разные файловые серверы, отвечающие за реализацию таблиц базы данных. В таком случае сам сервер базы данных обрабатывает только запросы.

Многоуровневая архитектура

Разделение на три логических уровня, о которых говорилось выше, предлагает ряд возможностей для физического распределения клиент-серверного приложения на нескольких машинах. Самая простая организация – иметь только два типа машин:

1) клиентский компьютер, содержащий только программы, реализующие уровень (часть) пользовательского интерфейса;
2) серверный компьютер, содержащий остальные программы, то есть программы, реализующие уровень обработки и данных.

В этой организации все обрабатывается сервером, пока клиент по сути не более чем тупой терминал, возможно, только с удобным графическим интерфейсом. Есть, однако, много других возможностей. Как объяснено в разделе 2.1, многие распределенные приложения делятся на три уровня: 1) уровень пользовательского интерфейса, 2) уровень обработки и 3) уровень данных.

Один из подходов к организации клиентов и серверов заключается в распределении этих уровней по разным машинам, как показано на рис. 2.16 (см. также Umar [1997]). В качестве первого шага мы делаем различие только между двумя типами машин: клиентскими и серверными машинами, что приводит к тому, что также называется (**физическая**) **двухуровневая архитектура** ((physically) two-tiered architecture).

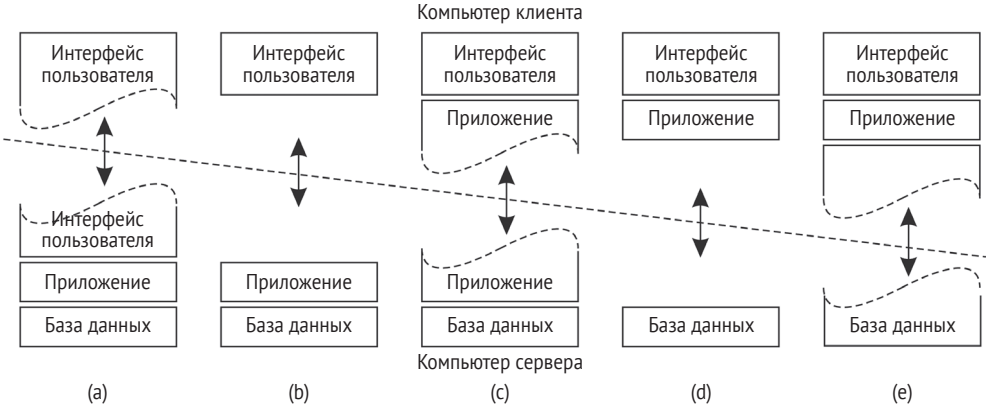


Рис. 2.16 ❖ Клиент-серверные организации в двухуровневой архитектуре

Одна из возможных организаций состоит в том, чтобы на клиентском компьютере была только зависимая от терминала часть пользовательского интерфейса, как показано на рис. 2.16a, и приложения могли удаленно контролировать представление своих данных. Альтернативой является размещение всего программного обеспечения пользовательского интерфейса на стороне клиента, как показано на рис. 2.16b. В таких случаях мы, по существу, делим приложение на графический интерфейс, который обменивается данными и остальной частью приложения (находящейся на сервере) через специальный протокол приложения. В этой модели интерфейс (клиентское программное обеспечение) не выполняет никакой обработки, кроме необходимой для представления интерфейса приложения.

Продолжая эту линию рассуждений, мы также можем переместить часть приложения в интерфейс, как показано на рис. 2.16c. Пример, где это имеет смысл, – это когда приложение использует форму, которую необходимо заполнить полностью, прежде чем ее можно будет обработать. Затем передний конец может проверить правильность и согласованность формы, а также при необходимости взаимодействовать с пользователем. Другим примером организации, показанной на рис. 2.16c, является текстовый процессор, в котором основные функции редактирования выполняются на стороне клиента, где они работают с локально кешированными данными или данными в памяти, но где используются расширенные средства поддержки, такие как проверка орфографии и грамматики, выполняющиеся на стороне сервера. Во многих клиент-серверных средах особенно популярны организации, показанные на рис. 2.16d и рис. 2.16e. Эти организации сред используются там, где кли-

ентский компьютер представляет собой ПК или рабочую станцию, подключенную через сеть к распределенной файловой системе или базе данных. По сути, большая часть приложения выполняется на клиентском компьютере, но все операции над файлами или записями базы данных отправляются на сервер. Например, многие банковские приложения выполняются на компьютере конечного пользователя, где пользователь готовит транзакции и т. п. После завершения приложение связывается с базой данных на сервере банка и загружает транзакции для дальнейшей обработки. На рис. 2.16e представлена ситуация, когда локальный диск клиента содержит часть данных. Например, при просмотре веб-страниц клиент может постепенно создать огромный кеш на локальном диске из последних проверенных веб-страниц.

Примечание 2.4 (дополнительная информация: существует ли лучшая организация?)

Мы отмечаем, что наблюдалась сильная тенденция отходить от конфигураций, показанных на рис. 2.16d и рис. 2.16e, в тех случаях, когда клиентское программное обеспечение размещается на компьютерах конечных пользователей. Вместо этого большая часть обработки и хранения данных обрабатывается на стороне сервера. Причина этого проста: хотя клиентские машины выполняют много задач, ими также и сложнее управлять. Наличие большей функциональности на клиентском компьютере означает, что широкому кругу конечных пользователей потребуется возможность обрабатывать это программное обеспечение. Это означает, что необходимо приложить больше усилий для обеспечения устойчивости программного обеспечения к поведению конечного пользователя. Кроме того, программное обеспечение на стороне клиента зависит от базовой платформы клиента (т. е. операционной системы и ресурсов), что может легко означать необходимость поддержки нескольких версий. С точки зрения системного управления наличие так называемых «**жирных клиентов**» (fat clients) не является оптимальным. «**Тощие клиенты**» (thin clients), представленные организациями, показанными на рис. 2.16a–c, намного проще, возможно, за счет менее сложных пользовательских интерфейсов и воспринимаемой клиентом производительности.

Означает ли это конец жирных клиентов? Ни в малой степени. Во-первых, существует множество приложений, для которых организация с толстым клиентом часто остается лучшей. Мы уже упоминали об офисных комплектах, но также и для многих мультимедийных приложений требуется, чтобы обработка выполнялась на стороне клиента. Во-вторых, с появлением усовершенствованной технологии просмотра веб-страниц теперь стало намного проще динамически размещать и управлять программным обеспечением на стороне клиента, просто загружая (иногда очень сложные) сценарии. В сочетании с тем фактом, что этот тип программного обеспечения на стороне клиента работает в хорошо определенных широко развернутых средах и, таким образом, зависимость от платформы представляет собой гораздо меньшую проблему, мы видим, что контраргумент сложности управления часто более не действителен.

Наконец, обратите внимание, что отказ от жирных клиентов не означает, что нам больше не нужны распределенные системы. Напротив, мы по-прежнему видим, что серверные решения становятся все более распределенными, поскольку один сервер заменяется несколькими серверами, работающими на разных компьютерах. Облачные вычисления являются хорошим примером в этом случае: вся серверная часть выполняется в центрах обработки данных и, как правило, на нескольких серверах.

Различая только клиентские и серверные машины, как мы делали до сих пор, мы упускаем тот момент, когда серверу иногда может потребоваться действовать как клиент, как показано на рис. 2.17, приводя к (**физической**) **трехуровневой архитектуре** ((physically) three-tiered architecture).

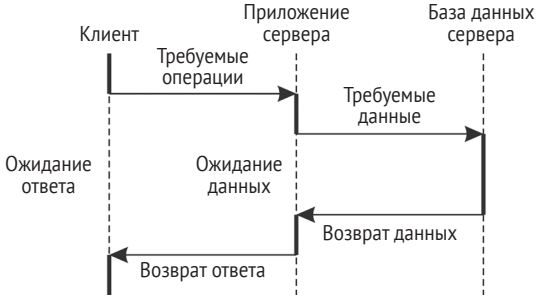


Рис. 2.17 ❖ Пример сервера, выступающего в роли клиента

В этой архитектуре программы, которые являются частью уровня обработки, традиционно выполняются отдельным сервером, но могут дополнительно частично распределяться по клиентским и серверным машинам. Типичный пример использования трехуровневой архитектуры – обработка транзакций. Отдельный процесс, называемый монитором обработки транзакций, координирует все транзакции на разных серверах данных.

Другой, но совершенно отличный от первого пример когда мы часто видим трехуровневую архитектуру в организации веб-сайтов. В этом случае веб-сервер выступает в качестве точки входа на сайт, передавая запросы на сервер приложений, где и происходит фактическая обработка. Этот сервер приложений, в свою очередь, взаимодействует с сервером базы данных. Например, сервер приложений может отвечать за выполнение кода для проверки доступного инвентаря некоторых товаров, предлагаемых электронным книжным магазином. Для этого может потребоваться взаимодействие с базой данных, содержащей необработанные данные инвентаризации.

Децентрализованные организации: одноранговые системы

Многоуровневые клиент-серверные архитектуры являются прямым следствием разделения распределенных приложений на пользовательский интерфейс, компоненты обработки и компоненты управления данными. Различные уровни соответствуют напрямую логической организации приложений. Во многих бизнес-средах распределенная обработка эквивалентна организации клиент-серверного приложения в виде многоуровневой архитектуры. Мы называем этот тип распределения **вертикальным распределением** (vertical distribution). Характерной особенностью вертикального распределения является то, что оно достигается путем размещения логически разных компонентов на разных машинах. Этот термин связан с концепцией вертикальной фрагментации, используемой в распределенных реляцион-

ных базах данных, где он означает, что таблицы разбиваются по столбцам и впоследствии распределяются по нескольким машинам [Özsu and Valduriez, 2011].

Опять же, с точки зрения системного управления, вертикальное распределение может помочь: функции логически и физически разделены на несколько машин, где каждая машина приспособлена для определенной группы функций. Однако вертикальное распределение – это только один из способов организации клиент-серверных приложений. В современных архитектурах часто учитывается распределение клиентов и серверов, которое мы называем **горизонтальным распределением** (horizontal distribution). В этом типе распределения клиент или сервер может быть физически разделен на логически эквивалентные части, но каждая часть работает со своей собственной частью полного набора данных, балансируя таким образом нагрузку. В данном разделе мы рассмотрим класс современных системных архитектур, которые поддерживают горизонтальное распределение, известный как **одноранговые системы** (peer-to-peer systems).

С точки зрения верхнего уровня, процессы, которые составляют одноранговую систему, все равны. Это означает, что функции, которые должны выполняться, представлены каждым процессом, составляющим распределенную систему. Как следствие большая часть взаимодействия между процессами симметрична: каждый процесс будет действовать как клиент и как сервер одновременно (что также называется действием **слуги** (servant)).

Учитывая это симметричное поведение, в одноранговых архитектурах задаются вопросом, как организовать процессы в **оверлейной сети** (overlay network) [Tarkoma, 2010] – сети, в которой узлы образованы процессами, а ссылки представляют возможные каналы связи (которые часто реализуются как соединения TCP). Узел может не иметь возможности напрямую связываться с произвольным другим узлом, и требуется отправлять сообщения через доступные каналы связи. Существуют два типа оверлейных сетей: структурированные и неструктурированные. Эти два типа подробно рассмотрены в [Lua et al., 2005] вместе с многочисленными примерами. В работе [Buford and Yu, 2010] дополнительно включен обширный список различных одноранговых систем. В работе [Aberer et al., 2005] предоставлена эталонная архитектура, которая позволяет более формально сравнивать различные типы одноранговых систем. Обзор распространенного контента представлен в [Androutsellis-Theotokis and Spinellis, 2004]. Наконец, работы [Buford et al., 2009], [Tarkoma, 2010] и [Vu et al., 2010] выходят за рамки обзоров и являются адекватными учебниками для начального или дальнейшего изучения.

Структурированные одноранговые системы

Как следует из названия, в структурированной одноранговой системе узлы (то есть процессы) организованы наложением, которое придерживается определенной, детерминированной топологии: кольцо, двоичное дерево, сетка и т. д. Эта топология используется для эффективного поиска данных. Для структурированных одноранговых систем характерно, что они обычно основаны на использовании так называемого индекса без семантики. Это

означает, что каждый элемент данных, который должен обслуживаться системой, однозначно связан с ключом, и этот ключ впоследствии применяется в качестве индекса. Для этого обычно используют хеш-функцию, поэтому мы получаем:

ключ(элемент данных) = хеш(значение элемента данных).

Теперь одноранговая система в целом отвечает за хранение пар (ключ, значение). С этой целью каждому узлу присваивается идентификатор из одного и того же набора всех возможных значений хеш-функции, и каждый узел отвечает за хранение данных, связанных с определенным подмножеством ключей. В сущности, система, таким образом, реализует распределенную хеш-таблицу, обычно сокращенно обозначаемую как **распределенная хеш-таблица** (Distributed Hash Table, DHT) [Balakrishnan et al., 2003]. Этот подход сводит объект структурированных одноранговых систем к возможности поиска элемента данных с помощью его ключа. Таким образом, система обеспечивает эффективную реализацию поиска функции, которая отображает ключ на существующий узел:

существующий узел = поиск(ключ).

Именно здесь топология структурированной одноранговой системы играет решающую роль. Любой узел может быть запрошен для поиска данного ключа, что затем сводится к эффективной маршрутизации этого запроса поиска к узлу, ответственному за хранение данных, связанных с этим ключом.

Чтобы прояснить эти вопросы, давайте рассмотрим простую одноранговую систему с фиксированным числом узлов, организованную в **гиперкубе**. Гиперкуб – это n-мерный куб. Гиперкуб, показанный на рис. 2.18, является четырехмерным. Его можно представить как два обычных куба, каждый с 8 вершинами и 12 ребрами. Чтобы расширить гиперкуб до пяти измерений, надо добавить к фигуре еще один набор из двух взаимосвязанных кубов, соединить соответствующие ребра в двух половинах и т. д.

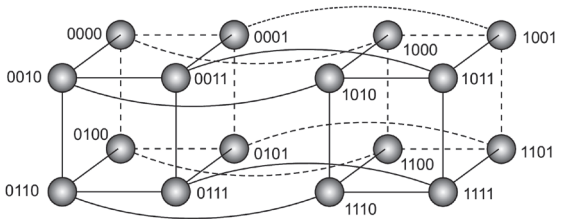


Рис. 2.18 ❖ Простая одноранговая система, организованная в виде четырехмерного гиперкуба

Для этой (в общем-то наивной) системы каждый элемент данных связан с одним из 16 узлов. Это может быть достигнуто путем хеширования значения элемента данных с ключом $k \in \{0, 1, 2, \ldots, 2^4 - 1]\}$. Теперь предположим,

что узел с идентификатором 0111 запрашивается для поиска данных, имеющих ключ 14, соответствующий двоичному значению 1110. В этом примере мы предполагаем, что узел с идентификатором 1110 отвечает за хранение всех элементов данных, которые имеют ключ 14. Что может просто сделать узел 0111, так это переслать запрос соседу, который находится ближе к узлу 1110. В этом случае это либо узел 0110, либо узел 1111. Если он выбирает узел 0110, этот узел затем направит запрос непосредственно на узел 1110, откуда данные могут быть получены.

Примечание 2.5 (пример: система Chord)

Предыдущий пример иллюстрирует две вещи: 1) использование функции хеширования для определения узла, ответственного за хранение некоторого элемента данных, и 2) маршрутизацию по топологии одноранговой системы, когда поиск элемента данных дает его ключ. Это не очень реалистичный пример, хотя бы по той причине, что мы предположили, что полный набор узлов фиксирован. Поэтому давайте рассмотрим более реалистичный пример структурированной одноранговой системы, которая также используется на практике.

В системе Chord (хорда) [Stoica et al., 2003] узлы логически организованы в кольцо так, что элемент данных с m-битным ключом k отображается на узле с наименьшим (опять же, m бит) идентификатором id > k. Этот узел называется **преемником** (*successor*) ключа k и обозначается как $succ(k)$. Ключи и идентификаторы обычно длиной 128 или 160 бит. Рисунок 2.19 показывает намного меньшее кольцо Chord, где $m = 5$, с девятью узлами {1, 4, 9, 11, 14, 18, 20, 21, 28}. Преемник ключа 7 равен 9. Аналогично $succ(5) = 9$, но также $succ(9) = 9$. В системе Chord каждый узел поддерживает ярлыки для других узлов. Ярлык отображается в виде направленного ребра от одного узла к другому. Как создаются эти ярлыки, объясняется в главе 5. Построение выполняется таким образом, что длина кратчайшего пути между любой парой узлов имеет порядок $O(\log N)$, где N – общее количество узлов.

Чтобы найти ключ, узел попытается переслать запрос «насколько это возможно», но не передавая его за пределы узла, ответственного за этот ключ. Для пояснения предположим, что в нашем примере кольца Chord узел 9 просят найти узел, ответственный за ключ 3 (который является узлом 4). Узел 9 имеет четыре ярлыка: для узлов 11, 14, 18 и 28 соответственно. Поскольку узел 28 является самым дальним узлом 9, который знает и все еще предшествует тому, который отвечает за ключ 3, он получит запрос поиска. Узел 28 имеет три ярлыка: для узлов 1, 4 и 14 соответственно. Обратите внимание, что узел 28 не знает о существовании узлов между узлами 1 и 4. По этой причине лучшее, что он может сделать, – это переслать запрос узлу 1. Последний знает, что его преемником в кольце является узел 4, и, таким образом, именно этот узел отвечает за ключ 3, которому он впоследствии будет пересылать запрос.

Теперь предположим, что узел с уникальным идентификатором u хочет присоединиться к Chord. Для этого он связывается с произвольным узлом и запрашивает у него поиск u, то есть возвращает значение $v = succ(u)$. В этот момент узел u просто должен вставить себя между предшественником v и самим v и, таким образом, стать новым предшественником v. Во время этого процесса будут созданы ярлыки от u к другим узлам, но также и некоторые существующие, ранее направленные к v теперь будут направлены к u (опять же, детали откладываются до последующих глав). Очевидно, что любой элемент данных с ключом k, хранящимся в v, но для которого $succ(k)$ теперь равен u, передается из v в u.

Выход так же прост: узел u сообщает о своем выходе своему предшественнику и преемнику и передает свои элементы данных в $succ(u)$.

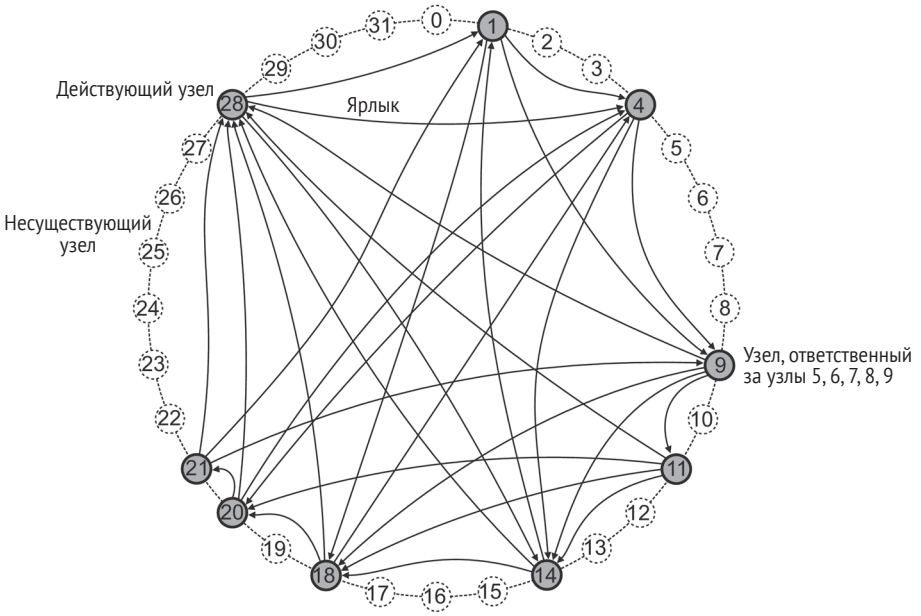


Рис. 2.19 ❖ Организация узлов и элементов данных в системе Chord

Неструктурированные одноранговые системы

Структурированные одноранговые системы пытаются поддерживать конкретную детерминированную оверлейную сеть. В неструктурированной одноранговой системе, напротив, каждый узел поддерживает специальный список соседей. Полученное наложение напоминает так называемый **случайный граф** (random graph): граф, в котором ребро (u, v) между двумя узлами u и v существует только с определенной вероятностью $\mathbb{P}[u, v]$. В идеале эта вероятность одинакова для всех пар узлов, но на практике наблюдается широкий диапазон распределений.

В неструктурированной одноранговой системе, когда узел присоединяется, он часто связывается с известным узлом, чтобы получить начальный список других одноранговых узлов в системе. Затем этот список можно использовать для поиска большего числа пиров, а также для игнорирования других и т. д. На практике узел обычно меняет свой локальный список почти непрерывно. Например, узел может обнаружить, что сосед больше не отвечает и что его нужно заменить. Могут быть и другие причины, которые мы опишем в ближайшее время.

В отличие от структурированных одноранговых систем, поиск данных не может следовать по заранее определенному маршруту, когда списки соседей создаются специальным образом. Вместо этого в неструктурированных одноранговых системах нам действительно необходимо прибегнуть к поиску данных [Risson and Moors, 2006]. Давайте рассмотрим две крайности и разберем случай, в котором нас просят найти конкретные данные (например, идентифицированные по ключевым словам).

Лавинная адресация (flooding): в случае лавинной адресации запрашивающий узел u просто передает запрос на элемент данных всем своим соседям. Запрос будет проигнорирован, когда принимающий запрос узел, скажем v, видел его раньше. В противном случае v выполняет локальный поиск запрошенного элемента данных. Если v имеет требуемые данные, он может либо напрямую ответить на запрашивающий узел u, либо отправить его обратно исходному серверу пересылки, который затем вернет его исходному серверу пересылки, и т. д. Если у v нет запрошенных данных, он пересылает запрос всем своим соседям.

Очевидно, что лавинная адресация может стать дорогим удовольствием, и поэтому у запроса часто есть связанное **время жизни** (time-to-live, TTL), определяющее максимальное количество прыжков пересылки запроса. Выбор правильного значения TTL имеет решающее значение: слишком маленькое количество означает, что запрос будет оставаться близко к эмитенту и, следовательно, может не достичь узла, имеющего данные. Слишком большое количество влечет за собой большие расходы на связь.

В качестве альтернативы настройке значений TTL узел также может начать поиск с начальным значением TTL, равным 1, что означает, что он сначала запросит только своих соседей. Если нет или недостаточно результатов, TTL увеличивается, и начинается новый поиск.

Случайные обходы (Random walks): на другом конце спектра поиска запрашивающий узел u может просто попытаться найти элемент данных, задавая случайно выбранному соседу, скажем v. Если v не имеет данных, он направляет запрос одному из своих случайно выбранных соседей и т. д. Результат известен как **случайный обход** (random walk) [Gkantsidis et al., 2006; Lv et al., 2002]. Очевидно, что при случайном обходе трафик сети значительно меньше, но может пройти гораздо больше времени, прежде чем будет достигнут узел, который имеет запрошенные данные. Чтобы уменьшить время ожидания, эмитент может просто запустить n случайных обходов одновременно. Действительно, исследования показывают, что в этом случае время, необходимое для достижения узла, в котором находятся данные, уменьшается примерно в n раз. Авторы [Lv et al, 2002] сообщают, что относительно небольшие значения n, такие как 16 или 64, оказываются эффективными.

Случайный обход также должен быть остановлен. Для этого мы можем либо снова использовать TTL, либо, в качестве альтернативы, когда узел получает запрос на поиск, уточнить у эмитента, необходима ли пересылка запроса другому случайно выбранному соседу.

Обратите внимание, что ни один из методов не использует конкретную методику сравнения для определения того, когда запрашиваемые данные были найдены. Для структурированных одноранговых систем мы предполагали использование ключей для сравнения; для двух только что описанных подходов подойдет любой метод сравнения.

Между лавинной адресацией и случайными обходами лежат основанные на политике методы поиска. Например, узел может решить отслеживать пиры, которые ответили положительно, эффективно превращая их в предпочтительных соседей для последующих запросов. Аналогичным образом мы можем захотеть ограничить количество лавинных адресаций меньшим

количеством соседей, но в любом случае отдать предпочтение соседям, имеющим много соседей.

Примечание 2.6 (дополнительно: лавинная адресация против случайных обходов)

Если задуматься над этим вопросом, то может удивить то, что случайный обход считается альтернативным способом поиска. Вначале это может показаться техникой, напоминающей поиск иголки в стоге сена. Однако мы должны понимать, что на практике имеем дело с реплицированными данными, и исследования показывают, что даже для очень малых коэффициентов репликации и различных распределений репликации развертывание случайных обходов не только эффективно, но и намного эффективнее по сравнению с лавинной адресацией.

Мы следуем модели, описанной в [Lv et al., 2002] и [Cohen and Shenker, 2002]. Чтобы понять, почему, предположим, что имеется всего N узлов и что каждый элемент данных реплицируется на r случайно выбранных узлов. Поиск состоит из многократного случайного выбора узла, пока не будет найден элемент. Если $\mathbb{P}[k]$ – вероятность того, что узел найден после k попыток, мы имеем:

$$\mathbb{P}[k] = \frac{r}{N}\left(1 - \frac{r}{N}\right)^{k-1}.$$

Пусть средний размер поиска S будет ожидаемым числом узлов, которые необходимо проверить перед поиском запрошенного элемента данных:

$$S = \sum_{k=1}^{N} k \cdot \mathbb{P}[k] = \sum_{k=1}^{N} k \cdot \frac{r}{N}\left(1 - \frac{r}{N}\right)^{k-1} \approx N/r \text{ для } 1 \ll r \leq N.$$

Простая репликация каждого элемента данных на каждый узел $S = 1$, и становится ясно, что случайный обход всегда будет превосходить лавинную адресацию даже для значений *TTL*, равных 1. Более реалистично, однако, предположить, что r/N относительно низкое, такое как 0,1 %, что означает, что средний размер поиска будет примерно 1000 узлов.

Чтобы сравнить это с лавинной адресацией, предположим, что каждый узел в среднем направляет запрос d случайно выбранным соседям. После одного шага запрос будет доставлен в d узлов, каждый из которых перенаправит его на другие $d - 1$ узлы (при условии что узел, с которого поступил запрос, пропущен), и т. д. Другими словами, после k шагов и с учетом того, что узел может получить запрос более одного раза, мы достигнем (не более)

$$R(k) = d(d-1)^{k-1}$$

узлов. Различные исследования показывают, что $R(k)$ является хорошей оценкой для фактического числа достигнутых узлов, при условии что у нас есть только несколько шагов лавинной адресации. Мы можем ожидать, что часть r/N из этих узлов будет иметь запрошенный элемент данных, а это означает, что когда $(r/N) \cdot R(k) \geq 1$, мы, скорее всего, найдем узел, который имеет элемент данных.

Чтобы проиллюстрировать это, пусть $r/N = 0{,}001 = 0{,}1$ %, что означает, что $S \approx 1000$. При затухании в среднем для $d = 10$ соседей нам потребуется по крайней мере 4 шага кодирования, достигая примерно 7290 узлов, что значительно больше, чем 1000 узлов, которые требуются при использовании случайного обхода. Только при $d = 33$ нам нужно будет связаться также приблизительно с 1000 узлов в $k = 2$ шагах кодирования и иметь $r/N\, R\,(k) \geq 1$.

Очевидный недостаток случайных обходов в том, что получение ответа может занять гораздо больше времени.

Иерархически организованные одноранговые сети

По мере роста сети в неструктурированных одноранговых системах поиск релевантных элементов данных может стать проблематичным. Причина этой проблемы масштабируемости проста: поскольку не существует детерминированного способа направления запроса на поиск к конкретному элементу данных, по сути, единственная техника, к которой может прибегнуть узел, – это поиск запроса с помощью лавинной адресации или случайного обхода сети. В качестве альтернативы во многих одноранговых системах предложили использовать специальные узлы, которые поддерживают индекс элементов данных.

Существуют и другие ситуации, в которых разумно отказаться от симметричной природы одноранговых систем. Рассмотрим совместную работу узлов, которые предлагают ресурсы друг другу. Например, в **сети совместной доставки контента** (collaborative content delivery network, CDN) узлы могут предлагать хранилище для размещения копий веб-документов, позволяя веб-клиентам получать доступ к ближайшим страницам и, таким образом, получать к ним быстрый доступ. Для этого необходимо выяснить, где лучше всего хранить документы. Использование **брокера** (посредника), который собирает данные об использовании и доступности ресурсов для ряда узлов, находящихся рядом друг с другом, позволит быстро выбрать узел с достаточными ресурсами.

Узлы, которые поддерживают индекс или выступают в качестве **брокера**, обычно называются супероднорангoвыми узлами, или **суперпирами** (super peers). Как следует из названия, суперпиры часто также организованы как одноранговые сети, что приводит к иерархической организации, как показано в работе [Yang and Garcia-Molina, 2003]. Простой пример такой организации представлен на рис. 2.20. В подобной организации каждый постоянный одноранговый узел, называемый теперь **слабым пиром** (weak peer), подключен как клиент к суперпиру. Все коммуникации от слабого пира и к нему происходят через ассоциированного суперпира этого слабого пира.

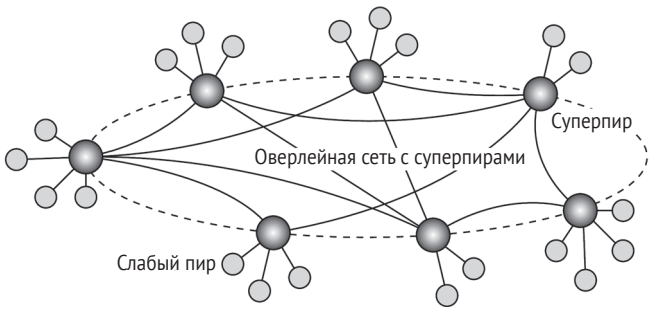


Рис. 2.20 ❖ Иерархическая организация узлов в супероднорангoвой сети

Во многих случаях связь между слабым пиром и его суперпиром фиксируется: всякий раз, когда слабый пир включается в сеть, он присоединяется

к одному из суперпиров и остается подключенным, пока не покинет сеть. Очевидно, ожидается, что суперузлы являются долгоживущими процессами с высокой доступностью. Чтобы компенсировать потенциальное нестабильное поведение суперпира, схемы резервного копирования можно развернуть, например связать каждого суперпира с другим и требовать, чтобы слабые пиры подключались к обоим.

Наличие фиксированной связи с суперпиром не всегда может быть лучшим решением. Например, в случае сетей с совместным использованием файлов для слабого пира может быть лучше подключиться к суперпиру, поддерживающему индекс файлов, в которых в данный момент заинтересован слабый пир. В этом случае шансы больше, чем когда слабый пир ищет конкретный файл, поскольку суперпир будет знать, где его найти. В работе [Garbacki et al., 2010] описывается относительно простая схема, в которой связь между слабым и сильным пирами может измениться, так как слабые пиры обнаруживают для общения лучших суперпиров. В частности, суперпир, возвращающий результат операции поиска, получает преимущество перед другими суперпирами.

Как мы уже видели, одноранговые сети предлагают гибкие средства для присоединения и выхода узлов из сети. Однако в супероднораноговых сетях возникает новая проблема, а именно как выбрать узлы, которые могут стать суперпирами. Эта проблема тесно связана с **проблемой выборов лидера** (leader-election problem), которую мы обсуждаем в разделе 6.4.

Примечание 2.7 (пример: сеть Skype)

Чтобы привести конкретный пример иерархически организованной одноранговой сети, давайте подробнее рассмотрим одну из самых успешных сетей: сеть VoIP (Voice over Internet Protocol) Skype. Хотя нет официальных публикаций о том, как организована сеть Skype, в [Baset and Schulzrinn, 2006] экспериментально проанализирована работа Skype, что позволяет определить архитектуру. Независимо от того, насколько точны их выводы (Skype значительно изменился со времени этой публикации), она остается интересным примером.

Сеть Skype организована очень похоже на то, как показано на рис. 2.20, с одним важным отличием: существует дополнительный централизованный **сервер входа в Skype** (Skype login server), с которым может общаться каждый узел (т. е. как слабый, так и суперузел).

Суперпиры (известные как **суперузлы Skype** (Skype super nodes)) имеют решающее значение для работы системы в целом. Помимо сервера входа в систему, существует ряд стандартных суперпиров Skype, которые можно использовать, чтобы начать, когда слабый пир начинается с нуля. Оказывается, что адрес каждого из этих суперпиров по умолчанию жестко задан в программном обеспечении Skype. Адрес состоит из пары (IP-адрес, номер порта).

Каждый слабый одноранговый узел имеет локальный список адресов доступных одноранговых узлов, называемый **кешем хоста** (host cache). Если ни один из кешированных суперузлов недоступен, он пытается по умолчанию подключиться к одному из суперузлов. Кеш хоста рассчитан на несколько сотен адресов. Чтобы подключиться к сети Skype, требуется слабый узел для установки соединения TCP с суперпиром. Это важно, особенно когда одноранговый узел работает за (NATed) межсетевым экраном (брандмауэром), поскольку суперузел может помочь в фактическом контакте с этим одноранговым узлом.

Давайте сначала рассмотрим ситуацию, когда один партнер A хочет связаться с другим (слабым) партнером B, для которого у него есть контактный адрес. Нам нужно выделить три случая, все они связаны с ситуацией, независимой от того, находятся ли одноранговые узлы за (NATed) межсетевыми экранами.

Как A, так и B находятся в общедоступном интернете: нахождение в общедоступном интернете означает, что с A и B можно связаться напрямую. В этом случае между A и B устанавливается TCP-соединение, которое используется для обмена контрольными пакетами. Фактический вызов происходит с использованием пакетов UDP (user diagram protocol) между согласованными портами вызывающего и вызываемого абонентов соответственно.

A **работает за брандмауэром, а** B **находится в общедоступном интернете:** в этом случае A сначала установит TCP-соединение с суперузлом S, после чего S установит TCP-соединение с B. Опять же, TCP-соединения используются для передачи контрольных пакетов между A и B (через S), после чего фактический вызов будет проходить через UDP и напрямую между A и B, не проходя через S. Однако кажется, что S необходим для обнаружения правильной пары номеров портов для межсетевого экрана в A, чтобы разрешить обмен пакетами UDP. В принципе, это также должно быть возможно с помощью B.

Как A, так и B работают за брандмауэром: это наиболее сложная ситуация, конечно, если мы также предположим, что брандмауэры ограничивают трафик UDP. В этом случае A подключится к онлайн-одноранговому узлу S через TCP, после чего S установит TCP-соединение с B. Эти соединения используются для обмена управляющими пакетами. Для фактического вызова связывается другой суперузел, который будет действовать как **ретранслятор** R: A устанавливает соединение с R и также с B. Весь голосовой трафик затем перенаправляется по двум соединениям TCP и через R.

Как пользователи находят друг друга? Как уже упоминалось, первое, что нужно сделать слабому одноранговому узлу, – это установить TCP-соединение с супероднораноговым узлом. Этот суперузел либо находится в кеше локального хоста, либо получен через один из суперузлов по умолчанию. Чтобы найти конкретного пользователя, слабый одноранговый узел связывается со своим супероднораноговым узлом, который возвращает несколько других одноранговых узлов для запроса. Если это не привело к каким-либо результатам, суперузел возвращает другой (на этот раз более длинный) список пиров, на которые должен быть переадресован поисковый запрос. Этот процесс повторяется до тех пор, пока пользователь не будет найден или запросчик не решит, что пользователь не существует. На самом деле это можно рассматривать как форму поиска на основе политики, о которой мы упоминали выше. Поиск пользователя означает, что его адрес или адрес связанного с ним суперпира возвращается. В принципе, если искомый пользователь находится в сети, тогда может быть установлено VoIP-соединение.

Наконец, сервер входа в Skype используется, чтобы убедиться, что только зарегистрированные пользователи могут использовать сеть. Это гарантирует, что идентификаторы пользователей уникальны, а также выполняет другие административные задачи, которые являются трудными, если не практически невозможными для децентрализованного управления.

Гибридные архитектуры

До сих пор мы сосредоточились на клиент-серверных архитектурах и ряде одноранговых архитектур. Многие распределенные системы объединяют архитектурные особенности, с чем мы уже сталкивались в супероднораноран-

говых сетях. В этом разделе мы рассмотрим некоторые конкретные классы распределенных систем, в которых клиент-серверные решения сочетаются с децентрализованными архитектурами.

Системы пограничных серверов

Важный класс распределенных систем, который организован в соответствии с гибридной архитектурой, сформирован **системами пограничного сервера** (edge-server systems). Эти системы развернуты в интернете, где серверы расположены «на краю» сети. Этот край образован границей между корпоративными сетями и фактическим интернетом, например как это осуществляется **поставщиком услуг интернета** (Internet Service Provider, ISP). Аналогично, когда конечные пользователи дома подключаются к интернету через своего интернет-провайдера, его можно рассматривать как находящегося на границе интернета.

Это приводит к общей организации, подобной той, что показана на рис. 2.21.

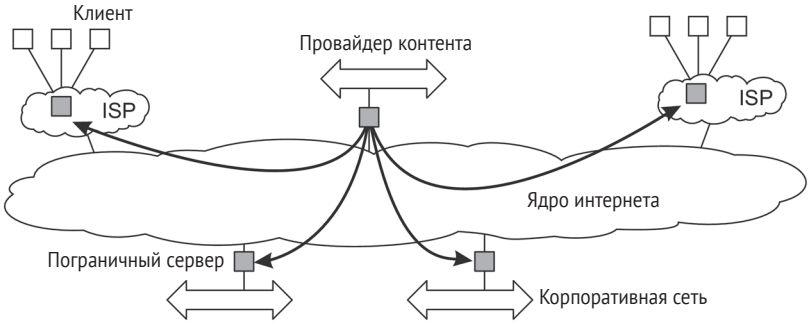


Рис. 2.21 ❖ Представление интернета в виде набора пограничных серверов

Конечные пользователи или клиенты в целом подключаются к интернету с помощью пограничного сервера. Основная цель пограничного сервера – обслуживать контент, возможно, после применения функций фильтрации и транскодирования. Более интересным является тот факт, что набор пограничных серверов можно использовать для оптимизации контента и распространения приложений. Базовая модель такова, что для конкретной организации один пограничный сервер действует как **исходный сервер** (origin server), из которого исходит весь контент. Этот сервер могут использовать другие пограничные серверы для репликации веб-страниц и т. д. [Leff and Rayfield, 2004; Nayate et al., 2004; Rabinovich and Spastscheck, 2002].

Данная концепция систем пограничных серверов в настоящее время часто позволяет идти еще дальше: облачные вычисления, реализованные в центре обработки данных в качестве ядра, используют дополнительные серверы на границе сети для оказания помощи в вычислениях и хранении, что по сути ведет к распределенному облаку системы. В случае **затуманивания в компьютерной графике** (fog computing) даже устройства конечных поль-

зователей являются частью системы и (частично) контролируются поставщиком облачных услуг [Yi et al., 2015].

Совместные распределенные системы

Гибридные структуры особенно применимы в совместных распределенных системах. Основной проблемой во многих из этих систем является первое начало работы, для которого часто используется традиционная схема клиент-сервер. Как только узел присоединился к системе, он может использовать полностью децентрализованную схему для совместной работы.

Чтобы конкретизировать ситуацию, давайте рассмотрим популярную систему совместного использования файлов BitTorrent [Cohen, 2003]. BitTorrent – это одноранговая система загрузки файлов. Ее основная идея показана на рис. 2.22 и состоит в том, что когда конечный пользователь ищет файл, он загружает фрагменты файла от других пользователей, пока загруженные фрагменты не могут быть собраны вместе, давая полный файл. Важной целью этой структуры является обеспечение сотрудничества. В большинстве систем совместного использования файлов значительная часть участников просто загружает файлы, но в остальном почти не вносит своего вклада [Adar and Huberman, 2000; Saroiu et al., 2003; Yang et al., 2005], – феномен, называемый **легкой прогулкой** (free riding). Чтобы избежать такой ситуации, в структуре BitTorrent файл может быть загружен только тогда, когда клиент загрузки сам предоставляет контент кому-либо еще.

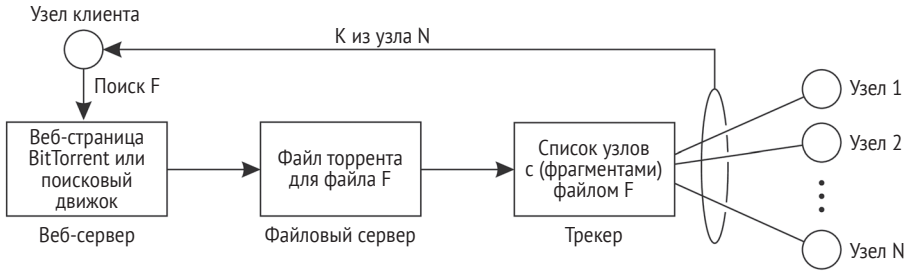


Рис. 2.22 ❖ Принцип работы BitTorrent
(адаптировано с разрешения [Pouwelse et al., 2005])

Чтобы загрузить файл, пользователю необходим доступ к глобальному каталогу, который обычно является одним из немногих известных веб-сайтов. Такой каталог содержит ссылки на так называемые **торрент-файлы** (torrent-file). В торрент-файле содержится информация, необходимая для загрузки конкретного файла. В частности, он содержит ссылку на так называемый **трекер** (tracker), являющийся сервером, в котором содержится точная учетная запись *активных* узлов, имеющих (фрагменты) запрошенного файла. Активный узел – это тот, который в настоящее время также загружает файл. Очевидно, что там будет много разных трекеров, хотя, как правило, только один трекер на один файл (или набор файлов).

Как только узлы были идентифицированы из того места, откуда могут быть загружены фрагменты, загружающий узел фактически становится активным. В этот момент он будет вынужден помогать другим, например предоставляя куски загружаемого файла, которого у других еще нет. Это принуждение исходит из очень простого правила: если узел P замечает, что узел Q загружает больше, P может решить уменьшить скорость, с которой он отправляет данные в Q. Подобная схема работает хорошо, если у P есть что-то для загрузки с B. По этой причине узлы часто снабжаются ссылками на многие другие узлы, что позволяет им лучше оперировать данными.

Очевидно, что BitTorrent объединяет централизованные и децентрализованные решения. Как оказалось, узкое место системы, что неудивительно, образовано трекерами. В альтернативной реализации BitTorrent'а узел также присоединяется к отдельной структурированной одноранговой системе (то есть DHT), чтобы помочь в отслеживании загрузок файлов. Фактически нагрузка центрального трекера теперь распределяется между участвующими узлами, причем каждый узел выступает в качестве трекера для относительно небольшого набора торрент-файлов. Первоначальная функция трекера, координирующего совместную загрузку файла, сохраняется. Однако надо отметить, что во многих системах BitTorrent, используемых сегодня, функциональность отслеживания фактически сведена к минимуму до одноразового предоставления одноранговых узлов, которые в это время участвуют в загрузке файла. С этого момента вновь участвующий одноранговый узел будет взаимодействовать только с этими одноранговыми узлами, а не с первоначальным трекером. Начальный трекер для запрашиваемого файла ищется в DHT через так называемую **магнитную связь** (magnet link). Мы вернемся к поискам на основе DHT в разделе 5.2.

Примечание 2.8 (дополнительно: BitTorrent под капотом)

Одним из заманчивых свойств BitTorrent является принудительное сотрудничество между узлами: получение файлового фрагмента требует его предоставления. Другими словами, у клиентов BitTorrent есть естественный стимул делать данные доступными для других, что позволяет обойти проблемы легкой прогулки, которые есть в других одноранговых системах.

Чтобы понять, как работает этот механизм, давайте немного углубимся в протокол BitTorrent.

Когда одноранговый узел A нашел трекер для файла F, трекер возвращает подмножество всех узлов, которые в настоящее время участвуют в загрузке F. Полный набор загружаемых узлов для конкретного файла F известен как **рой** (swarm) (для этого файла); подмножество узлов этого роя, с которым сотрудничает A, называется набором N_A соседей A. Обратите внимание, что если B находится в N_A, то A будет членом N_B: соседство является симметричным отношением. Набор соседей однорангового узла A периодически обновляется путем обращения к трекеру или когда новый одноранговый узел B присоединяется к рою и трекер передает N_A в B.

Узел, который имеет все фрагменты F и продолжает участвовать в своем рое, называется **сеялкой** (seeder); все остальные в рое известны как **пиявки** (leecher).

Как уже упоминалось, каждый файл делится на несколько кусков одинакового размера, которые в BitTorrent называются фрагментами. Размер фрагмента обыч-

но составляет 256 Кбайт, но также используются и другие размеры. Единицей передачи данных между двумя одноранговыми узлами является **блок**, размер которого обычно составляет 16 Кбайт. Узел может загрузить блок, только если у него есть весь фрагмент, к которому принадлежит этот блок. Сосед B принадлежит **потенциальному набору** (potential set) P_A в A, если у него есть блок, которого у A еще нет, *и наоборот*. Опять же, если $B \in P_A$ и $A \in P_B$, то A и B находятся в состоянии, в котором они могут *торговать* блоком.

С помощью этой терминологии загрузка файла может быть описана в терминах трех этапов, как подробно описано в [Rai et al., 2007].

Фаза начальной загрузки: узел A только что получил свой первый фрагмент, поместив его в позицию для начала торговых блоков. Эта первая часть получается с помощью механизма, называемого оптимистическим отцеплением (optimistic unchoking), с помощью которого узлы из N_A не всегда обеспечивают блоки части, чтобы запустить вновь прибывший узел. Если потенциальный набор P_A для A пуст, A должен будет ждать, пока в набор не будут добавлены новые соседи, у которых могут быть фрагменты для обмена.

Фаза торговли: на этом этапе $|P_A| > 0$, что означает, что всегда есть пир, с которым A может торговать блоками. На практике это фаза, на которой загрузка очень эффективна, так как достаточно узлов может быть спарено для обмена данными.

Последняя фаза загрузки: теперь размер потенциального набора снова упал до нуля, так что узел A полностью зависит от вновь прибывающих одноранговых узлов в своем наборе соседей, чтобы получить последние недостающие фрагменты. Обратите внимание, что набор соседей может быть изменен только при использовании трекера.

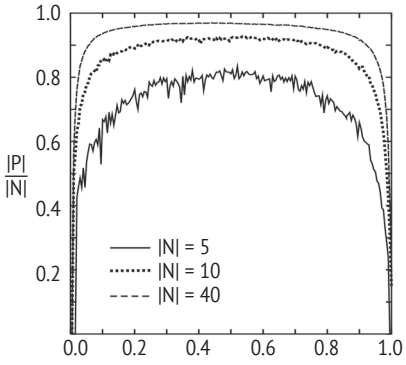


Рис. 2.23 ❖ Развитие размера потенциального набора |P| относительно размера |N| окрестности в BitTorrent по мере загрузки фрагментов

В большинстве описаний BitTorrent рассматривается только фаза торговли, которая обычно является самой длинной и наиболее эффективной фазой. На рис. 2.23 показано, как развиваются три фазы с точки зрения относительного размера набора потенциалов. Мы ясно видим, что на этапе начальной загрузки, а также на последней стадии загрузки потенциальный набор P является относительно небольшим. Фактически это означает, что после запуска может пройти некоторое время, и завершение загрузки может занять некоторое время по той простой причине, что сложно найти торгового партнера.

Чтобы найти подходящего торгового партнера, узел всегда выбирает его из своего потенциального набора, имеющего самую высокую скорость загрузки. Аналогично, чтобы ускорить распространение, узел обычно выбирает раздачу блоков самой редкой части в первом рое, прежде чем рассматривать другие части. Таким образом, мы можем ожидать быстрого и равномерного распределения фрагментов по всему рою.

2.4. Примеры архитектур

Файловая система сети

Многие распределенные файловые системы организованы в соответствии с архитектурой клиент-сервер, при этом **файловая система сети** (Network File System, NFS) Sun Microsystem является одной из наиболее распространенных для систем Unix. Здесь мы сосредоточимся на NFSv3, широко используемой третьей версии NFS [Callaghan, 2000], и NFSv4, самой последней, четвертой версии [Shepler et al., 2003]. Мы также обсудим различия между ними.

Основная идея NFS заключается в том, что каждый файловый сервер обеспечивает стандартизированное представление своей локальной файловой системы. Другими словами, не должно иметь значения, как эта локальная файловая система реализована; каждый сервер NFS поддерживает одну и ту же модель. Этот подход был принят и для других распределенных файловых систем. NFS поставляется с протоколом связи, который позволяет клиентам получать доступ к файлам, хранящимся на сервере, и таким образом позволяет гетерогенному набору процессов, возможно, работающих в разных операционных системах и машинах, использовать общую файловую систему.

Модель, лежащая в основе NFS и подобных систем, – это модель удаленного файлового сервиса. В этой модели клиентам предлагается прозрачный доступ к файловой системе, которой управляет удаленный сервер. Однако клиенты обычно не знают о фактическом местонахождении файлов. Вместо этого им предлагается интерфейс к файловой системе, аналогичный интерфейсу, предлагаемому обычной локальной файловой системой. В частности, клиенту предлагается только интерфейс, содержащий различные файловые операции, но отвечает за выполнение этих операций сервер. Такую модель поэтому также называют моделью удаленного доступа. Это показано на рис. 2.24a.

Напротив, в **модели загрузки/выгрузки** (upload/download model) клиент получает доступ к файлу локально после его загрузки с сервера, как показано на рис. 2.24b. Когда клиент завершает работу с файлом, файл снова загружается обратно на сервер, так что он может быть использован другим клиентом. Подобным образом в интернете может использоваться служба FTP, когда клиент загружает полный файл, изменяет его, а затем помещает обратно. NFS была реализована для большого количества различных операционных систем, хотя версии Unix преобладают.

Практически для всех современных систем Unix NFS обычно реализуется в соответствии с многоуровневой архитектурой, показанной на рис. 2.25.

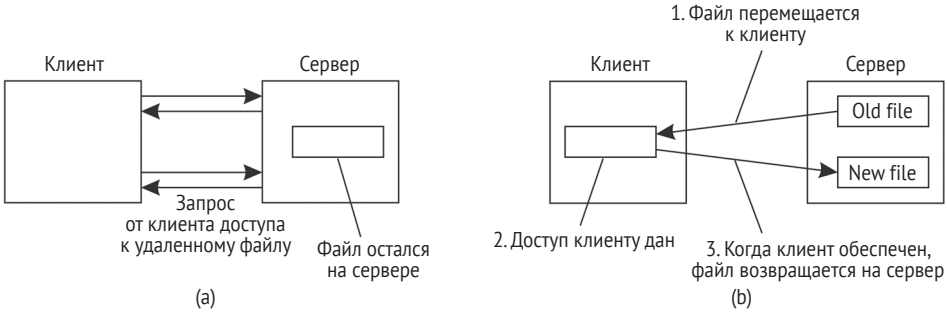


Рис. 2.24 ❖ Модели: a) удаленного доступа; b) загрузки/выгрузки

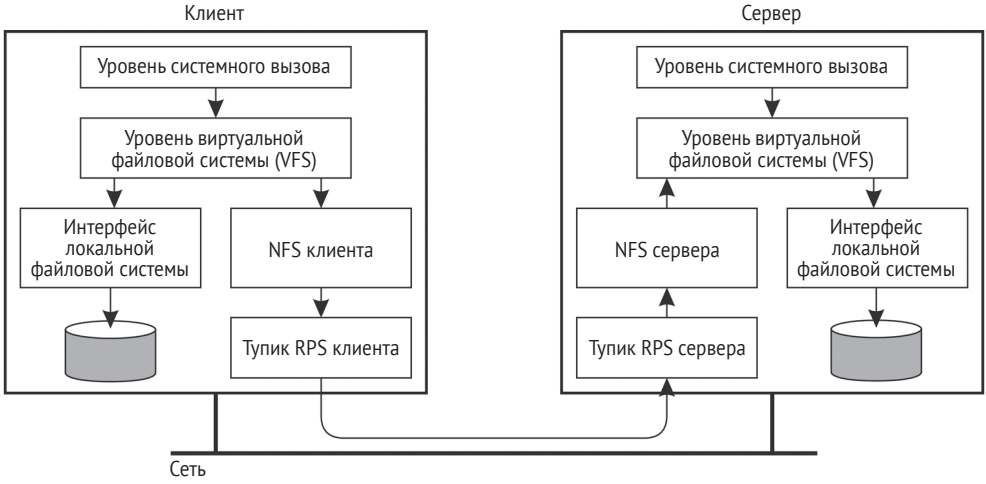


Рис. 2.25 ❖ Базовая архитектура NFS для систем Unix

Клиент получает доступ к файловой системе, используя системные вызовы, предоставляемые его локальной операционной системой. Однако локальный интерфейс файловой системы Unix заменяется интерфейсом с виртуальной **файловой системой** (Virtual File System, VFS), которая к настоящему времени де-факто является стандартом для взаимодействия с различными (распределенными) файловыми системами [Kleiman, 1986]. Практически все современные операционные системы предоставляют собой VFS, а ее отсутствие в той или иной мере вынуждает разработчиков в значительной степени переопределять огромные части операционной системы при принятии новой структуры файловой системы. В NFS операции на интерфейсе VFS передаются либо в локальную файловую систему, либо в отдельный компонент, известный как **клиент NFS**, который заботится об обработке доступа к файлам, хранящимся на удаленном сервере. В NFS все клиент-серверные соединения осуществляются через так называемые **удаленные вызовы процедур** (remote procedure calls, RPC). По сути, RPC – это стандартизированный способ, позволяющий клиенту на компьютере A совершать обычный вызов процедуры, которая реализована на другом компьютере B. Мы подробно об-

суждаем RPC в главе 4. Клиент NFS реализует системные операции NFS как вызовы удаленных процедур на сервер. Обратите внимание, что операции, предлагаемые интерфейсом VFS, могут отличаться от операций, предлагаемых клиентом NFS. Вся идея VFS состоит в том, чтобы скрыть различия между разными файловыми системами.

На стороне сервера мы видим похожую организацию. Сервер NFS отвечает за обработку входящих клиентских запросов. Компонент RPC на сервере преобразует входящие запросы в обычные операции с файлами VFS, которые впоследствии передаются на уровень VFS. Опять же, VFS отвечает за реализацию локальной файловой системы, в которой хранятся фактические файлы.

Важным преимуществом этой схемы является то, что NFS в значительной степени не зависит от локальных файловых систем. В принципе, действительно не имеет значения, реализует ли операционная система файловую систему Unix, файловую систему Windows или даже старую файловую систему MS-DOS на клиенте или сервере. Единственная важная проблема заключается в том, что эти файловые системы соответствуют модели файловой системы, предлагаемой NFS. Например, MS-DOS с ее короткими именами файлов не может быть использована для полной реализации NFS-сервера.

Примечание 2.9 (дополнительная информация: модель файловой системы NFS)

На рис. 2.26 показаны общие файловые операции, поддерживаемые NFS версий 3 и 4 соответственно. Операция `create` используется для создания файла, но имеет несколько разные значения в NFSv3 и NFSv4. В версии 3 эта операция используется для создания обычных файлов. Специальные файлы создаются с применением отдельных операций. Операция **ссылки** (`link`) используется для создания жестких ссылок. **Симлинк** (`Symulink`) используется для создания символических ссылок; `Mkdir` – для создания подкаталогов. Специальные файлы, такие как файлы устройств, сокеты и именованные каналы, создаются с помощью операции `mknod`.

Эта ситуация полностью изменилась в NFSv4, где `create` используется для создания *нерегулярных* файлов, которые включают символические ссылки, каталоги и специальные файлы. Жесткие ссылки по-прежнему создаются с применением отдельной операции ссылки, но регулярные файлы создаются с помощью операции открытия, которая является новой для NFS и основным отклонением от подхода к обработке файлов в более старых версиях. Вплоть до 4-й версии NFS была разработана так, чтобы позволить ее файловым серверам быть так называемыми серверами, **не имеющими состояния** (stateless): после завершения запроса сервер забудет о клиенте и запрошенной операции. По причинам, которые мы обсудим позже, этот критерий проектирования был отменен в NFSv4, в котором предполагается, что серверы обычно поддерживают состояние между операциями на одном и том же файле.

Операция **переименования** (`rename`) используется для изменения имени существующего файла так же, как в Unix.

Файлы удаляются с помощью операции **удаления** (`remove`). В версии 4 эта операция используется для удаления любого типа файла. В предыдущих версиях для удаления подкаталога требовалась отдельная операция `rmdir`. Файл удаляется по его имени, и количество жестких ссылок на него уменьшается на единицу. Когда количество ссылок падает до нуля, файл может быть уничтожен.

Версия 4 позволяет клиентам открывать и закрывать (обычные) файлы. Открытие несуществующего файла имеет побочный эффект создания нового файла. Что-

бы открыть файл, клиент предоставляет имя, а также различные значения для атрибутов. Например, клиент может указать, что файл должен быть открыт для доступа на запись. После того как файл был успешно открыт, клиент может получить к нему доступ с помощью своего дескриптора файла. Этот дескриптор также используется для закрытия файла, с помощью которого клиент сообщает серверу, что ему больше не нужно иметь доступ к файлу. Сервер, в свою очередь, может освободить любое состояние, которое он поддерживал, чтобы предоставить клиенту доступ к файлу.

Операция	v3	v4	Описание
create	Да	Нет	Создать обычный файл
create	Нет	Да	Создать нестандартную ссылку на файл
link	Да	Да	Создать жесткую ссылку на файл
symulink	Да	Нет	Создать символическую ссылку на файл
mkdir	Да	Нет	Создать подкаталог в указанном каталоге
mknod	Да	Нет	Создать специальный файл
rename	Да	Да	Изменить имя файла
remove	Да	Да	Удалить файл из файловой системы
rmdir	Да	Нет	Удалить пустой подкаталог из открытого каталога
open	Нет	Да	Открыть файл
close	Нет	Да	Закрыть файл
lookup	Да	Да	Посмотреть файл посредством имени файла
readdir	Да	Да	Чтение записей в каталоге
readlink	Да	Да	Чтение имени пути, хранящегося в символической ссылке
getattr	Да	Да	Получить значения атрибута для файла
setattr	Да	Да	Установить одно или несколько значений атрибута для файла
read	Да	Да	Чтение данных, содержащихся в файле
write	Да	Да	Запись данных в файл

Рис. 2.26 ❖ Неполный список операций файловой системы NFS

Операция поиска (lookup) используется для поиска дескриптора файла для заданного имени пути. В NFSv3 операция поиска не разрешает имя, выходящее за пределы так называемой **точки монтирования** (mount point) (которая по сути является местом в системе именования, подключающимся к другой системе именования. Мы вернемся к этому в главе 5). Например, предположим, что имя /remote/vu относится к точке монтирования в графе имен. При разрешении имени /remote/vu/mbox операция поиска в NFSv3 вернет дескриптор файла для точки монтирования /remote/vu вместе с оставшейся частью пути (т. е. mbox). Затем клиент должен явно смонтировать файловую систему, необходимую для завершения поиска имени. Файловая система в этом контексте – это набор файлов, атрибутов, каталогов и блоков данных, которые совместно реализованы как устройство логического блока [Tanenbaum and Woodhull, 2006].

В версии 4 все было упрощено. В этом случае поиск попытается разрешить полное имя, даже если это означает пересечение точек монтирования. Обратите внимание, что этот подход возможен, только если файловая система уже смонтирована в точках монтирования. Клиент может обнаружить, что точка монтирования была пере-

сечена, проверяя идентификатор файловой системы, который позже возвращается, когда поиск завершается.

Существует отдельная операция `readdir` для чтения записей в каталоге. Эта операция возвращает список пар ((*name*, *file handle*)), а также значения атрибутов, запрошенные клиентом. Клиент также может указать, сколько записей должно быть возвращено. Операция возвращает смещение, которое можно использовать при последующем вызове `readdir` для чтения следующей серии записей.

Операция `readlink` используется для считывания данных, связанных с символической ссылкой. Обычно эти данные соответствуют пути, который впоследствии можно найти. Обратите внимание, что операция поиска не может обрабатывать символические ссылки. Вместо этого, когда достигается символическая ссылка, разрешение имен останавливается, и клиенту необходимо сначала вызвать `readlink`, чтобы выяснить, где разрешение имен должно продолжаться.

Файлы имеют различные атрибуты, связанные с ними. Опять же, есть важные различия между NFS версий 3 и 4. Типичные атрибуты включают тип файла (указывающий, имеем ли мы дело с каталогом, символьной ссылкой, специальным файлом и т. д.), длину файла, идентификатор файловой системы, которая содержит файл, и время последнего изменения файла. Атрибуты файла могут быть прочитаны и установлены с использованием операций `getattr` и `setattr` соответственно.

Наконец, есть операции для чтения данных из файла и записи данных в файл. Чтение данных с помощью операции `read` осуществляется достаточно просто. Клиент определяет смещение и то количество байтов, которые будут прочитаны. Клиенту возвращается фактическое количество байтов, которое было прочитано, а также дополнительная информация о состоянии (например, достигнут ли конец файла).

Запись данных в файл выполняется с помощью операции `write`. Клиент снова указывает позицию в файле, с которой должна начинаться запись, количество записываемых байтов и данные. Кроме того, он может дать команду серверу обеспечить запись всех данных в стабильное хранилище (мы обсуждаем стабильное хранилище в главе 8). Серверы NFS необходимы для поддержки устройств хранения, которые могут выдерживать сбои питания, сбои операционной системы и сбои оборудования.

Веб

Архитектура распределенных систем на основе веб-технологий принципиально не отличается от других распределенных систем. Тем не менее интересно посмотреть, как развивалась первоначальная идея поддержки распределенных документов с момента ее создания в 1990-х годах. Документы превратились из чисто статических и пассивных в динамически генерируемый контент. Кроме того, в последние годы многие организации начали оказывать вспомогательные услуги вместо только предоставления документов.

Простые веб-системы

Многие веб-системы по-прежнему организованы как относительно простые клиент-серверные архитектуры. Ядро веб-сайта формируется процессом, который имеет доступ к локальной файловой системе, хранящей документы. Самый простой способ ссылки на документ – это ссылка, называемая **унифицированным указателем ресурса** (Uniform Resource Locator, URL). Она

указывает, где находится документ, встраивая DNS-имя связанного с ним сервера вместе с именем файла документа, по которому сервер может искать документ в своей локальной файловой системе. Кроме того, URL-адрес определяет протокол уровня приложения для передачи документа по сети.

Клиент взаимодействует с веб-серверами через браузер, который отвечает за правильное отображение документа. Кроме того, браузер принимает ввод от пользователя, в основном позволяя пользователю выбрать ссылку на другой документ, который он затем получает и отображает. Связь между браузером и веб-сервером стандартизирована: они оба придерживаются **протокола передачи гипертекста** (HyperText Transfer Protocol, HTTP). Это приводит к общей организации, показанной на рис. 2.27.

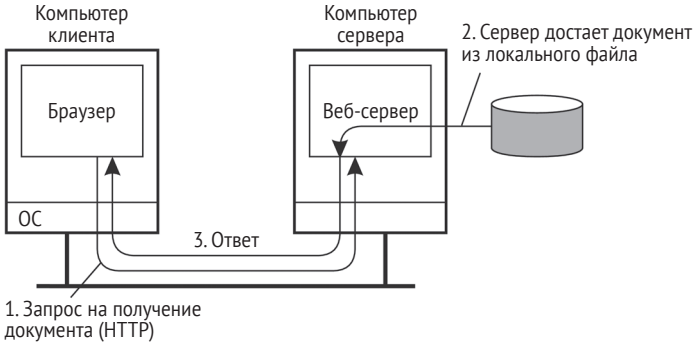


Рис. 2.27 ❖ Общая организация традиционного веб-сайта

Давайте немного усилим понимание того, что из себя представляет документ. Возможно, самая простая форма – это стандартный текстовый файл. В этом случае серверу и браузеру практически нечего делать: сервер копирует файл из локальной файловой системы и передает его в браузер. Последний, в свою очередь, просто отображает содержание файла дословно без каких-либо добавлений.

Более интересными являются веб-документы, которые были размечены, что обычно делается на **языке разметки гипертекста** (HyperText Markup Languag, HTML). В этом случае документ включает в себя различные инструкции, выражающие способ отображения его содержимого, аналогично тому, что можно ожидать от любой приличной системы обработки текста (хотя эти инструкции обычно скрыты от конечного пользователя). Например, указание текста, который нужно выделить, выполняется с помощью следующей разметки:

```
<emph>Emphasize this text</ emph>
```

Есть еще много таких инструкций по разметке. Дело в том, что браузер понимает эти инструкции и будет действовать соответствующим образом при отображении текста.

Документы могут содержать гораздо больше, чем просто инструкции по разметке. В частности, они могут иметь встроенные программы, из которых

Javascript является наиболее часто используемым. В этом случае браузер предупреждается, что есть некоторый код для выполнения, как показано в

```
<script type=''text/javascript''>....</script>
```

И пока браузер является подходящим встроенным интерпретатором для указанного языка, все между «<script>» и «</script>» будет выполняться как любая другая программа. Основным преимуществом включения сценариев является то, что они обеспечивают гораздо лучшее взаимодействие с конечным пользователем, включая отправку информации обратно на сервер. (Кстати, последнее всегда поддерживалось в HTML через **формы**.) Намного больше можно сказать о веб-документах, но здесь это не совсем к месту. Хорошее введение о том, как создавать веб-приложения, можно найти в [Sebesta, 2006].

Многоуровневые архитектуры

Интернет начинался как относительно простая двухуровневая система клиент-сервер, показанная на рис. 2.27. К настоящему времени эта простая архитектура была расширена для поддержки гораздо более сложных средств для документов. На самом деле можно с полным основанием утверждать, что термин «документ» теперь более не подходит. Во-первых, большинство из того, что мы видим в нашем браузере, было сгенерировано на месте в результате отправки запроса на веб-сервер. Содержимое хранится в базе данных на стороне сервера вместе со сценариями на стороне клиента, и составленный на лету документ впоследствии отправляется в браузер клиента. Таким образом, документы стали полностью динамичными.

Одним из первых улучшений базовой архитектуры стала поддержка простого взаимодействия с пользователем с помощью стандарта **интерфейса компьютерной графики** (Common Gateway Interface, CGI). CGI определяет стандартный способ, с помощью которого веб-сервер может выполнять программу, принимая пользовательские данные в качестве входных данных. Обычно пользовательские данные поступают из формы HTML; в ней указывается программа, которая должна выполняться на стороне сервера, а также значения параметров, которые вводит пользователь. После заполнения формы имя программы и собранные значения параметров отправляются на сервер, как показано на рис. 2.28.

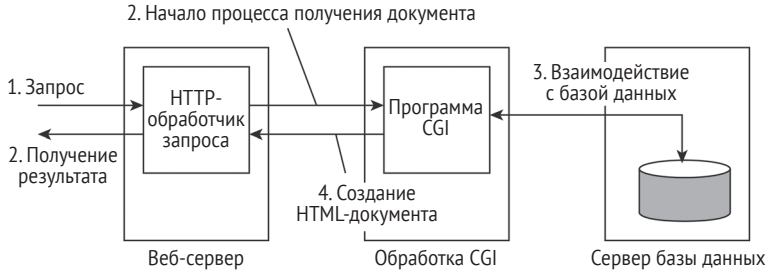


Рис. 2.28 ❖ Принцип использования CGI-программ на стороне сервера

Когда сервер видит запрос, он запускает программу, названную в запросе, и передает ей значения параметров. На этом этапе программа просто выполняет свою работу и обычно возвращает результаты в виде документа, который отправляется обратно в браузер пользователя для отображения.

Программы CGI могут быть настолько сложными, насколько этого хочет разработчик. Например, как показано на рис. 2.28, многие программы работают с базой данных, локальной для веб-сервера. После обработки данных программа генерирует документ HTML и возвращает этот документ на сервер. Затем сервер передаст документ клиенту. Интересное наблюдение состоит в том, что для сервера это выглядит так, как будто он просит программу CGI получить документ. Другими словами, сервер делает только делегирование извлечения документа внешней программе.

Раньше основной задачей сервера была обработка клиентских запросов путем простого извлечения документов. При использовании CGI-программ выборка документа может быть делегирована таким образом, что сервер не будет знать, был ли документ сгенерирован на лету или фактически прочитан из локальной файловой системы. Обратите внимание, что мы только что описали двухуровневую организацию серверного программного обеспечения.

Однако в наши дни серверы делают гораздо больше, чем просто извлекают документы. Одним из наиболее важных улучшений является то, что серверы также могут обрабатывать документ перед его передачей клиенту. В частности, документ может содержать **серверный скрипт** (server-side script), который выполняется сервером, когда документ был выбран локально. Результат выполнения скрипта отправляется вместе с остальной частью документа клиенту. Сам скрипт не отправляется. Другими словами, использование серверного сценария изменяет документ, по существу заменяя сценарий результатами его выполнения. Чтобы конкретизировать ситуацию, взгляните на очень простой пример динамической генерации документа. Предположим, что файл хранится на сервере со следующим содержимым:

```
<strong> <?php echo $_SERVER['REMOTE_ADDR']; ?> </strong>
```

Сервер проверит файл и затем обработает код PHP (между «<?php» и «?>»), заменив код адресом запрашивающего клиента. Возможны гораздо более сложные настройки, такие как доступ к локальной базе данных и последующий выбор контента из этой базы данных для объединения с другим динамически генерируемым контентом.

2.5. Резюме

Распределенные системы могут быть организованы различными способами. Мы можем отметить различие между архитектурой программного обеспечения и архитектурой системы. Последняя учитывает, где компоненты, составляющие распределенную систему, размещены на различных машинах. Первая больше заботится о логической организации программного обеспе-

чения: как взаимодействуют компоненты, как их можно структурировать, как сделать их независимыми и т. д.

Ключевым словом при разговоре об архитектуре является архитектурный стиль. Стиль отражает основной принцип, который следует при организации взаимодействия между программными компонентами, составляющими распределенную систему. К важным стилям относятся многоуровневые, объектно-ориентированные стили, ресурсно-ориентированные стили и стили, в которых обработка событий является превалирующей.

Существует много разных организаций распределенных систем. Важный класс – это класс, где машины делятся на клиентов и серверы. Клиент отправляет запрос на сервер, который затем выдаст результат, возвращающийся клиенту. Клиент-серверная архитектура отражает традиционный способ модуляции программного обеспечения, при котором модуль вызывает функции, доступные в другом модуле. Размещая разные компоненты на разных машинах, мы получаем естественное физическое распределение функций по совокупности машин.

Клиент-серверные архитектуры часто сильно централизованы. В децентрализованных архитектурах мы нередко видим равную роль процессов, составляющих одноранговую распределенную систему, известную также как система точка-точка. В одноранговых системах процессы организованы в рамках оверлейной сети, которая представляет собой логическую сеть, в которой каждый процесс имеет локальный список других одноранговых узлов, с которыми он может взаимодействовать. Оверлейная сеть может быть структурирована, и в этом случае могут быть развернуты детерминированные схемы для маршрутизации сообщений между процессами. В неструктурированных сетях список пиров является более или менее случайным, что подразумевает необходимость развертывания алгоритмов поиска для определения местоположения данных или других процессов.

В гибридных архитектурах элементы централизованных и децентрализованных организаций объединяются. Централизованный компонент часто используется для обработки начальных запросов, например для перенаправления клиента на сервер реплики, который, в свою очередь, может быть частью одноранговой сети, как в случае систем на основе BitTorrent.

Глава 3

Процессы

В этой главе мы подробнее рассмотрим, как различные типы процессов играют решающую роль в распределенных системах. Понятие процесса появилось в области операционных систем, где процесс обычно определяется как исполняемая программа. С точки зрения операционной системы управление и планирование процессов являются, пожалуй, наиболее важными для решения проблемами. Однако когда дело доходит до распределенных систем, другие проблемы оказываются одинаково или даже более важными.

Мы начнем с подробного обсуждения потоков и их роли в распределенных системах. Как оказалось, потоки играют решающую роль не только в получении производительности в многоядерных и многопроцессорных средах, но также помогают структурировать клиентов и серверы. Во многих случаях мы видим, что потоки заменяются процессами и используют базовую операционную систему для обеспечения защиты и облегчения связи. Вместе с тем когда на карту поставлена производительность, потоки продолжают играть важную роль.

В последние годы концепция виртуализации приобрела большую популярность. Виртуализация позволяет приложению, а также, возможно, и всей среде, включая операционную систему, работать одновременно с другими приложениями, но не зависит от базового оборудования и платформ, что приводит к высокой степени переносимости. Кроме того, виртуализация помогает изолировать сбои, вызванные ошибками или проблемами безопасности. Это важная концепция для распределенных систем, на которую мы обратим внимание в отдельном разделе.

Клиент-серверные организации важны в распределенных системах. В этой главе мы подробнее рассмотрим типичные организации как клиентов, так и серверов. Мы также обращаем внимание на общие вопросы проектирования серверов, в том числе те, которые обычно используются в объектно-распределенных системах. Широко используемым веб-сервером является веб-сервер Apache, которому мы уделяем отдельное внимание. Организация кластеров серверов остается важной областью, особенно когда им необходимо совместно создать иллюзию единой системы, и мы обсудим примеры того, как достичь этой перспективы, включая глобальные серверы, такие как PlanetLab.

Важной проблемой, особенно в глобальных распределенных системах, является перемещение процессов между различными машинами. Миграция процессов или, в частности, миграция кода может помочь в достижении

масштабируемости, но также и в динамическом конфигурировании клиентов и серверов. В этой главе также обсуждается, что на самом деле означает перенос кода и каковы его последствия.

3.1. Потоки

Хотя процессы образуют строительный блок в распределенных системах, практика показывает, что модульность процессов, обеспечиваемая операционными системами, на которых построены распределенные системы, недостаточна. Выясняется, что наличие большого уровня модульности в виде многих потоков управления в процессе значительно упрощает создание распределенных приложений и повышает производительность. В этом разделе мы рассмотрим роль потоков в распределенных системах и объясним, почему они так важны.

Подробнее о потоках и о том, как их можно использовать для создания приложений, можно узнать из [Lewis and Berg, 1998; Stevens, 1999; Robbins and Robbins, 2003], [Herlihy and Shavit, 2008]. Настоятельно рекомендуется узнать больше о многопоточном параллельном программировании в целом.

Введение в потоки

Чтобы понять роль потоков в распределенных системах, важно понять, что такое процесс и как связаны процессы и потоки. Для выполнения программы операционная система создает несколько **виртуальных процессоров** (virtual processors), каждый из которых предназначен для запуска другой программы. Для отслеживания этих виртуальных процессоров в операционной системе имеется **таблица процессов** (process table), содержащая записи для хранения значений регистров центрального процессора (ЦП), карт памяти, открытых файлов, учетной информации, привилегий и т. д. Вместе эти записи образуют **контекст процесса** (process context).

Контекст процесса можно рассматривать как программный аналог **контекста аппаратного процессора** (processor context). Последний состоит из минимальной информации, которая автоматически сохраняется аппаратным обеспечением для обработки прерывания и последующего возврата туда, где процессор остановился. Контекст процессора содержит, по крайней мере, программный счетчик, но иногда и другие значения регистра, такие как указатель стека.

Процесс часто определяется как выполняемая программа, то есть программа, которая в данный момент выполняется на одном из виртуальных процессоров операционной системы. Важной проблемой является то, что операционная система уделяет большое внимание обеспечению того, чтобы независимые процессы не могли злонамеренно или непреднамеренно повлиять на правильность поведения друг друга. Другими словами, тот факт,

что несколько процессов могут одновременно использовать один и тот же процессор и другие аппаратные ресурсы, становится прозрачным. Обычно операционной системе требуется аппаратная поддержка для обеспечения такого разделения.

Эта прозрачность параллелизма имеет свою цену. Например, каждый раз, когда создается процесс, операционная система должна создавать полное независимое адресное пространство. Распределение может означать инициализацию сегментов памяти путем, например, обнуления сегмента данных, копирования связанной программы в текстовый сегмент и установки стека для временных данных. Аналогично переключение ЦП между двумя процессами также может потребовать некоторых усилий. Помимо сохранения данных, хранящихся в настоящий момент в различных регистрах (включая программный счетчик и указатель стека), операционной системе также придется изменять регистры модуля управления памятью (memory management unit, MMU) и делать недействительными кеши преобразования адресов, например в буфере преобразования перевода (translation lookaside buffer, TLB). Кроме того, если операционная система поддерживает больше процессов, чем она может одновременно удерживать в основной памяти, ей, возможно, придется поменять местами процессы между основной памятью и диском, прежде чем произойдет фактическое переключение.

Как и процесс, поток выполняет свой собственный фрагмент кода независимо от других потоков. Однако, в отличие от процессов, не делается никаких попыток достичь высокой степени прозрачности параллелизма, если это приведет к снижению производительности. Следовательно, система потоков, как правило, поддерживает только минимальную информацию, чтобы позволить центральному процессору совместно использоваться несколькими потоками. В частности, **контекст потока** (thread context) часто состоит не более чем из контекста процессора вместе с некоторой другой информацией для управления потоками. Например, система потоков может отслеживать тот факт, что поток в настоящее время заблокирован в переменной мьютекса, чтобы не выбирать его для выполнения. Информация, которая не является строго необходимой для управления несколькими потоками, обычно игнорируется. По этой причине защита данных от несанкционированного доступа потоками внутри одного процесса полностью принадлежит разработчикам приложений. Таким образом, мы видим, что контекст процессора содержится в контексте потока и что, в свою очередь, контекст потока содержится в контексте процесса.

Как только что было сказано, у применения потоков есть два важных последствия. Во-первых, производительность многопоточного приложения вряд ли когда-либо будет хуже, чем у его однопоточного аналога. На самом деле во многих случаях многопоточность даже приводит к увеличению производительности. Во-вторых, поскольку потоки не защищены автоматически друг от друга как процессы, разработка многопоточных приложений требует дополнительных интеллектуальных усилий. Правильный дизайн и простота, как обычно, очень помогают. К сожалению, современная практика не демонстрирует, что этот принцип одинаково хорошо понят.

Использование потоков в нераспределенных системах

Прежде чем обсуждать роль потоков в распределенных системах, давайте сначала рассмотрим их использование в традиционных, нераспределенных системах. Многопоточные процессы имеют несколько преимуществ, которые увеличили популярность использования потоковых систем.

Наиболее важным преимуществом является тот факт, что в однопоточном процессе всякий раз, когда выполняется системный вызов блокировки, процесс в целом блокируется. Для иллюстрации рассмотрим такое приложение, как программа для работы с электронными таблицами, и предположим, что пользователь постоянно и в интерактивном режиме хочет изменять значения. Важным свойством программы работы с электронными таблицами является то, что она поддерживает функциональные зависимости между различными ячейками, часто из разных электронных таблиц. Следовательно, всякий раз, когда ячейка модифицируется, все зависимые ячейки автоматически обновляются. Когда пользователь изменяет значение в одной ячейке, такая модификация может инициировать большую серию вычислений. Если существует только один поток управления, вычисления не могут продолжаться, пока программа ожидает ввода. Аналогично нелегко обеспечить ввод данных во время вычисления зависимостей. Простое решение состоит в том, чтобы иметь как минимум два потока управления: один для управления взаимодействием с пользователем и один для обновления электронной таблицы. В то же время третий поток может быть использован для резервного копирования электронной таблицы на диск, в то время как два других выполняют свою работу.

Другое преимущество многопоточности заключается в том, что становится возможным использовать параллелизм при выполнении программы в многопроцессорной или многоядерной системе. В этом случае каждый поток назначается разным ЦП или ядрам, а общие данные хранятся в общей основной памяти. При правильной разработке такой параллелизм может быть прозрачным: процесс будет одинаково хорошо работать в однопроцессорной системе, хотя и медленнее. Многопоточность для параллелизма становится все более важной с наличием относительно дешевых многопроцессорных и многоядерных компьютеров. Такие компьютерные системы обычно используются для запуска серверов в клиент-серверных приложениях, но в настоящее время также широко используются в таких устройствах, как смартфоны.

Многопоточность полезна и в контексте больших приложений. Такие приложения часто разрабатываются как набор взаимодействующих программ, каждая из которых выполняется отдельным процессом. Этот подход типичен для среды Unix. Сотрудничество между программами осуществляется с помощью **средств обеспечения взаимодействия процессов** (interprocess communication, IPC). Для систем Unix эти механизмы обычно включают в себя (именованные) каналы, очереди сообщений и сегменты разделяемой памяти (см. также [Stevens and Rago, 2005]). Основным недостатком всех механизмов IPC является то, что для связи часто требуется относительно обширное переключение контекста, показанное в трех разных точках на рис. 3.1.

Поскольку IPC требует вмешательства ядра, процесс, как правило, сначала должен переключиться из режима пользователя в режим ядра, показанный как S1 на рис. 3.1.

Это требует изменения карты памяти в MMU, а также установки TLB. Внутри ядра происходит переключение контекста процесса (на рис. S2), после чего другую сторону можно активировать, снова переключаясь из режима ядра в режим пользователя (S3 на рис. 3.1). Последний переключатель опять требует изменения карты MMU и запуска TLB.

Вместо использования процессов приложение также может быть сконструировано так, чтобы разные части выполнялись отдельными потоками.

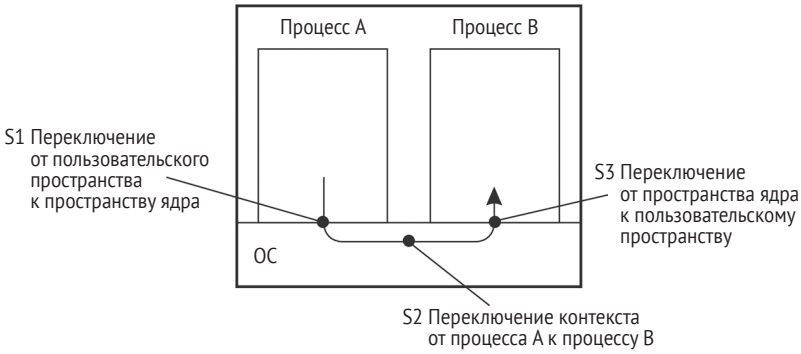


Рис. 3.1 ❖ Переключение контекста в результате IPC

Связь между этими частями полностью решается с помощью общих данных. Переключение потоков может иногда выполняться полностью в пользовательском пространстве, хотя в других реализациях ядро знает о потоках и планирует их. Результатом может быть резкое улучшение производительности.

И наконец, есть и прямая причина для использования потоков при разработке программного обеспечения: многие приложения проще структурировать как совокупность взаимодействующих потоков. Вспомните о приложениях, которые должны выполнять несколько (более или менее независимых) задач, например в рассмотренном ранее нашем примере с электронными таблицами.

Примечание 3.1 (дополнительно: стоимость переключения контекста)

Было проведено много исследований по измерению влияния на производительность переключения контекста. Как и во многих случаях с измерениями компьютерных систем, найти основную истину не так-то просто. В работе [Tsafrir 2007] отмечается, что обработка тактов стала более или менее повсеместной в операционных системах, что делает их отличным кандидатом для измерения затрат. Обработчик часов активируется один раз каждые T миллисекунд посредством прерывания часов. Общие значения для T находятся в диапазоне от 0,5 до 20 миллисекунд, что соответствует частоте прерывания 2000 Гц и 50 Гц соответственно. Обработчик обычно помогает в реализации различных служб синхронизации и использования

ЦП, отправляет сигналы тревоги и помогает в прерывании выполнения задач для справедливого совместного использования ЦП. Просто изменяя частоту, с которой оборудование генерирует прерывание, можно легко получить представление о затратах.

Чтобы измерить влияние производительности на прерывание, проводится различие между **прямыми и косвенными издержками** (direct overhead and indirect overhead). Прямые издержки состоят из времени, которое требуется для фактического переключения контекста, а также времени, которое требуется обработчику для выполнения своей работы, и последующего переключения обратно на прерванную задачу. Косвенные издержки – это все остальное, и в основном они вызваны возмущениями кеша (к которым мы вскоре вернемся). В работе [Tsafrir 2007] показывается, что для различных процессоров Intel время переключения контекста составляет порядка 0,5–1 микросекунды, а сам обработчик выполняет свою работу порядка 0,5–7 микросекунд, что сильно зависит от реализации.

Оказывается, однако, что прямые издержки на самом деле не так уж влиятельны. В другом исследовании [Liu and Solihin 2010] ясно дается понять, что переключение контекста может сильно нарушить кеш, что приведет к потере производительности по сравнению с ситуацией до возникновения прерывания. Фактически для простого случая прерывания тактовой частоты косвенные издержки оценивались приблизительно в 80 % [Tsafrir 2007].

Чтобы понять, что происходит, рассмотрим организации данных, как показано на рис. 3.2. Предположим, что кеш организован таким образом, что наименее недавно использованный блок данных удаляется из кеша, когда требуется место для нового блока данных.

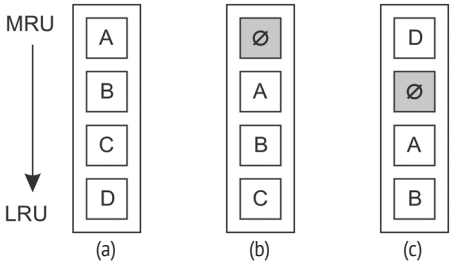


Рис. 3.2 ❖ Организация кеша при работе с прерываниями:
a) до переключения контекста, b) после переключения контекста
и c) после обращения к блоку D (адаптировано из [Liu and Solihin 2010]).
LRU (Least Recently Used) – наименее недавно использован

На рис. 3.2a показана ситуация до возникновения прерывания. После обработки прерывания блок D, возможно, был изгнан из кеша, оставляя дыру, как показано на рис. 3.2b. Доступ к блоку D скопирует его обратно в кеш, возможно, исключив блок C, и т. д. Другими словами, даже простое прерывание может вызвать значительную и относительно длительную реорганизацию кеша, что, в свою очередь, влияет на общую производительность приложения.

Реализация потоков

Потоки часто предоставляются в виде пакета потоков. Такой пакет содержит операции по созданию и уничтожению потоков, а также операции над переменными синхронизации, такими как мьютексы и условные переменные. Существует два основных подхода к реализации пакета потоков. Первый подход заключается в создании библиотеки потоков, которая выполняется полностью в пространстве пользователя. Второй подход – ядро должно быть осведомлено о потоках и планировать их.

Библиотека потоков на уровне пользователя имеет ряд преимуществ. Во-первых, создавать и уничтожать потоки дешево. Поскольку все управление потоками хранится в адресном пространстве пользователя, цена создания потока в первую очередь определяется стоимостью выделения памяти для настройки стека потоков. Аналогично уничтожение потока в основном включает освобождение памяти для стека, который больше не используется. Обе операции дешевы.

Второе преимущество потоков пользовательского уровня заключается в том, что переключение контекста потока часто можно выполнить всего за несколько инструкций. По сути, только значения регистров ЦП должны быть сохранены и впоследствии загружены с ранее сохраненными значениями потока, в который он переключается. Там нет необходимости менять карты памяти, использовать TLB, вести учет ЦП и т. д. Переключение контекста потока выполняется, когда необходимо синхронизировать два потока, например при входе в раздел общих данных. Однако, как обсуждалось в примечании 3.1, большая часть накладных расходов на переключение контекста вызвана возмущением кешей памяти.

Основным недостатком потоков пользовательского уровня является развертывание модели потоков «много к одному»: несколько потоков сопоставляются одному планируемому объекту. Как следствие вызов блокирующего системного вызова немедленно заблокирует весь процесс, которому принадлежит поток, и, следовательно, также все другие потоки в этом процессе. Как мы объясняли, потоки особенно полезны для структурирования больших приложений в частях, которые могут быть выполнены логически одновременно. В этом случае блокировка ввода-вывода не должна препятствовать выполнению других частей в это время. Для таких приложений потоки уровня пользователя не помогают.

Эти проблемы в основном можно обойти, реализуя потоки в ядре операционной системы, что приводит к так называемой **модели потоков «один к одному»** (one-to-one threading model), в которой каждый поток является планируемым объектом. Платой является то, что каждая операция с потоками (создание, удаление, синхронизация и т. д.) должна выполняться ядром, требующим системного вызова. Переключение контекстов потока теперь может стать таким же дорогим, как переключение контекстов процесса. Однако в свете того факта, что производительность переключения контекста, как правило, диктуется неэффективным использованием кешей памяти, а не

различием между моделями потоков «один к одному» или «много к одному», многие операционные системы сейчас предлагают последнюю модель, хотя бы за ее простоту.

Примечание 3.2 (дополнительно: облегченные процессы)

Альтернативой двум экстремальным вариантам является гибридная форма потоков на уровне пользователя и на уровне ядра, так называемая **модель потоков «много к многим»**. Модель реализована в виде **облегченных процессов** (lightweight processes, LWP). LWP выполняется в контексте одного (тяжеловесного) процесса, и в каждом процессе может быть несколько LWP. В дополнение к наличию LWP система также предлагает пакет потоков на уровне пользователя, предлагая приложениям обычные операции для создания и уничтожения потоков. Кроме того, пакет предоставляет средства для синхронизации потоков, такие как взаимные исключения и условные переменные. Важным пунктом является то, что пакет потоков реализуется полностью в пространстве пользователя. Другими словами, все операции с потоками выполняются без вмешательства ядра.

Пакет потока может использоваться совместно несколькими LWP, как показано на рис. 3.3.

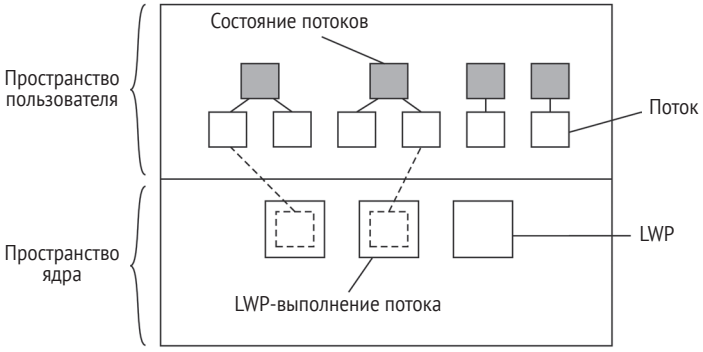


Рис. 3.3 ❖ Объединение процессов уровня ядра и потоков уровня пользователя

Это означает, что каждый LWP может запускать свой собственный (на уровне пользователя) поток. Многопоточные приложения создаются путем создания потоков и последующего назначения каждого потока LWP. Назначение потока в LWP обычно неявно и скрыто от программиста.

Комбинация потоков (уровня пользователя) и LWP работает следующим образом. Пакет потока имеет одну подпрограмму для планирования следующего потока. При создании LWP (который выполняется с помощью системного вызова) LWP присваивается свой собственный стек, и он получает указание выполнить подпрограмму планирования в поисках потока для выполнения. Если имеется несколько LWP, то каждый из них выполняет планировщик. Таким образом, таблицу потоков, которая используется для отслеживания текущего набора потоков, LWP используют совместно. Защита этой таблицы для обеспечения взаимоисключающего доступа осуществляется посредством мьютексов, которые полностью реализованы в пользовательском пространстве. Другими словами, синхронизация между LWP не требует поддержки ядра.

Когда LWP находит работающий поток, он переключает контекст на этот поток. Между тем другие LWP могут также искать иные выполняемые потоки. Если поток должен блокировать мьютекс или условную переменную, он выполняет необходимое администрирование и, в конце концов, вызывает процедуру планирования. Когда найден другой работающий поток, контекст переключается в этот поток. Прелесть всего этого в том, что LWP, выполняющий поток, не нужно информировать: переключение контекста полностью реализуется в пространстве пользователя и представляется LWP как нормальный программный код.

Теперь давайте посмотрим, что происходит, когда поток выполняет блокирующий системный вызов. В этом случае выполнение изменяется с пользовательского режима на режим ядра, но все еще продолжается в контексте текущего LWP. В тот момент, когда текущий LWP больше не может продолжаться, операционная система может решить переключить контекст на другой LWP, что также подразумевает, что контекст переключается обратно в режим пользователя. Выбранная LWP просто продолжит работу с того места, где она была ранее остановлена.

Есть несколько преимуществ использования LWP в сочетании с пакетом потоков на уровне пользователя. Во-первых, создание, уничтожение и синхронизация потоков относительно дешевы и вообще не требуют вмешательства ядра. Во-вторых, при условии что у процесса достаточно LWP, блокирующий системный вызов не приостановит весь процесс. В-третьих, приложению не нужно знать о LWP. Все, что он видит, – это потоки на уровне пользователя. В-четвертых, LWP можно легко использовать в многопроцессорных средах, выполняя разные LWP на разных процессорах. Эта многопроцессорная обработка может быть полностью скрыта от приложения. Единственный недостаток облегченных процессов в сочетании с потоками пользовательского уровня заключается в том, что нам все еще необходимо создавать и уничтожать LWP, что так же дорого, как и для потоков на уровне ядра. Однако создание и уничтожение LWP необходимо выполнять только изредка, и это часто полностью контролируется операционной системой.

Альтернативный, но схожий подход к облегченным процессам заключается в использовании **активаций планировщика** (scheduler activations) [Anderson et al., 1991]. Самое существенное отличие между активациями планировщика и LWP заключается в том, что когда поток блокирует системный вызов, ядро выполняет обратный вызов пакета потока, фактически вызывая подпрограмму планировщика для выбора следующего запускаемого потока. Эта же процедура повторяется, когда поток разблокирован. Преимущество данного подхода в том, что он сохраняет управление LWP ядром. Однако использование вызовов `upcall` считается менее элегантным, поскольку оно нарушает структуру многоуровневых систем, в которых разрешены вызовы только на следующий более низкий уровень.

Из-за сложностей, возникающих при развертывании модели «много к многим», и относительно низкого прироста производительности (который теперь считается определяемым поведением кеширования) часто предпочтительна простая модель «один к одному».

В заключение отметим, что важно понимать, что использование потоков является одним из способов организации одновременных и параллельных действий в приложении. На практике мы часто видим, что приложения создаются как совокупность параллельных процессов, совместно использующих средства межпроцессного взаимодействия, предлагаемые операционной системой (см. также [Robbins and Robbins, 2003; Stevens, 1999]). Хорошим примером такого подхода является организация веб-сервера Apache, который по умолчанию начинается с нескольких процессов для обработки входящих

запросов. Каждый процесс формирует однопоточную реализацию сервера, но способен взаимодействовать с другими экземплярами с помощью стандартных средств.

Как утверждается в [Srinivasan, 2010], использование процессов вместо потоков имеет важное преимущество разделения пространства данных: каждый процесс работает со своей собственной частью данных и защищен от вмешательства других через операционную систему. Преимущество такого разделения не следует недооценивать: программирование потоков считается крайне сложным, поскольку разработчик несет полную ответственность за управление одновременным доступом к совместно используемым данным. При использовании процессов пространства данных в конечном счете защищены аппаратной поддержкой. Если процесс пытается получить доступ к данным за пределами выделенной ему памяти, аппаратная часть выдает исключение, которое затем обрабатывается операционной системой. Такая поддержка недоступна для потоков, одновременно работающих в одном и том же процессе.

Потоки в распределенных системах

Важным свойством потоков является то, что они могут предоставлять удобные средства, позволяющие блокировать системные вызовы, не блокируя весь процесс, в котором выполняется поток. Это свойство делает потоки особенно привлекательными для использования в распределенных системах, поскольку значительно упрощает передачу данных в форме поддержки нескольких логических соединений одновременно. Мы проиллюстрируем этот момент более детальным рассмотрением многопоточных клиентов и серверов соответственно.

Многопоточные клиенты

Чтобы установить высокую степень прозрачности распределения, распределенным системам, работающим в глобальных сетях, может потребоваться на длительное время скрыть распространение сообщений между процессами. Задержка приема-передачи в глобальной сети может легко составлять порядка сотен миллисекунд, а иногда даже секунд.

Обычный способ скрыть задержки в коммуникации – это инициировать коммуникацию и немедленно приступить к чему-то другому. Типичный пример того, где это происходит, – веб-браузеры. Во многих случаях веб-документ состоит из HTML-файла, содержащего простой текст, а также коллекции изображений, значков и т. д. Чтобы получить каждый элемент веб-документа, браузер должен установить соединение TCP/IP, прочитать входящие данные и передать их компоненту отображения. Настройка соединения, а также чтение входящих данных по своей сути блокируют операции. При работе с дальней связью также существует недостаток, заключающийся в том, что время для завершения каждой операции может быть относительно долгим.

Веб-браузер часто начинается с выборки HTML-страницы и затем отображает ее. Чтобы максимально скрыть задержки связи, некоторые браузеры начинают отображать данные, когда они еще поступают. Когда текст доступен пользователю, включая средства для прокрутки и т. д., браузер продолжает извлекать другие файлы, которые составляют страницы, такие как изображения. Последние отображаются по мере их поступления. Таким образом, пользователю не нужно ждать, пока все компоненты всей страницы будут выбраны и страница станет доступной.

В сущности, видно, что веб-браузер одновременно выполняет несколько задач. Как оказалось, разработка браузера как многопоточного клиента значительно упрощает работу. Как только основной файл HTML был выбран, можно активировать отдельные потоки, чтобы позаботиться о получении других частей. Каждый поток устанавливает отдельное соединение с сервером и извлекает данные. Настройка соединения и чтение данных с сервера могут быть запрограммированы с использованием стандартных (блокирующих) системных вызовов, при условии что блокирующий вызов не приостанавливает весь процесс. Как показано в [Stevens, 1998], код для каждого потока одинаков и прежде всего прост. Между тем пользователь замечает только задержки при отображении изображений и т. п., но может просматривать документ.

Есть еще одно важное преимущество использования многопоточных веб-браузеров, в которых несколько соединений могут быть открыты одновременно. В предыдущем примере несколько соединений были настроены на один и тот же сервер. Если этот сервер сильно загружен или просто слишком медленный, реального улучшения производительности по сравнению с добавлением файлов, которые составляют страницу строго один за другим, не будет замечено.

Однако во многих случаях веб-серверы реплицируются на несколько компьютеров, где каждый сервер предоставляет одинаковый набор веб-документов. Реплицированные серверы расположены на одном сайте и известны под тем же именем. Когда поступает запрос на веб-страницу, запрос перенаправляется на один из серверов, часто с использованием стратегии циклического перебора или другого метода балансировки нагрузки. При использовании многопоточного клиента соединения могут быть настроены на разные реплики, что позволяет передавать данные параллельно, эффективно устанавливая, что весь веб-документ полностью отображается за гораздо более короткое время, чем с нереплицированным сервером. Такой подход возможен, только если клиент действительно может обрабатывать параллельные потоки входящих данных. Потоки идеально подходят для этой цели.

Примечание 3.3 (дополнительно: использование потоков на стороне клиента для повышения производительности)

Хотя существуют очевидные возможности использования потоков для достижения высокой производительности, интересно посмотреть, эффективно ли используется многопоточность. Чтобы увидеть, в какой степени несколько потоков задействуют многоядерный процессор, в работе [Blake et al., 2010] рассмотрено выполнение различных приложений на современных архитектурах. Браузеры, как и многие другие

клиентские приложения, по своей природе интерактивны, поэтому ожидаемое время простоя процессора может быть довольно высоким. Чтобы правильно измерить, в какой степени используется многоядерный процессор, в [Blake et al., 2010] применяется метрика, известная как **параллелизм на уровне потоков** (thread-level parallelism, TLP). Пусть c_i обозначает долю времени, когда ровно i потоков выполняются одновременно.

Параллелизм на уровне потока тогда определяется как

$$TLP = \frac{\sum_{i-1}^{N} i \cdot c_i}{1 - c_0},$$

где N – максимальное количество потоков, которые (могут) выполняться одновременно. В своем исследовании типичный веб-браузер имел значение TLP между 1,5 и 2,5, что означает, что для эффективного использования параллелизма клиентский компьютер должен иметь два или три ядра, или, аналогично, 2–3 процессора.

Эти результаты интересны, если учесть, что современные веб-браузеры создают сотни потоков и что одновременно активны десятки потоков (обратите внимание, что активный поток не обязательно работает; он может быть заблокирован в ожидании завершения запроса ввода-вывода). Таким образом, мы видим, что многопоточность используется для организации приложения, но эта многопоточность не приводит к значительному повышению производительности за счет использования аппаратного обеспечения. То, что браузеры могут быть эффективно спроектированы для использования параллелизма, показано, например, в [Meyerovich and Bodik, 2010]. Адаптируя существующие алгоритмы, авторам удается увеличить скорость в несколько раз.

Многопоточные серверы

Хотя многопоточные клиенты имеют важные преимущества, основное использование многопоточности в распределенных системах приходит со стороны сервера. Практика показывает, что многопоточность не только значительно упрощает серверный код, но и значительно упрощает разработку серверов, использующих параллелизм для достижения высокой производительности даже в однопроцессорных системах. С современными многоядерными процессорами многопоточность для параллелизма – очевидный путь. Чтобы понять преимущества потоков для написания серверного кода, рассмотрим организацию файлового сервера, который иногда должен блокировать ожидание диска. Файловый сервер обычно ожидает входящий запрос для файловой операции, выполняет запрос и затем отправляет ответ обратно. Одна из возможных и особенно популярных организаций показана на рис. 3.4. Здесь один поток, **диспетчер** (dispatcher), читает входящие запросы для файловой операции. Запросы отправляются клиентами в известную конечную точку для этого сервера. После изучения запроса сервер выбирает свободный (т. е. заблокированный) **рабочий поток** (worker thread) и передает ему запрос.

Рабочий поток продолжает выполнение блокирующего чтения в *локальной* файловой системе, что может привести к приостановке потока до тех пор, пока данные не будут извлечены с диска. Если поток приостановлен, другой поток выбирается для выполнения. Например, диспетчер может быть вы-

бран, чтобы запросить дополнительную работу. В качестве альтернативы можно выбрать другой рабочий поток, который уже готов к запуску.

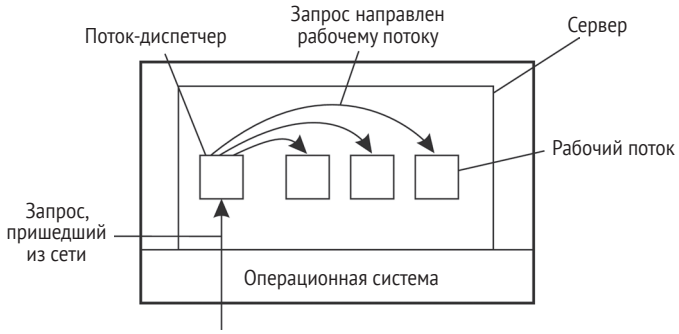


Рис. 3.4 ❖ Многопоточный сервер, организованный по модели диспетчер / рабочий поток

Теперь рассмотрим, как файл-сервер мог быть записан при отсутствии потоков. Одна возможность состоит в том, чтобы это работало как единственный поток. Основной цикл файлового сервера получает запрос, проверяет его и выполняет до завершения, прежде чем получить следующий. В ожидании диска сервер простаивает и не обрабатывает иные запросы. Следовательно, запросы от других клиентов не могут быть обработаны. Кроме того, если файловый сервер работает на выделенном компьютере, как это обычно бывает, процессор просто простаивает, пока файловый сервер ожидает диска. В результате получается, что за единицу времени можно обрабатывать гораздо меньше запросов. Таким образом, потоки получают существенную производительность, но каждый поток программируется, как обычно, последовательно.

До сих пор мы видели два возможных варианта проектирования: многопоточный файловый сервер и однопоточный файловый сервер. Третий вариант – запустить сервер как большой однопоточный конечный автомат. Когда приходит запрос, его проверяет один-единственный поток. Если это может быть обеспечено из внутренней памяти кеша, отлично, но если нет, поток должен быть обеспечен диском. Однако вместо запуска операции с блокированным диском поток планирует асинхронную (то есть неблокирующую) операцию с диском, которая впоследствии будет прервана операционной системой. Чтобы это работало, поток записывает состояние запроса (что у него есть ожидающая операция на диске) и продолжает проверять наличие других входящих запросов, требующих его внимания.

Как только ожидающая операция с диском будет завершена, операционная система уведомит поток, который затем своевременно просмотрит состояние соответствующего запроса и продолжит его обработку. В конечном итоге ответ будет отправлен исходному клиенту, вновь используя неблокирующий вызов для отправки сообщения по сети.

В этой схеме модель «последовательного процесса», которая у нас в первых двух случаях, была утрачена. Каждый раз, когда потоку нужно выполнить

операцию блокировки, ему необходимо записывать, где именно он находился при обработке запроса, возможно, также сохраняя дополнительное состояние. Как только это будет сделано, он может начать операцию и продолжить другую работу. Другая работа означает обработку вновь поступивших запросов или запросов постобработки, для которых завершена ранее запущенная операция. Конечно, если работа не выполняется, поток действительно может блокироваться. По сути, мы имитируем поведение нескольких потоков и их соответствующих стеков сложным способом. Процесс работает как конечный автомат, который получает событие, а затем реагирует на него, в зависимости от того, что в нем находится.

Модель	Характеристики
Многопоточность	Параллелизм, блокировка системных вызовов
Однопоточный процесс	Нет параллелизма, блокировка системных вызовов
Конечный автомат	Параллелизм, неблокирующие системные вызовы

Рис. 3.5 ❖ Три способа построения сервера

Теперь должно быть ясно, что могут предложить потоки. Они позволяют сохранить идею последовательных процессов, которые блокируют системные вызовы и все же достигают параллелизма. Блокировка системных вызовов упрощает программирование, поскольку они выглядят как обычные вызовы процедур. Кроме того, несколько потоков допускают параллелизм и, следовательно, повышение производительности. Однопоточный сервер сохраняет простоту и простоту блокировки системных вызовов, но может серьезно снизить производительность с точки зрения количества запросов, которые могут быть обработаны за единицу времени. Подход с использованием конечного автомата обеспечивает высокую производительность благодаря параллелизму, но использует неблокирующие вызовы, которые обычно сложно программировать и, следовательно, поддерживать. Эти модели приведены на рис. 3.5.

Опять же, обратите внимание, что вместо применения потоков мы также можем использовать несколько процессов для организации сервера (что приводит к ситуации, когда у нас фактически есть многопроцессорный сервер). Преимущество заключается в том, что операционная система может предложить больше защиты от случайного доступа к общим данным. Однако если процессам нужно много общаться, в этом случае можно получить заметное влияние на производительность по сравнению с использованием потоков.

3.2. Виртуализация

Потоки и процессы можно рассматривать как способ делать в одно и то же время больше. По сути, они позволяют нам создавать (части) программы, которые, как кажется, выполняются одновременно. На однопроцессорном компьютере такое одновременное выполнение – конечно, иллюзия. Посколь-

ку имеется только один процессор, одновременно будет выполняться только инструкция из одного потока или процесса. Быстрое переключение между потоками и процессами создает иллюзию параллелизма.

Это разделение между наличием одного ЦП и возможностью притворяться, что их больше, может быть распространено и на другие ресурсы, что приводит к так называемой виртуализации ресурсов. Эта виртуализация применялась в течение многих десятилетий, но получила новый интерес, поскольку (распределенные) компьютерные системы стали более распространенными и сложными, и это привело к ситуации, когда прикладное программное обеспечение в большинстве случаев всегда живет дольше своих базовых системных программ и аппаратных средств.

Принцип виртуализации

На практике каждая (распределенная) компьютерная система предлагает программный интерфейс для программного обеспечения более высокого уровня, как показано на рис. 3.6a. Существует много различных типов интерфейсов, начиная от базового набора команд, предлагаемого ЦП, до обширной коллекции интерфейсов прикладного программирования, которые поставляются со многими современными системами промежуточного программного обеспечения. По своей сути виртуализация имеет дело с расширением или заменой существующего интерфейса, чтобы имитировать поведение другой системы, как показано на рис. 3.6b. Вскоре мы обсудим технические детали виртуализации, но давайте сначала сосредоточимся на том, почему виртуализация важна.

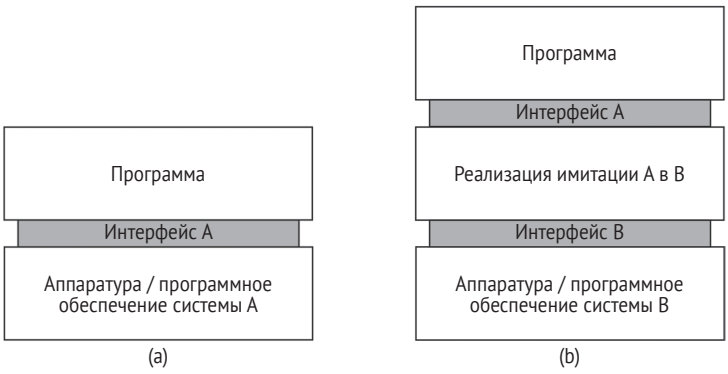


Рис. 3.6 ❖ Общая организация:
a) между программой, интерфейсом и системой;
b) виртуализации системы A поверх B

Виртуализация и распределенные системы

Одной из наиболее важных причин внедрения виртуализации еще в 1970-х годах было то, что устаревшее программное обеспечение работало на дорогом

оборудовании мейнфреймов. Программное обеспечение включало в себя не только различные приложения, но и операционные системы, для которых они были разработаны. Этот подход к поддержке устаревшего программного обеспечения был успешно применен на мейнфреймах IBM 370 (и их преемниках), которые предлагали виртуальную машину, на которую были перенесены различные операционные системы.

По мере того как аппаратное обеспечение дешевело, компьютеры становились все более мощными, а количество различных операционных систем уменьшалось, потребность в виртуализации становилась все меньше.

Однако с конца 1990-х годов ситуация снова изменилась. Во-первых, в то время как аппаратное обеспечение и программное обеспечение систем низкого уровня изменялись достаточно быстро, программное обеспечение на более высоких уровнях абстракции (например, промежуточное программное обеспечение и приложения) часто оставалось намного более стабильным. Другими словами, мы сталкиваемся с ситуацией, когда устаревшее программное обеспечение нельзя поддерживать в том же темпе, что и платформы, на которые оно опирается. Виртуализация может помочь в этом, перенося устаревшие интерфейсы на новые платформы и, таким образом, открывая последние для многих классов существующих программ.

Примечание 3.4 (обсуждение: стабильное программное обеспечение?)

Хотя действительно существует много устаревшего программного обеспечения, которое может использовать преимущества стабильных интерфейсов для быстро меняющегося базового оборудования, ошибочно полагать, что программное обеспечение для широко доступных служб почти не меняется. В связи с растущим сдвигом в сторону серверных вычислений в форме **программного обеспечения как услуги** (Software-as-a-Service, SaaS) большое количество программного обеспечения можно использовать для относительно однородной платформы, полностью принадлежащей организации, предлагающей соответствующие услуги. Как следствие обслуживание программных продуктов может быть намного проще, поскольку существует гораздо меньшая потребность в распространении изменений среди потенциальных миллионов клиентов. Фактически изменения могут быстро сменять друг друга при изменениях в доступном оборудовании и платформе, и без того, чтобы клиент замечал простои [Barroso and Hölze, 2009].

Не менее важным является тот факт, что создание сетей стало широко распространенным явлением. Трудно представить себе, чтобы современный компьютер не был подключен к сети. На практике это подключение требует, чтобы системные администраторы имели большой и разнородный набор серверных компьютеров, на каждом из которых выполнялись бы совершенно разные приложения, к которым клиенты могли бы обращаться. В то же время различные ресурсы должны быть легкодоступны для этих приложений. Виртуализация может сильно помочь: разнообразие платформ и машин может быть уменьшено за счет того, что каждое приложение может работать на своей собственной виртуальной машине, возможно, включая связанные библиотеки и операционную систему, которые, в свою очередь, работают на общей платформе.

Этот последний тип виртуализации обеспечивает высокую степень мобильности и гибкости. Например, чтобы реализовать сети доставки контента, которые могут легко поддерживать репликацию динамического контента. В работе [Awadallah and Rosenblum, 2002] утверждаетя, что управление станет намного проще, если пограничные серверы будут поддерживать виртуализацию, позволяя создать полноценный сайт, включая динамическое копирование его среды. Эти аргументы остаются в силе, и действительно, переносимость является, пожалуй, самой важной причиной, почему виртуализация играет столь ключевую роль во многих распределенных системах.

Типы виртуализации

Существует много разных способов реализации виртуализации. Обзор этих различных подходов описан в [Smith and Nair, 2005a]. Чтобы понять различия в виртуализации, важно понимать, что компьютерные системы обычно предлагают четыре различных типа интерфейсов на трех разных уровнях.

1. Интерфейс между аппаратным и программным обеспечением, называемый **архитектурой набора инструкций** (instruction set architecture, ISA), формирующий набор машинных инструкций. Этот набор разделен на два подмножества:
   - привилегированные инструкции, которые разрешено выполнять только операционной системе;
   - общие инструкции, которые могут быть выполнены любой программой.
2. Интерфейс, состоящий из системных вызовов, предлагаемых операционной системой.
3. Интерфейс, состоящий из библиотечных вызовов, обычно формирующих то, что известно как **интерфейс прикладного программирования** (application programming interface, API). Во многих случаях упомянутые системные вызовы скрыты API.

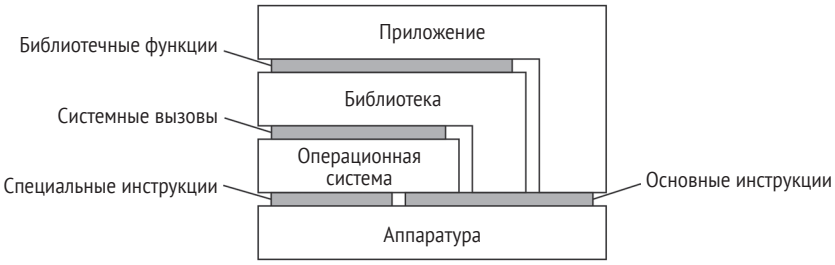


Рис. 3.7 ❖ Различные интерфейсы, предлагаемые компьютерными системами

Эти различные типы показаны на рис. 3.7. Суть виртуализации заключается в том, чтобы имитировать поведение этих интерфейсов.

Виртуализация может осуществляться двумя различными способами. Во-первых, мы можем построить систему времени выполнения, по существу

предоставляющую абстрактный набор команд, который должен использоваться для выполнения приложений. Инструкции можно интерпретировать (как в случае среды выполнения в Java), но также можно эмулировать, как это делается для запуска приложений Windows на платформах Unix. Обратите внимание, что в последнем случае эмулятор также должен будет имитировать поведение системных вызовов, что, как правило, далеко не тривиально. Этот тип виртуализации, показанный на рис. 3.8a, приводит к тому, что в [Smith and Nair, 2005a] называют **виртуальной машиной процесса** (process virtual machine), подчеркивая, что виртуализация предназначена только для одного процесса.

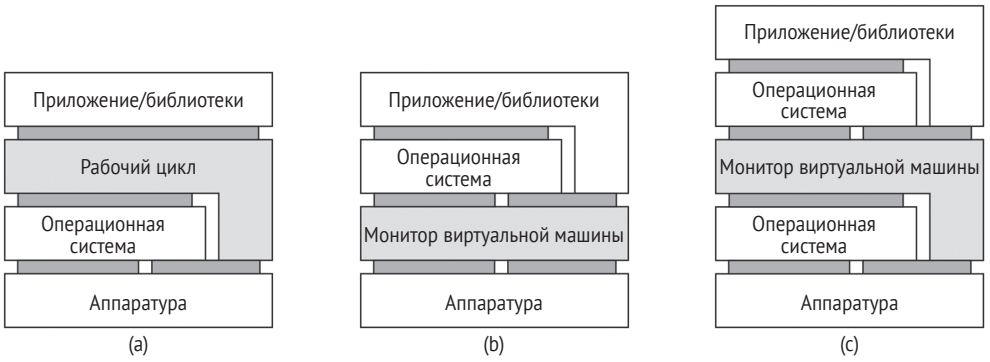


Рис. 3.8 ❖ a) Виртуальная машина процесса;
b) собственный монитор виртуальной машины;
c) размещенный монитор виртуальной машины

Альтернативный подход к виртуализации, показанный на рис. 3.8b, заключается в предоставлении системы, которая реализована в виде уровня, экранирующего исходное оборудование, но предлагающего полный набор инструкций того же (или другого) оборудования в качестве интерфейса. Это приводит к тому, что известно как **собственный монитор виртуальной машины** (native virtual machine monitor). Он называется собственным, потому что реализован непосредственно поверх базового оборудования. Обратите внимание, что интерфейс, предлагаемый монитором виртуальной машины, может предлагаться *одновременно* различным программам. В результате теперь можно иметь несколько и разные **гостевые операционные системы** (guest operating systems), работающие независимо и одновременно на одной платформе.

Собственный монитор виртуальной машины должен обеспечивать и регулировать доступ к различным ресурсам, таким как внешнее хранилище и сети. Как и в любой другой операционной системе, это означает, что для этих ресурсов потребуется реализовать драйверы устройств. Вместо того чтобы делать все это заново, **размещенный монитор виртуальной машины** (hosted virtual machine monitor) будет работать поверх **доверенной операционной системы хоста** (trusted host operating system), как показано на рис. 3.8c. В этом случае монитор виртуальной машины может исполь-

зовать существующие средства, предоставляемые операционной системой хоста. Как правило, ему должны быть даны специальные привилегии вместо работы в качестве приложения пользовательского уровня. Применение размещенного монитора виртуальной машины очень популярно в современных распределенных системах, таких как центры обработки данных и облака.

Как утверждается в работе [Rosenblum and Garfinkel, 2005], виртуальные машины приобретают все большее значение в контексте надежности и безопасности (распределенных) систем. Поскольку они допускают изоляцию всего приложения и его среды, сбой, вызванный ошибкой или атакой безопасности, больше не будет отражаться на всей машине. Кроме того, как мы уже упоминали ранее, мобильность значительно улучшена, поскольку виртуальные машины обеспечивают дальнейшее разделение аппаратного и программного обеспечения, позволяя перемещать всю среду с одного компьютера на другой. Мы вернемся к миграции в разделе 3.5.

Примечание 3.5 (дополнительно: о производительности виртуальных машин)

Виртуальные машины работают на удивление хорошо. Фактически многие исследования показывают, что современные виртуальные машины работают близко к запущенным приложениям непосредственно в операционной системе хоста. Давайте подробнее рассмотрим, что происходит под капотом виртуальных машин. Подробный и полный отчет о виртуальных машинах предоставлен в [Smith and Nair, 2005b].

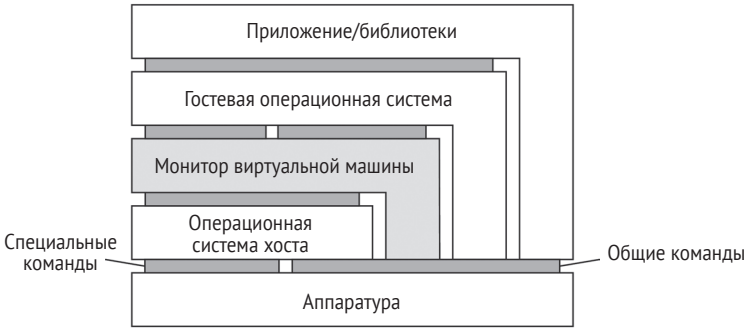


Рис. 3.9 ❖ Приложения, гостевая операционная система, монитор виртуальной машины и хост-система на одной аппаратной платформе

Часть ответа на проблему производительности показана на рис. 3.9, который образует расширение на рис. 3.8c: большая часть кода, составляющая монитор виртуальной машины, гостевую операционную систему и приложение, изначально работает на базовом оборудовании. В частности, все общие (то есть неспециальные) машинные команды непосредственно выполняются базовой машиной, а не интерпретируются или эмулируются.

Этот подход не нов и основан на исследованиях [Popek and Goldberg, 1974], которые формализовали требования для эффективного выполнения виртуальных машин. В сущности, Попек (Popek) и Гольдберг (Goldberg) предположили, что базовый компьютер обеспечивает по крайней мере два режима работы (системный и пользовательский), что подмножество команд может быть выполнено только

в системном режиме и что адресация памяти была относительной (т. е. физический адрес был получен путем добавления относительного адреса к смещению, найденному в регистре перемещения). Далее было проведено различие между двумя типами инструкций. **Специальная** (**привилегированная**) команда (control-sensitive instruction) – это инструкция, которая характеризуется тем, что если, и только если, она выполняется пользователем, то является для операционной системы причиной **ловушки** (trap). Все остальные команды являются **непривилегированными**.

Учитывая эти формальные предположения, Попек и Гольдберг определили два класса специальных команд. **Чувствительная к управлению инструкция** (control-sensitive instruction) – это инструкция, которая может повлиять на конфигурацию машины. Типичным примером является инструкция, которая влияет на структуру памяти, например путем изменения смещения памяти, хранящегося в регистре перемещения. Другим примером являются инструкции, которые влияют на таблицу прерываний, содержащую указатели на обработчики прерываний.

Инструкция, чувствительная к поведению (behavior-sensitive instruction) – это команда, действие которой частично определяется контекстом, в котором она выполняется. Например, процессоры Intel x86 имеют инструкции, которые могут влиять или не влиять на определенные регистры, в зависимости от того, выполняется ли эта инструкция в системном режиме или режиме пользователя. Примером, приведенным в [Smith and Nair, 2005b], является инструкция POPF, которая может устанавливать флаг с поддержкой прерываний, но только при выполнении в системном режиме.

Теперь у нас есть следующий важный результат:

> *Для любого обычного компьютера может быть создан монитор виртуальной машины, если набор чувствительных инструкций для этого компьютера является подмножеством набора привилегированных инструкций*.

Это говорит о том, что до тех пор, пока конфиденциальные инструкции перехватываются при выполнении в пользовательском режиме, мы можем безопасно выполнять все нечувствительные инструкции непосредственно на базовом оборудовании. Это также означает, что при проектировании наборов инструкций, если мы позаботимся о том, чтобы вышеуказанное требование было выполнено, мы не будем создавать ненужных препятствий для эффективной виртуализации этого набора инструкций.

К сожалению, не все наборы инструкций имеют конфиденциальные инструкции только для привилегированных пользователей, в том числе, пожалуй, самый популярный – набор инструкций Intel x86. Как оказалось, в этом наборе 17 конфиденциальных инструкций, которые не являются привилегированными [Robin and Irvine, 2000]. Другими словами, каждая из этих инструкций может быть выполнена в пользовательском режиме, не вызывая ловушку для операционной системы, но влияет на то, как операционная система управляет своими ресурсами. В этих случаях существует два решения.

Первое решение состоит в том, чтобы эмулировать все инструкции. Конечно, это может серьезно повлиять на производительность. Чтобы обойти проблемы, подход, реализованный в виртуальной машине VMWare [Sugerman et al., 2001], состоит в том, чтобы сканировать исполняемый файл и вставлять код в непривилегированные конфиденциальные инструкции, чтобы перенаправить управление на монитор виртуальной машины. Там будет происходить соответствующая эмуляция, например с учетом контекста, в котором должна была выполняться инструкция. В результате может произойти полная виртуализация, а это означает, что выполнение может происходить без изменения гостевой операционной системы или самого приложения.

Альтернативное решение – применить **паравиртуализацию** (paravirtualization), которая требует модификации гостевой операционной системы. В частности, гостевая операционная система модифицирована таким образом, что рассматриваются все побочные эффекты запуска непривилегированных конфиденциальных инструкций в пользовательском режиме, которые обычно выполняются в системном режиме. Например, код можно переписать так, чтобы эти инструкции просто больше не встречались, или, если они это делают, их семантика одинакова независимо от того, выполняется в пользовательском или системном режиме. Паравиртуализация была адаптирована Xen [Barham et al., 2003; Chisnall, 2007].

Применение виртуальных машин в распределенных системах

С точки зрения распределенных систем наиболее важным применением виртуализации являются облачные вычисления. Как мы уже упоминали в разделе 1.3, облачные провайдеры предлагают примерно три различных типа услуг:

- **инфраструктура как услуга** (Infrastructure-as-a-Service, IaaS), охватывающая базовую инфраструктуру;
- **платформа как услуга** (Platform-as-a-Service, PaaS), охватывающая услуги системного уровня;
- **программное обеспечение как услуга** (Software-as-a-Service, SaaS), содержащее реальные приложения.

Виртуализация играет ключевую роль в IaaS. Вместо того чтобы сдавать в аренду физическую машину, поставщик облачных услуг будет сдавать в аренду виртуальную машину (монитор), которая может или не может совместно использовать физическую машину с другими клиентами. Прелесть виртуализации заключается в том, что она обеспечивает практически полную изоляцию между клиентами, у которых действительно будет иллюзия, что они только что арендовали выделенную физическую машину. Однако изоляция никогда не будет полной хотя бы потому, что фактические физические ресурсы используются совместно, что, в свою очередь, приводит к заметному снижению производительности.

Чтобы конкретизировать ситуацию, давайте рассмотрим облако **Amazon Elastic Compute Cloud**, или просто **EC2**. Облако **EC2** позволяет создать среду, состоящую из нескольких сетевых виртуальных серверов, которые вместе образуют основу распределенной системы. Проще говоря, доступно (большое) количество предварительно сконфигурированных образов машин, называемых **образами машин Amazon** (Amazon Machine Images), или просто **AMI**. AMI – это устанавливаемый программный пакет, состоящий из ядра операционной системы и ряда сервисов. Примером простого базового AMI является образ **LAMP**, состоящий из ядра Linux, веб-сервера Apache, системы баз данных MySQL и библиотек PHP. Также доступны более сложные образы, содержащие дополнительное программное обеспечение, а также образы, основанные на других ядрах Unix или Windows. В этом смысле AMI – по сути то же, что и загрузочный диск (хотя есть несколько важных отличий, к которым мы вскоре вернемся).

Клиент EC2 должен выбрать AMI, возможно, после его адаптации или настройки. Затем AMI может быть *запущен*, что станет тем, что называется экземпляром EC2: реальной виртуальной машиной, которую можно использовать для размещения приложений клиента. Важной проблемой является то, что клиент вряд ли когда-либо точно знает, где в действительности выполняется экземпляр машины. Очевидно, что он работает на одной физической машине, но там, где эта машина находится, остается скрытым. Самое близкое к тому местоположению, где запускается экземпляр, и которое можно определить – это один из нескольких регионов, предоставляемых Amazon (США, Южная Америка, Европа, Азия).

Для связи каждый экземпляр получает два IP-адреса: частный, который можно использовать для внутренней связи между различными экземплярами, применяя внутренние сетевые средства EC2, и общедоступный IP-адрес, позволяющий любым интернет-клиентам связываться с экземпляром. Общий адрес сопоставляется с частным с использованием **стандартной технологии преобразования сетевых адресов** (Network Address Translation, NAT). Простой способ управления экземпляром – использовать SSH-соединение, для которого Amazon предоставляет средства для генерации соответствующих ключей.

Среда EC2, в которой выполняется экземпляр, предоставляет различные уровни следующих служб:

- **ЦП**: позволяет выбрать количество и тип ядер, включая графические процессоры;
- **память**: определяет, сколько основной памяти выделено для экземпляра;
- **хранилище**: определяет, сколько выделено локального хранилища;
- **платформа**: различает 32- и 64-разрядные архитектуры;
- **сеть**: задает пропускную способность, которую можно использовать.

Кроме того, могут быть запрошены дополнительные ресурсы, такие как дополнительный сетевой интерфейс. Локальное хранилище, которое поставляется с экземпляром, является временным: когда экземпляр останавливается, все данные, хранящиеся локально, теряются. Чтобы предотвратить потерю данных, клиенту необходимо явно сохранить данные в постоянном хранилище, например с помощью Amazon Simple Storage Service (S3). Альтернативой является подключение устройства хранения, предлагаемого Amazon's **Elastic Block Store** (**Amazon EBS**). Это еще одна служба, и она может быть использована в виде виртуального блочного устройства, которое просто монтируется, как если бы был подключен дополнительный жесткий диск. Когда экземпляр остановлен, все данные, которые были сохранены в EBS, сохранятся. И как и следовало ожидать, устройство EBS может быть (пере)подключено к любому другому экземпляру.

Не вдаваясь в какой-либо значительный уровень детализации, теперь должно быть ясно, что IaaS, предлагаемый EC2, позволяет клиенту создавать (потенциально большое) количество виртуальных машин, каждая из которых сконфигурирована с необходимыми ресурсами и способна обеспечить обмен сообщениями через IP-сеть. Кроме того, к этим виртуальным машинам можно получить доступ из любого места через интернет (при условии что

у клиента есть надлежащие учетные данные). Таким образом, Amazon EC2, как и многие другие поставщики IaaS, предлагает средства для конфигурирования полной распределенной системы, состоящей из сетевых виртуальных серверов и запуска предоставляемых заказчиком распределенных приложений. В то же время этим клиентам не нужно будет поддерживать какую-либо физическую машину, что само по себе часто является огромным преимуществом, как не раз увидим в этой книге. Можно утверждать, что виртуализация лежит в основе современных облачных вычислений.

3.3. Клиенты

В предыдущих главах мы обсуждали модель клиент-сервер, роли клиентов и серверов и способы их взаимодействия. Давайте теперь подробнее рассмотрим анатомию соответственно клиентов и серверов. Мы начнем в этом разделе с обсуждения клиентов. Серверы обсуждаются в следующем разделе.

Сетевые пользовательские интерфейсы

Основная задача клиентских машин – предоставить пользователям средства взаимодействия с удаленными серверами. Есть два способа поддержать это взаимодействие. Во-первых, для каждой удаленной службы клиентская машина будет иметь отдельного партнера, который может связаться со службой по сети. Типичным примером является календарь, работающий на смартфоне пользователя, который должен синхронизироваться с удаленным, возможно, общим календарем. В этом случае протокол уровня приложения будет обрабатывать синхронизацию, как показано на рис. 3.10.

Второе решение – предоставить прямой доступ к удаленным сервисам, предлагая только удобный пользовательский интерфейс. Фактически это означает, что клиентский компьютер используется только в качестве терминала без необходимости локального хранения, что приводит к нейтральному для приложения решению, как показано на рис. 3.10b. В случае если сетевые интерфейсы пользовательские, все они обрабатываются и хранятся на сервере.

Этот **подход тонкого клиента** (thin-client approach) получил большое распространение с увеличением интернет-соединений и использованием мобильных устройств. Решения для тонких клиентов также популярны, поскольку облегчают задачу управления системой.

Пример: система X Window

Возможно, одним из самых старых и все еще широко используемых сетевых пользовательских интерфейсов является система **X Window**. Система X Window, обычно называемая просто **X**, используется для управления бит-отображаемыми терминалами, которые включают монитор, клавиату-

ру и указатель типа мышь. Помимо поддержки традиционных терминалов, которые можно найти на настольных компьютерах и рабочих станциях, X также поддерживает современные устройства, такие как сенсорные экраны на планшетах и смартфонах. В некотором смысле X можно рассматривать как часть операционной системы, которая управляет терминалом. Сердцем системы является то, что мы будем называть **ядром X** (X kernel). Оно содержит все специфичные для терминала драйверы устройств и потому, как правило, сильно зависит от оборудования.

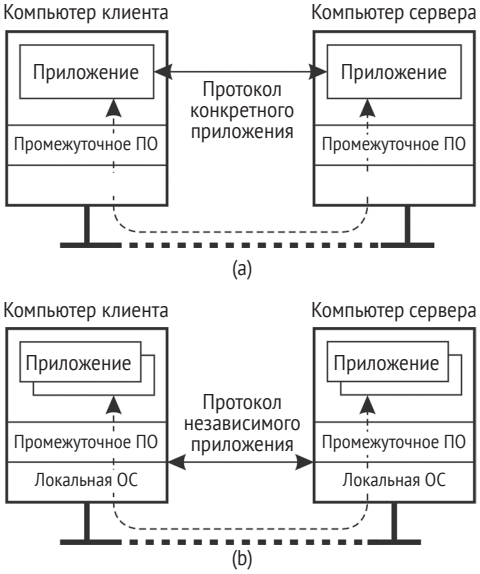


Рис. 3.10 ❖ a) Сетевое приложение со своим собственным протоколом; b) общее решение, позволяющее получить доступ к удаленным приложениям

Ядро X предлагает относительно низкоуровневый интерфейс для управления экраном, а также захвата событий с клавиатуры и мыши. Этот интерфейс доступен для приложений в виде библиотеки под названием Xlib. Общая организация показана на рис. 3.11. Обратите внимание, что Xlib практически никогда не используется непосредственно приложениями, которые вместо этого развертывают более простые в использовании наборы инструментов, реализованные поверх Xlib.

Интересный аспект X состоит в том, что ядро X и приложения X не обязательно должны находиться на одной машине. В частности, X предоставляет протокол X, который является протоколом связи прикладного уровня, с помощью которого экземпляр Xlib может обмениваться данными и событиями с ядром X. Например, Xlib, среди многих других запросов, может отправлять запросы к ядру X для создания или уничтожения окна, установки цветов и определения типа курсора для отображения.

В свою очередь, ядро X будет реагировать на локальные события, такие как ввод с клавиатуры и мыши, отправляя пакеты событий обратно в Xlib.

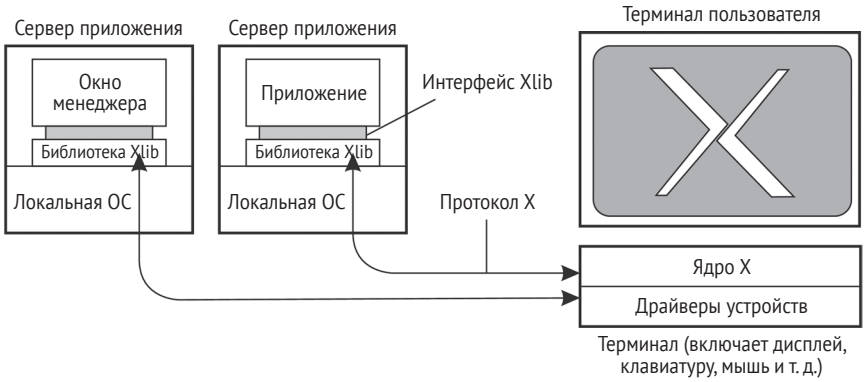


Рис. 3.11 ❖ Базовая организация системы X Windows

Несколько приложений могут одновременно взаимодействовать с ядром X. Существует одно конкретное приложение, которому предоставляются специальные права, известное как **оконный менеджер** (window manager). Это приложение может диктовать «внешний вид» дисплея так, как он отображается пользователю. Например, диспетчер окон может предписывать, как каждое окно украшено дополнительными кнопками, как размещать окна на дисплее и т. д. Другие приложения должны будут придерживаться этих правил. На практике это означает, что большая часть взаимодействия между приложением и X-терминалом перенаправляется через оконный менеджер.

Интересно отметить, как система X Window фактически вписывается в клиент-серверные вычисления. Из того, что мы описали до сих пор, должно быть ясно, что ядро X получает запросы для управления отображением. Оно получает эти запросы от (возможно, удаленных) приложений. В этом смысле ядро X действует как сервер, а приложения играют роль клиентов. Эта терминология была принята в X, и хотя, строго говоря, и верна, она может легко привести к путанице.

Сетевые вычисления для тонких клиентов

Очевидно, что приложения управляют отображением, используя специальные команды отображения, предлагаемые X. Эти команды обычно отправляются по сети, где они впоследствии выполняются ядром X. По своей природе приложения, написанные для X, должны предпочтительно отделять логику приложения от команд пользовательского интерфейса. К сожалению, это часто не так. Как сообщают в [Lai и Nieh, 2002], оказывается, что большая часть логики приложения и взаимодействия с пользователем тесно связаны, что означает, что приложение отправит много запросов к ядру X, на которые оно будет ожидать ответа, прежде чем сможет сделать следующий шаг. Такое синхронное поведение может отрицательно повлиять на производительность при работе в глобальной сети с большими задержками.

Есть несколько решений этой проблемы. Одним из них является реинжиниринг реализации протокола X, как это делается в NX [Pinzari, 2003]. Важной частью этой работы является уменьшение пропускной способности за счет

уменьшения размера X-сообщений. С этой целью считается, что сообщения состоят из фиксированной части, которая рассматривается как идентификатор, и переменной части. Во многих случаях несколько сообщений будут иметь один и тот же идентификатор, и в этом случае они часто будут содержать одинаковые данные. Это свойство может использоваться для отправки только различий между сообщениями, имеющими один и тот же идентификатор. Если отправитель и получатель поддерживают идентификаторы, декодирование в получателе может быть легко применено. Сообщалось об уменьшении пропускной способности до 1000 раз, что позволяет X также работать по каналам с низкой пропускной способностью всего в 9600 Кбит/с.

В качестве альтернативы использованию X исследователи и практики также стремились позволить приложению *полностью* управлять удаленным дисплеем, т. е. повысить уровень пикселей. Изменения в растровом изображении затем отправляются по сети на дисплей, где они немедленно передаются в локальный буфер кадров. Хорошо известным примером этого подхода является **Virtual Network Computing** (**VNC**) [Richardson et al., 1998], который существует с конца 1990-х годов. Очевидно, чтобы приложение могло управлять дисплеем, требуются сложные методы кодирования для предотвращения появления проблем полосы пропускания. Например, рассмотрим отображение видеопотока со скоростью 30 кадров в секунду на простом экране 320×240. Если каждый пиксель кодируется 24 битами, то без эффективной схемы кодирования нам потребуется ширина полосы приблизительно 53 Мбит/с. На практике используются различные методы кодирования, однако выбор лучшего из них обычно зависит от приложения.

Недостаток отправки необработанных пиксельных данных по сравнению с протоколами более высокого уровня, такими как X, состоит в том, что невозможно использовать семантику приложения, поскольку они эффективно теряются на этом уровне. В работе [Baratto et al., 2005] предлагается другая техника. В своем решении, называемом THINC, авторы предоставляют несколько высокоуровневых команд отображения, которые работают на уровне драйверов видеоустройств. Таким образом, эти команды зависят от устройства, более мощны, чем операции с необработанными пикселями, но менее мощны по сравнению с тем, что предлагает такой протокол, как X. В результате серверы отображения могут быть намного проще, что хорошо для использования ЦП, но в то же время зависящие от приложения оптимизации могут использоваться для уменьшения пропускной способности и синхронизации.

Клиентское программное обеспечение для прозрачности распространения

Клиентское программное обеспечение состоит не только из пользовательских интерфейсов. Во многих случаях части обработки и уровня данных в клиент-серверном приложении также выполняются на стороне клиента. Специальный класс формируется встроенным клиентским программным обеспечением, например для банкоматов, кассовых аппаратов, считывателей

штрих-кода, телевизионных приставок и т. д. В этих случаях пользовательский интерфейс является относительно небольшой частью клиентского программного обеспечения, в отличие от местных средств обработки и связи.

Помимо пользовательского интерфейса и другого прикладного программного обеспечения, клиентское программное обеспечение содержит компоненты для обеспечения прозрачности распространения. В идеале клиент не должен знать, что он взаимодействует с удаленными процессами. Напротив, распределение часто менее прозрачно для серверов по причинам производительности и правильности.

Прозрачность доступа обычно обрабатывается путем создания **клиентской заглушки** (client stub) из определения интерфейса того, что сервер может предложить. Заглушка обеспечивает тот же интерфейс, что и на сервере, но скрывает возможные различия в архитектуре машины, а также фактическую связь. Клиентская заглушка преобразует локальные вызовы в сообщения, отправляемые на сервер, и наоборот – преобразует сообщения с сервера, чтобы возвращать значения, как и следовало ожидать при вызове обычной процедуры.

Существуют разные способы управления прозрачностью местоположения, миграции и перемещения. Решающее значение имеет использование удобной системы именования. Во многих случаях также важно сотрудничество с программным обеспечением на стороне клиента. Например, когда клиент уже связан с сервером, он может быть напрямую проинформирован, когда сервер меняет местоположение. В этом случае промежуточное ПО клиента может скрыть текущее сетевое местоположение сервера от пользователя, а также прозрачно выполнить повторную привязку к серверу, если это необходимо. В худшем случае клиентское приложение может заметить временную потерю производительности.

Аналогичным образом многие распределенные системы реализуют прозрачность репликации с помощью решений на стороне клиента. Например, представьте распределенную систему с реплицированными серверами. Такая репликация может быть достигнута путем пересылки запроса каждой реплике, как показано на рис. 3.12. Программное обеспечение на стороне клиента может прозрачно собирать все ответы и передавать один ответ клиентскому приложению.

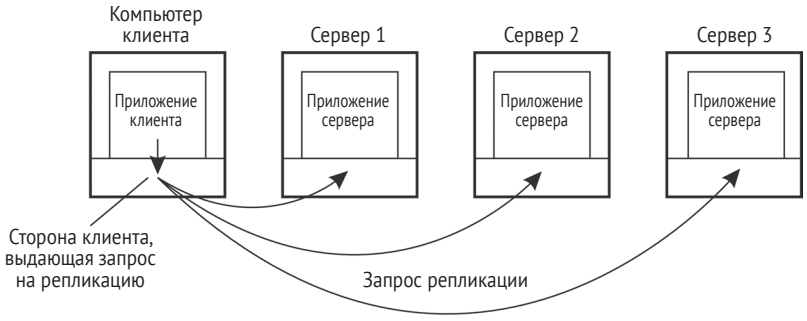


Рис. 3.12 ❖ Прозрачная репликация сервера с использованием решения на стороне клиента

Что касается прозрачности сбоев, то маскирование сбоев связи с сервером обычно выполняется через клиентское промежуточное ПО. Например, клиентское промежуточное программное обеспечение может быть настроено на неоднократных попытках подключения к серверу или возможности после нескольких попыток использовать другой сервер. Существуют даже ситуации, когда клиентское промежуточное ПО возвращает данные, которые он кешировал во время предыдущего сеанса, как это иногда делают веб-браузеры, которые не могут подключиться к серверу.

Наконец, прозрачность параллелизма может быть обработана через специальные промежуточные серверы, особенно мониторы транзакций, и требует меньшей поддержки со стороны клиентского программного обеспечения.

3.4. Серверы

Давайте теперь подробнее рассмотрим организацию серверов. На следующих страницах мы сначала сосредоточимся на ряде общих вопросов проектирования серверов, а затем обсудим кластеры серверов.

Общие вопросы дизайна

Сервер – это процесс, реализующий конкретный сервис от имени группы клиентов. По сути, каждый сервер организован одинаково: он ожидает входящего запроса от клиента и впоследствии гарантирует, что запрос обработан, после чего ожидает следующего входящего запроса.

Параллельный сервер против итеративного сервера

Существует несколько способов организации серверов. **Итеративный сервер** (iterative serve) сам обрабатывает запросы и, если необходимо, возвращает ответ запросившему клиенту. **Параллельный сервер** (concurrent server) не обрабатывает сам запрос, а передает его отдельному потоку или другому процессу, после чего он немедленно ожидает следующего входящего запроса. Многопоточный сервер является примером параллельного сервера. Альтернативная реализация параллельного сервера – это создание нового процесса для каждого нового входящего запроса. Этот подход используется во многих системах Unix. Поток или процесс, который обрабатывает запрос, отвечает за возврат ответа запрашивающему клиенту.

Связь с сервером: конечные точки

Другая проблема заключается в том, в каком месте клиенты связываются с сервером. Во всех случаях клиенты отправляют запросы в **конечную точку** (end point), также называемую **по́ртом** (port), на компьютере, на котором работает сервер. Каждый сервер слушает конкретную конечную точку. Как клиенты узнают о конечной точке службы? Одним из подходов является глобаль-

ное назначение конечных точек для известных услуг. Например, серверы, обрабатывающие FTP-запросы в интернете, всегда прослушивают TCP-порт 21. Аналогично HTTP-сервер для Всемирной паутины всегда будет прослушивать TCP-порт 80. Эти конечные точки были назначены **Управлением назначения номеров интернета** (Internet Assigned Numbers Authority, IANA) и документированы в [Reynolds and Postel, 1994]. С назначенными конечными точками клиент должен найти только сетевой адрес машины, на которой работает сервер. Для этой цели могут быть использованы службы имен.

Существует множество служб, которые не требуют предварительно назначенной конечной точки. Например, сервер времени суток может использовать конечную точку, которая динамически назначается ему локальной операционной системой. В этом случае клиент должен сначала найти конечную точку. Одним из решений является запуск специального демона на каждом компьютере, на котором работают серверы. Демон отслеживает текущую конечную точку каждого сервиса, реализованного на одном и том же сервере. Сам демон слушает известную конечную точку. Клиент сначала свяжется с демоном, запросит конечную точку, а затем свяжется с конкретным сервером, как показано на рис. 3.13a.

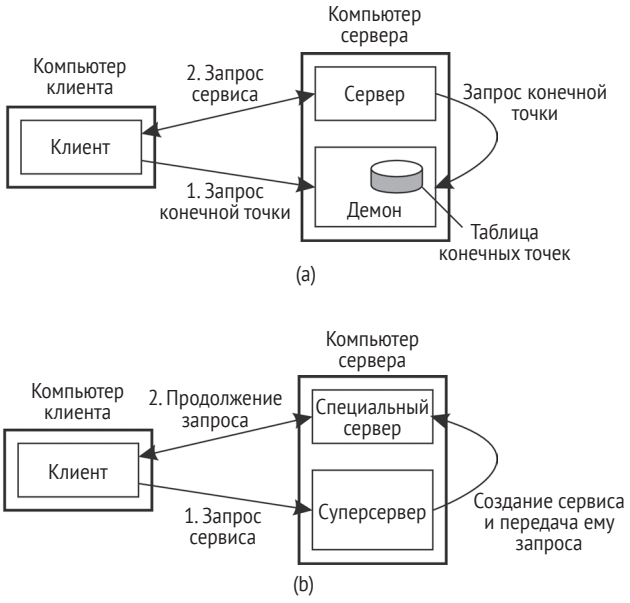


Рис. 3.13 ❖ Привязка клиент-сервер с использованием: a) демона; b) суперсервера

Обычно конечную точку ассоциируют с конкретной службой. Однако фактическая реализация каждой услуги с помощью отдельного сервера может быть пустой тратой ресурсов. Например, в типичной системе Unix часто работают одновременно несколько серверов, причем большинство из них пассивно ожидает, пока не поступит запрос клиента. Вместо того чтобы от-

слеживать столько пассивных процессов, зачастую более эффективно иметь один суперсервер, прослушивающий каждую конечную точку, связанную с конкретной службой, как показано на рис. 3.13b. Например, демон `inetd` в Unix прослушивает несколько известных портов для интернет-сервисов. Когда приходит запрос, демон разветвляет процесс для его обработки. Этот процесс заканчивается после завершения обработки.

Прерывание сервера

Другая проблема, которую необходимо учитывать при разработке сервера, заключается в том, может ли сервер быть прерван и каким образом. Например, рассмотрим пользователя, который только что решил загрузить огромный файл на FTP-сервер. Затем, внезапно осознав, что это неправильный файл, он хочет прервать работу сервера, чтобы отменить дальнейшую передачу данных. Есть несколько способов сделать это. Один из подходов, который хорошо работает в текущем интернете (и иногда является единственной альтернативой), заключается в том, что пользователь внезапно завершает работу клиентского приложения (которое автоматически разорвет соединение с сервером), немедленно перезапускает его и делает вид, что ничего не произошло. Сервер со временем разорвет старое соединение, думая, что клиент, вероятно, сломался.

Гораздо лучший подход к обработке прерываний связи заключается в разработке клиента и сервера таким образом, чтобы их можно было отправлять вне **полосы данных** (out-of-band), которые должны были быть обработаны сервером перед любыми другими данными от этого клиента. Одно из решений состоит в том, чтобы позволить серверу прослушивать отдельную контрольную точку управления, в которую клиент отправляет внеполосные данные, и в то же время прослушивать (с более низким приоритетом) конечную точку, через которую проходят обычные данные. Другое решение заключается в отправке внеполосных данных по тому же соединению, через которое клиент отправляет исходный запрос. В TCP, например, можно передавать срочные данные. Когда срочные данные принимаются на сервере, последний прерывается (например, посредством сигнала в системах Unix), после чего он может проверять данные и обрабатывать их соответствующим образом.

Серверы без сохранения состояния против серверов, сохраняющих состояние

И наконец, важная проблема проектирования – сохраняет ли сервер состояние. Сервер без сохранения состояния не хранит информацию о состоянии своих клиентов и может изменять свое собственное состояние без необходимости информирования об этом какого-либо клиента [Birman, 2012]. Веб-сервер, например, не имеет состояния. Он просто отвечает на входящие HTTP-запросы, которые могут быть либо для загрузки файла на сервер, либо (чаще всего) для выборки файла. Когда запрос обработан, веб-сервер полностью забывает клиента. Аналогичным образом набор файлов, которыми

управляет веб-сервер (возможно, в сотрудничестве с файловым сервером), может быть изменен без необходимости информирования клиентов.

Обратите внимание, что во многих проектах без сохранения состояния на самом деле сервер хранит информацию о своих клиентах, но важным является тот факт, что если эта информация будет потеряна, это не приведет к нарушению службы, предлагаемой сервером. Допустим, веб-сервер обычно регистрирует все клиентские запросы. Эта информация полезна, например, для принятия решения о том, следует ли тиражировать определенные документы и куда их надо реплицировать. Очевидно, что если журнал потерян, нет никакого другого штрафа, кроме, возможно, неоптимальной производительности.

Конкретной формой проекта без сохранения состояния является то, что сервер поддерживает так называемое **мягкое состояние** (soft state). В этом случае сервер обещает поддерживать состояние от имени клиента, но только в течение ограниченного времени. После того как это время истекло, сервер возвращается к поведению по умолчанию, таким образом отбрасывая любую информацию, которую он держал на счету связанного клиента. Примером такого типа состояний является сервер, обещающий информировать клиента об обновлениях, но только в течение ограниченного времени. После этого клиент должен опросить сервер на наличие обновлений. Подходы с мягким состоянием берут свое начало от разработки протокола в компьютерных сетях, но могут в равной степени применяться к проектированию сервера [Clark, 1989; Lui et al., 2004].

Напротив, **сервер с сохранением состояния** (stateful server) обычно хранит постоянную информацию о своих клиентах. Это означает, что информация должна удаляться сервером явно. Типичным примером является файловый сервер, который позволяет клиенту сохранять локальную копию файла даже для выполнения операций обновления. Такой сервер будет поддерживать таблицу, содержащую (клиент, файл) записи. Подобная таблица позволяет серверу отслеживать, какой клиент в настоящее время имеет разрешения на обновление, для какого файла и, следовательно, возможно, также самую последнюю версию этого файла.

Такой подход может улучшить производительность операций чтения и записи, воспринимаемых клиентом. Повышение производительности по сравнению с серверами без сохранения состояния часто является важным преимуществом серверов с сохранением состояния. Однако пример также иллюстрирует основной недостаток серверов с сохранением состояния. Если сервер дает сбой, он должен восстановить свою таблицу (*клиент, файл*) записей, или иначе он не может гарантировать, что обработал самые последние обновления в файле. В общем случае серверу с состоянием необходимо полностью восстановить свое состояние, как это было до сбоя. Включение восстановления может привести к значительной сложности, как мы обсудим это в главе 8. В схеме без сохранения состояния не требуется никаких специальных мер для восстановления аварийного сервера. Он просто снова запускается и ожидает поступления клиентских запросов.

В работе [Ling et al., 2004] авторы утверждают, что на самом деле следует проводить различие между (вре́менным) **состоянием сеанса** и **постоянным состоянием** (session state и permanent state). Приведенный выше пример ти-

пичен для состояния сеанса: он связан с серией операций одного пользователя и должен поддерживаться в течение некоторого времени, но не неопределенно долго. Как выясняется, состояние сеанса часто поддерживается в трехуровневых клиент-серверных архитектурах, где серверу приложений действительно необходимо получить доступ к серверу базы данных через серию запросов, прежде чем он сможет ответить запрашивающему клиенту. Проблема здесь в том, что никакого реального вреда не будет, если состояние сеанса потеряно, при условии что клиент может просто повторно выполнить прежний запрос. Это позволяет более простое и менее надежное хранение состояния.

При постоянном сохранении состояния обычно остается информация о клиенте, ключах, связанных с приобретенным программным обеспечением и т. д., которая хранится в базе данных. Однако для большинства распределенных систем поддержание состояния сеанса уже подразумевает структуру с сохранением на сервере состояния, требующую специальных мер в случае возникновения сбоев и делающую явным предположения о продолжительности хранения состояния. Мы еще вернемся к этим вопросам при обсуждении отказоустойчивости.

При проектировании сервера выбор дизайна без сохранения состояния или с сохранением состояния не должен влиять на сервисы, предоставляемые сервером. Например, если файлы должны быть открыты до того, как их можно будет прочитать или записать, то сервер без сохранения состояния должен так или иначе имитировать это поведение. Распространенным решением является такое, при котором сервер отвечает на запрос на чтение или запись, сначала открывая указанный файл, затем выполняет фактическую операцию чтения или записи и немедленно снова закрывает файл.

В других случаях сервер может захотеть вести учет поведения клиента, чтобы он мог более эффективно отвечать на его запросы. Например, веб-серверы иногда предлагают возможность сразу направить клиента на его любимые страницы. Такой подход возможен только в том случае, если на сервере имеется хронологическая информация об этом клиенте. Когда сервер не может поддерживать состояние, то обычное решение состоит в том, чтобы позволить клиенту отправлять дополнительную информацию о своих предыдущих доступах. В случае интернета эта информация часто прозрачно хранится браузером клиента в так называемом файле cookie, который представляет собой небольшой фрагмент данных, содержащий специфичную для клиента информацию, которая представляет интерес для сервера. Cookie никогда не выполняются браузером; они просто хранятся.

Когда клиент обращается к серверу первый раз, последний отправляет cookie вместе с запрошенными веб-страницами обратно в браузер, после чего браузер безопасно убирает cookie. Каждый раз, когда клиент обращается к серверу, его cookie для этого сервера отправляется вместе с запросом.

Объектные серверы

Давайте посмотрим на общую организацию объектных серверов, необходимых для распределенных объектов. Важное различие между обычным объ-

ектным сервером и другими (более традиционными) серверами заключается в том, что объектный сервер сам по себе не обеспечивает конкретной службы. Конкретные сервисы реализуются объектами, которые находятся на сервере. По сути, сервер предоставляет только средства для вызова локальных объектов на основе запросов от удаленных клиентов. Как следствие изменить услуги относительно несложно, просто добавляя и удаляя объекты.

Таким образом, объектный сервер действует как место, где объекты находятся. Объект состоит из двух частей: данные, представляющие его состояние, и код для выполнения его методов. Разделены эти части или нет, являются ли методы реализации общими, зависит от сервера объектов. Кроме того, существуют различия в том, как объектный сервер вызывает свои объекты. Например, в многопоточном сервере каждому объекту может быть назначен отдельный поток, или отдельный поток может использоваться для каждого запроса вызова. Эти и другие вопросы обсуждаются далее.

Для вызова объекта объектному серверу необходимо знать, какой код выполнять, с какими данными он должен работать, должен ли он запускать отдельный поток, чтобы заботиться о вызове, и т. д. Простой подход состоит в том, чтобы предположить, что все объекты выглядят одинаково и что существует только один способ вызвать объект. К сожалению, такой подход, как правило, негибкий и часто излишне ограничивает разработчиков распределенных объектов.

Гораздо лучший подход для сервера – это поддержка различных политик. Рассмотрим, например, **временный** (transient) **объект**: объект, который существует только до тех пор, пока существует его сервер, но, возможно, и в течение более короткого периода времени. Копия файла, доступная только для чтения, обычно может быть реализована как временный объект. Аналогично калькулятор также может быть реализован как временный объект. Разумная политика состоит в том, чтобы создать временный объект при первом запросе вызова и уничтожить его, как только к нему больше не будут привязаны клиенты.

Преимущество этого подхода состоит в том, что временный объект будет нуждаться в ресурсах сервера только до тех пор, пока объект действительно нужен. Недостатком является то, что вызов может занять некоторое время, потому что сначала должен быть создан объект. Поэтому альтернативной политикой иногда является создание за счет потребления ресурсов всех временных объектов во время инициализации сервера, даже когда ни один клиент не использует объект.

Аналогичным образом сервер может следовать политике, согласно которой каждый из его объектов помещается в отдельный сегмент памяти. Другими словами, объекты не разделяют ни код, ни данные. Такая политика может быть необходима, когда реализация объекта не разделяет код и данные или когда объекты необходимо разделить по соображениям безопасности. В последнем случае серверу необходимо будет принять специальные меры или потребовать поддержки от базовой операционной системы, чтобы гарантировать, что границы сегмента не нарушаются.

Альтернативный подход состоит в том, чтобы позволить объектам по крайней мере совместно использовать свой код. Например, база данных, со-

держащая объекты, принадлежащие к одному и тому же классу, может быть эффективно реализована путем загрузки реализации класса только один раз на сервер. Когда приходит запрос на вызов объекта, серверу нужно только извлечь состояние этого объекта и выполнить запрошенный метод.

Существует множество различных политик в отношении потоков. Самый простой подход – реализовать сервер только с одним потоком управления. В качестве альтернативы сервер может иметь несколько потоков, по одному для каждого из своих объектов. Всякий раз, когда поступает запрос вызова для объекта, сервер передает запрос потоку, ответственному за этот объект. Если поток в данный момент занят, запрос ставится в очередь.

Преимущество этого подхода заключается в том, что объекты автоматически защищены от одновременного доступа: все вызовы поступают последовательно через один поток, связанный с объектом. Аккуратно и просто. Конечно, также можно использовать отдельный поток для каждого запроса вызова, требуя, чтобы объекты уже были защищены от одновременного доступа. Независимо от использования потока для объекта или потока для метода зависит выбор того, создаются ли потоки по требованию или сервер поддерживает пул потоков. Как правило, нет единой лучшей политики. Какую из них использовать, зависит от того, доступны ли потоки, насколько важна производительность, и других подобных факторов.

Решения о том, как вызывать объект, обычно называют **политиками активации** (activation policies), чтобы подчеркнуть, что во многих случаях сам объект должен сначала быть перенесен в адресное пространство сервера (то есть активирован), прежде чем он будет фактически вызван. В таком случае необходим механизм группировки объектов по политике. Такой механизм иногда называют **объектным адаптером** (object adapter), или, альтернативно, **упаковщиком объекта** (object wrapper). Объектный адаптер лучше всего рассматривать как программное обеспечение, реализующее определенную политику активации. Объектные адаптеры являются общими компонентами, помогающими разработчикам распределенных объектов, которые необходимо настроить только для конкретной политики.

Адаптер объекта контролирует один или несколько объектов. Поскольку сервер должен быть способен одновременно поддерживать объекты, для которых требуются разные политики активации, несколько адаптеров объектов могут находиться на одном сервере. Когда запрос вызова доставляется на сервер, он сначала отправляется соответствующему объектному адаптеру, как показано на рис. 3.14.

Важно заметить, что объектные адаптеры не знают о конкретных интерфейсах объектов, которыми они управляют. В противном случае они никогда не могли бы быть общими. Единственная проблема, которая важна для объектного адаптера, состоит в том, что он может извлечь ссылку на объект из запроса вызова и впоследствии отправить запрос объекту, на который ссылаются, но теперь следуя определенной политике активации. Как также показано на рис. 3.14, вместо того чтобы передавать запрос непосредственно объекту, адаптер передает запрос на вызов заглушке этого объекта на стороне сервера. Заглушка, также называемая скелетом, обычно генерируется из определений интерфейса объекта, отменяет маршализацию запроса и вызывает соответствующий метод.

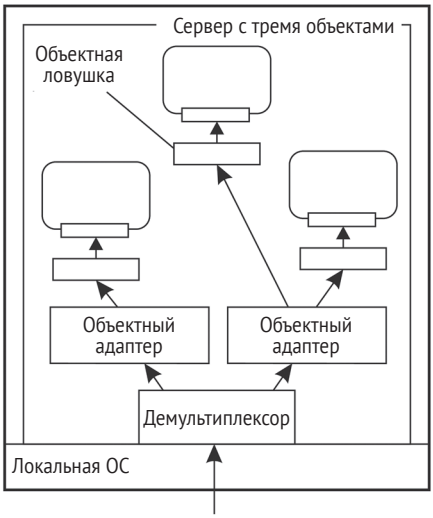


Рис. 3.14 ❖ Объектный сервер, поддерживающий разные политики активации

Адаптер объекта может поддерживать различные политики активации, просто конфигурируя его во время выполнения. Например, в CORBA-совместимых системах [OMG, 2001] можно указать, должен ли объект продолжать существовать после остановки соответствующего адаптера. Аналогично адаптер может быть сконфигурирован для генерации идентификаторов объектов или для того, чтобы приложение могло их предоставить. Адаптер может быть настроен для работы в однопоточном или многопоточном режиме, как мы объяснили выше.

Обратите внимание, что хотя на рис. 3.14 мы говорим об объектах, мы ничего не сказали о том, что эти объекты представляют собой на самом деле. В частности, следует подчеркнуть, что в рамках реализации такого объекта сервер может (косвенно) обращаться к базам данных или вызывать специальные библиотечные процедуры. Детали реализации скрыты для объектного адаптера, который взаимодействует только с ловушкой. Таким образом, фактическая реализация может не иметь ничего общего с тем, что мы часто видим относительно объектов уровня языка (то есть при компиляции). По этой причине обычно используется другая терминология. **Слуга** (servant) – общий термин для фрагмента кода, который формирует реализацию объекта.

Примечание 3.6 (пример: система поддержки выполнения программ Ice)

Давайте кратко рассмотрим систему распределенных объектов Ice, которая была частично разработана в ответ на сложности коммерческих распределенных систем на основе объектов [Henning, 2004]. Объектный сервер в Ice – это не что иное, как обычный процесс, который просто начинается с инициализации **системы поддержки выполнения** Ice (runtime system, RTS). Основу среды выполнения составляет то, что называется *коммуникатором*. Коммуникатор – это компонент, который управляет рядом основных ресурсов, наиболее важный из которых состоит из пула

потоков. Он также будет связан с динамически распределенной памятью и прочим. Кроме того, коммуникатор предоставляет средства для настройки среды. Например, можно указать максимальную длину сообщения, максимальное количество повторных попыток вызова и т. д.

Обычно объектный сервер имеет только один коммуникатор. Однако, когда разные приложения должны быть полностью отделены и защищены друг от друга, отдельный коммуникатор (возможно, с другой конфигурацией) может быть создан в одном и том же процессе. По крайней мере, при таком подходе отдельные пулы потоков будут разделены так, что если одно приложение израсходует все свои потоки, это не повлияет на другое приложение.

Коммуникатор также можно использовать для создания адаптера объекта, как показано на рис. 3.15. Отметим, что код упрощен и неполон. Дополнительные примеры и подробную информацию об Ice можно найти в [Henning and Spruiell, 2005].

```
main(int argc, char* argv[]) {
Ice::Communicator ic;
Ice::ObjectAdapter adapter;
Ice::Object object;

ic = Ice::initialize(argc, argv);
adapter = ic->createObjectAdapterWithEnd Points( «MyAdapter»,»tcp -p 10000»);
object = new MyObject;
adapter->add(object, objectID);
adapter->activate();
ic->waitForShutdown();
```

Рис. 3.15 ❖ Пример создания объектного сервера в Ice

В этом примере мы начинаем с создания и инициализации среды выполнения. Когда это будет сделано, объектный адаптер будет создан. В этом случае рекомендуется прослушивать входящие TCP-соединения через порт 10 000. Обратите внимание, что адаптер создается в контексте только что созданного коммуникатора. Теперь мы можем создать объект и впоследствии добавить этот объект в адаптер. Наконец, адаптер активируется, и это означает, что под капотом активируется поток, который начнет прослушивать входящие запросы.

Этот код еще не демонстрирует значительных различий в политиках активации. Политику можно изменить, изменив свойства адаптера. Одно семейство свойств связано с поддержанием специфичного для адаптера набора потоков, которые используются для обработки входящих запросов. Например, можно указать, что всегда должен быть только один поток, эффективно организуя последовательный порядок всех обращений к объектам, которые были добавлены в адаптер.

Опять же, обратите внимание, что мы не определили MyObject. Как и раньше, это может быть простой объект C++, но также тот, который обращается к базам данных и другим внешним службам, которые совместно реализуют объект. При регистрации MyObject с помощью адаптера такие подробности реализации полностью скрываются от клиентов, которые теперь считают, что они вызывают удаленный объект.

В приведенном выше примере объект создается как часть приложения, после чего он добавляется в адаптер. Фактически это означает, что адаптеру может потребоваться поддержка множества объектов одновременно, что может привести к потенциальным проблемам с масштабируемостью. Альтернативное решение – динамически загружать объекты в память, когда они необходимы. Для этого Ice

обеспечивает поддержку специальных объектов, известных как *локаторы*. Локатор вызывается, когда адаптер получает входящий запрос для объекта, который не был явно добавлен. В этом случае запрос направляется локатору, задача которого заключается в дальнейшей обработке запроса.

Чтобы конкретизировать ситуацию, предположим, что локатору передан запрос на объект, локатор которого знает, что его состояние хранится в системе реляционной базы данных. Конечно, здесь нет ничего волшебного: локатор был явно запрограммирован для обработки таких запросов. В этом случае идентификатор объекта может соответствовать ключу записи, в которой хранится это состояние. Затем локатор просто выполнит поиск этого ключа, выберет состояние и сможет продолжить обработку запроса.

К адаптеру может быть добавлено несколько локаторов. В этом случае адаптер будет отслеживать, какие идентификаторы объектов будут принадлежать одному и тому же локатору. Использование нескольких локаторов позволяет поддерживать множество объектов одним адаптером. Конечно, объекты (или, скорее, их состояние) должны быть загружены во время выполнения, но это динамическое поведение, возможно, сделает сам сервер относительно простым.

Примечание 3.7 (пример: технология Enterprise Java Beans)

Язык программирования Java и связанная с ним модель легли в основу многочисленных распределенных систем и приложений. Его популярность можно объяснить простой поддержкой ориентации объекта в сочетании с внутренней поддержкой метода удаленного вызова. Java обеспечивает высокую степень прозрачности доступа, что делает его более простым в использовании, чем, например, сочетание языка C с удаленным вызовом процедур.

С момента его появления существовал сильный стимул для предоставления средств, которые облегчили бы разработку распределенных приложений. Эти средства выходят далеко за рамки языковой поддержки и требуют среды выполнения, которая поддерживает традиционные многоуровневые архитектуры клиент-сервер. С этой целью была проделана большая работа по разработке технологии **(Enterprise) Java Beans (EJB)**.

EJB – это, по сути, объект Java, который размещается на специальном сервере и предлагает удаленным клиентам различные способы вызова данного объекта. Важно то, что этот сервер обеспечивает поддержку для отделения функциональности приложения от системно-ориентированной функциональности. Последняя включает в себя функции для поиска объектов, хранения объектов, позволяет объектам быть частью транзакции и т. д. Как разрабатывать EJB, подробно описано в [Schildt, 2010].

Учитывая это разделение, EJB могут быть изображены, как показано на рис. 3.16. Важной проблемой является то, что EJB встроен в контейнер, эффективно обеспечивающий интерфейсы для базовых сервисов, которые реализуются сервером приложений. Контейнер может более или менее автоматически связывать EJB с этими сервисами, что означает, что правильные ссылки легкодоступны программисту. Типичные службы включают службы для метода удаленного вызова (remote method invocation, RMI), доступа к базе данных (database access. DBC), именования (naming, JNDI) и обмена сообщениями (messaging, JMS). Использование этих сервисов более или менее автоматизировано, но требует, чтобы программист делал различие между четырьмя типами EJB:

1) сессионные компоненты без состояния;
2) сессионные компоненты с сохранением состояния;

3) объектные компоненты;
4) управляемые сообщениями компоненты.

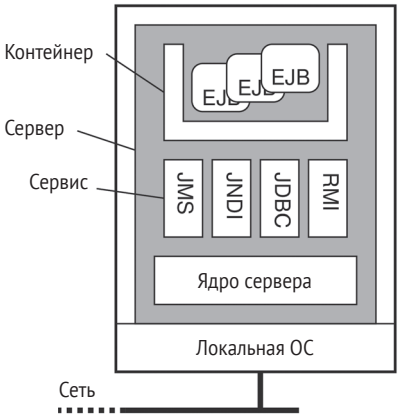


Рис. 3.16 ❖ Общая архитектура сервера EJB

Как следует из названия, сессионный **компонент без сохранения состояния** (stateless session bean) – это объект, который вызывается один раз и выполняет свою работу, после чего он сбрасывает любую информацию, необходимую для выполнения услуги, предлагаемой клиенту. Например, сессионный компонент без сохранения состояния может использоваться для реализации службы, которая перечисляет книги с самым высоким рейтингом. В этом случае компонент обычно состоит из SQL-запроса, который передается в базу данных. Результаты будут помещены в специальный формат, который может обработать клиент, после чего его работа будет завершена и перечисленные книги будут сброшены.

Напротив, **сессионный компонент с сохранением состояния** (stateful session bean) поддерживает состояние, связанное с клиентом. Каноническим примером является компонент объектной модели bean, реализующий электронную корзину для покупок. В этом случае клиент, как правило, может положить вещи в корзину, удалить предметы и использовать корзину для перехода к электронному оформлению заказа. Bean, в свою очередь, обычно получает доступ к базам данных для получения текущих цен и информации о количестве товаров, которые еще есть в наличии. Однако его время жизни все еще будет ограничено, поэтому его называют сессионным компонентом: когда клиент завершает работу (возможно, несколько раз вызвав объект), компонент автоматически уничтожается.

Объектные компоненты Bean (**entity bean**) можно считать долгоживущим *постоянным* объектом. Таким образом, объектный компонент обычно будет храниться в базе данных, а также нередко будет частью транзакций. Как правило, сущности компонентов хранят информацию, которая может понадобиться в следующий раз, когда конкретный клиент получит доступ к серверу. В настройках электронной торговли объектный компонент может использоваться для записи информации о клиенте, например адреса доставки, адреса выставления счета, информации о кредитной карте и т. д. В этих случаях, когда клиент входит в систему, его связанный объектный компонент будет восстановлен и использован для дальнейшей обработки.

Наконец, **управляемые сообщениями компоненты Bean** (message-driven beans) используются для программирования объектов, которые должны реагировать на входящие сообщения (а также иметь возможность отправлять сообщения). Компоненты, управляемые сообщениями, не могут быть вызваны напрямую клиентом, а скорее подходят для способа публикации-подписки. Все сводится к тому, что управляемый сообщениями компонент автоматически вызывается сервером при получении конкретного сообщения *m*, на которое сервер (или, скорее, приложение, которое он размещает) ранее подписался, то есть заявил, что он хочет быть уведомленным, когда такое сообщение прибывает. Bean содержит код приложения для обработки сообщения, после чего сервер просто сбрасывает его. Таким образом, компоненты bean, управляемые сообщениями, рассматриваются как не имеющие состояния.

Пример: веб-сервер Apache

Интересным примером сервера, который уравновешивает разделение между политиками и механизмами, является веб-сервер Apache. Это также чрезвычайно популярный сервер, который, по оценкам, используется для размещения примерно в 50 % всех веб-сайтов. Apache – это сложная часть программного обеспечения, и благодаря многочисленным улучшениям типов документов, которые сейчас предлагаются в интернете, важно, чтобы сервер был легко конфигурируемым и расширяемым и в то же время в значительной степени независимым от конкретных платформ.

Обеспечение независимости серверной платформы осуществляется путем предоставления собственной базовой среды выполнения, которая затем реализуется для различных операционных систем. Эта среда выполнения, известная как **Apache Portable Runtime** (**APR**), представляет собой библиотеку, которая обеспечивает независимый от платформы интерфейс для обработки файлов, работы в сети, блокировки, потоков и т. д. При расширении Apache переносимость в значительной степени гарантируется при условии, что выполняются только вызовы к APR и что избегаются обращения к библиотекам, специфичным для платформы.

С определенной точки зрения, Apache можно рассматривать как полностью общий сервер, предназначенный для получения ответа на входящий запрос. Конечно, есть все виды скрытых зависимостей и предположений, благодаря которым Apache оказывается в первую очередь подходящим для обработки запросов на веб-документы. Например, как мы уже упоминали, веб-браузеры и серверы используют HTTP в качестве протокола связи. Протокол HTTP практически всегда реализуется поверх TCP, поэтому ядро Apache предполагает, что все входящие запросы соответствуют протоколу связи на основе TCP. Запросы на основе UDP не могут быть обработаны без изменения ядра Apache.

Однако ядро Apache делает несколько предположений о том, как входящие запросы должны быть обработаны. Его общая организация показана на рис. 3.17.

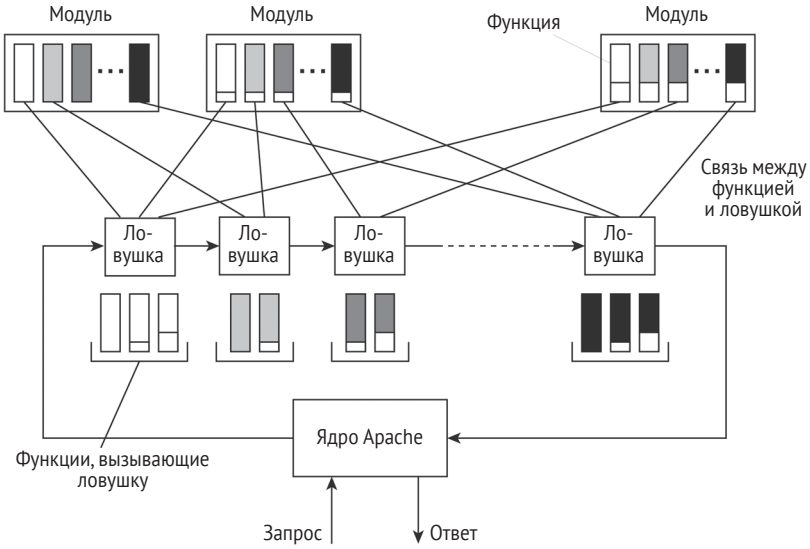


Рис. 3.17 ❖ Общая организация веб-сервера Apache

Основой этой организации является концепция ловушки, которая является не чем иным, как заполнителем для определенной группы функций. Ядро Apache предполагает, что запросы обрабатываются в несколько этапов, каждый из которых состоит из нескольких перехватчиков. Таким образом, каждая ловушка представляет группу похожих действий, которые необходимо выполнить как часть обработки запроса.

Например, есть ловушка для перевода URL-адреса в имя локального файла. Такой перевод почти наверняка потребуется при обработке запроса. Аналогично есть ловушка для записи информации в журнал, ловушка для проверки идентификации клиента, ловушка для проверки прав доступа и ловушка для проверки, к какому типу MIME относится запрос (например, чтобы убедиться, что запрос может быть правильно обработан). Как показано на рис. 3.17, ловушки обрабатываются в заранее определенном порядке. Здесь мы явно видим, как Apache обеспечивает особый контроль над обработкой запросов.

Все функции, связанные с ловушкой, предоставляются отдельными модулями. Хотя, в принципе, разработчик может изменить набор ловушек, которые будут обрабатываться Apache, гораздо чаще пишутся модули, содержащие функции, которые необходимо вызывать как часть обработки стандартных ловушек, предоставляемых немодифицированным Apache. Основной принцип довольно прост. Каждая ловушка может содержать набор функций, соответствующих конкретному прототипу функции (т. е. список параметров и тип возвращаемого значения). Разработчик модуля пишет функции для определенных ловушек. При компиляции Apache разработчик определяет, какая функция должна быть добавлена к какой ловушке. Последняя на рис. 3.17 демонстрирует, как образуются различные связи между функциями и ловушками.

Поскольку могут присутствовать десятки модулей, каждая ловушка обычно будет содержать несколько функций. Как правило, модули считаются взаимозависимыми, поэтому функции в одной ловушке будут выполняться в произвольном порядке. Вместе с тем Apache может обрабатывать зависимости модулей, позволяя разработчику определять порядок, в котором должны обрабатываться функции из разных модулей. В целом в результате получается веб-сервер, который чрезвычайно универсален. Подробная информация о настройке Apache, а также введение в его расширение можно найти в [Laurie and Laurie, 2002].

Кластеры серверов

В главе 1 мы кратко обсудили кластерные вычисления как одно из многих проявлений распределенных систем. Теперь мы более подробно рассмотрим организацию кластеров серверов, а также основные проблемы проектирования. Сначала рассмотрим общие кластеры серверов, которые организованы в локальных сетях. Особую группу составляют глобальные кластеры серверов, которые мы впоследствии обсудим.

Локальные кластеры

Проще говоря, кластер серверов – это не что иное, как набор машин, соединенных через сеть, где на каждом компьютере работает один или несколько серверов. Рассматриваемые здесь кластеры серверов – это те, в которых машины подключены через локальную сеть, часто предлагая высокую пропускную способность и низкую задержку.

Общая организация

Во многих случаях кластер серверов логически организован в три уровня, как показано на рис. 3.18. Первый уровень состоит из (логического) коммутатора, через который маршрутизируются клиентские запросы. Такой переключатель может широко варьироваться. Например, коммутаторы транспортного уровня принимают входящие запросы на соединение TCP и передают запросы на один из серверов в кластере. Совершенно другой пример – веб-сервер, который принимает входящие HTTP-запросы, но частично передает запросы на серверы приложений для дальнейшей обработки только для последующего сбора результатов с этих серверов и возврата HTTP-ответа.

Как и в любой многоуровневой архитектуре клиент-сервер, многие кластеры серверов также содержат серверы, предназначенные для обработки приложений. В кластерных вычислениях это обычно серверы, работающие на высокопроизводительном оборудовании, предназначенном для обеспечения вычислительной мощности. Тем не менее в случае кластеров корпоративных серверов это может быть случай, когда приложения должны работать только на компьютерах с относительно низким уровнем производительности, поскольку требуемая вычислительная мощность не является узким местом, но доступ к памяти таковым является.

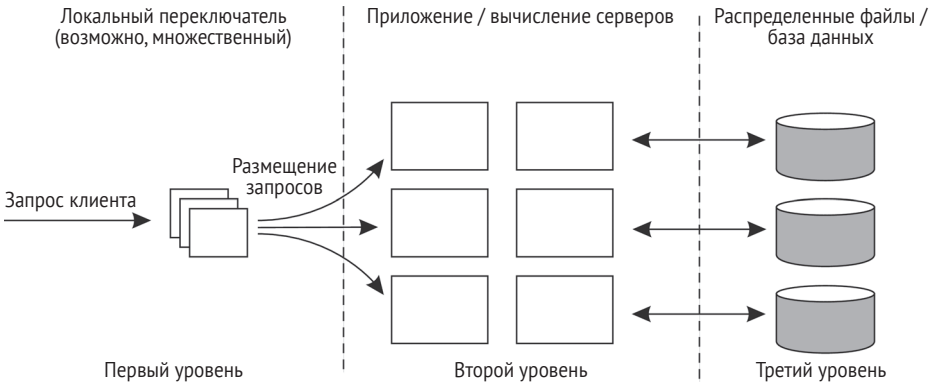


Рис. 3.18 ❖ Общая организация трехуровневого серверного кластера

Это приводит нас к третьему уровню, который состоит из серверов обработки данных, в частности файловых серверов и серверов баз данных. Опять же, в зависимости от использования кластера серверов эти серверы могут работать на специализированных компьютерах, настроенных для высокоскоростного доступа к диску и имеющих большие кеши данных на стороне сервера.

Конечно, не все кластеры серверов будут следовать этому строгому разделению. Часто каждый компьютер оснащен собственным локальным хранилищем, нередко интегрируя приложения и обработку данных на одном сервере, что приводит к двухуровневой архитектуре. Например, когда речь идет о потоковой передаче мультимедиа с помощью кластера серверов, обычно развертывается двухуровневая архитектура системы, где каждая машина выступает в качестве выделенного медиасервера [Steinmetz and Nahrstedt, 2004].

Когда кластер серверов предлагает несколько служб, может случиться, что на разных машинах работают разные серверы приложений. Как следствие коммутатор должен будет иметь возможность различать службы, или, иначе, он не может пересылать запросы на соответствующие машины. Как следствие мы можем обнаружить, что некоторые машины временно бездействуют, в то время как другие получают перегрузку запросов. Что было бы полезно, так это временно перенести службы на незанятые машины. Решение состоит в том, чтобы использовать виртуальные машины, позволяющие относительно легко переносить код на реальные машины.

Отправка запросов. Давайте теперь подробнее рассмотрим первый уровень, состоящий из коммутатора, также известного как **внешний интерфейс** (front end). Важной целью проектирования кластеров серверов является скрытие того факта, что существует несколько серверов. Другими словами, клиентские приложения, работающие на удаленных машинах, не должны знать ничего о внутренней организации кластера. Такая прозрачность доступа неизменно обеспечивается с помощью единой точки доступа, которая, в свою очередь, реализуется с помощью какого-либо аппаратного коммутатора, такого как выделенный компьютер.

Коммутатор формирует точку входа для кластера серверов, предлагая единый сетевой адрес. Для масштабируемости и доступности кластер серверов может иметь несколько точек доступа, где каждая точка доступа затем реализуется отдельной выделенной машиной. Мы рассматриваем только случай одной точки доступа.

Стандартный способ доступа к кластеру серверов – установить TCP-соединение, по которому запросы уровня приложения затем отправляются как часть сеанса. Сеанс заканчивается разрывом соединения. В случае **коммутаторов транспортного уровня** (transport-layer switches) коммутатор принимает входящие запросы на соединение TCP и передает эти соединения на один из серверов. Существуют два основных способа работы переключателя [Cardellini et al., 2002].

В первом способе клиент устанавливает соединение TCP таким образом, чтобы все запросы и ответы проходили через коммутатор. Коммутатор, в свою очередь, установит TCP-соединение с выбранным сервером, передаст запросы клиента на этот сервер, а также примет ответы сервера (которые он передаст клиенту). По сути, коммутатор находится в середине TCP-соединения между клиентом и выбранным сервером, перезаписывая адреса источника и назначения при прохождении сегментов TCP. Этот подход является **формой трансляции сетевых адресов** (network address translation, NAT) [Srisuresh and Holdrege, 1999].

Альтернативно, коммутатор может фактически передавать соединение с выбранным сервером, так что все ответы напрямую передаются клиенту без прохождения через сервер [Hunt et al., 1997; Pai et al., 1998]. Принцип работы **передачи обслуживания** (TCP handoff) показан на рис. 3.19.

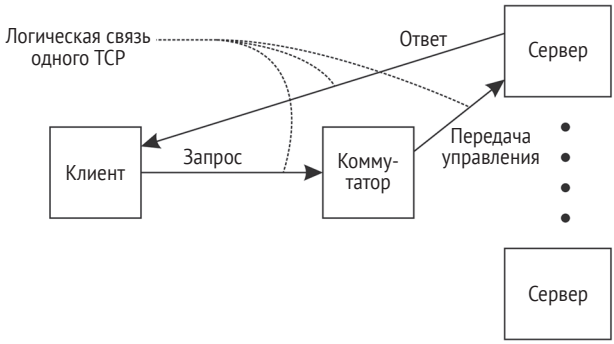


Рис. 3.19 ❖ Принцип передачи обслуживания TCP

Когда коммутатор получает запрос на соединение TCP, он сначала определяет лучший сервер для обработки этого запроса и перенаправляет пакет запроса на этот сервер. Сервер, в свою очередь, отправит подтверждение обратно запрашивающему клиенту, но вставит IP-адрес коммутатора в качестве поля источника заголовка IP-пакета, несущего сегмент TCP. Обратите внимание, что эта перезапись адресов необходима клиенту для продолжения выполнения протокола TCP: он ожидает ответ от коммутатора, а не от какого-

либо произвольного сервера, о котором он никогда раньше не слышал. Очевидно, что реализация TCP-передачи обслуживания требует модификации уровня операционной системы. Передача TCP особенно эффективна, когда ответы намного больше, чем запросы, как в случае веб-серверов.

Уже видно, что коммутатор может играть важную роль в распределении нагрузки между различными серверами. Решая, куда направить запрос, коммутатор также решает, какой сервер должен обрабатывать дальнейшую обработку запроса. Простейшей политикой балансировки нагрузки, которой может следовать коммутатор, является циклическая перестановка: каждый раз, когда он выбирает следующий сервер из своего списка для пересылки запроса. Конечно, коммутатор должен будет отслеживать, какому серверу он передал соединение TCP, по крайней мере до тех пор, пока это соединение не будет разорвано. Оказывается, что поддержание этого состояния и передача последующих сегментов TCP, принадлежащих одному и тому же TCP-соединению, фактически может замедлить коммутатор.

Также могут быть развернуты более сложные критерии выбора сервера. Например, предположим, что кластер серверов предлагает несколько услуг. Если коммутатор может различать эти сервисы при поступлении запроса, он может принять обоснованное решение о том, куда направить запрос. Этот выбор сервера может по-прежнему осуществляться на транспортном уровне, если службы различаются по номеру порта. В случае **коммутаторов транспортного уровня** (transport-level switches), как мы уже обсуждали, решения о том, куда направлять входящий запрос, основаны только на информации транспортного уровня. Еще один шаг заключается в том, чтобы коммутатор действительно проверял полезную нагрузку входящего запроса. Такое распределение запросов с учетом содержания может применяться только в том случае, если известно, как выглядит эта полезная нагрузка. Например, в случае веб-серверов коммутатор может ожидать HTTP-запрос, на основании которого он может затем решить, кто должен его обрабатывать.

Примечание 3.8 (дополнительно: эффективное распространение запросов с учетом содержания)

Очевидно, что дизайн коммутатора имеет решающее значение, так как он может легко стать узким местом во всем трафике, проходящем через него. Переключатели транспортного уровня, как правило, очень эффективны. Вместе с тем контент-зависимые коммутаторы, работающие на прикладном уровне, проверяющие полезную нагрузку входящего запроса, на самом деле могут нуждаться в большой обработке.

Распространение запросов с учетом содержания имеет несколько преимуществ. Например, если коммутатор всегда перенаправляет запросы на один и тот же файл на один и тот же сервер, этот сервер может быть способен эффективно кешировать файл, что приводит к увеличению времени отклика. Аналогичным образом мы можем решить распространять набор файлов на конкретные серверы, делая различие, например, между потоковыми медиафайлами, изображениями, текстовыми файлами, но также, возможно, и фактическими базами данных. Возможность сделать такое различие позволит установить выделенные серверы второго уровня.

В идеале коммутатор должен быть таким же эффективным, как обычный коммутатор транспортного уровня, но при этом иметь функции для распространения с учетом содержимого. В работе [Aron et al., 2000] предлагается схема, по которой

проверка полезной нагрузки входящего запроса распределяется по нескольким серверам и объединяет это распределение с коммутацией на транспортном уровне. У переключателя теперь есть две задачи. Во-первых, когда изначально приходит запрос, он должен решить, какой сервер будет обрабатывать остальную часть связи с клиентом. Во-вторых, коммутатор должен пересылать клиентские TCP-сообщения, связанные с передаваемым TCP-соединением.

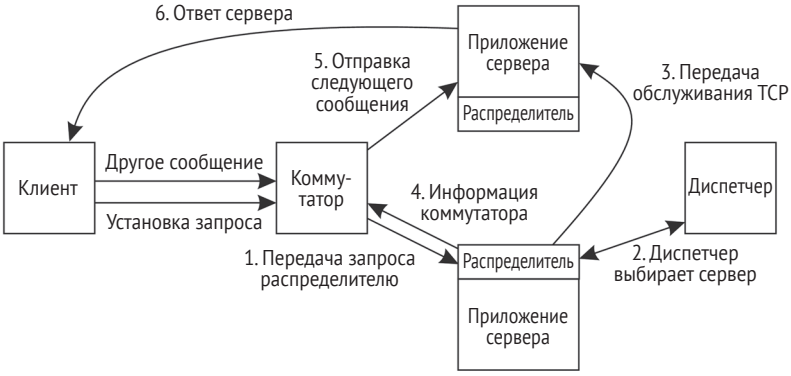


Рис. 3.20 ❖ Распределение запросов с учетом содержимого с использованием передачи обслуживания TCP

Эти две задачи могут быть распределены, как показано на рис. 3.20. Когда приходит запрос, первым делом коммутатор передает его произвольному распределителю, который работает на одном из серверов. Основная задача распределителя – управлять передачей TCP. В свою очередь, распределитель передает запрос диспетчеру, который проверяет полезную нагрузку, чтобы выбрать лучший сервер приложений. После того как этот сервер был выбран, начальный распределитель будет выполнять администрирование для передачи TCP-соединения: он связывается с выбранным сервером для принятия соединения и в конечном итоге сообщает коммутатору, на какой сервер должен впоследствии пересылать связанные сегменты TCP.

Глобальные кластеры

Характерной особенностью кластеров локальных серверов является то, что они принадлежат одной организации. Развертывание кластеров в глобальной сети традиционно было довольно обременительным, поскольку обычно приходилось иметь дело с несколькими административными организациями, такими как интернет-провайдеры (ISP). С появлением облачных вычислений все изменилось, и в настоящее время мы наблюдаем рост глобальных распределенных систем, в которых серверы (или кластеры серверов) распространяются через интернет. Проблемы, связанные с необходимостью иметь дело с несколькими организациями, эффективно обходятся путем использования средств одного облачного поставщика.

Облачные провайдеры, такие как Amazon и Google, управляют несколькими центрами данных, расположенными в разных местах по всему миру.

Таким образом, они могут предложить конечному пользователю возможность создать глобальную распределенную систему, состоящую из потенциально большой совокупности сетевых виртуальных машин, разбросанных по интернету. Важной причиной для желания иметь такие распределенные системы является обеспечение локальности: предоставление данных и услуг, которые близки к клиентам. Примером такой важности являются потоковые мультимедиа: чем ближе к серверу расположен видеосервер, тем легче обеспечить высококачественные потоки.

Обратите внимание, что если локальность не является критической, она может быть достаточной, или даже лучше разместить виртуальные машины в одном центре обработки данных, чтобы межпроцессорное взаимодействие могло выиграть от малой задержки в локальных сетях. Платой может быть более высокая задержка между клиентами и службой, работающей в удаленном центре обработки данных.

Диспетчеризация запроса. Если локальность является проблемой, то диспетчеризация запроса становится важной: когда клиент обращается к сервису, его запрос должен быть перенаправлен на соседний сервер, то есть сервер, который позволит установить быструю связь с этим клиентом. Решение о том, какой сервер должен обрабатывать запрос клиента, является вопросом политики перенаправления [Sivasubramanian et al., 2004b]. Если мы предположим, что клиент сначала свяжется с диспетчером запросов, аналогичным коммутатору в нашем обсуждении локальных кластеров, то этот диспетчер должен будет оценить задержку между клиентом и несколькими серверами. Как такая оценка может быть сделана, обсуждается в разделе 6.5.

После выбора сервера диспетчер должен будет сообщить об этом клиенту. Возможны несколько **механизмов перенаправления** (redirection mechanisms). Популярным является случай, когда диспетчером фактически является DNS-сервер имен. Интернет или веб-сервисы часто ищутся в **системе доменных имен** (Domain Name System, DNS). Клиент предоставляет доменное имя, такое как service.organization.org, локальному DNS-серверу, который в конечном итоге возвращает IP-адрес связанной службы, возможно, после установления связи с другими DNS-серверами. При отправке запроса на поиск имени клиент также отправляет свой собственный IP-адрес (запросы DNS отправляются в виде пакетов UDP). Другими словами, DNS-сервер также будет знать IP-адрес клиента, который он впоследствии сможет использовать для выбора лучшего сервера для этого клиента и возврата близкого IP-адреса.

К сожалению, эта схема не идеальна по двум причинам. Во-первых, вместо отправки IP-адреса клиента происходит то, что локальный DNS-сервер, с которым связывается клиент, действует как прокси для этого клиента. Другими словами, для определения местоположения клиента используется не IP-адрес клиента, а адрес локального DNS-сервера. В работе [Mao et al., 2002] показано, что в этом случае могут быть огромные дополнительные расходы на связь, поскольку локальный DNS-сервер часто не слишком локальный.

Во-вторых, в зависимости от схемы, используемой для разрешения доменного имени, может даже оказаться, что адрес локального DNS-сервера и не используется. Вместо этого может случиться так, что DNS-сервер, который

решает, какой IP-адрес вернуть, может быть обманут тем фактом, что запрашивающий является еще одним DNS-сервером, действующим в качестве промежуточного звена между исходным клиентом и решающим DNS-сервером. В этих случаях местная осведомленность полностью теряется.

Несмотря на то что перенаправление на основе DNS не всегда может быть очень точным, оно широко применяется, хотя бы потому, что его относительно легко реализовать, а также является прозрачным для клиента. Кроме того, нет необходимости полагаться на клиентское программное обеспечение с учетом местоположения клиента.

Примечание 3.9 (дополнительно: альтернатива для организации кластеров глобальных серверов)

Как мы уже упоминали, большинство кластеров серверов предлагают одну точку доступа. При сбое данной точки кластер становится недоступным. Чтобы устранить эту потенциальную проблему, может быть предоставлено несколько точек доступа, адреса которых сделаны общедоступными. **Система доменных имен** (Domain Name System, DNS) может возвращать несколько адресов, принадлежащих одному и тому же имени хоста, используя простую стратегию циклического перебора. Этот подход все еще требует, чтобы клиенты делали несколько попыток, если один из адресов не работает. Кроме того, это не решает проблему необходимости статических точек доступа.

Наличие стабильности в виде долгоживущей точки доступа является желательной функцией с точки зрения клиента и сервера. С другой стороны, также желательно иметь высокую степень гибкости при настройке кластера серверов, включая коммутатор. Это наблюдение привело к созданию **распределенного сервера** (distributed server), который фактически представляет собой не что иное, как, возможно, динамически изменяющийся набор машин, с также возможно изменяющимися точками доступа, но который тем не менее представляется внешнему миру как единая, мощная машина. В работе [Szymaniak et al., 2007] представлен проект такого распределенного сервера. Мы кратко опишем его здесь.

Основная идея распределенного сервера заключается в том, что клиенты получают выгоду от надежного, высокопроизводительного и стабильного сервера. Эти свойства часто могут быть предоставлены мейнфреймами высокого класса, у некоторых из которых среднее время между отказами превышает 40 лет. Однако, прозрачно группируя более простые машины в кластер и не полагаясь на доступность одной машины, можно добиться большей степени стабильности, чем для каждого компонента в отдельности. Например, такой кластер может быть динамически сконфигурирован из компьютеров конечного пользователя, как в случае совместной распределенной системы. Отметим также, что во многих центрах обработки данных используются относительно дешевые и простые машины [Barroso and Hölze, 2009].

Давайте сосредоточимся на том, как в такой системе может быть достигнута стабильная точка доступа. Основная идея состоит в том, чтобы использовать доступные сетевые сервисы, в частности поддержку мобильности для процессора MIPv6 версии 6. В MIPv6 (mashine-instraction processor) предполагается, что мобильный узел имеет **домашнюю сеть** (home network), в которой он обычно находится и для которой он имеет связанный стабильный адрес, известный как его **домашний адрес** (home-of address, HoA). К этой домашней сети подключен специальный маршрутизатор, известный как **домашний агент** (home agent), который будет брать на себя заботу о трафике на мобильный узел, когда тот отсутствует. С этой целью, когда

мобильный узел подключается к чужой сети, он получает **временный служебный адрес** (care-of address, CoA), по которому он может быть достигнут. Этот служебный адрес сообщается домашнему агенту узла, который затем следит за тем, чтобы весь трафик пересылался на мобильный узел. Обратите внимание, что приложения, взаимодействующие с мобильным узлом, будут видеть только адрес, связанный с домашней сетью узла. Они никогда не увидят адрес для передачи.

Этот принцип может использоваться для предоставления стабильного адреса распределенного сервера. В этом случае один уникальный **контактный адрес** (contact address) изначально назначается кластеру серверов. Контактный адрес будет адресом времени жизни сервера, который будет использоваться при любом общении с внешним миром. В любой момент один узел на распределенном сервере будет работать как точка доступа, используя этот контактный адрес, но эту роль может легко занять другой узел. Происходит следующее: точка доступа записывает свой собственный адрес как адрес для передачи в домашнем агенте, связанном с сервером распределения. В этой точке весь трафик будет направлен к точке доступа, которая затем позаботится о распределении запросов среди участвующих в это время узлов. Если точка доступа выходит из строя, появляется простой механизм переключения при сбое, с помощью которого другая точка доступа сообщает новый адрес для передачи.

Эта простая конфигурация сделает домашний агент, а также точку доступа потенциальным узким местом, поскольку весь трафик будет проходить через эти две машины. Подобной ситуации можно избежать, используя функцию MIPv6, известную как **оптимизация маршрута** (optimization route). Оптимизация маршрута работает следующим образом. Всякий раз, когда мобильный узел с домашним адресом HA сообщает о своем текущем служебном адресе, например CA, домашний агент может пересылать CA клиенту. Последний затем будет локально хранить пару (HA, CA). С этого момента связь будет напрямую направляться в CA. Хотя приложение на стороне клиента все еще может использовать домашний адрес, основное программное обеспечение поддержки для MIPv6 преобразует этот адрес в CA и использует его.

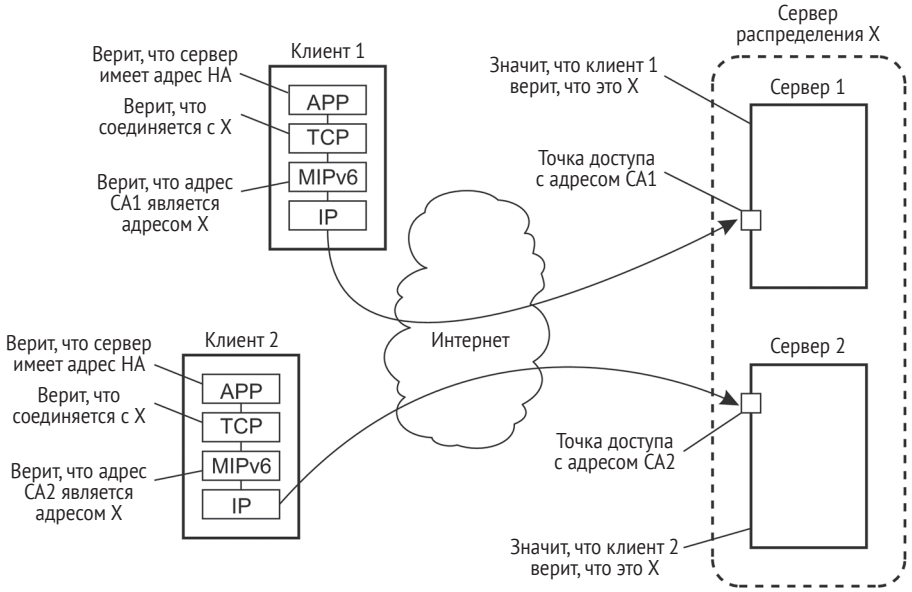


Рис. 3.21 ❖ Оптимизация маршрута на сервере распределения

Оптимизация маршрута может использоваться для того, чтобы разные клиенты полагали, что они обмениваются данными с одним сервером, когда фактически каждый клиент обменивается данными с различным узлом-участником сервера распределения, как показано на рис. 3.21. С этой целью, когда точка доступа сервера распределения пересылает запрос от клиента C_1, скажем, к узлу S_1 (с адресом CA_1), она передает достаточно информации в S_1, чтобы позволить ей инициировать процедуру оптимизации маршрута, с помощью которой в конечном итоге у клиента создается впечатление, что адрес для передачи – CA_1. Это позволит C_1 сохранить пару (HA, CA_1). Во время этой процедуры точка доступа (а также домашний агент) туннелирует большую часть трафика между C_1 и S_1. Это не позволит домашнему агенту поверить в то, что адрес для обслуживания изменился, и он продолжит связь с точкой доступа.

Конечно, пока идет процедура оптимизации маршрута, запросы от других клиентов все еще могут поступать. Они остаются в состоянии ожидания в точке доступа до тех пор, пока их нельзя будет переадресовать. Запрос от другого клиента C_2 затем может быть перенаправлен на узел S_2 (с адресом CA_2), что позволяет последнему позволить клиенту C_2 хранить пару (HA, CA_2). В результате разные клиенты будут иметь прямую связь с различными членами сервера распределения, где у каждого клиентского приложения все еще есть иллюзия, что этот сервер имеет адрес HA. Домашний агент продолжает общаться с точкой доступа, общаясь с контактным адресом.

Пример использования: PlanetLab

Давайте теперь подробнее рассмотрим несколько необычный кластерный сервер. PlanetLab – это распределенная система для совместной работы, в которой каждая организация жертвует один или несколько компьютеров, что в сумме составляет сотни узлов.

Вместе эти компьютеры образуют одноуровневый кластер серверов, где доступ, обработка и хранение могут осуществляться на каждом узле индивидуально. Управление PlanetLab по необходимости почти полностью распределено.

Общая организация. В PlanetLab участвующая организация жертвует один или несколько узлов (компьютеров), которые впоследствии совместно используются всеми пользователями PlanetLab. Каждый узел организован, как показано на рис. 3.22. Есть два важных компонента [Bavier et al., 2004; Peteson et al., 2006]. Первый – это монитор виртуальной машины (virtual machine monitor, VMM), который является улучшенной операционной системой Linux. Улучшения в основном включают в себя настройки для поддержки второго компонента, а именно (Linux) **Vservers** (V-серверы). На данный момент V-сервер лучше всего рассматривать как отдельную среду, в которой выполняется группа процессов. Мы вернемся к V-серверам далее.

Linux VMM обеспечивает разделение V-серверов: процессы на разных V-серверах выполняются одновременно и независимо друг от друга, каждый из которых использует только пакеты программ и программы, доступные в их собственной среде. Разделение между процессами в разных серверах строгое. Например, два процесса в разных V-серверах могут иметь один и тот же идентификатор пользователя, но это не означает, что они происходят от

одного и того же пользователя. Такое разделение значительно облегчает поддержку пользователей из разных организаций, которые хотят использовать PlanetLab в качестве, например, испытательного стенда для экспериментов с совершенно разными распределенными системами и приложениями.

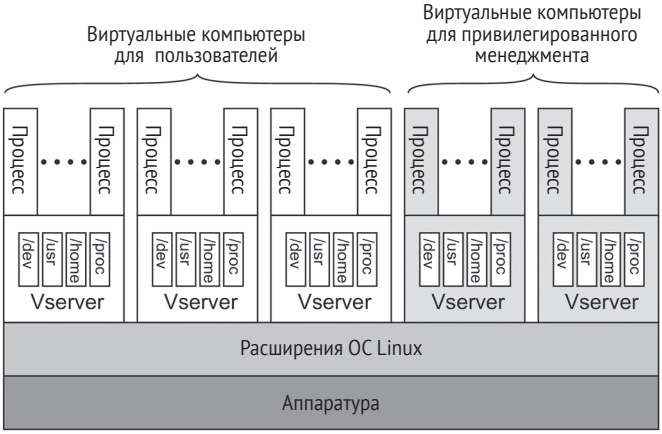


Рис. 3.22 ❖ Базовая организация узла PlanetLab

Для поддержки таких экспериментов PlanetLab использует **слои**, причем каждый слой представляет собой набор V-серверов и каждый V-сервер работает на отдельном узле, как показано на рис. 3.23. Таким образом, слой можно рассматривать как кластер виртуальных серверов, реализованный посредством набора виртуальных машин.

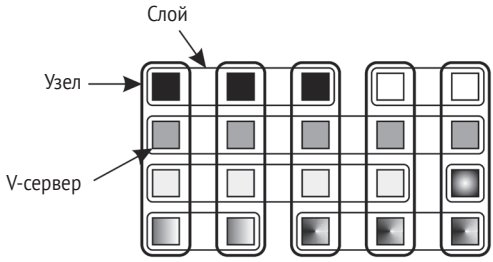


Рис. 3.23 ❖ Принцип слоев PlanetLab, показывающий наборы связанных V-серверов на разных узлах

Центральным элементом управления ресурсами PlanetLab является **менеджер узлов** (node manager). У каждого узла есть такой менеджер, реализованный с помощью отдельного V-сервера, единственной задачей которого является создание других V-серверов на узле, которым он управляет, и управление распределением ресурсов. Для создания нового слоя каждый узел также будет запускать **службу создания слоев** (slice creation service, SCS), которая, в свою очередь, может связаться с менеджером узла, запросив

его создать V-server и выделить ресурсы. С самим менеджером узла нельзя связаться напрямую через сеть, что позволяет ему сосредоточиться только на управлении локальными ресурсами. В свою очередь, SCS ни от кого не будет принимать запросы на создание слоев. Только определенные **компоненты управления слоями** (slice authorities) имеют право запрашивать создание слоя. Каждый компонент управления слоями будет иметь права доступа к коллекции узлов. Простейшая модель состоит в том, что существует только один централизованный компонент управления слоями, которому разрешено запрашивать создание слоев на всех узлах. Этот компонент обычно используется для запуска работы пользователя на PlanetLab.

Отслеживание ресурсов осуществляется с помощью спецификации ресурса, или rspec для краткости. В rspec указывается интервал времени, в течение которого были выделены определенные ресурсы. Ресурсы включают дисковое пространство, файловые дескрипторы, пропускную способность входящей и исходящей сетей, конечные точки транспортного уровня, основную память и использование ЦП. Спецификация Rspec идентифицируется через глобальный уникальный 128-битный идентификатор, известный как ресурсная способность (rcap). Имея rcap, менеджер узлов может найти связанный rspec в локальной таблице.

Ресурсы привязаны к слоям. Другими словами, чтобы использовать ресурсы, необходимо создать слой. Каждый слой связан с **поставщиком услуг** (service provider), который лучше всего рассматривать как объект, имеющий учетную запись в PlanetLab. Каждый слой затем может быть идентифицирован парой (Principal_id, slice_tag), где Principal_id идентифицирует провайдера, а slice_tag является идентификатором, выбранным провайдером.

V-серверы

Давайте теперь обратим наше внимание на сервер Vservers PlanetLab, который был описан и оценен в [Soltesz et al. 2007]. V-сервер организован в соответствии с так называемым **контейнерным подходом** (container-based approach). Основное отличие от традиционных виртуальных машин, о которых говорилось в разделе 3.2, заключается в том, что они опираются на одно общее ядро операционной системы. Это также означает, что управление ресурсами в основном осуществляется только базовой операционной системой, а не любым из V-серверов. Следовательно, основная задача V-сервера – просто поддерживать группу процессов и держать эту группу изолированной от процессов, работающих под юрисдикцией другого V-сервера. По существу, V-сервер формирует изолированный контейнер для группы процессов и выделенных ресурсов. К настоящему времени контейнеры стали довольно популярными для предоставления облачных услуг.

Эта изоляция технически установлена базовым VMM, для чего была адаптирована операционная система Linux. Одна из наиболее важных адаптаций касается разделения независимых пространств имен. Например, чтобы создать иллюзию того, что V-сервер действительно является одной машиной, Unix-процесс init всегда традиционно получает идентификатор процесса ID 1 (родительский элемент которого имеет идентификатор ID 0). Очевидно, что

в Linux VMM уже будет запущен такой процесс, но ему нужно создать еще один процесс инициализации для каждого V-сервера. Каждый из этих процессов также получит идентификатор процесса ID1, в то время как ядро отслеживает соответствие между таким идентификатором виртуального процесса и действительным, фактически назначенным идентификатором процесса. Другие примеры последовательных имен в V-серверах легко приходят на ум.

Аналогично изоляция также устанавливается путем предоставления каждому V-серверу своего собственного набора библиотек, но с использованием структуры каталогов, которую ожидают эти библиотеки, то есть с каталогами с именами /dev, /home, /proc, /usr и т. д. (которые также изображены на рис. 3.22). В принципе, такое разделенное пространство имен может быть достигнуто с помощью стандартной команды chroot, эффективно предоставляя каждому V-серверу свой собственный корневой каталог. Однако на уровне ядра требуются специальные меры для предотвращения несанкционированного доступа одного V-сервера к дереву каталогов другого V-сервера, как объяснено в [Soltesz et al., 2007].

Важное преимущество основанного на контейнерах подхода к виртуализации по сравнению с запуском отдельных гостевых операционных систем заключается в том, что распределение ресурсов в целом может быть намного проще. В частности, возможно избыточное резервирование ресурсов за счет динамического распределения ресурсов, как это делается при распределении ресурсов обычными процессами. Обычно при использовании гостевой операционной системы гостю необходимо заранее выделить фиксированное количество ресурсов (особенно основной памяти). Учитывая, что узлы, предоставляемые участвующими организациями PlanetLab, должны иметь только несколько гигабайтов основной памяти, нетрудно представить, что память может стать дефицитным ресурсом. Поэтому необходимо динамически распределять память, чтобы позволить десяткам виртуальных машин работать одновременно на одном узле. V-серверы идеально подходят для этого типа управления ресурсами. Операционные системы в таких случаях гораздо сложнее поддерживать. Конечно, это не может помешать V-серверу использовать слишком много памяти на занятом узле. Политика PlanetLab в этом случае проста: память V-сервера, когда пространство подкачки почти заполнено, сбрасывается.

3.5. Миграция кода

До сих пор нас интересовали в основном распределенные системы, в которых связь ограничена передачей данных. Однако существуют ситуации, когда текущие программы, иногда даже во время их выполнения, упрощают проект распределенной системы. В этом разделе мы подробно рассмотрим миграцию кода. Мы начнем с рассмотрения различных подходов к переносу кода, а затем обсудим, как обращаться с локальными ресурсами, которые использует переносимая программа. Особенно трудной проблемой является перенос кода в гетерогенных системах, что также будет обсуждаться.

Причины переноса кода

Традиционно миграция кода в распределенных системах происходила в форме миграции процессов, при которой весь процесс перемещался с одного узла на другой [Milojicic et al., 2000]. Перемещение запущенного процесса на другую машину является дорогостоящей и сложной задачей, и для этого должна быть веская причина. Этой причиной всегда была производительность. Основная идея заключается в том, что общая производительность системы может быть улучшена, если процессы переносятся с сильно загруженных на слабо загруженные машины. Нагрузка часто выражается через длину очереди центрального процессора (ЦП) или загрузку ЦП, но используются и другие показатели производительности. В заключение обзора [Milojicic et al., 2000] сделан вывод, что миграция процесса более не является жизнеспособным вариантом для улучшения распределенных систем.

Однако вместо разгрузки машин мы можем переместить код. В частности, миграция полных виртуальных машин с их набором приложений на менее загруженные машины с целью минимизации общего числа используемых узлов является обычной практикой оптимизации использования энергии в центрах обработки данных. Интересно, что хотя миграция виртуальных машин может потребовать больше ресурсов, сама задача гораздо менее сложна, чем миграция процесса, как мы обсуждаем в примечании 3.11.

В общем, алгоритмы распределения нагрузки, с помощью которых принимаются решения относительно распределения и перераспределения задач по набору машин, играют важную роль в вычислительных системах. Однако во многих современных распределенных системах оптимизация вычислительных мощностей является меньшей проблемой, чем, например, попытка свести к минимуму обмен данными. Более того, из-за неоднородности базовых платформ и компьютерных сетей повышение производительности за счет миграции кода часто основано на качественных рассуждениях, а не на математических моделях.

Рассмотрим в качестве примера клиент-серверную систему, в которой сервер управляет огромной базой данных. Если клиентскому приложению необходимо выполнить много операций с базой данных, связанных с большими объемами данных, может быть лучше отправить часть клиентского приложения на сервер и по сети отправить только результаты. В противном случае сеть может быть завалена передачей данных с сервера клиенту. В этом случае миграция кода основана на предположении, что обычно имеет смысл обрабатывать данные вблизи того места, где эти данные находятся.

По этой же причине можно использовать для переноса части сервера на клиента. Например, во многих интерактивных приложениях баз данных клиенты должны заполнять формы, которые впоследствии преобразуются в серию операций базы данных. Обработка формы на стороне клиента и отправка на сервер только заполненной формы иногда позволяют избежать необходимости прохождения через сеть относительно большого количества небольших сообщений. В результате клиент получает лучшую производительность, и в то же время сервер тратит меньше времени на обработку форм

и обмен данными. В случае смартфонов перемещение кода, который будет выполняться на контроллере вместо сервера, может быть единственным жизнеспособным решением для получения приемлемой производительности как для клиента, так и для сервера (см. обзор подсчета разгрузки в [Kumar et al., 2013]).

Поддержка миграции кода также может помочь повысить производительность за счет использования параллелизма, но без обычных сложностей, связанных с параллельным программированием. Типичным примером является поиск информации в интернете. Относительно просто реализовать поисковый запрос в виде небольшой мобильной программы, называемой **мобильным агентом** (mobile agent), которая перемещается с сайта на сайт. Сделав несколько копий такой программы и отправив каждую на разные сайты, мы сможем добиться линейного ускорения по сравнению с использованием только одного экземпляра программы. Однако в работе [Carzaniga et al., 2007] сделан вывод, что мобильные агенты никогда не были успешными и не имели очевидного преимущества перед другими технологиями. Более того, что очень важно, оказалось, что мобильный код такого типа практически не может работать безопасно.

Помимо повышения производительности, есть и другие причины для поддержки миграции кода. Наиболее важным из них является гибкость. Традиционный подход к построению распределенных приложений состоит в том, чтобы разделить приложение на разные части и заранее решить, где каждая часть должна быть выполнена. Этот подход, например, привел к различным многоуровневым клиент-серверным приложениям, рассмотренным в разделе 2.3.

Если код может перемещаться между различными машинами, становится возможным динамически конфигурировать распределенные системы. Например, предположим, что сервер реализует стандартизированный интерфейс для файловой системы. Чтобы разрешить удаленным клиентам доступ к файловой системе, сервер использует собственный протокол. Обычно клиентская реализация интерфейса файловой системы, основанная на этом протоколе, должна быть связана с клиентским приложением. Этот подход требует, чтобы программное обеспечение было легко доступно клиенту во время разработки клиентского приложения.

Альтернатива состоит в том, чтобы позволить серверу обеспечить реализацию клиента не раньше, чем это строго необходимо, то есть когда клиент связывается с сервером. В этот момент клиент динамически загружает реализацию, проходит необходимые шаги инициализации и впоследствии вызывает сервер. Этот принцип показан на рис. 3.24 (отметим, что хранилище кода обычно располагается как часть сервера). Эта модель динамически перемещаемого кода с удаленного сайта требует стандартизации протокола загрузки и инициализации кода. Кроме того, необходимо, чтобы загруженный код мог быть выполнен на компьютере клиента. Как правило, скрипты, которые запускаются на виртуальной машине, встроенной, например, в веб-браузер, справляются с задачей. Возможно, эта форма переноса кода была ключом к успеху динамического интернета. Эти и другие решения обсуждаются ниже и в последующих главах.

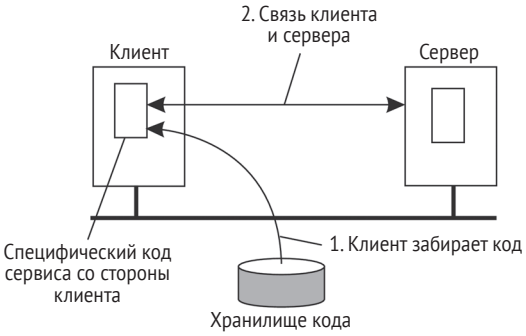


Рис. 3.24 ❖ Принцип динамического конфигурирования клиента для связи с сервером

Важным преимуществом этой модели динамической загрузки программного обеспечения на стороне клиента является то, что клиентам не нужно предварительно устанавливать все программное обеспечение для взаимодействия с серверами. Вместо этого программное обеспечение можно перемещать по мере необходимости, а также сбрасывать, когда оно больше не требуется. Другое преимущество состоит в том, что пока интерфейсы стандартизированы, мы можем изменять протокол клиент-сервер и его реализацию так часто, как нам нравится. Изменения не затронут существующие клиентские приложения, которые зависят от сервера.

Есть, конечно, и недостатки. Самый серьезный, обсуждаемый в главе 9, имеет отношение к безопасности. Слепое доверие тому, что загруженный код реализует только объявленный интерфейс при доступе к незащищенному жесткому диску и не отправляет самые сочные части неизвестно кому, не всегда может быть хорошей идеей. К счастью, хорошо известно, как защитить клиента от вредоносного загруженного кода.

Примечание 3.10 (дополнительная информация: отказаться от тонких клиентов?)

К настоящему времени стало гораздо больше знаний и опыта в отношении прозрачной и безопасной динамической миграции кода к клиентам. В результате частично вернулась тенденция, которую мы описали в примечании 2.4, о переходе к вычислениям на тонких клиентах, поскольку управление программным обеспечением на стороне клиента часто оказывалось обременительным. Благодаря динамическому переносу программного обеспечения на сторону клиента, но при этом сохраняя управление этим программным обеспечением полностью на стороне сервера (или, скорее, у его владельца), использование «более богатого» программного обеспечения на стороне клиента стало практически осуществимым.

Примечание 3.11 (дополнительно: модели для переноса кода)

Хотя миграция кода предполагает, что мы перемещаем только код между компьютерами, этот термин на самом деле охватывает гораздо более обширную область. Традиционно коммуникации в распределенных системах связаны с обменом дан-

ными между процессами. Миграция кода в самом широком смысле связана с перемещением программ между компьютерами с намерением, чтобы эти программы выполнялись в целевой системе. В некоторых случаях, например при миграции процессов, статус выполнения программы, ожидающие сигналы и другие составляющие среды также должны быть перемещены.

Чтобы лучше понять различные модели миграции кода, мы используем инфраструктуру, предложенную в [Fuggetta et al., 1998]. В этой структуре процесс состоит из трех сегментов. **Сегмент кода** (code segment) – это часть, которая содержит набор инструкций, составляющих выполняемую программу. **Сегмент ресурса** (resource segment) содержит ссылки на внешние ресурсы, необходимые процессу, такие как файлы, принтеры, устройства, другие процессы и т. д. Наконец, **сегмент выполнения** (execution segment) используется для хранения текущего состояния выполнения процесса, состоящего из личных данных, стека и, конечно, счетчика программы.

Еще одно различие может быть сделано между миграцией, **инициированной отправителем** (sender-initiated migration) и **получателем** (receiver-initiated migration). При миграции, инициированной отправителем, миграция инициируется на компьютере, на котором в данный момент находится или выполняется код. Как правило, инициируемая отправителем миграция выполняется при загрузке программ на вычислительный сервер. Другой пример – отправка запроса или пакета запросов на удаленный сервер базы данных.

При миграции, инициированной получателем, инициатива по миграции кода принадлежит целевой машине. Java-апплеты являются примером такого подхода.

Миграция, инициированная получателем, проще, чем миграция, инициированная отправителем. Во многих случаях миграция кода происходит между клиентом и сервером, где клиент берет на себя инициативу по миграции. Для безопасной загрузки кода на сервер, как это делается при миграции, инициированной отправителем, часто требуется, чтобы клиент был предварительно зарегистрирован и аутентифицирован на этом сервере. Другими словами, сервер должен знать всех своих клиентов, и причина в том, что клиент, вероятно, захочет иметь доступ к ресурсам сервера, таким как его диск. Защита таких ресурсов имеет важное значение. Напротив, загрузка кода, как в случае, инициированном приемником, часто может выполняться анонимно. Более того, сервер, как правило, не заинтересован в ресурсах клиента. Вместо этого миграция кода к клиенту выполняется только для повышения производительности на стороне клиента. Для этого необходимо защитить только ограниченное количество ресурсов, таких как память и сетевые соединения. Мы вернемся к безопасному переносу кода в главе 9.

Это приводит нас к четырем различным парадигмам мобильности кода, как показано на рис. 3.25. Вслед за [Following Fuggetta et al., 1998] мы делаем различие между **клиент-серверными вычислениями** (client-server computing), **удаленной оценкой** (remote evaluation), **кодом по требованию** (code-on-demand) и **мобильными агентами** (mobile agents). На рис. 3.25 показана ситуация на клиенте и сервере соответственно до и после выполнения мобильного кода.

В случае клиент-серверных вычислений код, состояние выполнения и сегмент ресурсов все расположены на сервере, и после выполнения, как правило, изменяется только состояние выполнения на сервере. Это изменение состояния обозначено на рисунке звездочкой. При инициированной отправителем **удаленной оценке** клиент переносит код на сервер, где этот код выполняется, и приводит к изменению состояния выполнения на сервере. **Код по требованию** – это схема, инициированная получателем, по которой клиент получает код от сервера с изменением его состояния на стороне клиента и работы с клиентскими ресурсами. Наконец, **мобильные агенты** обычно следуют методу, инициированному отправителем, перемещая код,

а также состояние выполнения с клиента на сервер, работая как на клиентских, так и на серверных ресурсах. Работа мобильного агента обычно приводит к модификации ассоциированного состояния выполнения.

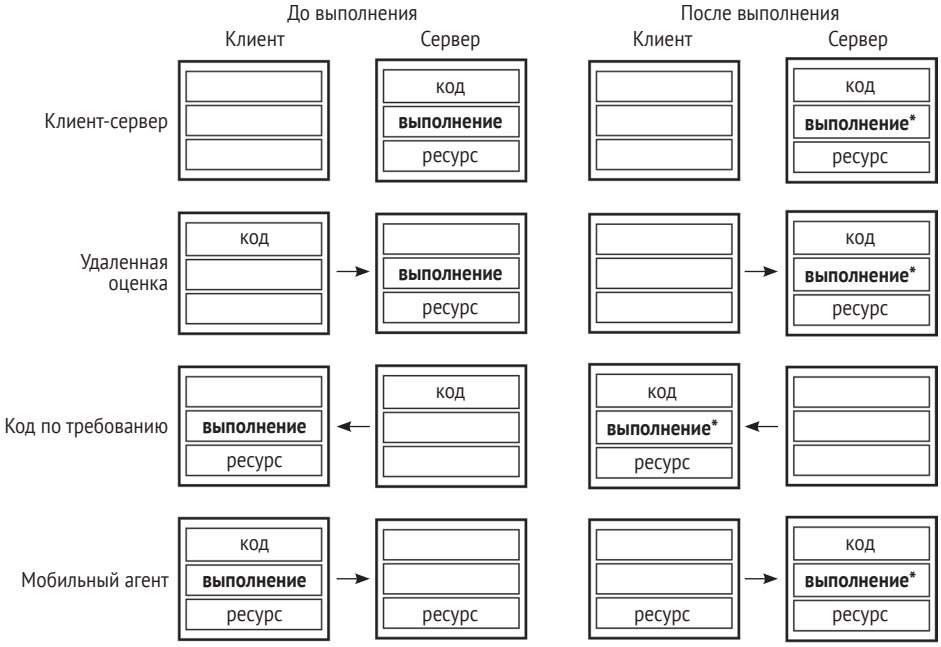


Рис. 3.25 ❖ Четыре различные парадигмы мобильности кода

Минимум для миграции кода – обеспечить только **слабую мобильность** (weak mobility). В этой модели можно передавать лишь сегмент кода вместе с некоторыми данными инициализации. Характерной особенностью слабой мобильности является то, что переданная программа всегда запускается заново. Это то, что происходит, например, с апплетами Java, которые начинаются с того же начального состояния. Другими словами, базовое промежуточное ПО не поддерживает историю, из которой перенесенный код был прерван в предыдущем месте. Если такая история должна быть сохранена, она должна быть закодирована как часть самого мобильного приложения. Преимущество слабой мобильности заключается в ее простоте, поскольку она требует только того, чтобы целевая машина могла выполнять сегмент кода. По сути, это сводится к тому, чтобы сделать код переносимым. Мы возвращаемся к этим вопросам при обсуждении миграции в гетерогенных системах.

В отличие от слабой мобильности, в системах, которые поддерживают **сильную мобильность** (strong mobility), также может быть передан исполнительный сегмент. Характерной особенностью сильной мобильности является то, что запущенный процесс может быть остановлен, впоследствии перемещен на другую машину и затем выполнение возобновлено именно там, где он остановился. Очевидно, что сильная мобильность является гораздо более общей, чем слабая мобильность, но ее сложнее реализовать. В частности, при миграции процесса исполнительный сегмент обычно также содержит данные, которые сильно зависят от конкретной реализации базовой операционной системы. Например, он может опираться на информацию, обычно находящуюся в таблице процессов операционной системы. Как

следствие переход на другую операционную систему, даже ту, которая принадлежит к тому же семейству, что и источник, может вызвать много головной боли.

В случае слабой мобильности также имеет значение, выполняется ли перенесенный код целевым процессом или запускается как отдельный процесс. Например, Java-апплеты просто загружаются веб-браузером и выполняются в адресном пространстве браузера. Преимущество этого подхода заключается в том, что нет необходимости запускать отдельный процесс, что позволяет избежать межпроцессорного взаимодействия на целевой машине. Очевидно, что основным недостатком является то, что целевой процесс должен быть защищен от вредоносного или непреднамеренного выполнения кода, что может быть достаточной причиной, чтобы изолировать перенесенный код в отдельном процессе.

Вместо перемещения запущенного процесса, называемого миграцией процесса, удаленное клонирование также может поддерживать сильную мобильность. В отличие от процесса миграции, клонирование дает точную копию исходного процесса, но теперь выполняется на другом компьютере. Клонированный процесс выполняется параллельно с исходным процессом. В системах Unix удаленное клонирование происходит путем разветвления дочернего процесса и продолжения этого дочернего процесса на удаленной машине. Преимущество клонирования заключается в том, что модель очень похожа на ту, которая уже используется во многих приложениях. Разница лишь в том, что клонированный процесс выполняется на другом компьютере. В этом смысле миграция путем клонирования – это простой способ улучшить прозрачность распространения.

Миграция в гетерогенных системах

До сих пор мы молчаливо предполагали, что перенесенный код может быть легко выполнен на целевой машине. Это предположение выполняется при работе с гомогенными системами. В целом, однако, распределенные системы построены на разнородной совокупности платформ, каждая из которых имеет свою собственную операционную систему и архитектуру машины.

Проблемы, возникающие из-за неоднородности, во многом такие же, как проблемы переносимости. Неудивительно, что и решения также очень похожи. Например, в конце 1970-х годов простым решением для облегчения многих проблем переноса Pascal на разные машины было создание независимого от машины промежуточного кода для абстрактной виртуальной машины [Barron, 1981]. Эта машина, конечно, должна была бы быть реализована на многих платформах, но тогда она позволяла бы запускать программы на Pascal где угодно. Хотя эта простая идея широко использовалась в течение нескольких лет, она так и не стала широко распространенным решением проблем переносимости для других языков, особенно C.

Спустя 25 лет миграция кода в гетерогенных системах решается с помощью языков сценариев и таких легко переносимых языков, как Java. По сути, в этих решениях используется тот же подход, что и при переносе в Pascal. Общим для всех таких решений является то, что они опираются на (процессную) виртуальную машину, которая либо напрямую интерпретирует исходный код (как в случае языков сценариев), либо иным образом интерпретирует промежуточный код, сгенерированный компилятором (как в Java). Быть в нужном месте в нужное время также важно для разработчиков языка.

Дальнейшие разработки ослабили зависимость от языков программирования. В частности, были предложены решения для миграции не только процессов, но и целых вычислительных сред. Основная идея состоит в том, чтобы разделить общую среду и предоставить процессам в одной и той же части свой взгляд на свою вычислительную среду. Это разделение происходит в форме мониторов виртуальных машин, работающих под управлением операционной системы и набора приложений.

С миграцией виртуальной машины становится возможным отделить вычислительную среду от базовой системы и фактически перенести ее на другую машину (обзор механизмов миграции для виртуальных машин см. [Medina and Garcia, 2014]). Основным преимуществом этого подхода является то, что процессы могут оставаться в неведении относительно самой миграции: их не нужно прерывать при выполнении, и при этом они не должны испытывать никаких проблем с используемыми ресурсами. Последние либо мигрируют вместе с процессом, либо способ, которым процесс обращается к ресурсу, остается незатронутым (по крайней мере, для этого процесса).

В качестве примера см. работу [Clark et al., 2005], в которой внимание концентрируется на миграции виртуализированной операционной системы в режиме реального времени, что обычно было бы удобно в кластере серверов, где достигается тесная связь через единую общую локальную сеть. В этих условиях миграция включает в себя две основные проблемы: перенос всего образа памяти и перенос привязок на локальные ресурсы.

Что касается первой проблемы, в принципе, есть три способа обработки миграции (которые можно объединить):

1) перенос страниц памяти на новую машину и повторная отправка страниц, которые позже были изменены в процессе миграции;
2) остановка текущей виртуальной машины; перенести память и запустить новую виртуальную машину;
3) разрешение новой виртуальной машине загружать новые страницы по мере необходимости, то есть разрешать процессам запускаться на новой виртуальной машине немедленно и копировать страницы памяти по требованию.

Второй вариант может привести к недопустимому простою, если на мигрирующей виртуальной машине запущен работающий сервис, то есть тот, который предлагает непрерывный сервис. С другой стороны, подход по требованию, представленный третьим вариантом, может значительно продлить период миграции, что также может привести к снижению производительности, поскольку требуется много времени, прежде чем рабочий набор перенесенных процессов будет перемещен в новую машину.

В качестве альтернативы в [Clark et al., 2005] предлагается использовать подход предварительного копирования, который сочетает в себе первый вариант, а также краткий этап остановки и копирования, представленный вторым вариантом. Как выясняется, такая комбинация может привести к очень низким простоям обслуживания (см. примечание 3.12).

Что касается локальных ресурсов, то вопросы упрощаются при работе только с кластерным сервером. Во-первых, поскольку существует единственная сеть, все, что нужно сделать, – это объявить о новой привязке сети к MAC-

адресу, чтобы клиенты могли связываться с перенесенными процессами через корректный сетевой интерфейс. Наконец, если можно предположить, что хранилище предоставляется в качестве отдельного уровня (как мы показали на рис. 3.18), то перенос привязки к файлам аналогично прост, поскольку он фактически означает восстановление сетевых подключений.

Примечание 3.12 (дополнительно: о производительности живой миграции виртуальных машин).

Одна потенциальная проблема с миграцией виртуальной машины состоит в том, что это может занять значительное время. Это само по себе не должно быть плохим, если службы, работающие на переносимой виртуальной машине, могут продолжать работать. Практический подход был кратко описан выше. Во-первых, страницы памяти копируются на целевой компьютер, возможно отправляя обновления страниц, которые были изменены во время копирования (помните, что копирование большого количества памяти может занять десятки секунд, даже в высокоскоростной локальной сети). Во-вторых, когда большинство страниц были благополучно скопированы, текущий компьютер останавливается, оставшиеся «грязные» страницы копируются в целевое устройство, где теперь можно начать запуск точных копий полученного оригинала.

Время простоя, при котором необходимо скопировать оставшиеся «грязные» страницы, зависит от приложений, запущенных на виртуальной машине. В отчете [Clark et al., 2005] время простоя для конкретных конфигураций составило от 60 мс до менее 4 с.

В [Voorsluys et al., 2009] приходят к аналогичным значениям. Однако что может быть более интересным – это наблюдать за временем отклика службы, работающей на виртуальной машине, во время миграции последней. Моделью в таком случае служит то, что служба продолжает работать на исходном компьютере, пока не завершится полная миграция. Однако нельзя игнорировать тот факт, что сама миграция является ресурсоемкой операцией, требующей значительной вычислительной мощности, а также пропускной способности сети.

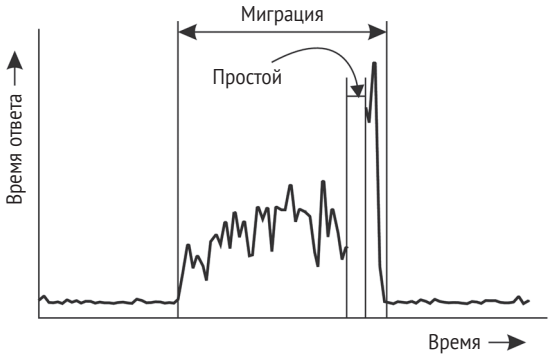


Рис. 3.26 ❖ Влияние на время отклика сервиса при миграции его базовой виртуальной машины (адаптировано из [Voorsluys et al., 2009])

В работе [Voorsluys et al., 2009] отмечено, что полная миграция может фактически занять десятки секунд, что приводит к увеличению времени отклика в 10–20 раз.

Кроме того, следует понимать, что во время миграции служба будет полностью недоступна (т. е. не будет отвечать), возможно, в течение 4 с. Хорошей новостью является то, что время отклика завершения миграции значительно возрастает, как показано на рис. 3.26, только после простоя.

Во многих случаях виртуальные машины переносятся для оптимизации использования реальных машин. Однако может быть также желательно клонировать виртуальную машину, например потому, что рабочая нагрузка для текущей машины становится слишком высокой. Такое клонирование очень похоже на использование нескольких процессов на параллельных серверах, с помощью которых процесс диспетчера создает рабочие процессы для обработки входящих запросов. Эта схема была объяснена на рис. 3.4 при обсуждении многопоточных серверов.

При клонировании для этого типа реализации часто имеет смысл не копировать первые страницы памяти, а фактически начинать с как можно меньшего количества страниц, так как служба, работающая на клонированной машине, по существу, запустится заново. Обратите внимание, что это поведение очень похоже на обычное поведение типа «родители-потомки», которое мы наблюдаем при разветвлении процесса Unix. А именно дочерний процесс начнет с загрузки своего собственного исполняемого файла, тем самым эффективно очистив память, которую он унаследовал от своего родительского узла. Эта аналогия вдохновила авторов [Lagar-Cavilla et al., 2009] разработать аналогичный механизм для *разветвления* виртуальных машин. Однако, в отличие от механизма, который традиционно использовался для миграции виртуальных машин, сначала требуются копии страниц их разветвленной виртуальной машины. Результатом является чрезвычайно эффективный механизм клонирования.

Таким образом, видно, что не существует единственного лучшего способа размещения копий виртуальной машины на разных физических машинах: это очень сильно зависит от того, как и почему размещается виртуальная машина.

3.6. Резюме

Процессы играют фундаментальную роль в распределенных системах, поскольку они формируют основу для связи между различными машинами. Важным вопросом является то, как процессы организованы внутри и, в частности, поддерживают ли они несколько потоков управления. Потоки в распределенных системах особенно полезны для продолжения использования ЦП, когда выполняется блокирующая операция ввода-вывода. Таким образом, становится возможным создать высокоэффективные серверы, которые запускают несколько потоков параллельно, из которых некоторые могут блокироваться, чтобы дождаться завершения ввода-вывода диска или сетевого взаимодействия. В общем случае потоки предпочтительнее использования процессов, когда на карту поставлена производительность.

Виртуализация с давних времен была важной областью компьютерных наук, но в связи с появлением облачных вычислений она привлекла еще большее внимание. Популярные схемы виртуализации позволяют пользователям запускать набор приложений поверх своей любимой операционной системы и настраивать виртуальные распределенные системы в облаке. Впечатляет, что производительность остается близкой к производительно-

сти приложений в операционной системе хоста, если только эта система не используется совместно с другими виртуальными машинами. Гибкое применение виртуальных машин привело к различным типам сервисов для облачных вычислений, включая инфраструктуры, платформы и программное обеспечение – все они работают в виртуальных средах.

Организация распределенного приложения в показателях клиентов и серверов оказалась полезной. Клиентские процессы обычно реализуют пользовательские интерфейсы, которые могут варьироваться от очень простых дисплеев до сложных интерфейсов, обрабатывающих составные документы. Кроме того, клиентское программное обеспечение нацелено на достижение прозрачности распределения за счет сокрытия подробностей, касающихся связи с серверами, расположения этих серверов и независимо от того, реплицированы серверы или нет. Кроме того, клиентское программное обеспечение частично отвечает за сокрытие сбоев и восстановление после сбоев.

Серверы часто более сложны, чем клиенты, но тем не менее подвержены лишь относительно небольшому числу проблем проектирования. Например, серверы могут быть итеративными или параллельными, реализовывать одну или несколько служб и могут быть без сохранения состояния или с сохранением состояния. Другие проблемы проектирования касаются адресации сервисов и механизмов для прерывания работы сервера, после того как запрос на обслуживание был выдан и, возможно, уже обрабатывается.

Особое внимание необходимо уделить при организации серверов в кластере. Общая цель состоит в том, чтобы скрыть внутренние компоненты кластера от внешнего мира. Это означает, что организация кластера должна быть защищена от приложений. Для этого большинство кластеров используют одну точку входа, которая может передавать сообщения на серверы в кластере. Сложной проблемой является прозрачная замена этой единственной точки входа полностью распределенным решением.

Для размещения удаленных объектов были разработаны расширенные объектные серверы. Объектный сервер предоставляет множество услуг базовым объектам, в том числе средства хранения объектов или обеспечения упорядочивания входящих запросов. Еще одна важная роль заключается в создании для внешнего мира иллюзии, что коллекция данных и процедур, работающих с этими данными, соответствует концепции объекта. Эта роль реализуется с помощью объектных адаптеров. Объектно-ориентированные системы достигли уровня, в котором мы можем создавать целые фреймворки с расширением для поддержки конкретных приложений. Java доказал, что предоставляет мощные средства для настройки более общих сервисов, примером которых является популярная концепция Enterprise Java Beans и ее реализация.

Типичным сервером для веб-систем является сервер от Apache. Сервер Apache можно рассматривать как общее решение для обработки множества HTTP-запросов. Предлагая правильные методы, мы, по сути, получаем гибко настраиваемый веб-сервер. Apache служит примером не только для традиционных веб-сайтов, но и для настройки кластеров совместных веб-серверов, даже в глобальных сетях.

Важной темой для распределенных систем является перенос кода между различными машинами. Двумя важными причинами поддержки миграции кода является повышение производительности и гибкости. Когда связь дорогая, мы иногда можем уменьшить связь, отправляя вычисления с сервера на клиента и позволяя клиенту выполнять как можно больше локальной обработки. Гибкость увеличивается, если клиент может динамически загружать программное обеспечение, необходимое для взаимодействия с конкретным сервером. Загруженное программное обеспечение может быть специально предназначено для этого сервера, не заставляя клиента предварительно его устанавливать.

Миграция кода влечет за собой проблемы, связанные с использованием локальных ресурсов, для которых требуется либо миграция ресурсов и новые привязки к локальным ресурсам на целевом компьютере, либо использование общесистемных сетевых ссылок. Другая проблема состоит в том, что кодирование требует, чтобы мы учли гетерогенность. Современная практика показывает, что лучшим решением для управления неоднородностью является использование виртуальных машин. Они могут принимать форму виртуальных машин процессов, как, например, в случае Java, или с помощью мониторов виртуальных машин, которые позволяют эффективно переносить набор процессов вместе с базовой операционной системой.

Глава 4

Коммуникации

Межпроцессорное взаимодействие лежит в основе всех распределенных систем. Нет смысла изучать распределенные системы без тщательного изучения способов, которыми процессы на разных машинах могут обмениваться информацией. Связь в распределенных системах всегда традиционно основывалась на передаче сообщений низкого уровня, предлагаемой базовой сетью. Осуществление связи через передачу сообщений сложнее, чем использование примитивов на основе разделяемой памяти, доступных для нераспределенных платформ. Современные распределенные системы часто состоят из тысяч или даже миллионов процессов, разбросанных по сети с ненадежной связью, такой как интернет. Если примитивные средства связи компьютерных сетей не будут заменены чем-то другим, разработка крупномасштабных распределенных приложений будет крайне затруднена.

В этой главе мы начнем с обсуждения правил, которых должны придерживаться коммуникационные процессы, известные как протоколы, и сосредоточимся на структурировании этих протоколов в виде уровней. Затем рассмотрим две широко используемые модели связи: удаленный вызов процедур (Remote Procedure Call, RPC) и промежуточное программное обеспечение, ориентированное на сообщения (Message-Oriented Middleware, MOM). Мы также обсудим общую проблему отправки данных нескольким получателям, которая называется многоадресной рассылкой.

Нашей первой моделью для связи в распределенных системах является удаленный вызов процедур (RPC). RPC стремится скрыть большинство тонкостей передачи сообщений и идеально подходит для клиент-серверных приложений. Однако реализацию RPC прозрачным осуществить не просто. Мы рассмотрим ряд важных деталей, которые нельзя игнорировать, и углубимся в собственно код, чтобы проиллюстрировать, в какой степени можно реализовать прозрачность распределения, дабы производительность оставалась приемлемой.

Во многих распределенных приложениях связь не следует довольно строгой схеме взаимодействия клиент-сервер. В этих случаях оказывается, что мышление в терминах сообщений более приемлемо. Коммуникационные средства компьютерных сетей низкого уровня во многих отношениях не подходят опять-таки из-за отсутствия прозрачности распределения. Альтернативой является использование модели очереди сообщений высокого уровня, в которой обмен данными происходит почти так же, как в системах электронной почты. Коммуникация, ориентированная на сообщениях, яв-

ляется предметом, достаточно важным, чтобы уделить ему отдельный раздел. Мы рассмотрим многочисленные аспекты, включая маршрутизацию на уровне приложений.

Наконец, поскольку наше понимание настройки многоадресных средств улучшилось, появились новые и элегантные решения для распространения данных. Мы уделяем специальное внимание этому вопросу в последнем разделе данной главы, обсуждая традиционные детерминированные средства многоадресной рассылки, а также вероятностные подходы, используемые в лавинной маршрутизации и обмене сообщениями. Последние получают повышенное внимание в последние годы из-за их элегантности и простоты.

4.1. Основы

Прежде чем мы начнем наше обсуждение коммуникации в распределенных системах, мы вкратце рассмотрим некоторые фундаментальные проблемы, связанные с коммуникацией.

В следующем разделе мы кратко обсудим сетевые протоколы связи, поскольку они формируют основу для любой распределенной системы. После этого используем другой подход, классифицируя различные типы связи, которые обычно применяются в распределенных системах.

Многоуровневые протоколы

Из-за отсутствия общей памяти вся связь в распределенных системах основана на отправке и получении (низкоуровневых) сообщений. Когда процесс P хочет связаться с процессом Q, он сначала создает сообщение в своем собственном адресном пространстве. Затем выполняет системный вызов, который заставляет операционную систему отправлять сообщение по сети в Q. Хотя эта базовая идея звучит достаточно просто, чтобы предотвратить хаос, P и Q должны согласовать значение отправляемых битов.

Эталонная модель OSI

Чтобы упростить работу с многочисленными уровнями и проблемами, связанными с коммуникацией, Международная организация стандартов (International Standards Organization, ISO) разработала эталонную модель, которая четко идентифицирует различные уровни, дает им стандартные названия и указывает, какой уровень должен выполнять какую работу. Эта модель называется **эталонной моделью взаимодействия открытых систем** (Open Systems Interconnection Reference Model) [Day and Zimmerman, 1983], обычно сокращенно **ISO OSI**, или иногда просто **модель OSI**. Следует подчеркнуть, что протоколы, которые были разработаны как часть модели OSI, никогда широко не использовались и по сути являются мертвыми. Однако сама базовая модель оказалась весьма полезной для понимания компьютерных сетей. Хотя мы не намерены давать здесь полное описание данной мо-

дели и всех ее последствий, краткое введение будет полезно. Для получения более подробной информации см. [Tanenbaum and Wetherall, 2010].

Модель OSI предназначена для обеспечения взаимодействия открытых систем. Открытая система – это система, которая готова взаимодействовать с любой другой открытой системой с использованием стандартных правил, определяющих формат, содержание и значение отправляемых и получаемых сообщений. Эти правила формализованы в так называемых **протоколах связи** (communication service). Чтобы позволить группе компьютеров обмениваться данными по сети, все они должны согласовать используемые протоколы. Говорят, что протокол предоставляет **услуги связи** (connection-oriented service). Существует два типа таких услуг. В случае службы, ориентированной на установление соединения, перед обменом данными отправитель и получатель сначала явно устанавливают соединение и, возможно, согласовывают конкретные параметры протокола, который они будут использовать. Когда это сделано, они освобождают (прекращают) соединение. Телефон является типичной услугой связи с установлением соединения. При использовании **услуг без установления соединения** (connectionless services) предварительная настройка не требуется. Отправитель просто передает первое сообщение, когда оно готово. Удаление письма в почтовом ящике является примером использования службы связи без установления соединения. В компьютерах распространены как ориентированные на соединение, так и не требующие соединения коммуникации.

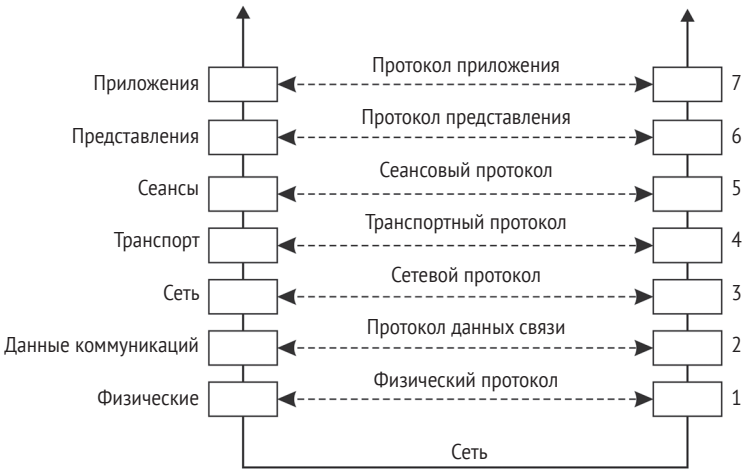


Рис. 4.1 ❖ Уровни, интерфейсы и протоколы в модели OSI

В модели OSI связь делится на семь уровней или слоев, как показано на рис. 4.1. Каждый уровень предлагает одну или несколько конкретных услуг связи для уровня над ним. Таким образом, проблема получения сообщения от A до B может быть разделена на управляемые части, каждая из которых может быть решена независимо от других. Каждый слой предоставляет **интерфейс** для уровня над ним. Интерфейс состоит из набора операций, ко-

торые вместе определяют сервис, который уровень готов предложить. Семь уровней OSI:

- **физический уровень.** Имеет дело со стандартизацией того, как два компьютера связаны и как представлены 0 и 1;
- **канальный уровень данных.** Обеспечивает средства для обнаружения и, возможно, исправления ошибок передачи, а также протоколы поддержания отправителя и получателя в одном и том же темпе;
- **сетевой уровень.** Содержит протоколы маршрутизации сообщения через компьютерную сеть, а также протоколы для обработки перегрузки;
- **транспортный уровень.** В основном содержит протоколы для непосредственной поддержки приложений, таких как те, которые устанавливают надежную связь или поддерживают потоковую передачу данных в реальном времени;
- **сеансовый уровень.** Обеспечивает поддержку для сеансов между приложениями;
- **уровень представления.** Описывает, как данные представляются независимо от хостов, на которых запущены взаимодействующие приложения;
- **уровень приложения.** По существу, все остальные протоколы: электронной почты, протоколы веб-доступа, протоколы передачи файлов и т. д.

Когда процесс P хочет установить связь с каким-либо удаленным процессом Q, он создает сообщение и передает его на уровень приложения, как ему предлагается средствами интерфейса. Интерфейс обычно отображается в форме библиотечной процедуры. Программное обеспечение прикладного уровня затем добавляет **заголовок** (header) в начало сообщения и передает полученное сообщение через интерфейс уровня 6/7 на уровень представления. Уровень представления, в свою очередь, добавляет свой собственный заголовок и передает результат на уровень сеанса и т. д. Некоторые уровни добавляют не только заголовок спереди, но и трейлер в конце. Когда он достигает дна, физический уровень фактически передает сообщение (которое сейчас может выглядеть, как показано на рис. 4.2), помещая его в физическую среду передачи.

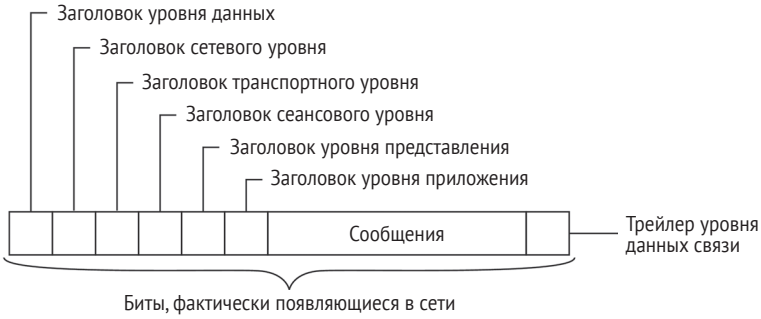


Рис. 4.2 ❖ Типичное сообщение в сети

Когда сообщение поступает на удаленный компьютер, на котором размещается Q, оно передается вверх, при этом каждый уровень удаляется и проверяет свой собственный заголовок. Наконец, сообщение поступает в получатель, процесс Q, который может ответить на него, используя обратный путь. Информация в *n*-заголовке используется для протокола *n*-уровня.

Примечание 4.1 (дополнительная информация: протоколы в модели OSI)

Давайте кратко рассмотрим каждый из уровней OSI по очереди, начиная с нижней части. Вместо того чтобы приводить примеры протоколов OSI, на каждом уровне там, где это уместно, мы укажем некоторые используемые интернет-протоколы.

Протоколы нижнего уровня. Три набора нижнего уровня протоколов OSI реализуют основные функции, которые охватывают компьютерную сеть.

Физический уровень связан с передачей 0 и 1. Какое напряжение в вольтах использовать для 0 и 1, сколько битов в секунду может быть отправлено и может ли передача происходить одновременно в обоих направлениях, являются ключевыми проблемами на физическом уровне. Кроме того, здесь важны размер и форма сетевого разъема (штекера), а также количество контактов и значение каждого из них.

Протокол физического уровня имеет дело со стандартизацией электрических, оптических, механических и сигнальных интерфейсов, так что когда одна машина отправляет 0 бит, он фактически принимается как 0 бит, а не 1 бит. Было разработано много стандартов физического уровня (для разных носителей), например стандарт USB для последовательных линий связи.

Физический уровень просто отправляет биты. Пока ошибок не происходит, все хорошо. Однако реальные сети связи подвержены ошибкам, поэтому необходим какой-то механизм для их обнаружения и исправления. Этот механизм является главной задачей канального уровня. Что он делает, так это группирует биты в единицы, иногда называемые кадрами, и видит, что каждый кадр принят правильно.

Уровень канала передачи данных выполняет свою работу, помещая специальный битовый шаблон в начало и конец каждого кадра, чтобы пометить их, а также вычисляет контрольную сумму путем сложения всех байтов в кадре определенным образом. Уровень канала передачи данных добавляет контрольную сумму к кадру. Когда приходит кадр, получатель повторно вычисляет контрольную сумму из данных и сравнивает результат с контрольной суммой, следующей за кадром. Если оба согласны, кадр считается правильным и принимается. Если они не согласны, получатель просит отправителя повторно передать его. Фреймам присваиваются порядковые номера (в заголовке), поэтому каждый может сказать, что есть что.

В локальной сети (LAN) отправителю обычно не требуется определять местонахождение получателя. Он просто отправляет сообщение в сеть, а получатель вынимает его. Однако глобальная сеть состоит из большого количества машин, каждая из которых имеет определенное количество линий, соединяющих машины, подобно крупномасштабной карте, на которой показаны крупные города и дороги, соединяющие их. Чтобы получить сообщение от отправителя к получателю, может потребоваться сделать несколько переключений, при этом каждый выбирает исходящую линию для использования. Вопрос, как выбрать лучший путь, называется маршрутизацией и, по сути, является основной задачей сетевого уровня.

Проблема осложняется тем, что самый короткий маршрут – не всегда лучший маршрут. Что действительно важно, так это величина задержки на данном маршруте, которая, в свою очередь, связана с объемом трафика и количеством сообщений, поставленных в очередь для передачи по различным линиям. Таким образом, задержка может изменяться с течением времени. Некоторые алгоритмы маршру-

тизации пытаются адаптироваться к изменяющимся нагрузкам, тогда как другие довольствуются принятием решений на основе долгосрочных средних.

В настоящее время наиболее широко используемым сетевым протоколом является **IP** (интернет-протокол) без установления соединения, который является частью набора интернет-протоколов. IP-пакет (IP-packet – технический термин для сообщения на сетевом уровне) может быть отправлен без какой-либо настройки. Каждый IP-пакет направляется по назначению независимо от всех других. Внутренний путь не выбирается и не запоминается.

Транспортные протоколы. Транспортный уровень формирует последнюю часть того, что можно назвать базовым стеком сетевых протоколов, в том смысле, что он реализует все те сервисы, которые не предоставляются на интерфейсе сетевого уровня, но которые разумно необходимы для создания сетевых приложений. Другими словами, транспортный уровень превращает базовую сеть во что-то, что разработчик приложений может использовать.

Пакеты могут быть потеряны на пути от отправителя к получателю. Некоторые приложения могут самостоятельно обрабатывать сообщения об ошибках, другие предпочитают надежное соединение. Работа транспортного уровня заключается в предоставлении этой услуги. Идея состоит в том, что уровень приложения должен быть в состоянии доставить сообщение на транспортный уровень, ожидая, что оно будет доставлено без потерь.

Получив сообщение от уровня приложения, транспортный уровень разбивает его на части, достаточно маленькие для передачи, присваивает каждому порядковый номер и затем отправляет их. В заголовке транспортного уровня указывается, какие пакеты были отправлены, какие были получены, сколько у получателя еще есть места для принятия сообщений, которые должны быть повторно переданы, и тому подобные темы.

Надежные транспортные соединения (которые по определению ориентированы на установление соединения) могут быть построены поверх ориентированных на установление соединения или сетевых услуг без установления соединения. В первом случае все пакеты будут доставлены в правильной последовательности (если они вообще будут получены), но во втором случае один пакет может выбрать другой маршрут и прибыть раньше, чем пакет, отправленный до него. Это зависит от программного обеспечения транспортного уровня, чтобы вернуть все обратно и сохранить иллюзию, что транспортное соединение подобно большой трубе – вы помещаете в него сообщения, и они выходят неповрежденными и в том же порядке, в котором вошли. Такое сквозное поведение связи является важным аспектом транспортного уровня.

Транспортный протокол интернета называется **TCP** (Transmission Control Protocol) и подробно описан в [Comer, 2013]. Комбинация TCP/IP теперь используется в качестве фактического стандарта для сетевого взаимодействия. Пакет интернет-протоколов также поддерживает транспортный протокол без установления соединения, называемый **протоколом универсальной дейтаграммы** (Universal Datagram Protocol, UDP), который по сути просто IP с некоторыми незначительными дополнениями. Пользовательские программы, которым не нужен протокол с установлением соединения, обычно используют UDP.

Регулярно предлагаются дополнительные транспортные протоколы. Например, для поддержки передачи данных в реальном времени был определен **транспортный протокол в реальном времени** (Real-time Transport Protocod, RTP). RTP является базовым протоколом в том смысле, что он определяет форматы пакетов для данных в реальном времени, не предоставляя фактических механизмов, гарантирующих доставку данных. Кроме того, он определяет протокол для мониторинга и управления передачей данных пакетов RTP [Schulzrinne et al., 2003]. **Протокол**

передачи управления потоком (Streaming Control Transmission Protocol, SCTP) был предложен в качестве альтернативы TCP [Stewart, 2007]. Основное различие между SCTP и TCP состоит в том, что SCTP группирует данные в сообщения, тогда как TCP просто перемещает байты между процессами. Это может упростить разработку приложений.

Протоколы более высокого уровня. Выше транспортного уровня OSI выделяет три дополнительных уровня. На практике используется только уровень приложения. Фактически в наборе протоколов интернета все, что находится выше транспортного уровня, сгруппировано. При рассмотрении систем промежуточного программного обеспечения мы увидим, что ни подход, ни OSI, ни интернет не являются действительно подходящими.

Сеансовый уровень, по сути, является расширенной версией транспортного уровня. Он обеспечивает управление диалогом, чтобы отслеживать, какая сторона в данный момент передает, и предоставляет средства синхронизации. Последние полезны, чтобы позволить пользователям вставлять контрольные точки в длительные передачи, так что в случае сбоя необходимо вернуться только к последней контрольной точке, а не полностью к началу. На практике лишь немногие приложения заинтересованы в уровне сеанса, и он редко поддерживается. Его даже нет в наборе интернет-протоколов. Однако в контексте разработки решений промежуточного программного обеспечения концепция сеанса и связанных с ним протоколов оказалась весьма актуальной, особенно при определении высокоуровневых протоколов связи.

В отличие от нижних уровней, которые связаны с надежной и эффективной передачей битов от отправителя к получателю, уровень представления связан со значением битов. Большинство сообщений состоят не из случайных битовых строк, а из более структурированной информации, такой как имена людей, адреса, суммы денег и т. д. На уровне представления можно определить записи, содержащие подобные поля, и затем заставить отправителя уведомить получателя о том, что сообщение содержит конкретную запись в определенном формате. Это облегчает машинам с различными внутренними представлениями взаимодействие друг с другом.

Уровень приложения OSI изначально предназначался для хранения набора стандартных сетевых приложений, таких как приложения для электронной почты, передачи файлов и эмуляции терминала. К настоящему времени он стал контейнером для всех приложений и протоколов, которые так или иначе не попадают в один из нижележащих уровней. С точки зрения эталонной модели OSI практически все распределенные системы являются просто приложениями.

В этой модели отсутствует четкое различие между приложениями, протоколами для конкретных приложений и протоколами общего назначения. Например, **протокол передачи файлов** через интернет (File Transfer Protocol, FTP) [Postel and Reynolds, 1985; Horowitz and Lunt, 1997] определяет протокол для передачи файлов между клиентом и сервером. Протокол не следует путать с программой ftp, которая является приложением конечного пользователя для передачи файлов и которая также (не полностью по совпадению) реализует интернет FTP.

Другим примером типичного протокола, специфичного для приложения, является **протокол передачи гипертекста** (HyperText Transfer Protocol, HTTP) [Fielding and Reschke, 2014], который предназначен для удаленного управления и обработки передачи веб-страниц. Протокол реализуется такими приложениями, как веб-браузеры и веб-серверы. Однако HTTP теперь также используется системами, которые не связаны с интернетом. Например, механизм вызова объектов Java может использовать HTTP для запроса вызова удаленных объектов, которые защищены межсетевым экраном (брандмауэром).

Существует также много протоколов общего назначения, которые полезны для многих приложений, но которые нельзя квалифицировать как транспортные протоколы. Во многих случаях такие протоколы попадают в категорию протоколов промежуточного программного обеспечения.

Протоколы промежуточного программного обеспечения

Промежуточное программное обеспечение – это приложение, которое логически находится (в основном) на уровне приложений OSI, но содержит много протоколов общего назначения, которые требуют своих собственных уровней, независимо от других, более специфических приложений. Давайте кратко рассмотрим некоторые примеры.

Система доменных имен (Domain Name System, DNS) [Liu and Albitz, 2006] – это распределенная служба, которая используется для поиска сетевого адреса, связанного с именем, таким как адрес так называемого **доменного имени** (domain name), например www.distributed-systems.net. С точки зрения эталонной модели OSI, DNS является приложением и, следовательно, логически размещается на уровне приложений. Однако совершенно очевидно, что DNS предлагает универсальный, независимый от приложений сервис и, возможно, является частью промежуточного программного обеспечения.

Другой пример. Существуют различные способы установить аутентификацию, то есть предоставить подтверждение заявленной идентичности. Протоколы аутентификации не тесно связаны с каким-либо конкретным приложением, но вместо этого могут быть интегрированы в систему промежуточного программного обеспечения в качестве общего сервиса. Аналогично протоколы авторизации, по которым аутентифицированным пользователям и процессам предоставляется доступ только к тем ресурсам, для которых они имеют авторизацию, имеют тенденцию иметь общий, независимый от приложения характер. Будучи помечены как приложения в эталонной модели OSI, они являются явными примерами из промежуточного программного обеспечения.

Протоколы распределенной фиксации устанавливают, что в группе процессов, возможно, распределенных по нескольким машинам, либо все процессы выполняют определенную операцию, либо эта операция вообще не выполняется. Это явление также называется **атомарностью** (atomicity) и широко применяется в транзакциях. Как выясняется, протоколы фиксации могут представлять интерфейс независимо от конкретных приложений, обеспечивая, таким образом, универсальный сервис транзакций. В подобной форме они обычно относятся к промежуточному программному обеспечению, а не к уровню приложений OSI.

В качестве последнего примера рассмотрим протокол распределенной блокировки, с помощью которого ресурс может быть защищен от одновременного доступа набора процессов, которые распределены по нескольким машинам. Нетрудно представить, что такие протоколы могут быть разработаны независимо от приложения и доступны через относительно простой, опять же независимый от приложения интерфейс. Как таковые они обычно принадлежат промежуточному программному обеспечению.

Эти примеры протоколов не связаны напрямую со связью, но есть также много протоколов связи промежуточного программного обеспечения. Например, при так называемом **удаленном вызове процедуры** (remote procedure call) процессу предлагается средство для *локального* вызова процедуры, которое эффективно реализуется на удаленной машине. Эта коммуникационная услуга относится к одному из самых старых типов сервисов промежуточного программного обеспечения и используется для обеспечения прозрачности доступа. Аналогичным образом существуют высокоуровневые службы связи для настройки и синхронизации потоков для передачи данных в реальном времени, например необходимых для мультимедийных приложений. В качестве последнего примера некоторые системы промежуточного программного обеспечения предлагают надежные многоадресные услуги, которые масштабируются до тысяч получателей, распределенных по глобальной сети.

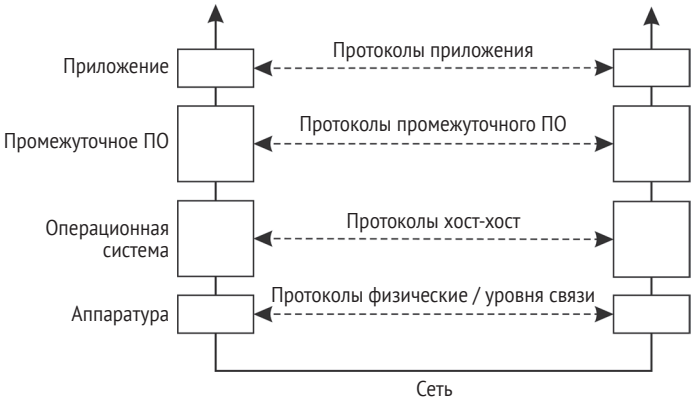


Рис. 4.3 ❖ Адаптированная эталонная модель для сетевого взаимодействия

Принятие этого подхода к многоуровневой структуре приводит к адаптированной и упрощенной эталонной модели для коммуникации, как показано на рис. 4.3. По сравнению с моделью OSI уровень сеансов и презентаций заменен одним уровнем промежуточного программного обеспечения, который содержит протоколы, не зависящие от приложений. Эти протоколы не относятся к нижним уровням, которые мы только что обсуждали. Сетевые и транспортные службы были сгруппированы в службы связи, обычно предлагаются операционной системой, которая, в свою очередь, управляет определенным оборудованием самого низкого уровня, используемым для установления связи.

Типы коммуникаций

В оставшейся части этой главы мы сосредоточимся на высокоуровневых услугах связи промежуточного программного обеспечения. Прежде чем сделать это, рассмотрим другие общие критерии для различения (промежуточного

программного обеспечения) коммуникаций. Чтобы понять различные альтернативы в коммуникациях, которые промежуточное программное обеспечение может предложить приложениям, мы будем рассматривать промежуточное программное обеспечение в качестве дополнительной услуги в вычислениях клиент-сервер, как показано на рис. 4.4. Рассмотрим, например, систему электронной почты. В принципе, ядро системы доставки почты можно рассматривать как службу связи промежуточного программного обеспечения. На каждом хосте работает пользовательский агент, позволяющий пользователям создавать, отправлять и получать электронную почту. Отправляющий пользовательский агент передает такую почту в систему доставки почты, ожидая, что он, в свою очередь, в конечном итоге доставит почту предполагаемому получателю. Аналогично пользовательский агент на стороне получателя подключается к системе доставки почты, чтобы увидеть, поступило ли какое-либо письмо. Если это так, сообщения передаются пользовательскому агенту, чтобы пользователь мог их отображать и читать.

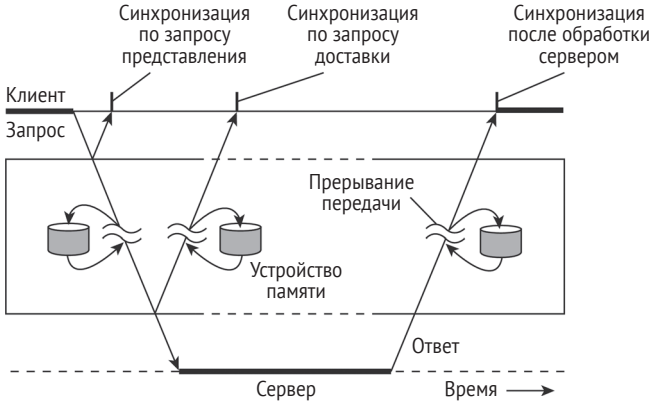


Рис. 4.4 ❖ Взгляд на промежуточное программное обеспечение в качестве промежуточного (распределенного) сервиса при обмене данными на уровне приложений

Система электронной почты является типичным примером, в котором общение является постоянным. При **постоянной связи** (persistent communication) сообщение, которое было передано для передачи, сохраняется связующим программным обеспечением до тех пор, пока оно доставляется получателю. В этом случае промежуточное программное обеспечение будет хранить сообщение в одном или нескольких хранилищах, показанных на рис. 4.4. Как следствие отправляющему приложению нет необходимости продолжать выполнение после отправки сообщения. Аналогично приложение-получатель не должно выполняться при отправке сообщения.

Напротив, при **перехо́дной** (вре́менной) **связи** (transient communication) сообщение сохраняется системой связи только до тех пор, пока выполняется отправляющее и получающее приложения. Точнее, как видно на рис. 4.4, если промежуточное ПО не может доставить сообщение из-за прерывания передачи или из-за того, что получатель в данный момент не активен, оно

просто будет сброшено. Как правило, все услуги связи транспортного уровня предлагают только переходную связь. В этом случае система связи состоит из традиционных маршрутизаторов с промежуточным хранением. Если маршрутизатор не может доставить сообщение следующему или целевому хосту, он просто сбросит сообщение.

Помимо того что связь постоянна или временна, связь может быть асинхронной или синхронной. Характерной особенностью асинхронной связи является то, что отправитель продолжает работу сразу после того, как он отправил свое сообщение для передачи. Это означает, что сообщение (временно) сохраняется промежуточным программным обеспечением после отправки. При синхронной связи отправитель блокируется до тех пор, пока известно, что его запрос не принят. По сути, есть три момента, где может иметь место синхронизация. Во-первых, отправитель может быть заблокирован до тех пор, пока промежуточное программное обеспечение не сообщит, что оно примет на себя передачу запроса. Во-вторых, отправитель может синхронизироваться до тех пор, пока его запрос не будет доставлен предполагаемому получателю. В-третьих, синхронизация может иметь место, позволяя отправителю подождать, пока его запрос не будет полностью обработан, то есть до момента, когда получатель вернет ответ.

На практике встречаются различные комбинации постоянства и синхронизации. Популярными являются постоянные в сочетании с синхронизацией при отправке запроса, что является общей схемой для многих систем очередей сообщений, которые мы обсудим позже в этой главе. Аналогичным образом широко используется переходная связь с синхронизацией после полной обработки запроса. Эта схема соответствует вызовам удаленных процедур, которые мы рассмотрим далее.

4.2. Удаленный вызов процедуры

Многие распределенные системы основаны на явном обмене сообщениями между процессами. Однако операции отправки и получения не скрывают связь вообще, что важно для обеспечения прозрачности доступа в распределенных системах. Эта проблема известна давно, но с ней мало что было сделано до тех пор, пока исследователи в 1980-х годах [Birrell and Nelson, 1984] не представили способ обработки сообщений. Хотя идея очень проста (если кто-то так подумал), последствия часто неуловимы. В этом разделе мы рассмотрим концепцию, ее реализацию, ее сильные и слабые стороны.

В двух словах: предложение состояло в том, чтобы позволить программам вызывать процедуры, расположенные на других машинах. Когда процесс на машине A вызывает процедуру на машине B, вызывающий процесс на A приостанавливается, и выполнение вызываемой процедуры происходит на B. Информация может передаваться от вызывающей стороны к вызываемой стороне в параметрах и может возвращаться в результате процедуры. Передача сообщений вообще не видна программисту. Этот метод известен как **вызов удаленной процедуры** (Remote Procedure Call, RPC).

Хотя основная идея звучит просто и элегантно, существуют тонкости.

Начнем с того, что вызывающие и вызываемые процедуры выполняются на разных машинах, они выполняются в разных адресных пространствах, что вызывает сложности. Параметры и результаты также должны быть переданы, что может быть сложно, особенно если машины не идентичны. Наконец, один или оба компьютера могут дать сбой, и каждый из возможных сбоев вызывает различные проблемы. Тем не менее с большинством из них можно справиться, и RPC является широко используемым методом, который лежит в основе многих распределенных систем.

Основная операция RPC

Идея RPC состоит в том, чтобы сделать вызов удаленной процедуры максимально похожим на локальный. Другими словами, мы хотим, чтобы RPC был прозрачным – вызывающая процедура не должна знать, что вызываемая процедура выполняется на другом компьютере, или наоборот. Предположим, что программа имеет доступ к базе данных, которая позволяет ей добавлять данные в сохраненный список, после чего она возвращает ссылку на измененный список. Операция становится доступной для программы с помощью стандартного добавления:

```
newlist = append(data, dbList)
```

В традиционной (однопроцессорной) системе `append` извлекается из библиотеки компоновщиком и вставляется в объектную программу. В принципе, это может быть короткая процедура, которая может быть реализована с помощью нескольких файловых операций для доступа к базе данных.

Несмотря на то что в конце концов `append` выполняет только несколько основных файловых операций, он вызывается обычным способом, помещая свои параметры в стек. Программист не знает деталей реализации `append`, и это, конечно, так, как и должно быть.

Примечание 4.2 (дополнительная информация: вызовы обычной процедуры)

Пониманию того, как работает RPC, и некоторых из его подводных камней может помочь знание о работе вызова обычной процедуры (т. е. одной машины). Рассмотрим следующую операцию:

```
newlist = append(data, dbList);
```

Мы предполагаем, что цель этого вызова состоит в том, чтобы взять *глобально* определенный объект списка, называемый здесь `dbList`, и добавить к нему простой элемент данных, представленный переменной `data`. Важным наблюдением является то, что в различных языках программирования, таких как C, `dbList` реализован как ссылка на объект списка (то есть указатель), тогда как данные могут быть представлены непосредственно его значением (что, мы предполагаем, имеет место здесь). При вызове `append` оба представления данных и `dbList` помещаются в стек, делая эти представления доступными для реализации `append`. Для данных это означает, что переменная следует **за политикой копирования по значению** (copy-by-value policy), политика для `dblist` является **копированием по ссылке** (copy-by-reference policy), что происходит до и во время вызова, показано на рис. 4.5.

Стоит отметить следующее. Во-первых, значения параметра, такого как `data`, являются просто инициализированной локальной переменной. Вызываемая процедура может изменить его, но такие изменения не влияют на исходное значение на вызывающей стороне.

Когда такой параметр, как `dbList`, на самом деле является указателем на переменную, а не значением переменной, происходит что-то еще. В стек помещается адрес объекта списка, который хранится в основной памяти. Когда значение данных добавляется в список, вызов `append` действительно изменяет объект списка. Разница между вызовом по значению и вызовом по ссылке довольно важна для RPC.

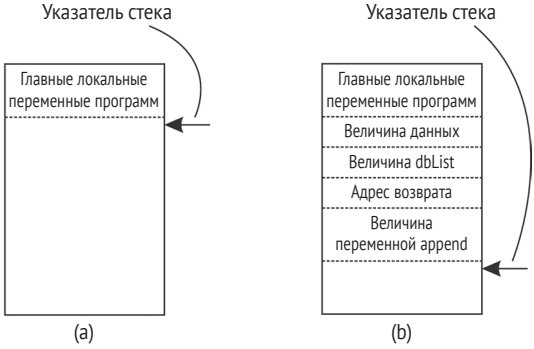


Рис. 4.5 ❖ Передача параметров в вызове локальной процедуры:
a) стек перед вызовом присоединения;
b) стек в то время, как вызываемая процедура активна

Существует еще один механизм передачи параметров, хотя он не используется в большинстве языков программирования. Он называется **вызов по копии/возврат** (call-by-copy/restore) и состоит в том, что переменная копируется в стек вызывающей стороной, как в вызове по значению, а затем копируется обратно после вызова, перезаписывая первоначальное значение вызывающей стороны. В большинстве случаев это обеспечивает тот же эффект, что и вызов по ссылке, но в некоторых ситуациях, например когда один и тот же параметр присутствует несколько раз в списке параметров, семантика различна.

Решение о том, какой механизм передачи параметров использовать, обычно принимается разработчиками языка и является фиксированным свойством языка. Иногда это зависит от типа передаваемых данных. Например, в C целые числа и другие скалярные типы всегда передаются по значению, тогда как массивы всегда передаются по ссылке. Некоторые компиляторы Ada используют копирование/восстановление для параметров `inout`, но другие используют вызов по ссылке. Определение языка допускает любой выбор, что делает семантику немного размытой. В языке Python все переменные передаются по ссылке, но некоторые фактически копируются в локальные переменные, имитируя таким образом поведение копирования по значению.

Вызов RPC достигает своей прозрачности аналогичным образом. Когда на самом деле `append` является удаленной процедурой, вызывающему клиенту предлагается другая версия `append`, называемая **клиентской заглушкой** (client stub). Как и исходная, она тоже вызывается с использованием обычной

вызывающей последовательности. Однако, в отличие от исходной, она не выполняет операцию добавления. Вместо этого оно упаковывает параметры в сообщение и запрашивает это сообщение для отправки на сервер, как показано на рис. 4.6.

После вызова для отправки клиентская заглушка вызывает `receive`, блокируясь до тех пор, пока ответ не вернется.

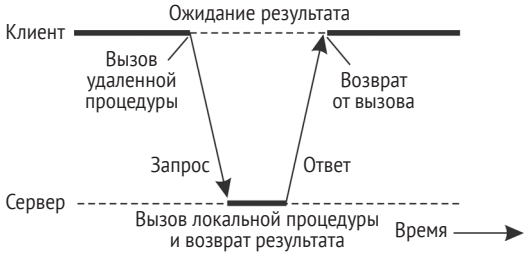


Рис. 4.6 ❖ Принцип RPC между клиентской и серверной программами

Когда сообщение поступает на сервер, операционная система сервера передает его на **серверную заглушку** (server stub). Серверная заглушка – это эквивалент клиентской заглушки на стороне сервера: часть кода, который преобразует запросы, поступающие по сети, в локальные вызовы процедур. Обычно серверная заглушка вызывает `receive` и будет заблокирована в ожидании входящих сообщений. Серверная заглушка распаковывает параметры из сообщения и затем вызывает серверную процедуру обычным способом. С точки зрения сервера это выглядит так, как будто он вызывается непосредственно клиентом – параметры и адрес возврата находятся в стеке, к которому они принадлежат, и нет ничего необычного. Сервер выполняет свою работу, а затем возвращает результат вызывающей стороне (в данном случае серверной заглушке) обычным способом.

Когда серверная заглушка получает управление после завершения вызова снова, она упаковывает результат в сообщение и вызывает `send`, чтобы вернуть его клиенту. После этого серверная заглушка обычно снова вызывает `receive`, чтобы ждать следующего входящего запроса.

Когда сообщение о результате прибывает на клиентский компьютер, операционная система передает его через операцию `receive`, которая была вызвана ранее, клиентской заглушке, и клиентский процесс будет разблокирован. Клиентская заглушка проверяет сообщение, распаковывает результат, копирует его вызывающей стороне и возвращает обычным способом. Когда вызывающая сторона получает контроль после вызова `append`, все, что она знает, – это то, что некоторые данные добавлены в список. Она понятия не имеет, что работа была выполнена удаленно на другой машине.

Это блаженная неосведомленность клиента – красота всей схемы. К удаленным службам обращаются путем выполнения обычных (то есть локальных) вызовов процедур, а не путем вызова `send` и `receive`. Все детали передачи сообщений скрыты в двух процедурах библиотеки, так же как детали

фактического выполнения системных вызовов скрыты в традиционных библиотеках.

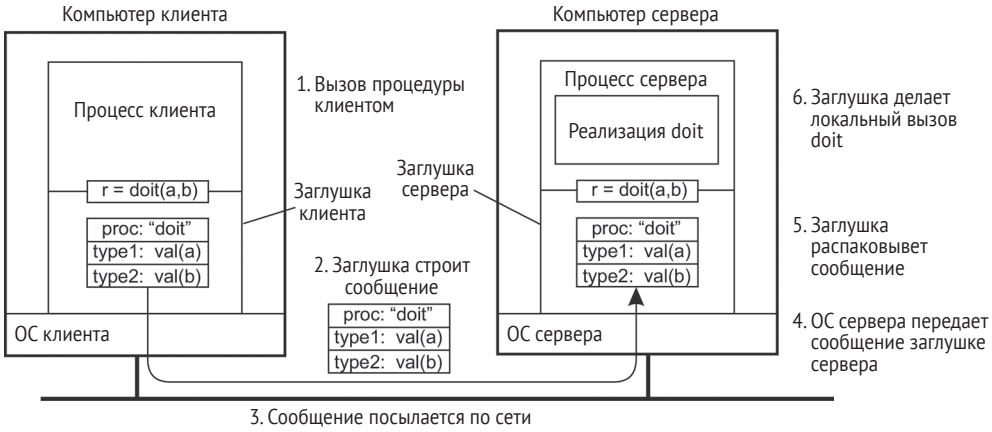


Рис. 4.7 ❖ Шаги, связанные с вызовом удаленной процедуры `doit(a, b)`. Путь возврата для результата не показан

Подводя итог. Следующие шаги выполняются при удаленном вызове процедуры:

1) клиентская процедура вызывает клиентскую заглушку обычным способом;
2) клиентская заглушка создает сообщение и вызывает локальную операционную систему;
3) клиентская операционная система (ОС) отправляет сообщение на удаленную ОС;
4) удаленная ОС выдает сообщение на серверную заглушку;
5) серверная заглушка распаковывает параметр(ы) и вызывает сервер;
6) сервер выполняет работу и возвращает результат в заглушку;
7) серверная заглушка упаковывает результат в сообщение и вызывает локальную ОС;
8) ОС сервера отправляет сообщение в ОС клиента;
9) ОС клиента передает сообщение клиентской заглушке;
10) заглушка распаковывает результат и возвращает его клиенту.

Первые шаги показаны на рис. 4.7 для абстрактной двухпараметрической процедуры `doit(a, b)`, где мы предполагаем, что параметр a имеет тип `type1`, а параметр `b` имеет тип `type2`. Общий эффект всех этих шагов заключается в преобразовании локального вызова клиентской процедуры в клиентской заглушке в локальный вызов серверной процедуры, при которой клиент и сервер не знают о промежуточных шагах или о существовании сети.

Примечание 4.3 (дополнительная информация: пример на языке Python)

Чтобы конкретизировать ситуацию, давайте рассмотрим, как можно реализовать удаленный вызов процедуры для операции добавления, рассмотренной ранее. Взгляните на код Python, показанный на рис. 4.8 (из которого мы опускаем несущественные фрагменты кода).

Класс class DBList просто представляет собой объект списка, имитирующий то, что можно было бы ожидать в версии, найденной в среде базы данных. Клиентская заглушка, представленная классом class Client, состоит из реализации добавления (append). При вызове с параметрами data и dbList происходит следующее. Вызов преобразуется в кортеж (APPEND, data, dbList), содержащий всю информацию, которая понадобится серверу для его работы. Клиентская заглушка затем отправляет запрос на сервер и впоследствии ожидает ответа. Когда приходит ответ, он завершается передачей результата программе, которая первоначально назвала заглушку.

```
 1 import channel, pickle
 2
 3 class DBList:
 4   def append(self, data):
 5     self.value = self.value + [data]
 6     return self
 7
 8 class Client:
 9   def append(self, data, dbList):
10     msglst = (APPEND, data, dbList)          # загрузить сообщения
11     self.chan.sendTo(self.server, msglst)    # отправить msg серверу
12     msgrcv = self.chan.recvFrom(self.server) # ожидать ответа
13     return msgrcv[1]                         # отправить его вызывающему
14
15 class Server:
16   def append(self, data, dbList):
17     return dbList.append(data)
18
19   def run(self):
20     while True:
21       msgreq = self.chan.recvFromAny() # ожидать других запросов
22       client = msgreq[0]               # посмотреть, кто является запрашивающим
23       msgrpc = msgreq[1]               # запросить вызов и параметры
24       if APPEND == msgrpc[0]:          # проверить то, что запрашивается
25         result = self.append(msgrpc[1], msgrpc[2]) # сделать локальный вызов
26         self.chan.sendTo([client],result)          # ответ
```

Рис. 4.8 ❖ Простой пример RPC для операции append

На стороне сервера мы видим, что в серверной заглушке сервер ждет любого входящего сообщения и проверяет, какую операцию ему необходимо вызвать. Предполагая, что он получил запрос вызова append, он просто выполняет локальный вызов своей реализации append с соответствующим параметром.

Передача параметров

Функция клиентской заглушки состоит в том, чтобы взять ее параметры, упаковать их в сообщение и отправить их на серверную заглушку. Это не так просто, как кажется на первый взгляд.

Упаковка параметров в сообщение называется **маршалингом параметров** (parameter marshaling). Возвращаясь к нашей операции добавления `append`, мы должны убедиться, что ее два параметра (`data` и `dbList`) передаются по сети и корректно интерпретируются сервером. Здесь нужно понять, что, в конце концов, сервер увидит только серию байтов, которые составляют исходное сообщение, отправленное клиентом. Однако дополнительная информация о том, что означают эти байты, обычно сообщением не предоставляется, не говоря уже о том, что мы снова столкнемся с той же проблемой: как метаинформация должна распознаваться сервером как таковая?

Помимо этой проблемы интерпретации, нам также необходимо рассмотреть случай, когда размещение байтов в памяти может отличаться в зависимости от архитектуры машины. В частности, нам необходимо учитывать тот факт, что некоторые машины, такие как Intel Pentium, нумеруют свои байты справа налево, тогда как многие другие, более старые процессоры ARM нумеруют их по-другому (ARM теперь поддерживает оба). Формат Intel называется порядок байтов **little endian**, а (более старый) формат ARM называется порядок байтов **big endian**. Порядок байтов также важен для работы в сети: здесь мы также можем столкнуться с тем, что машины используют разный порядок при передаче (и, следовательно, получении) битов и байтов. Тем не менее big endian – это то, что обычно используется для передачи байтов по сети.

Решение данной проблемы состоит в том, чтобы преобразовать данные, которые должны быть отправлены, в независимый от машины и сети формат, а также убедиться, что обе связывающиеся стороны ожидают передачи одного и того же *типа данных сообщения*.

Последнее обычно может быть решено на уровне языков программирования. Первое достигается с помощью машинно *зависимых* подпрограмм, которые преобразуют данные как в машинно независимые форматы, так и из них.

Маршалинг и демаршалинг – это преобразование в нейтральные форматы, которое является неотъемлемой частью удаленных вызовов процедур.

Теперь мы подошли к трудной проблеме: как передаются указатели или вообще ссылки? Ответ таков: только с величайшими трудностями, если вообще возможно. Указатель имеет смысл лишь в адресном пространстве процесса, в котором он используется. Возвращаясь к нашему примеру добавления, мы заявили, что второй параметр, dbList, реализован посредством ссылки на список, хранящийся в базе данных. Если эта ссылка является просто указателем на локальную структуру данных где-то в основной памяти вызывающего, мы не можем просто передать ее на сервер. Переданное значение указателя, скорее всего, будет ссылаться на что-то совершенно другое.

Примечание 4.4 (дополнительная информация: вновь пример в Python)

Нетрудно увидеть, что решение для удаленного вызова процедур, как показано на рис. 4.8, не будет работать вообще. Только если клиент и сервер работают на машинах, которые подчиняются одинаковым правилам упорядочения байтов *и* имеют одинаковые машинные представления для структур данных, будут обеспечены правильная интерпретации и обмен сообщениями. Надежное решение демонстрируется на рис. 4.9 (где мы снова опустили код для краткости).

В этом примере мы используем библиотеку Python pickle для маршалинга и немаршалинга структур данных. Обратите внимание, что код почти не меняется по сравнению с тем, что мы показали на рис. 4.8. Единственные изменения происходят непосредственно перед отправкой и после получения сообщения. Также обратите внимание, что и клиент, и сервер запрограммированы на работу с одинаковыми структурами данных, как обсуждалось ранее.

```
 1 import channel, pickle
 2
 3 class Client:
 4   def append(self, data, dbList):
 5     msglst = (APPEND, data, dbList)          # полезная нагрузка сообщения
 6     msgsnd = pickle.dumps(msglst)            # вызов обертки
 7     self.chan.sendTo(self.server, msgsnd)    # отправить запрос на #сервер
 8     msgrcv = self.chan.recvFrom(self.server) # ждать ответа
 9     retval = pickle.loads(msgrcv[1])         # развернуть возвращаемое значение
10     return retval                            # передать его вызывающей стороне

12 class Server:
13   def run(self):
14     while True:
15     msgreq = self.chan.recvFromAny() # ожидание любого запроса
16     client = msgreq[0]               # посмотрите, кто является вызывающим абонентом
17     msgrpc = pickle.loads(msgreq[1]) # развернуть вызов
18     if APPEND == msgrpc[0]:          # проверить, что запрашивается
19       result = self.append(msgrpc[1], msgrpc[2]) # сделать локальный вызов
20       msgres = pickle.dumps(result)              # обернуть результат
21       self.chan.sendTo([client],msgres)          # отправить ответ
```

Рис. 4.9 ❖ Простой пример RPC для операции добавления, но теперь с правильным маршалингом

Одно из решений – просто запретить указатели и ссылочные параметры в целом. Однако они настолько важны, что это решение крайне нежелательно. На самом деле это часто и не нужно. Во-первых, ссылочные параметры часто используются с типами данных фиксированного размера, такими как статические массивы, или с динамическими типами данных, для которых легко вычислить их размер во время выполнения, таких как строки или динамические массивы. В подобных случаях мы можем просто скопировать всю структуру данных, на которую ссылается параметр, эффективно заменив механизм копирования по ссылке на функцию копирования по значение/восстановление. Хотя это семантически не всегда идентично, часто это до-

статочно хорошо. Очевидная оптимизация состоит в том, что, когда клиентская заглушка знает, что упомянутые данные будут только прочитаны, нет необходимости копировать их обратно, когда вызов завершен. Копирование по значению, таким образом, приемлемо.

Часто могут поддерживаться и более сложные типы данных, и, конечно, если язык программирования поддерживает эти типы данных. Например, такие языки, как Python или Java, поддерживают пользовательские классы, что позволяет языковой системе полностью автоматизировать маршалинг и демаршалинг этих типов данных. Обратите внимание, что, как только мы имеем дело с очень большими, вложенными, или иными сложными динамическими структурами данных, автоматический (не)маршалинг может быть недоступным или даже нежелательным.

Проблема с указателями и ссылками, как обсуждалось ранее, заключается в том, что они имеют смысл только локально: они относятся к областям памяти, которые имеют значение только для вызывающего процесса. Проблемы могут быть устранены с помощью глобальных ссылок: ссылок, которые имеют значение для вызывающего и вызываемого процессов. Например, если клиент и сервер имеют доступ к одной и той же файловой системе, может помочь передача дескриптора файла вместо указателя. Есть одно важное наблюдение: оба процесса должны точно знать, что делать, когда передается глобальная ссылка. Другими словами, если мы рассмотрим глобальную ссылку, имеющую связанный тип данных, вызывающий и вызываемый процессы должны иметь абсолютно одинаковую картину операций, которые могут быть выполнены. Более того, оба процесса должны иметь соглашение о том, что делать, когда передается дескриптор файла. Опять же, это, как правило, проблемы, которые могут быть решены путем надлежащей поддержки языка программирования.

Примечание 4.5 (дополнительно: передача параметров в объектно-ориентированных системах)

Системы на основе объектов часто используют глобальные ссылки. Рассмотрим ситуацию, когда все объекты в системе могут быть доступны с удаленных компьютеров. В этом случае мы можем последовательно использовать ссылки на объекты в качестве параметров в вызовах методов. Ссылки передаются по значению и, таким образом, копируются с одного компьютера на другой. Когда процессу присваивается ссылка на объект в результате вызова метода, он может просто привязаться к объекту, на который ссылаются позже, когда это необходимо (см. также раздел 2.1).

К сожалению, использование только распределенных объектов может быть крайне неэффективным, особенно когда объекты маленькие, такие как целые числа или, что еще хуже, логические значения. Каждый вызов клиентом, который не расположен на том же сервере, что и объект, генерирует запрос между разными адресными пространствами или, что еще хуже, между разными машинами. Поэтому ссылки на удаленные объекты и ссылки на локальные объекты часто обрабатываются поразному.

При вызове метода со ссылкой на объект в качестве параметра эта ссылка копируется и передается в качестве параметра значения только тогда, когда он ссылается на удаленный объект. В этом случае объект буквально передается по ссылке. Однако когда ссылка ссылается на локальный объект, то есть объект в том же адресном

пространстве, что и клиент, указанный объект копируется целиком и передается вместе с вызовом. Другими словами, объект передается по значению.

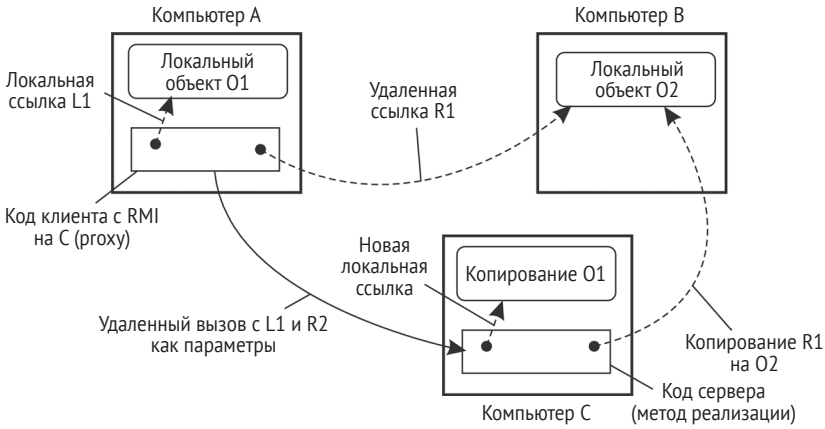


Рис. 4.10 ❖ Передача объекта по ссылке или по значению

Эти две ситуации проиллюстрированы на рис. 4.10, где показана клиентская программа, работающая на компьютере A, и серверная программа на компьютере C. У клиента есть ссылка на локальный объект O1, который он использует в качестве параметра при вызове серверной программы на машине C. Кроме того, она содержит ссылку на удаленный объект O2, находящийся на машине B, который также используется в качестве параметра. При вызове сервера копия O1 передается на сервер на компьютере C только вместе с копией ссылки на O2.

Обратите внимание, что независимо от того, имеем ли мы дело со ссылкой на локальный объект или со ссылкой на удаленный объект, это может быть очень прозрачно, как, например, в Java. В Java это различие видно только потому, что локальные объекты по существу имеют другой тип данных, чем удаленные объекты. В противном случае оба типа ссылок обрабатываются практически одинаково (см. также [Wollrath et al., 1996]). С другой стороны, при использовании традиционных языков программирования, таких как C, ссылка на локальный объект может быть такой же простой, как указатель, который никогда не может быть использован для ссылки на удаленный объект.

Побочным эффектом вызова метода со ссылкой на объект в качестве параметра является то, что мы можем копировать объект. Очевидно, что скрывать этот аспект недопустимо, поэтому мы вынуждены проводить явное различие между локальными и распределенными объектами. Понятно, что это различие не только нарушает прозрачность распространения, но и усложняет написание распределенных приложений.

Теперь мы также можем легко объяснить, как глобальные ссылки могут быть реализованы при использовании переносимых интерпретируемых языков, таких как Python или Java: используйте клиентскую заглушку в качестве ссылки. Ключевое наблюдение заключается в том, что клиентская заглушка часто является просто еще одной структурой данных, которая компилируется в (переносимый) байт-код. Этот скомпилированный код может фактически передаваться по сети и выполняться на стороне получателя. Другими словами, больше нет необходимости в явном связывании; простого копирования клиентской заглушки получателю достаточно, чтобы последний мог вызвать связанный объект на стороне сервера.

Поддержка приложений на основе RPC

Из того, что мы объяснили до сих пор, ясно, что скрытие удаленного вызова процедуры требует, чтобы вызывающая и вызываемая стороны согласовывали формат сообщений, которыми они обмениваются, и чтобы они выполняли те же шаги, когда речь идет, например, о передаче сложных структур данных. Другими словами, обе стороны в RPC должны следовать одному и тому же протоколу, иначе RPC не будет работать правильно. Существует как минимум два способа поддержки разработки приложений на основе RPC. Во-первых, разработчик должен указать, что именно нужно вызывать удаленно, из чего могут быть созданы полные заглушки на стороне клиента и на стороне сервера. Второй подход заключается во внедрении удаленного вызова процедур как части среды языка программирования.

Генерация заглушки

Рассмотрим функцию someFunction на рис. 4.11a. Она имеет три параметра: символ, число с плавающей точкой и массив из пяти целых чисел. Предполагая, что слово составляет четыре байта, протокол RPC может предписывать, чтобы мы передавали символ в крайнем правом байте слова (оставляя следующие три байта пустыми), слово float как целое слово и массив как группу слов, равную длине массива, которому предшествует слово, дающее ему длину, как показано на рис. 4.11b.

Таким образом, учитывая эти правила, клиентская заглушка для функции `someFunction` знает, что она должна использовать формат, показанный на рис. 4.11b, а серверная заглушка знает, что входящие сообщения для `someFunction` будут иметь формат, показанный на рис. 4.11b.

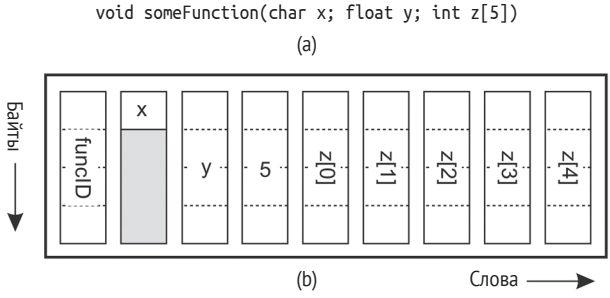


Рис. 4.11: a) Функция; b) соответствующее сообщение и порядок, в котором байты и слова отправляются через сеть

Определение формата сообщения является одним из аспектов протокола RPC, но этого недостаточно. Нам также нужно, чтобы клиент и сервер согласовали представление простых структур данных, таких как целые числа, символы, логические значения и т. д. Например, протокол мог бы предписывать, чтобы целые числа были представлены в виде дополнения к двум, символы

в виде 16-битных чисел Юникода и floats в стандартном формате IEEE #754, где все хранится в порядке байтов. С помощью этой дополнительной информации сообщения могут быть однозначно интерпретированы.

Теперь, когда правила кодирования закреплены до последнего бита, остается только сделать так, чтобы вызывающая сторона и вызываемый абонент согласовали фактический обмен сообщениями. Например, может быть решено использовать транспортную службу с установлением соединения, такую как TCP/IP. Альтернатива состоит в том, чтобы использовать ненадежную службу дейтаграмм и позволить клиенту и серверу реализовать схему контроля ошибок как часть протокола RPC. На практике существует несколько вариантов, и разработчик должен указать предпочтительный базовый коммуникационный сервис.

Как только протокол RPC будет полностью определен, клиентская и серверная заглушки должны быть реализованы. К счастью, заглушки для одного и того же протокола, но разных процедур, как правило, отличаются только интерфейсом приложений. Интерфейс состоит из набора процедур, которые могут быть вызваны клиентом и реализованы сервером. Интерфейс обычно доступен на том же языке программирования, на котором написан клиент или сервер (хотя это, строго говоря, не обязательно). Чтобы упростить задачу, интерфейсы часто определяются с помощью **языка определения интерфейса** (Interface Definition Language, IDL). Интерфейс, указанный в такой IDL, затем впоследствии компилируется в клиентскую заглушку и серверную заглушку вместе с соответствующими интерфейсами времени компиляции или выполнения.

Практика показывает, что использование языка определения интерфейса значительно упрощает клиент-серверные приложения на основе RPC. Поскольку легко генерировать клиентские и серверные заглушки, все системы промежуточного программного обеспечения на базе RPC предлагают IDL для поддержки при разработке приложений. В некоторых случаях использование IDL даже обязательно.

Языковая поддержка

Подход, описанный до сих пор, в значительной степени не зависел от конкретного языка программирования. В качестве альтернативы мы можем также встроить удаленный вызов процедуры в сам язык. Основное преимущество заключается в том, что разработка приложений часто становится намного проще. Кроме того, достижение высокой степени прозрачности доступа часто проще, поскольку многие проблемы, связанные с передачей параметров, можно вообще обойти.

Хорошо известным примером, в котором удаленный вызов процедуры полностью встроен, является язык Java, где RPC называется **удаленным вызовом метода** (remote method invocation, RMI) По сути, клиент, выполняемый своей собственной (Java) виртуальной машиной, может вызывать метод объекта, управляемого другой виртуальной машиной. Просто читая исходный код приложения, может быть трудно или даже невозможно увидеть, является ли вызов метода локальным или удаленным объектом.

Примечание 4.6 (дополнительная информация: RPC на основе языка в Python)

Давайте посмотрим на примере, как удаленный вызов процедуры может быть интегрирован в язык. Мы использовали язык Python для большинства наших примеров и продолжим это делать и сейчас. На рис. 4.12a мы показываем простой сервер для нашей структуры данных `DBList`. В этом случае у него есть две открытые операции: `visible_append` для добавления элементов и `visible_value` для отображения того, что в данный момент находится в списке. Мы используем пакет Python RPyC для встраивания RPC.

Клиент показан на рис. 4.12b. Когда будет установлено соединение с сервером, будет создан новый экземпляр `DBList`, и клиент сможет немедленно добавить значения в список. Выставленные операции можно вызывать без лишних слов.

```
1 import rpyc
2 from rpyc.utils.server import ForkingServer
3
4 class DBList(rpyc.Service):
5   value = []
6
7   def exposed_append(self, data):
8     self.value = self.value + [data]
9     return self.value
10
11  def exposed_value(self):
12    return self.value
13
14 if __name__ == "__main__":
15   server = ForkingServer(DBList, port = 12345)
16   server.start()
```

Рис. 4.12 ❖ a) Встраивание RPC в язык: сервер

```
1 import rpyc
2
3 class Client :
4   conn = rpyc.connect (SERVER, PORT) # Подключиться к серверу
5   conn.root.exposed_append (2)       # Вызвать открытую операцию
6   conn.root.exposed_append (4)       # и добавить два элемента
7   print conn.root.exposed_value ()   # Распечатать результат
```

Рис. 4.12 ❖ b) Встраивание RPC в язык: клиент

Этот пример иллюстрирует также тонкую проблему: очевидно, экземпляр DBList, созданный для клиента, является новым и уникальным. Другими словами, как только клиент разорвет соединение с сервером, список будет утерян. Это, так сказать, временный объект, и необходимо принять особые меры, чтобы сделать его постоянным объектом.

Вариации RPC

Как и в случае обычных вызовов процедуры, когда клиент вызывает удаленную процедуру, он блокируется до тех пор, пока не будет получен ответ. Такое строгое поведение «запрос-ответ» не требуется, если нет результата для возврата, или может препятствовать эффективности, когда необходимо выполнить несколько RPC. Далее мы рассмотрим два варианта схемы RPC, которые обсуждали до сих пор.

Асинхронный RPC

Для поддержки ситуаций, в которых просто нет результата для возврата клиенту, системы RPC могут предоставлять средства для так называемых **асинхронных RPC** (asynchronous RPC). При использовании асинхронных RPC сервер в принципе немедленно отправляет ответ обратно к клиенту в момент получения запроса RPC, после чего он локально вызывает запрошенную процедуру. Ответ действует как подтверждение клиента, что сервер собирается обработать RPC. Клиент продолжит работу без дальнейшей блокировки, как только получит подтверждение от сервера. На рис. 4.13b показано, как клиент и сервер взаимодействуют в случае асинхронных RPC. Для сравнения на рис. 4.13a показано нормальное поведение запрос-ответ.

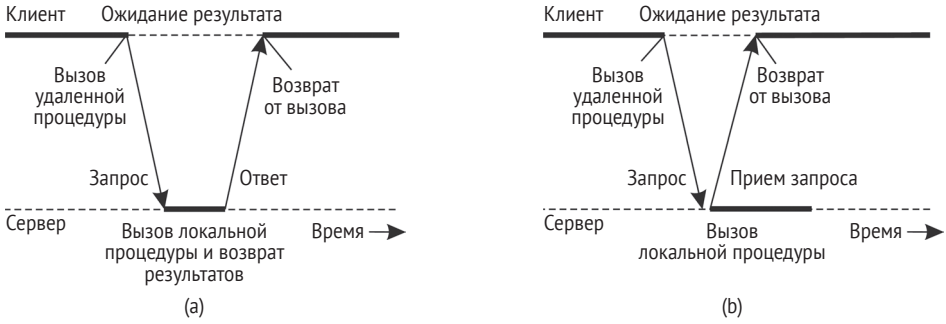


Рис. 4.13 ❖ Взаимодействие:
a) между клиентом и сервером в традиционном RPC;
b) с использованием асинхронного RPC

Асинхронные RPC также могут быть полезны, когда ответ будет возвращен, но клиент не готов ждать его и ничего не делать в это время. Типичный случай, когда клиенту необходимо связаться с несколькими серверами независимо. В этом случае он может отправлять запросы вызова один за другим, эффективно устанавливая, что серверы работают более или менее параллельно. После того как все запросы на вызов были отправлены, клиент может начать ждать, пока будут возвращены различные результаты. В та-

ких случаях имеет смысл организовать связь между клиентом и сервером через асинхронный RPC в сочетании с обратным вызовом, как показано на рис. 4.14. В этой схеме, также называемой **отложенным синхронным RPC** (deferred synchronous, RPC), клиент сначала вызывает сервер, ожидает принятия и продолжает работу. Когда результаты становятся доступными, сервер отправляет ответное сообщение, которое приводит к обратному вызову на стороне клиента. Обратный вызов – это определяемая пользователем функция, которая вызывается, когда происходит специальное событие, такое как входящее сообщение. Простая реализация – создать отдельный поток и позволить ему блокировать возникновение события, пока основной процесс продолжается. Когда происходит событие, поток будет разблокирован и вызовет функцию.

Следует отметить, что существуют варианты асинхронных RPC, в которых клиент продолжает выполнение сразу после отправки запроса на сервер. Другими словами, клиент не ожидает подтверждения приема запроса сервером. Мы называем такие RPC **односторонними** (one-way RPC). Проблема с этим подходом состоит в том, что когда надежность не гарантируется, клиент не может точно знать, будет ли обработан его запрос.

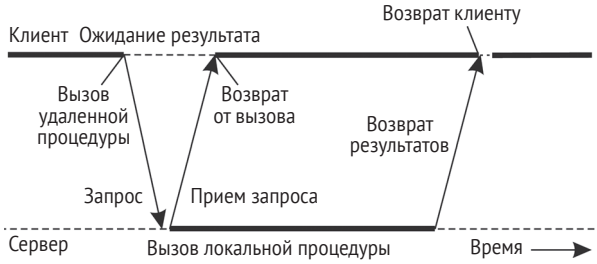


Рис. 4.14 ❖ Клиент и сервер, взаимодействующие через асинхронные RPC

Мы вернемся к этим вопросам в главе 8. Аналогично в случае отложенного синхронного RPC клиент может опросить сервер, чтобы узнать, доступны ли результаты, вместо того чтобы позволить серверу перезванивать клиенту.

Многоадресный RPC

Асинхронные и отложенные синхронные RPC обеспечивают еще одну альтернативу вызовам удаленных процедур, а именно выполнение нескольких RPC одновременно. Принимая односторонние RPC (то есть когда сервер не сообщает клиенту, что он принял его запрос вызова, но немедленно начинает его обрабатывать), многоадресный RPC сводится к отправке запроса RPC группе серверов. Этот принцип показан на рис. 4.15. В данном примере клиент отправляет запрос двум серверам, которые впоследствии обрабатывают этот запрос независимо и параллельно. Когда это сделано, результат возвращается клиенту, где происходит обратный вызов.

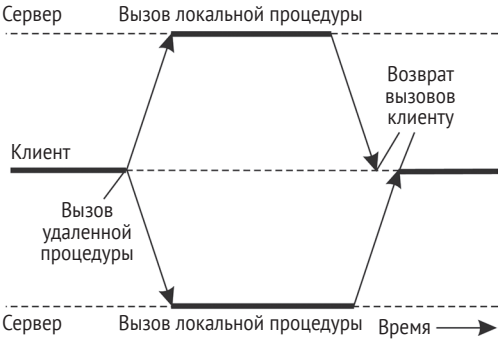


Рис. 4.15 ❖ Принцип многоадресного RPC

Есть несколько вопросов, которые мы должны рассмотреть. Во-первых, как и прежде, клиентское приложение может не знать о том факте, что RPC фактически пересылается более чем на один сервер. Например, чтобы повысить отказоустойчивость, мы можем решить, чтобы все операции выполнялись сервером резервного копирования, который может вступить в действие при сбое основного сервера. То, что сервер был реплицирован, может быть полностью скрыто от клиентского приложения соответствующей заглушкой. Однако даже заглушке не нужно знать, что сервер реплицируется, например потому, что мы используем групповой адрес транспортного уровня.

Во-вторых, нам нужно подумать, что делать с ответами. В частности, будет ли клиент продолжать работу после получения всех ответов или ждать только одного? Все это взаимозависимо. Когда сервер был реплицирован для обеспечения отказоустойчивости, мы можем решить подождать только первый ответ или, возможно, пока большинство серверов не выдаст тот же результат. С другой стороны, если серверы были реплицированы для выполнения одной и той же работы, но для разных частей ввода, их результаты, возможно, должны быть объединены, прежде чем клиент сможет продолжить работу. Опять же, такие вопросы могут быть скрыты в заглушке на стороне клиента, однако разработчику приложения, как минимум, придется указать назначение многоадресного RPC.

Пример: распределенная вычислительная среда RPC

Удаленные вызовы процедур широко используются в качестве основы промежуточного программного обеспечения и распределенных систем в целом. В этом разделе мы подробнее рассмотрим **распределенную вычислительную среду** (Distributed Computing Environment, DCE), разработанную компанией Open Software Foundation (OSF), которая теперь называется «Открытая группа» (Open Group). RPC формирует основу для распределенной вычислительной среды Microsoft DCOM [Eddon and Eddon, 1998] и используется в Samba, файловом сервере и сопутствующем наборе протоколов, позволяю-

щем получить доступ к файловой системе Windows через удаленные вызовы процедур из систем, отличных от Windows.

Хотя DCE RPC, возможно, не самый современный способ управления RPC, стоит обсудить некоторые его детали, в частности потому, что он характерен для большинства традиционных систем RPC, которые используют комбинацию спецификаций интерфейса и явных привязок к различным языкам программирования. Мы начнем с краткого введения в DCE, после чего рассмотрим его основные принципы работы. Подробную информацию о том, как разрабатывать приложения на основе RPC, можно найти в [Stevens, 1999].

Введение в распределенную вычислительную среду DCE

DCE – это настоящая система промежуточного программного обеспечения в том смысле, что она предназначена для выполнения в качестве уровня абстракции между существующими (сетевыми) операционными системами и распределенными приложениями. Первоначально разработанная для Unix, она теперь была перенесена на все основные операционные системы. Идея состоит в том, что заказчик может взять набор существующих компьютеров, добавить программное обеспечение DCE, а затем иметь возможность запускать распределенные приложения, не нарушая существующие (нераспределенные) приложения. Хотя большая часть пакета DCE выполняется в пространстве пользователя, в некоторых конфигурациях часть (распределенной файловой системы) должна быть добавлена в ядро базовой операционной системы.

Модель программирования, лежащая в основе DCE, является моделью клиент-сервер. Пользовательские процессы действуют как клиенты для доступа к удаленным сервисам, предоставляемым серверными процессами. Некоторые из этих сервисов являются частью самого DCE, но другие принадлежат приложениям и написаны программистами приложений. Все общение между клиентами и серверами происходит с помощью RPC.

Цели DCE RPC

Цели распределенной вычислительной системы RPC относительно традиционны. Прежде всего система RPC позволяет клиенту получить доступ к удаленному сервису, просто вызвав локальную процедуру. Этот интерфейс дает возможность писать клиентские (то есть прикладные) программы простым способом, знакомым большинству программистов. Это также облегчает запуск больших объемов существующего кода в распределенной среде с небольшими изменениями, если таковые имеются.

Система RPC должна скрывать все детали от клиентов, а в некоторой степени и от серверов. Для начала система RPC может автоматически найти нужный сервер и впоследствии настроить связь между клиентским и серверным программным обеспечением (обычно это называется **привязкой** (binding)). Она также может обрабатывать передачу сообщений в обоих направлениях, фрагментируя и собирая их по мере необходимости (например, если один из параметров представляет собой большой массив). Наконец, система RPC мо-

жет автоматически обрабатывать преобразования типов данных между клиентом и сервером, даже если они работают на разных архитектурах и имеют разный порядок байтов.

Вследствие способности системы RPC скрывать детали клиенты и серверы независимы друг от друга. Клиент может быть написан на Java, а сервер – на C, или наоборот. Клиент и сервер могут работать на разных аппаратных платформах и использовать разные операционные системы. Также поддерживаются различные сетевые протоколы и представления данных, все без какого-либо вмешательства со стороны клиента или сервера.

Написание клиента и сервера

Система DCE RPC состоит из ряда компонентов, в том числе языков, библиотек, демонов и служебных программ. Вместе они позволяют писать клиентов и серверы. В этом разделе мы опишем части и то, как они соединяются. Весь процесс написания и использования RPC-клиента и сервера представлен на рис. 4.16.

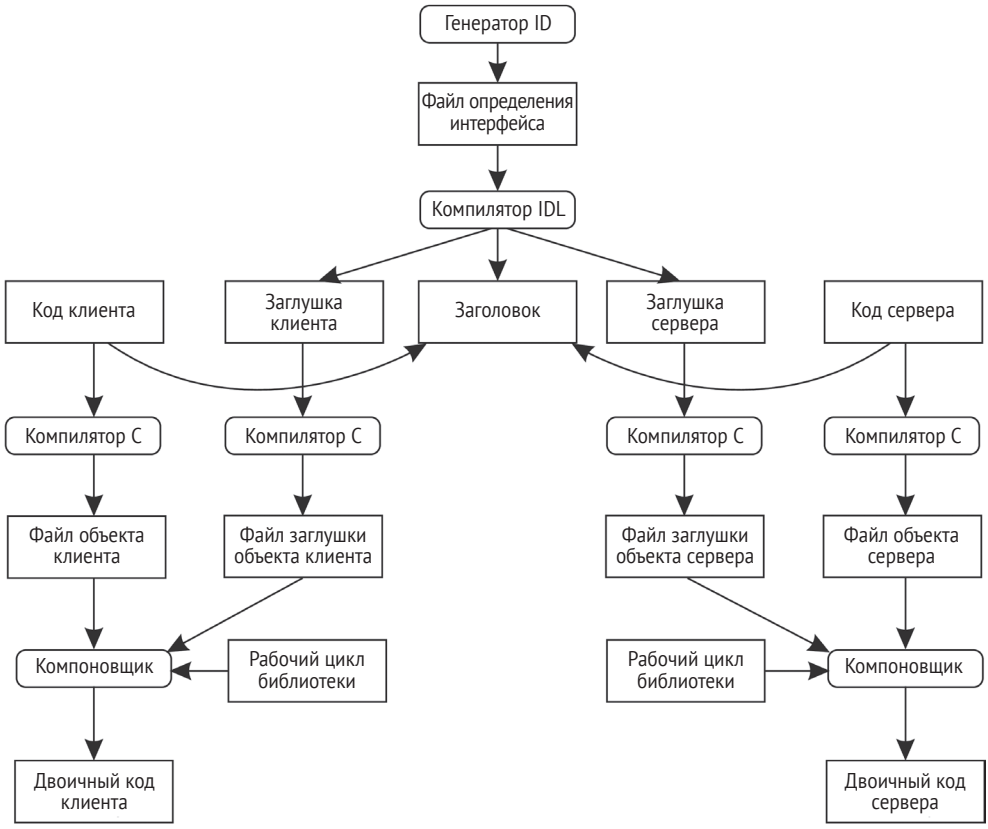


Рис. 4.16 ❖ Шаги при написании клиента и сервера в DCE RPC

В клиент-серверной системе связующим звеном является определение интерфейса, как указано в языке определения интерфейса, или IDL. Он допускает объявления процедур в форме, очень похожей на прототипы функций в ANSI C. Файлы IDL могут также содержать определения типов, объявления констант и другую информацию, необходимую для правильного маршалинга параметров и демаршалинга результатов. В идеале определение интерфейса должно также содержать формальное определение того, что делают процедуры, но такое определение выходит за рамки современного уровня разработки, поэтому определение интерфейса просто определяет синтаксис вызовов, а не семантику.

Важным элементом в каждом файле IDL является глобально уникальный идентификатор для указанного интерфейса. Клиент отправляет этот идентификатор в первом сообщении RPC, и сервер проверяет, правильно ли это. Таким образом, если клиент непреднамеренно пытается привязаться к неправильному серверу или даже к более старой версии правильного сервера, сервер обнаружит ошибку, и привязка не будет выполнена.

Определения интерфейса и уникальные идентификаторы тесно связаны в DCE. Как показано на рис. 4.16, первый шаг в написании клиент-серверного приложения обычно вызывает программу uuidgen с просьбой сгенерировать прототип IDL-файла, содержащий идентификатор интерфейса, который гарантированно никогда больше не будет использоваться в любом интерфейсе, сгенерированном uuidgen. Уникальность обеспечивается за счет кодирования в нем места и времени создания. Он состоит из 128-битного двоичного числа, представленного в файле IDL в виде строки ASCII в шестнадцатеричном формате.

Следующим шагом является редактирование файла IDL, заполнение имен удаленных процедур и их параметров. Стоит отметить, что RPC не является полностью прозрачным. Например, клиент и сервер не могут совместно использовать глобальные переменные. Правила IDL делают невозможным выражение конструкций, которые не поддерживаются.

Когда файл IDL завершен, для его обработки вызывается компилятор IDL. Выходные данные компилятора IDL состоят из трех файлов:

1) файла заголовка (например, `interface.h` в терминах C);
2) клиентской заглушки;
3) серверной заглушки.

Файл заголовка содержит уникальный идентификатор, определения типов, определения констант и прототипы функций. Он должен быть включен (используя `#include`) в коде клиента и сервера. Клиентская заглушка содержит фактические процедуры, которые будет вызывать клиентская программа. Эти процедуры отвечают за сбор и упаковку параметров в исходящее сообщение, а затем за вызов системы выполнения для его отправки. Клиентская заглушка также обрабатывает распаковку ответа и возвращает значения клиенту. Заглушка сервера содержит процедуры, вызываемые циклом выполнения на сервере, когда поступает входящее сообщение. Они, в свою очередь, вызывают фактические серверные процедуры, которые выполняют работу.

Следующим шагом разработчика приложения является написание кода клиента и сервера. Затем они оба компилируются, как и две процедуры-

заглушки. Полученный клиентский код и файлы-заглушки клиента затем связываются с библиотекой выполнения для создания исполняемого двоичного файла для клиента. Аналогично код сервера и заглушка сервера компилируются и связываются для создания двоичного файла сервера. Во время выполнения клиент и сервер запускаются, так что приложение фактически выполняется.

Привязка клиента к серверу

Чтобы позволить клиенту вызывать сервер, необходимо, чтобы сервер был зарегистрирован и был готов принимать входящие вызовы. Регистрация сервера позволяет клиенту найти сервер и выполнить привязку к нему. Поиск местоположения сервера выполняется в два этапа:

1) найдите компьютер сервера;
2) найдите сервер (т. е. правильный процесс) на этом компьютере.

Второй шаг несколько более тонкий. По сути, все сводится к тому, что для связи с сервером клиент должен знать **порт** на компьютере сервера, на который он может отправлять сообщения. Порт используется операционной системой сервера для различения входящих сообщений для разных процессов. В DCE таблица пар (сервер, порт) поддерживается на каждом сервере с помощью процесса, называемого **демоном DCE** (DCE daemon). Прежде чем он станет доступен для входящих запросов, сервер должен запросить у операционной системы порт. Затем он регистрирует этот порт с помощью демона DCE. Демон DCE записывает эту информацию (включая протоколы, на которых говорит сервер) в таблицу портов для будущего использования.

Сервер также регистрируется в службе каталогов, предоставляя ему сетевой адрес компьютера сервера и имя, под которым можно искать сервер. Привязка клиента к серверу происходит, как показано на рис. 4.17.

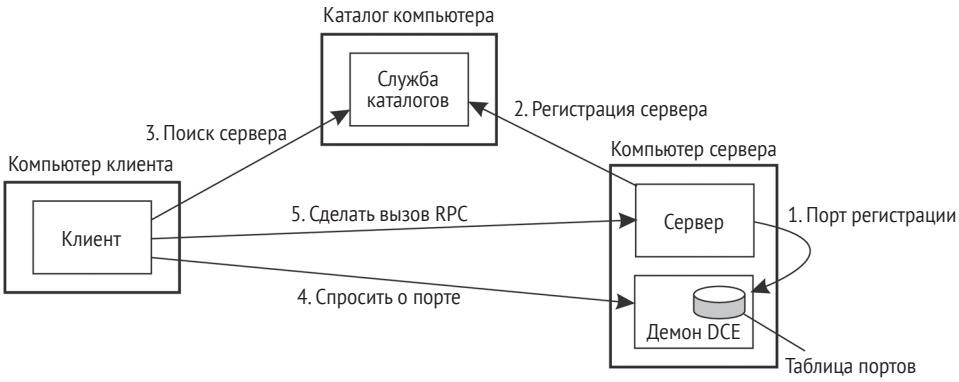


Рис. 4.17 ❖ Привязка клиент-сервер в DCE

Предположим, что клиент хочет привязаться к видеосерверу, который локально известен под именем /local/multimedia/video/movies. Он передает это имя серверу каталогов, возвращающему сетевой адрес компьютера, на ко-

тором работает видеосервер. Затем клиент переходит к демону DCE на этом компьютере (у которого есть известный порт) и просит его найти порт видеосервера в своей таблице портов. Вооружившись этой информацией, RPC теперь может состояться. Для последующих RPC этот поиск не требуется. DCE также дает клиентам возможность выполнять более сложные поиски подходящего сервера, когда это необходимо. Безопасный RPC также является вариантом, где конфиденциальность или целостность данных имеют решающее значение.

Выполнение RPC

Фактический RPC выполняется прозрачно и обычным способом. Клиентская заглушка направляет параметры в библиотеку цикла выполнения для передачи, используя протокол, выбранный во время привязки. Когда сообщение поступает на серверную часть, оно направляется на правильный сервер на основе порта, содержащегося во входящем сообщении. Библиотека цикла выполнения передает сообщение заглушке сервера, которая осуществляет демаршалинг параметров и вызывает сервер. Ответ возвращается по обратному маршруту.

DCE предоставляет несколько семантических опций. По умолчанию используется **не более одной операции** (at-most-once operation), и в этом случае вызов не выполняется более одного раза, даже при наличии системных сбоев. На практике это означает, что если во время вызова RPC происходит сбой сервера, а затем быстрое восстановление, клиент не повторяет операцию, опасаясь, что она уже могла быть выполнена один раз.

В качестве альтернативы можно пометить удаленную процедуру как **идемпотентную** (idempotent) (в файле IDL), и в этом случае она может повторяться без вреда несколько раз. Например, чтение указанного блока из файла можно повторять до тех пор, пока это не удастся. Когда происходит сбой идемпотентного RPC из-за сбоя сервера, клиент может подождать, пока сервер перезагрузится, а затем повторить попытку. Доступна и другая семантика (но редко используемая), включая широковещательную передачу RPC на все машины в локальной сети. Мы возвращаемся к семантике RPC в разделе 8.3 при обсуждении RPC при наличии сбоев.

4.3. Коммуникации, ориентированные на сообщения

Удаленные вызовы процедур и вызовы удаленных объектов способствуют скрытию связи в распределенных системах, то есть повышают прозрачность доступа. К сожалению, нет одного механизма, приемлемого во всех случаях. В частности, когда нельзя предположить, что принимающая сторона выполняет в момент выдачи запроса, необходимы альтернативные услуги связи. Аналогичным образом присущий RPC синхронный характер, при котором клиент блокируется до тех пор, пока его запрос не будет обработан, может нуждаться в замене чем-то другим.

Этим чем-то является обмен сообщениями. В данном разделе мы сосредоточимся на коммуникациях, ориентированных на сообщения, в распределенных системах, сначала подробно рассмотрев, что такое синхронное поведение и каковы его последствия. Затем обсудим системы обмена сообщениями, в которых предполагается, что стороны выполняют свою работу во время связи. Наконец, рассмотрим системы очередей сообщений, которые позволяют процессам обмениваться информацией, даже если другая сторона не выполняется в момент начала связи.

Простой временный обмен сообщениями с сокетами

Многие распределенные системы и приложения построены непосредственно на основе простой ориентированной на сообщения модели, предлагаемой транспортным уровнем. Чтобы лучше понять и оценить системы, ориентированные на сообщения, как часть решений промежуточного программного обеспечения, мы сначала обсудим обмен сообщениями через сокеты транспортного уровня.

Особое внимание было уделено стандартизации интерфейса транспортного уровня, чтобы позволить программистам использовать весь набор его протоколов (сообщений) посредством простого набора операций. Кроме того, стандартные интерфейсы упрощают перенос приложения на другой компьютер. В качестве примера мы кратко обсудим **интерфейс сокетов** (socket interface), представленный в 1970-х годах в Berkeley Unix и принятый в качестве стандарта POSIX (с очень небольшим количеством адаптаций).

Концептуально сокет является конечной точкой связи, в которую приложение может записывать данные, которые должны быть отправлены по базовой сети и из которых можно читать входящие данные. Сокет формирует абстракцию над реальным портом, который используется локальной операционной системой для конкретного транспортного протокола. В последующем мы сконцентрируемся на операциях с сокетами для TCP, которые показаны на рис. 4.18.

Операция	Описание
`socket`	Создание новой привязки конечной точки связи
`bind`	Подсоединение локального адреса к сокету
`listen`	Сообщение операционной системе, какое максимальное число ожидающих запросов на соединение должно быть принято
`accept`	Блокировка вызывающего, пока не придет запрос на связь
`connect`	Попытка установить связь
`send`	Отправка данных
`receive`	Получение данных
`close`	Завершение связи

Рис. 4.18 ❖ Операции с сокетами для TCP/IP

Серверы обычно выполняют первые четыре операции в указанном порядке. При вызове операции сокета вызывающая сторона создает новую конечную точку связи для определенного транспортного протокола. Внутренне создание конечной точки связи означает, что локальная операционная система резервирует ресурсы для отправки и получения сообщений для указанного протокола.

Операция связывания (`bind`) связывает локальный адрес со вновь созданным сокетом. Например, сервер должен привязать IP-адрес своего компьютера вместе с (возможно, общеизвестным) номером порта к сокету. Привязка сообщает операционной системе, что сервер хочет получать сообщения только по указанным адресу и порту. В случае связи с установлением соединения адрес используется для получения входящих запросов на соединение.

Операция прослушивания (`listen`) вызывается только в случае связи с установлением соединения. Это неблокирующий вызов, который позволяет локальной операционной системе зарезервировать достаточное количество буферов для указанного максимального числа ожидающих запросов на подключение, которые вызывающий абонент готов принять.

Вызов `accept` блокирует вызывающего до тех пор, пока не поступит запрос на соединение. Когда приходит запрос, локальная операционная система создает новый сокет с теми же свойствами, что и исходный, и возвращает его вызывающей стороне. Такой подход позволит серверу, например, отключить процесс, который впоследствии будет обрабатывать фактическую связь через новое соединение. Сервер может вернуться назад и ждать другого запроса на подключение к исходному сокету.

Давайте теперь посмотрим на клиентскую сторону. Здесь также сначала должен быть создан сокет с использованием операции сокета, но явная привязка сокета к локальному адресу не требуется, поскольку операционная система может динамически распределять порт при установке соединения. Операция соединения требует, чтобы вызывающая сторона указывала адрес транспортного уровня, на который должен быть отправлен запрос на соединение. Клиент блокируется до тех пор, пока соединение не будет успешно установлено, после чего обе стороны могут начать обмен информацией посредством операции отправки и получения. Наконец, закрытие соединения симметрично при использовании сокетов и устанавливается путем вызова клиентом и сервером операции закрытия. Хотя из этого правила есть много исключений, общая схема, используемая клиентом и сервером для связи с установлением соединения с использованием сокетов, показана на рис. 4.19. Подробности о сетевом программировании с использованием сокетов и других интерфейсов в Unix можно найти в [Stevens, 1998].

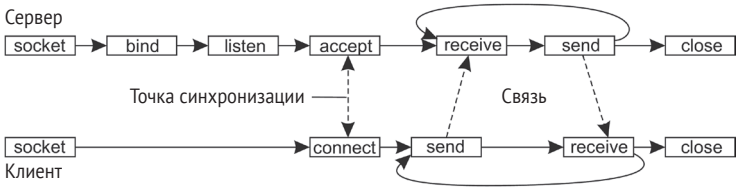


Рис. 4.19 ❖ Ориентированный на соединение шаблон связи с использованием сокетов

Примечание 4.7 (пример: простая система клиент-сервер на основе сокетов)

В качестве иллюстрации повторяющегося шаблона на рис. 4.19 рассмотрим простую систему клиент-сервер на основе сокетов, показанную ниже (см. также примечание 2.1). Мы видим, что сервер [рис. 4.20a] начинает с создания сокета, а затем привязывает адрес к этому сокету. Он вызывает операцию прослушивания и ожидает входящего запроса на соединение. Когда сервер принимает соединение, библиотека сокетов создает отдельное соединение conn, которое используется для получения данных и отправки ответа подключенному клиенту. Сервер входит в цикл приема и отправки сообщений, пока не будет получено больше данных. Затем он закрывает соединение.

```
 1 from socket import *
 2 s = socket(AF_INET, SOCK_STREAM)
 3 s.bind((HOST, PORT))
 4 s.listen(1)
 5 (conn, addr) = s.accept()  # возвращает новый сокет и адрес клиента
 6 while True:                # всегда
 7   data = conn.recv(1024)   # получает данные от клиента
 8   if not data: break       # останавливается, если останавливается клиент
 9   conn.send(str(data)+»*») # возвращает присланные данные с «*»
10 conn.close()              # завершает связь
```

Рис. 4.20 ❖ a) Простая система клиент-сервер на основе сокетов: сервер

Клиент, показанный на рис. 4.20b, снова следует шаблону из рис. 4.19. Он создает сокет и вызывает connect для запроса соединения с сервером. Как только соединение установлено, оно отправляет сообщение, ожидает ответа и после печати результата закрывает соединение.

```
1 from socket import *
2 s = socket(AF_INET, SOCK_STREAM)
3 s.connect((HOST, PORT)) # соединиться с сервером (блокировать, пока нет приема)
4 s.send('Hello, world')  # отправить те же данные
5 data = s.recv(1024)     # получить ответ
6 print data               # напечатать результат
7 s.close()               # завершить соединение
```

Рис. 4.20 ❖ b) Простая система клиент-сервер на основе сокетов: клиент

Примечание 4.8 (дополнительно: реализация заглушек как пересмотр глобальных ссылок)

Чтобы глубже понять работу сокетов, давайте рассмотрим более сложный пример, а именно использование заглушек в качестве глобальных ссылок. Мы возвращаемся к нашему примеру реализации общего списка, который мы сейчас делаем с помощью сервера списков, реализованного в форме класса Python, показанного на рис. 4.21b. На рис. 4.21a представлена реализация заглушки общего списка. Опять же, мы опустили код для удобства чтения.

```
class DBClient:
  def sendrecv(self, message):
    sock = socket()                        # создать сокет
    sock.connect((self.host, self.port))   # соединение с сервером
    sock.send(pickle.dumps(message))       # отправить данные
    result = pickle.loads(sock.recv(1024)) # получить ответ
    sock.close()                           # завершить соединение
    return result

  def create(self):
    self.listID = self.sendrecv([CREATE])
    return self.listID

  def getValue(self):
    return self.sendrecv([GETVALUE, self.listID])

  def appendData(self, data):
    return self.sendrecv([APPEND, data, self.listID])
```

Рис. 4.21 ❖ a) Реализация сервера списков в Python: структура списка

```
class Server
  def __init__(self, port=PORT):
    self.host = 'localhost'                # этот компьютер
    self.port = port                       # порт, который он будет прослушивать
    self.sock = socket()                   # сокет для входящих вызовов
    self.sock.bind((self.host, self.port)) # привязать сокет к адресу
    self.sock.listen(5)                    # максимальное количество соединений
    self.setOfLists = {}                   # init: нет списков для управления

  def run(self):
    while True:
      (conn, addr) = self.sock.accept()  # принять входящий вызов
      data = conn.recv(1024)             # получить данные от клиента
      request = pickle.loads(data)       # развернуть запрос
      if request[0] == CREATE:           # создать список
        istID = len(self.setOfLists) + 1 # разместить listID
        self.setOfLists[listID] = []     # инициализировать для пустых
        conn.send(pickle.dumps(listID))  # вернуть идентификатор

      elif request[0] == APPEND:              # запрос на добавление
        listID = request[2]                   # выборка listID
        data = request[1]                     # выборка данных для добавления
        self.setOfLists[listID].append (data) # добавить его к списку
        conn.send(pickle.dumps(OK))           # вернуть OK

      elif request[0] == GETVALUE:       # запрос на чтение
        listID = request[1]              # выборка listID
        result = self.setOfLists[listID] # получить элементы
        conn.send(pickle.dumps(result))  # вернуть список
      conn.close()                       # завершить соединение
```

Рис. 4.21 ❖ b) Реализация сервера списков в Python: сервер

Класс DBClient представляет заглушку на стороне клиента, которая после маршалинга может передаваться между процессами. Он предоставляет три операции, связанные со списком: create, getValue и append с очевидной семантикой. Предполагается, что DBClient связан с одним конкретным списком, управляемым сервером. Идентификатор этого списка возвращается при создании списка. Обратите внимание, как (внутренняя) операция sendrecv следует шаблону на стороне клиента, описанному на рис. 4.19.

Сервер поддерживает списки, как показано на рис. 4.21b. Его внутренняя структура данных представляет собой setOfLists, где каждый элемент является ранее созданным списком. Сервер просто ожидает входящие запросы, отменяет маршалинг запроса и проверяет, какая операция запрашивается. Результаты отправляются обратно запрашивающему клиенту (который всегда выдает операцию sendrecv, реализованную как часть DBClient). Вновь мы видим, что сервер следует шаблону, показанному на рис. 4.19: он создает сокет, привязывает к нему адрес, информирует операционную систему о том, сколько подключений он должен прослушивать, а затем ожидает принятия входящего запроса на подключение. Как только соединение установлено, сервер получает данные, отправляет ответ и снова закрывает соединение.

Чтобы использовать заглушку в качестве глобальной ссылки, мы представляем каждое клиентское приложение с помощью класса Client, показанного на рис. 4.21c. Класс создается в том же процессе, в котором запущено приложение (например, значением self.host), и будет прослушивать на конкретном порте сообщения от других приложений, а также от сервера. В противном случае он просто отправляет и получает сообщения, закодированные с помощью операций sendTo и recvAny соответственно.

```
 1 class Client:
 2   def __init__(self, port):
 3     self.host = 'localhost'                # этот компьютер
 4     self.port = port                       # порт, который будет прослушивать
 5     self.sock = socket()                   # сокет для входящих вызовов
 6     self.sock.bind((self.host, self.port)) # привязать сокет к адресу
 7     self.sock.listen(2)                    # максимальное чило соединений
 8
 9   def sendTo(self, host, port, data):
10     sock = socket()
11     sock.connect((host, port))     # подключиться к серверу (блокирующий вызов)
12     sock.send(pickle.dumps (data)) # отправить некоторые данные
13     sock.close()
14
15   def recvAny(self):
16     (conn, addr) = self.sock.accept()
17     return conn.recv(1024)
```

Рис. 4.21 ❖ c) Реализация сервера списков в Python: клиент

Теперь рассмотрим код, показанный на рис. 4.21d, который имитирует два клиентских приложения. Первый создает новый список и добавляет к нему данные. Затем обратите внимание, как dbClient1 просто отправляется другому клиенту. Под капотом мы теперь знаем, что он маршалируется в операции sendTo (строка 12) класса Client, показанной на рис. 4.21c.

Второй клиент просто ожидает входящего сообщения (строка 12), отменяет маршалирование результата, зная, что это экземпляр DBClient, и впоследствии добавляет еще несколько данных в тот же список, что и данные, добавленные первым клиентом. Действительно, экземпляр DBClient рассматривается как глобальная ссылка, по-видимому, наряду со всеми операциями, которые идут со связанным классом.

```
 1 pid = os.fork()
 2 if pid == 0:
 3   client1 = Client(CLIENT1)                    # создать клиента
 4   dbClient1 = DBClient(HOST, PORT)             # создать ссылку
 5   dbClient1.create()                           # создать новый список
 6   dbClient1. appendData('Client 1')            # добавить некоторые данные
 7   client1.sendTo(HOSTCL2, CLIENT2, dbClient1) # отправить другому клиенту
 8
 9 pid = os.fork()
10 if pid == 0:
11   client2 = Client(CLIENT2)        # создать нового клиента
12   data = client2.recvAny()         # блок до отправки данных
13   dbClient2 = pickle.loads(data)   # получить ссылку
14   dbClient2.appendData('Client 2') # добавить данные в тот же список
```

Рис. 4.21 ❖ d) Передача заглушек в качестве ссылок

Расширенный переходный обмен сообщениями

Стандартный, основанный на сокетах подход к временным сообщениям является очень базовым и, как таковой, довольно хрупким: легко допустить ошибку. Кроме того, сокеты, по сути, поддерживают только TCP или UDP, что означает, что любое дополнительное средство для обмена сообщениями должно быть реализовано отдельно прикладным программистом.

На практике нам часто нужны более продвинутые подходы к коммуникациям, ориентированным на сообщения, чтобы упростить сетевое программирование, выйти за пределы функциональности, предлагаемой существующими сетевыми протоколами, лучше использовать локальные ресурсы и т. д.

Использование шаблонов обмена сообщениями: ZeroMQ

Один из подходов к упрощению сетевого программирования основан на том наблюдении, что многие приложения обмена сообщениями или их компоненты могут быть эффективно организованы в соответствии с несколькими простыми шаблонами связи. Последующее предоставление улучшений для сокетов, для каждого из этих шаблонов, может упростить разработку распределенного сетевого приложения. Этот подход был использован в ZeroMQ и задокументирован в [Hintjens, 2013; Akgul, 2013].

Как и в подходе Беркли, ZeroMQ также предоставляет сокеты, через которые происходит вся связь. Фактическая передача сообщений обычно осу-

ществляется по TCP-соединениям, и, как и по протоколу TCP, все коммуникации по существу ориентированы на соединение, то есть сначала будет установлено соединение между отправителем и получателем до того, как может произойти передача сообщения. Тем не менее настройка и поддержание соединений в основном скрыты: программисту приложений не нужно беспокоиться об этих проблемах. Чтобы еще больше упростить задачу, сокет может быть связан с несколькими адресами, что позволяет серверу эффективно обрабатывать сообщения из самых разных источников через единый интерфейс. Например, сервер может прослушивать несколько портов с помощью одной операции получения блокировки. Таким образом, сокеты ZeroMQ могут поддерживать связь «многие к одному», а не просто связь «один к одному», как в случае со стандартными сокетами Беркли. В завершение: сокеты ZeroMQ также поддерживают связь «один ко многим», то есть многоадресную рассылку.

Для ZeroMQ важно то, что связь является асинхронной: отправитель обычно продолжает работу после отправки сообщения в базовую подсистему связи. Интересным побочным эффектом объединения асинхронной связи с установлением соединения является то, что процесс может запросить настройку соединения и впоследствии отправить сообщение, даже если получатель еще не запущен и готов принять входящие запросы на соединение, не говоря уже о входящих сообщениях. Конечно, происходит то, что запрос на подключение и последующие сообщения помещаются в очередь на стороне отправителя, в то время как отдельный поток как часть библиотеки ZeroMQ позаботится о том, чтобы в конечном итоге соединение было установлено и сообщения передавались получателю.

Упрощая, ZeroMQ устанавливает более высокий уровень абстракции в коммуникациях на основе сокетов путем *сопряжения* сокетов: определенный тип сокета, используемый для отправки сообщений, соединяется с соответствующим типом сокета для приема сообщений. Каждая пара типов сокетов соответствует шаблону связи. Три наиболее важных коммуникационных шаблона, поддерживаемых ZeroMQ: *запрос-ответ*, *публикация-подписка* и *конвейер*.

Шаблон **запрос-ответ** используется в традиционной связи клиент-сервер, аналогичной той, которая обычно используется для удаленных вызовов процедур. Клиентское приложение применяет **сокет запроса** (request socket) (типа REQ) для отправки сообщения запроса на сервер и ожидает, что последний ответит. Предполагается, что сервер использует **сокет ответа** (reply socket) (типа REP). Шаблон запрос-ответ упрощает вопросы для разработчиков, избегая необходимости вызывать операцию прослушивания, а также операцию приема. Кроме того, когда сервер получает сообщение, последующий вызов для отправки автоматически направляется к исходному отправителю. Аналогично, когда клиент вызывает операцию recv (для получения сообщения) после отправки сообщения, ZeroMQ предполагает, что клиент ожидает ответа от исходного получателя. Обратите внимание, что этот подход был эффективно закодирован в локальной операции `sendrecv` на рис. 4.21, которая обсуждается в примечании 4.8.

Примечание 4.9 (пример: шаблон запрос-ответ)

Давайте посмотрим на простой пример программирования, чтобы проиллюстрировать шаблон запроса-ответа. На рис. 4.22a показан сервер, который добавляет звездочку к полученному сообщению. Как и прежде, он создает сокет и связывает его с комбинацией протокола (в данном случае TCP), а также хоста и порта. В нашем примере сервер готов принимать входящие запросы на подключение к двум разным портам. Затем он ожидает входящих сообщений. Шаблон запроса-ответа эффективно связывает получение сообщения с последующим ответом. Другими словами, когда сервер вызывает `send`, он будет передавать сообщение тому же клиенту, от которого ранее получил сообщение. Конечно, эта простота может быть достигнута только в том случае, если программист действительно соблюдает шаблон запроса-ответа.

```
1 import zmq
2 context = zmq.Context()
3
4 p1 = "tcp://"+ HOST +":"+ PORT1 # как и где подключиться
5 p2 = "tcp://"+ HOST +":"+ PORT2 # как и где подключиться
6 s = context.socket (zmq.REP)    # создать сокет ответа
7
8 s.bind(p1)                      # связать сокет с адресом
9 s.bind(p2)                      # связать сокет с адресом
10 while True:
11   message = s.recv()           # ожидать входящего сообщения
12   if not "STOP" in message:    # если «нет» остановить...
13     s.send(message + "*")      # добавить «*» к сообщению
14   else:                        # иначе...
15     break                      # выйти из цикла и завершить
```

Рис. 4.22 ❖ a) Система клиент-сервер ZeroMQ на основе: сервер

Клиент, показанный на рис. 4.22b, делает то, что ожидается: он создает сокет и подключается к связанному серверу. Когда он отправляет сообщение, он может ожидать получить от того же сервера ответ. Отправляя строку `"STOP"`, он сообщает серверу, что это сделано, после чего сервер фактически остановится.

Интересно, что асинхронная природа ZeroMQ позволяет запустить клиента перед запуском сервера. Подразумевается, что если в этом примере мы запустим сервер, затем клиента и через некоторое время второго клиента, то последний будет заблокирован, пока сервер не будет перезапущен. Кроме того, обратите внимание, что ZeroMQ не требует от программиста указания количества ожидаемых байтов. В отличие от TCP, ZeroMQ использует сообщения вместо байтовых потоков в качестве базовой модели связи.

```
 1 import zmq
 2 context = zmq.Context()
 3
 4 p1 = "tcp://"+ HOST +":"+ PORT1 # как и где подключиться
 5 s = context.socket(zmq.REQ)     # создать сокет запроса
 6
 7 s.connect(p1)                  # блокировать до подключения
 8 s.send("Hello world 1")        # отправить сообщение
 9 message = s.recv()             # блокировать до ответа
10 s.send("STOP")                 # сообщить серверу остановиться
11 print message                  # распечатать результат
```

Рис. 4.22 ❖ b) Система клиент-сервер ZeroMQ: клиент

В случае **шаблона публикации-подписки** (publish-subscribe pattern) клиенты подписываются на определенные сообщения, которые публикуются серверами. Мы столкнулись с этой моделью кратко в главе 1, когда обсуждали координацию на основе событий. По сути, будут передаваться только те сообщения, на которые подписан клиент. Если сервер публикует сообщения, на которые никто не подписан, эти сообщения будут потеряны. В своей простейшей форме этот шаблон устанавливает многоадресные сообщения от сервера нескольким клиентам. Предполагается, что сервер использует сокет типа PUB, в то время как каждый клиент должен использовать сокеты типа SUB. Каждый клиентский сокет подключен к сокету сервера. По умолчанию клиент не подписывается на конкретное сообщение. Это означает, что до тех пор, пока не будет предоставлена явная подписка, клиент не будет получать сообщение, опубликованное сервером.

Примечание 4.10 (пример: шаблон публикации-подписки)

Давайте опять сделаем эту модель более конкретной на простом примере.

```
1 import ZMQ, time
2
3 context = zmq.Context()
4 s = context.socket(zmq.PUB)       # создать сокет издателя
5 p = "tcp://"+ HOST +":"+ PORT     # как и где общаться
6 c. bind(p)                        # привязать сокет к адресу
7 while True:
8   time.sleep(5)                   # ждать каждые 5 секунд
9   s.send("TIME" + time.asctime()) # опубликовать текущее время
```

Рис. 4.23 ❖ a) Сервер времени на основе многоадресного сокета

На рис. 4.23a показан заведомо наивный сервер времени, который публикует свое текущее местное время через PUB-сокет. Местное время публикуется каждые пять секунд для любого заинтересованного клиента.

Клиент одинаково прост, как показано на рис. 4.23b. Сначала создается SUB-сокет, который он подключает к соответствующему PUB-сокету сервера. Для получения соответствующих сообщений необходимо подписаться на сообщения, в которых в качестве тега указано TIME. В нашем примере клиент просто распечатает первые пять сообщений, полученных от сервера. Обратите внимание, что у нас может быть столько клиентов, сколько мы хотим: сообщение сервера будет многоадресным для всех подписчиков.

```
 1 import ZMQ
 2
 3 context = zmq.Context()
 4 s = context.socket(zmq.SUB)          # создать абонентский сокет
 5 p = "tcp://"+ HOST +":"+ PORT        # как и где общаться
 6 c.connect(p)                         # подключиться к серверу
 7 s.setsockopt(zmq.SUBSCRIBE, "TIME") # подписаться на сообщения TIME
 8
 9 for i in range(5):                   # пять итераций
10 time = s.recv()                      # получить сообщение
11 print time
```

Рис. 4.23 ❖ b) Клиент для сервера времени на основе сокетов многоадресной рассылки

Наконец, **схема конвейера** (pipeline pattern) характеризуется тем, что процесс хочет выдать свои результаты, предполагая, что есть другие процессы, которые хотят получить эти результаты. Суть шаблона конвейера состоит в том, что продвигающийся процесс на самом деле не заботится о том, какой другой процесс извлекает свои результаты: первый доступный процесс сделает это. Аналогично любой процесс, извлекающий результаты из множества других процессов, будет делать это с самого начала процесса, делая свои результаты доступными. Таким образом, намерение шаблона конвейера состоит в том, чтобы поддерживать максимально возможное количество процессов, как можно быстрее продвигая результаты через конвейер процессов.

Примечание 4.11 (пример: схема конвейера)

В качестве нашего последнего примера рассмотрим следующий шаблон для сохранения коллекции рабочих задач. Рисунок 4.24a показывает код для так называемой **задачи фермера** (farmer task): процесс, генерирующий задачи, которые должны быть подхвачены другими. В этом примере мы моделируем задачу, позволяя фермеру выбрать случайное число, моделирующее продолжительность или нагрузку выполняемой работы. Эта рабочая нагрузка затем отправляется в push-сокет, фактически ставя в очередь, пока другой процесс не подхватит ее.

Такие другие процессы известны как **задачи работников** (worker task), эскиз которых приведен на рис. 4.24b. Эта задача соединяет один выдвижной сокет с, в этом случае двумя конкретными фермерскими задачами. Как только он берет на себя какую-то работу, он имитирует, что он на самом деле что-то делает, спя в течение нескольких десятков миллисекунд, пропорциональных полученной рабочей нагрузке.

```
 1 import ZMQ, time, pickle, sys, random
 2
 3 context = zmq.Context()
 4 me = str(sys.argv[1])
 5 s = context.socket(zmq.PUSH)              # создать push-сокет
 6 src = SRC1  if me == '1' else SRC2        # проверьте хост источника задачи
 7 prt = PORT1 if me == '1' else PORT2       # проверьте порт источника задачи
 8 p = "tcp://"+ src +":"+ prt               # как и где подключиться
 9 s.bind(p)                                 # привязать сокет к адресу
10
11 for i in range(100):                      # сгенерировать 100 рабочих нагрузок
12   workload = random.randint(1, 100)      # вычислить рабочую нагрузку
13   s.sforend(pickle.dumps((me, workload))) # отправить рабочую нагрузку работнику
```

Рис. 4.24 ❖ a) Задача, имитирующая генерацию работы

Семантика этого двухтактного шаблона такова, что первый доступный работник получит работу от любого фермера, и аналогично, если есть несколько рабочих, готовых взять работу, каждому из них будет дано задание. То, насколько распределение работы на самом деле выполняется честно, требует особого внимания, которое мы не будем здесь обсуждать.

```
 1 import ZMQ, time, pickle, sys, random
 2
 3 context = zmq.Context()
 4 me = str (sys.argv[1])
 5 r = context.socket(zmq.PULL)    # создать вытяжной сокет
 6 p1 = "tcp://"+ SRC1 +":"+ PORT1 # адрес первого источника задачи
 7 p2 = "tcp://"+ SRC2 +":"+ PORT2 # адрес второго источника задачи
 8 r.connect(p1)                   # подключиться к источнику задачи 1
 9 r.connect(p2)                   # подключиться к источнику задачи 2
10
11 while True:
12   work = pickle.loads(r.recv()) # получает работу из источника
13   time.sleep(work[1]*0.01)      # претендует на работу
```

Рис. 4.24 ❖ b) Задача работников

Интерфейс передачи сообщений (MPI)

С появлением высокопроизводительных мультикомпьютеров разработчики искали операции, ориентированные на сообщения, которые позволили бы им легко создавать высокоэффективные приложения. Это означает, что операции должны быть на удобном уровне абстракции (чтобы упростить разработку приложений) и что их реализация требует минимальных затрат. Сокеты были сочтены недостаточными по двум причинам. Во-первых, они были на недостаточном уровне абстракции, поддерживая только простые операции отправки и получения. Во-вторых, сокеты были разработаны для связи между сетями с использованием стеков протоколов общего назначения, таких как TCP/IP. Они не считались подходящими для запатентованных

протоколов, разработанных для высокоскоростных сетей присоединения, таких как те, которые используются в высокопроизводительных кластерах серверов. Эти протоколы требовали интерфейса, который мог бы обрабатывать более продвинутые функции, такие как различные формы буферизации и синхронизации.

В результате большинство межсетевых сетей и высокопроизводительных мультикомпьютеров были оснащены собственными коммуникационными библиотеками. Эти библиотеки предлагали множество высокоуровневых и в целом эффективных операций связи. Конечно, все библиотеки были несовместимы друг с другом, поэтому у разработчиков приложений теперь возникла проблема переносимости.

Необходимость быть независимой от аппаратного обеспечения и платформы в конечном итоге привела к определению стандарта для передачи сообщений, называемого просто **интерфейсом передачи сообщений** (Message-Passing Interface, MPI). Интерфейс MPI разработан для параллельных приложений и как таковой приспособлен для переходной коммуникации. Это позволяет прямое использование базовой сети. Также предполагается, что серьезные сбои, такие как сбои процессов или сетевых разделов, являются фатальными и не требуют автоматического восстановления.

MPI предполагает, что общение происходит в рамках известной группы процессов. Каждой группе присваивается идентификатор. Каждому процессу в группе также присваивается (локальный) идентификатор. Поэтому пара (`groupID`, `processID`) однозначно идентифицирует источник или место назначения сообщения и используется вместо адреса транспортного уровня. Может быть несколько, возможно перекрывающихся, групп процессов, участвующих в вычислении, и все они выполняются одновременно.

В основе MPI лежат операции обмена сообщениями для поддержки переходных коммуникаций. Наиболее интуитивно понятные операции представлены на рис. 4.25.

Операция	Описание
`MPI_bsend`	Добавить исходящее сообщение в локальный буфер отправки
`MPI_send`	Отправить сообщение и дождаться копирования на локальный или удаленный буфер
`MPI_ssend`	Отправить сообщение и дождаться начала передачи
`MPI_sendrecv`	Отправить сообщение и дождаться ответа
`MPI_isend`	Передать ссылку на исходящее сообщение и продолжить
`MPI_recv`	Получить сообщение; заблокировать, если его нет
`MPI_irecv`	Проверить, есть ли входящее сообщение, но не блокировать

Рис. 4.25 ❖ Некоторые из наиболее интуитивно понятных операций передачи сообщений MPI

Переходная асинхронная связь поддерживается с помощью операции `MPI_bsend`. Источник отправляет сообщение для передачи, которое обычно в процессе выполнения MPI сначала копируется в локальный буфер. Когда

сообщение скопировано, отправитель продолжает. Локальная система выполнения MPI удалит сообщение из своего локального буфера и позаботится о передаче, как только получатель вызовет операцию приема.

Существует также блокирующая операция отправки, называемая `MPI_send`, семантика которой зависит от реализации.

Операция `MPI_send` может блокировать вызывающую сторону до тех пор, пока указанное сообщение не будет скопировано в систему выполнения MPI на стороне отправителя, или до тех пор, пока получатель не инициирует операцию приема. Синхронная связь, посредством которой отправитель блокируется до тех пор, пока его запрос не будет принят для дальнейшей обработки, доступна через операцию `MPI_ssend`. Наконец, поддерживается также самая сильная форма синхронной связи: когда отправитель вызывает `MPI_sendrecv`, он отправляет запрос получателю и блокирует, пока последний не вернет ответ. По сути, эта операция соответствует обычному RPC.

И `MPI_send`, и `MPI_ssend` имеют варианты, позволяющие избежать копирования сообщений из пользовательских буферов в буферы, внутренние для локальной системы исполнения MPI. Эти варианты по существу соответствуют форме асинхронного общения. При использовании `MPI_isend` отправитель передает указатель на сообщение, после чего система времени выполнения MPI заботится о связи. Отправитель немедленно продолжает. Чтобы предотвратить перезапись сообщения до завершения связи, MPI, если требуется, предлагает операции для проверки завершения или даже для блокировки. Как и в случае `MPI_send`, вопрос о том, было ли сообщение фактически передано получателю или просто скопировано локальной системой выполнения MPI во внутренний буфер, не определено.

Аналогично с `MPI_issend`, отправитель также передает только указатель на систему выполнения MPI. Когда система выполнения указывает, что обработала сообщение, отправителю гарантируется, что получатель принял сообщение и теперь работает над ним.

Операция `MPI_recv` вызывается для получения сообщения; она блокирует вызывающего до тех пор, пока не прибудет сообщение. Существует также асинхронный вариант, называемый `MPI_irecv`, с помощью которого получатель указывает, что он готов принять сообщение. Получатель может проверить, действительно ли сообщение прибыло, или заблокировать, пока оно не поступило.

Семантика операций связи MPI не всегда прямолинейна, и различные операции иногда могут взаимно заменяться, не влияя на правильность программы. Официальная причина, почему поддерживается так много форм связи, следующая: она предоставляет разработчикам систем MPI достаточно возможностей для оптимизации производительности. Циники могли бы сказать, что комитет не мог составить свое коллективное мнение, поэтому он добавил все. В настоящее время MPI находится в третьей версии с более чем 440 доступными операциями. Возможно, легче понять наличие такого разнообразия, приняв во внимание его разработку для высокопроизводительных параллельных приложений. Подробнее о MPI можно почитать в [Gropp et al., 2016]. Полную ссылку на MPI-3 можно найти в [Message Passing Interface Forum, 2015].

Постоянная связь, ориентированная на сообщения

Теперь мы подошли к важному классу сервисов промежуточного программного обеспечения, ориентированных на сообщения, известному как **системы очередей сообщений** (message-queuing systems), или просто как **промежуточное программное обеспечение, ориентированное на сообщения** (Message-Oriented Middleware, MOM). Эти системы сообщений обеспечивают обширную поддержку постоянной асинхронной связи. Суть этих систем заключается в том, что они предлагают среднесрочную память для хранения сообщений, не требуя, чтобы отправитель или получатель был активен во время передачи сообщения. Важным отличием от сокетов и MPI является то, что системы очередей сообщений обычно предназначены для поддержки передачи сообщений, которая может занимать минуты вместо секунд или миллисекунд.

Модель очереди сообщений

Основная идея системы очереди сообщений заключается в том, что приложения взаимодействуют друг с другом, вставляя сообщения в определенные очереди. Эти сообщения пересылаются через ряд коммуникационных серверов и в конечном итоге доставляются к месту назначения, даже если оно было недоступно при отправке сообщения. На практике большинство коммуникационных серверов напрямую связаны друг с другом. Другими словами, сообщение обычно передается непосредственно на целевой сервер. В принципе, каждое приложение имеет собственную частную очередь, в которую другие приложения могут отправлять сообщения. Очередь может быть прочитана только связанным приложением, но несколько приложений также могут совместно использовать одну очередь.

Важным аспектом систем очередей сообщений является то, что отправителю, как правило, дают только гарантии того, что его сообщение будет в конечном итоге вставлено в очередь получателя. Не дается никаких гарантий относительно того, когда и даже будет ли сообщение действительно прочитано, что полностью определяется поведением получателя.

Эта семантика позволяет коммуникациям быть свободно связанными во времени. Таким образом, нет необходимости, чтобы получатель выполнялся, когда сообщение отправляется в его очередь. Аналогично отправителю не нужно выполнять в тот момент, когда его сообщение получено получателем. Отправитель и получатель могут работать совершенно независимо друг от друга. Фактически, после того как сообщение было помещено в очередь, оно будет оставаться там до тех пор, пока не будет удалено, независимо от того, выполняется ли его отправитель или получатель. Это дает нам четыре комбинации с учетом режима выполнения отправителя и получателя, как показано на рис. 4.26.

На рис. 4.26a отправитель и получатель выполняются в течение всей передачи сообщения. На рис. 4.26b выполняется только отправитель, а получатель

пассивен, то есть находится в состоянии, в котором доставка сообщения невозможна. Тем не менее отправитель может отправлять сообщения. Комбинация пассивного отправителя и исполняющего получателя показана на рис. 4.26c. В этом случае получатель может читать сообщения, которые были ему отправлены, но не обязательно, чтобы его соответствующие отправители также выполнялись. Наконец, на рис. 4.26d мы видим ситуацию, когда система хранит (и, возможно, передает) сообщения, даже когда отправитель и получатель пассивны. Можно утверждать, что только если поддерживается эта последняя конфигурация, система очереди сообщений действительно обеспечивает постоянный обмен сообщениями.

Рис. 4.26 ❖ Четыре комбинации для слабосвязанных коммуникаций с использованием очередей

В принципе, сообщения могут содержать любые данные. Единственный важный аспект с точки зрения промежуточного программного обеспечения – это то, что сообщения адресуются правильно. На практике адресация осуществляется путем предоставления общесистемного уникального имени очереди назначения. В некоторых случаях размер сообщения может быть ограничен, хотя также возможно, что базовая система заботится о фрагментации и сборке больших сообщений таким образом, чтобы они были полностью прозрачны для приложений. Результатом этого подхода является то, что базовый интерфейс, предлагаемый приложениям, может быть чрезвычайно простым, как показано на рис. 4.27.

Операция `put` вызывается отправителем для передачи сообщения в базовую систему, которое должно быть добавлено в указанную очередь. Как мы объясняли, это неблокирующий вызов. Операция `get` – это блокирующий вызов, с помощью которого авторизованный процесс может удалить самое длинное ожидающее сообщение в указанной очереди. Процесс блокируется, только если очередь пуста. Изменения в этом вызове позволяют искать конкретное сообщение в очереди, например с использованием приоритета или соответствующего шаблона. Неблокирующий вариант задается операцией `poll`.

Операция	Описание
put	Добавить сообщение в указанную очередь
get	Блокировать до тех пор, пока указанная очередь не пустая, и удалить первое сообщение
poll	Проверить указанную очередь на наличие сообщений и удалить первое сообщение. Никогда не блокировать
notify	Установите обработчик, который будет вызываться при помещении сообщения в указанную очередь

Рис. 4.27 ❖ Базовый интерфейс для очереди в системе очередей сообщений

Если очередь пуста, или если конкретное сообщение не может быть найдено, вызывающий процесс просто продолжается.

Наконец, большинство систем массового обслуживания также позволяют процессу устанавливать обработчик как *функцию обратного вызова* (*callback function*), которая автоматически вызывается всякий раз, когда сообщение помещается в очередь. Обратные вызовы также можно использовать для автоматического запуска процесса, который будет извлекать сообщения из очереди, если в данный момент не выполняется ни один процесс. Этот подход часто реализуется с помощью демона на стороне получателя, который постоянно отслеживает очередь на наличие входящих сообщений и обрабатывает соответственно.

Общая архитектура системы очереди сообщений

Давайте теперь подробнее рассмотрим, как выглядит общая система очереди сообщений. Прежде всего очереди управляются **администраторами очередей** (queue managers). Администратор очередей является отдельным процессом или реализуется с помощью библиотеки, связанной с приложением. Во-вторых, как правило, приложение может помещать сообщения только в локальную очередь. Аналогично получение сообщения возможно путем извлечения его только из *локальной* очереди. Как следствие если администратор очередей QM_A, обрабатывающий очереди для приложения A, работает как отдельный процесс, оба процесса QM_A и A обычно размещаются на одной машине или, в худшем случае, в одной и той же локальной сети. Также обратите внимание, что если все администраторы очередей связаны с соответствующими приложениями, мы больше не можем говорить о постоянной асинхронной системе обмена сообщениями.

Если приложения могут помещать сообщения только в локальные очереди, то очевидно, что каждое сообщение должно нести информацию о своем назначении. Задача администратора очередей – убедиться, что сообщение достигает пункта назначения. Это подводит нас к рассмотрению ряда вопросов.

Во-первых, нам нужно рассмотреть вопрос о том, как очередь назначения переадресована. Очевидно, что для повышения прозрачности местоположения предпочтительно, чтобы очереди имели логические, независимые от местоположения имена. Предполагая, что менеджер очереди реализован как отдельный процесс, использование логических имен подразумевает, что

каждое имя должно быть связано с **адресом контакта** (contact address), таким как пара (хост, порт), и что сопоставление имени с адресом является легкодоступным для администратора очередей, как показано на рис. 4.28. На практике контактный адрес несет больше информации, в частности используемый протокол, такой как TCP или UDP. Мы встречались с подобными контактными адресами в наших примерах расширенных сокетов, например в примечании 4.9.

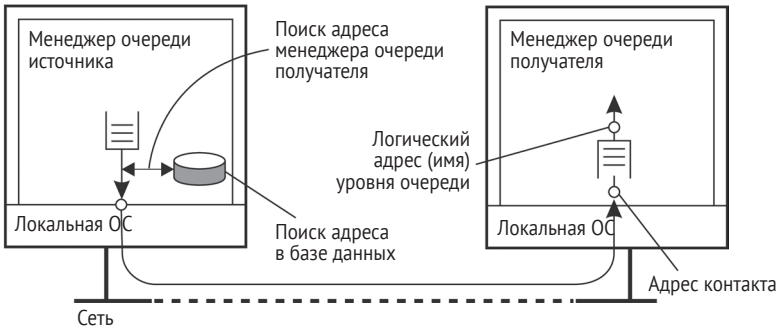


Рис. 4.28 ❖ Соотношение между именами на уровне очереди и адресацией на сетевом уровне

Вторая проблема, которую мы должны рассмотреть, заключается в том, как сопоставление имени с адресом фактически делается доступным для администратора очередей. Обычный подход – просто реализовать сопоставление в виде таблицы поиска и скопировать эту таблицу всем менеджерам. Очевидно, это приводит к проблеме обслуживания, при каждом добавлении или именовании новой очереди многие, если не все, таблицы должны обновляться. Существуют различные способы решения таких проблем, и мы обсудим их в главе 5.

Это подводит нас к рассмотрению третьего вопроса, связанного с проблемами эффективного поддержания отображения имени на адрес. Мы неявно предположили, что если очередь назначения в диспетчере QM_B известна администратору очередей QM_A, то QM_A может напрямую связаться с QM_B для передачи сообщений. Фактически это означает, что (контактный адрес) каждого администратора очередей должен быть известен всем остальным. Очевидно, что при работе с очень большими системами очередей сообщений у нас будет проблема с масштабируемостью. На практике часто существуют специальные администраторы очередей, которые работают как **маршрутизаторы** (routers): они пересылают входящие сообщения другим администраторам очередей. Таким образом, система очереди сообщений может постепенно превратиться в полноценную **оверлейную сеть** (overlay network) уровня приложения.

Если только несколько маршрутизаторов должны знать о топологии сети, то администратору очереди источника нужно только знать, к какому соседнему маршрутизатору, скажем, R он должен переслать сообщение с учетом очереди назначения. Маршрутизатору R, в свою очередь, может понадобить-

ся отслеживать только смежные маршрутизаторы, чтобы увидеть, куда переслать сообщение, и т. д. Конечно, нам все еще нужно иметь сопоставления имени и адреса для всех администраторов очередей, включая маршрутизаторы, но не трудно представить, что такие таблицы могут быть намного меньше и проще в обслуживании.

Брокеры сообщений

Важной областью применения систем очереди сообщений является интеграция существующих и новых приложений в единую согласованную распределенную информационную систему. Если мы предполагаем, что связь с приложением происходит через сообщения, то интеграция требует, чтобы приложения могли понимать сообщения, которые они получают. На практике это требует, чтобы отправитель имел свои исходящие сообщения в том же формате, что и получатель, но также чтобы его сообщения придерживались той же семантики, что и ожидаемая получателем. По сути, отправитель и получатель должны говорить на одном языке, то есть придерживаться одного и того же протокола обмена сообщениями.

Проблема с таким подходом состоит в том, что каждый раз, когда приложение A добавляется в систему, имеющую собственный протокол обмена сообщениями, для каждого другого приложения B, которое должно связываться с A, нам потребуется предоставить средства для преобразования их соответствующих сообщений. Таким образом, в системе с N приложениями нам потребуется $N \times N$ конвертеров протоколов обмена сообщениями.

Альтернативой является согласование общего протокола обмена сообщениями, как это делается с традиционными сетевыми протоколами. К сожалению, этот подход, как правило, не будет работать для систем очереди сообщений. Проблема заключается в уровне абстракции, на которой работают эти системы. Общий протокол обмена сообщениями имеет смысл только в том случае, если совокупность процессов, использующих этот протокол, действительно имеет достаточно много общего. Если набор приложений, составляющих распределенную информационную систему, весьма разнообразен (как это часто бывает), то создание единого решения для всех просто не сработает.

Если мы сосредоточимся только на формате и значении сообщений, общности можно достичь, подняв уровень абстракции, как это делается с сообщениями XML. В этом случае сообщения несут информацию об их собственной организации, и что стандартизировано, является способом, с помощью которого они могут описать свой контент. Как следствие приложение может предоставить информацию об организации своих сообщений, которая может быть автоматически обработана. Конечно, этой информации, как правило, недостаточно: нам также необходимо убедиться, что семантика сообщений понятна.

Учитывая эти проблемы, общий подход заключается в том, чтобы научиться жить с различиями и попытаться предоставить средства, позволяющие сделать преобразования максимально простыми. В системах очередей сообщений преобразования обрабатываются специальными узлами в сети

очередей, известными как **брокеры сообщений** (message brokers). Брокер сообщений действует как посредник сообщений, шлюз уровня приложения в системе очередей сообщений. Его основная цель – конвертировать входящие сообщения, чтобы их можно было понять по месту назначения приложения. Обратите внимание, что для системы очередей сообщений брокер сообщений – это просто другое приложение, как показано на рис. 4.29. Иными словами, брокер сообщений обычно не считается неотъемлемой частью системы обслуживания очередей.

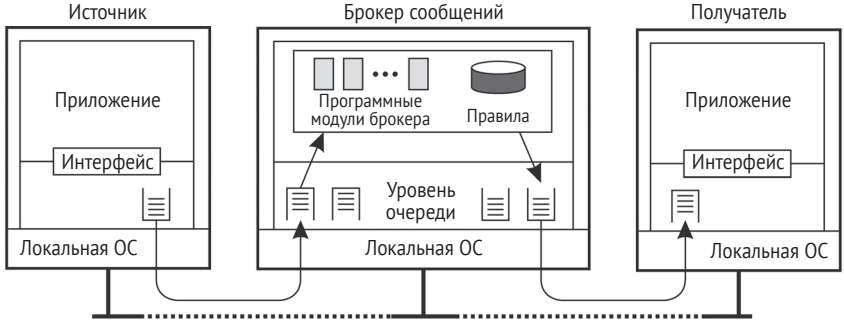


Рис. 4.29 ❖ Общая организация брокера сообщений в системе очередей сообщений

Брокер сообщений может просто переформатировать сообщения. Например, предположим, что входящее сообщение содержит таблицу из базы данных, в которой записи разделены специальным разделителем конца записи, а поля в записи имеют известную фиксированную длину. Если целевое приложение ожидает другого разделителя между записями, а также ожидает, что поля имеют переменную длину, для преобразования сообщений в формат, ожидаемый адресатом, может использоваться брокер сообщений.

В более продвинутой настройке брокер сообщений может действовать как шлюз уровня приложений, в котором была закодирована информация о протоколе обмена сообщениями нескольких приложений. В целом для каждой пары приложений у нас будет отдельная подпрограмма, способная конвертировать сообщения между двумя приложениями. На рис. 4.29 эта подпрограмма изображена как программный модуль (plugin), чтобы подчеркнуть, что такие части могут быть динамически подключены или удалены из брокера.

Наконец, обратите внимание, что во многих случаях для расширенной **интеграции корпоративных приложений** (enterprise application integration, EAI) используется брокер сообщений, как мы обсуждали в разделе 1.3. В этом случае вместо (только) преобразования сообщений брокер отвечает за сопоставление приложений на основе сообщений, которыми происходит обмен. В такой модели **публикации-подписки** (publish-subscribe) приложения отправляют сообщения в форме публикации. В частности, они могут опубликовать сообщение по теме X, которое затем отправляется брокеру. Приложения, которые заявили о своей заинтересованности в сообщениях по теме X, то

есть подписались на эти сообщения, будут затем получать эти сообщения от брокера. Также возможны более продвинутые формы посредничества.

В основе брокера сообщений лежит хранилище правил для преобразования сообщений одного типа в другой. Задача заключается в том, чтобы определить правила и программные модули брокера. Большинство продуктов брокеров сообщений поставляются со сложными инструментами разработки, но важно, чтобы хранилище было заполнено экспертами. Здесь мы видим прекрасный пример, когда коммерческие продукты часто вводят в заблуждение, якобы обеспечивая «интеллект» там, где на самом деле интеллект разве что в головах самих экспертов.

Примечание 4.12 (дополнительная информация: примечание о системах очередей сообщений)

Учитывая то, что мы сказали о системах очередей сообщений, может показаться, что они давно существуют в виде реализаций для служб электронной почты. Системы электронной почты обычно реализуются через набор почтовых серверов, которые хранят и пересылают сообщения от имени пользователей на хосты, напрямую подключенные к серверу. Маршрутизация обычно не учитывается, поскольку системы электронной почты могут напрямую использовать базовые транспортные службы. Например, в почтовом протоколе для интернета, SMTP [Postel, 1982], сообщение передается путем установки прямого TCP-соединения с почтовым сервером назначения.

Что делает почтовые системы особенными по сравнению с системами очередей сообщений, так это то, что они в первую очередь направлены на оказание прямой поддержки конечным пользователям. Это объясняет, например, почему ряд приложений для групповой работы основаны непосредственно на системе электронной почты [Khoshafin and Buckiewicz, 1995]. Кроме того, системы электронной почты могут иметь очень специфические требования, такие как автоматическая фильтрация сообщений, поддержка для расширенных баз данных обмена сообщениями (например, для простого извлечения ранее сохраненных сообщений) и т. д.

Общие системы очереди сообщений не нацелены на поддержку только конечных пользователей. Важной проблемой является то, что они настроены для обеспечения постоянной связи между процессами, независимо от того, выполняет ли процесс пользовательское приложение, обрабатывает ли доступ к базе данных, выполняет ли вычисления и т. д. Этот подход приводит к иному набору требований к системам очередей сообщений, чем к чисто системам электронной почты. Например, системы электронной почты, как правило, не должны обеспечивать гарантированную доставку сообщений, приоритеты сообщений, средства ведения журналов, эффективную многоадресную рассылку, балансировку нагрузки, отказоустойчивость и т. д. для общего использования.

Таким образом, универсальные системы очередей сообщений имеют широкий спектр приложений, включая электронную почту, рабочий процесс, групповое программное обеспечение и пакетную обработку. Однако, как мы уже говорили ранее, наиболее важной областью применения является интеграция (возможно, широко рассредоточенного) набора баз данных и приложений в интегрированную информационную систему [Hohpe and Woolf, 2004]. Например, запрос, расширяющий несколько баз данных, может потребоваться разбить на подзапросы, которые перенаправляются в отдельные базы данных. Системы очереди сообщений помогают, предоставляя основные средства для упаковки каждого подзапроса в сообщение и маршрутизации его в соответствующую базу данных. Другие средства связи, которые обсуждались в этой главе, гораздо менее приемлемы.

Пример: система очереди сообщений IBM WebSphere

Чтобы понять, как системы очереди сообщений работают на практике, давайте посмотрим на систему очереди сообщений, которая является частью IBM WebSphere. Ранее известная как MQSeries, теперь она называется **WebSphere MQ**. На систему WebSphere MQ имеется множество документации, и в дальнейшем мы можем обсудить только основные принципы. Краткое введение можно найти в WebSphere MQ [Тейлор, 2012], а дополнительные подробности – в работах [Davies and Broadhurst, 2005; Aranha et al., 2013].

Обзор

Базовая архитектура сети массового обслуживания MQ довольно проста и показана на рис. 4.30. Все очереди управляются **администраторами очередей** (queue managers). Администратор очередей отвечает за удаление сообщений из своих очередей отправки и их пересылку другим администраторам очередей. Аналогично администратор очередей отвечает за обработку входящих сообщений, отбирая их из базовой сети и впоследствии сохраняя каждое сообщение в соответствующей входной очереди. Чтобы составить представление о том, что может означать обмен сообщениями: максимальный размер сообщения по умолчанию составляет 4 МБ, но его можно увеличить до 100 МБ. Очередь обычно ограничена 2 Гб данных, но в зависимости от базовой операционной системы этот максимум может быть легко установлен выше.

Администраторы очередей попарно связаны через **каналы сообщений** (message channels), которые являются абстракцией соединений транспортного уровня. Канал сообщений – это однонаправленное, надежное соединение между отправляющим и принимающим администраторами очередей, по которому передаются сообщения в очереди. Например, интернет-канал сообщений реализован как TCP-соединение. Каждый из двух концов канала сообщений управляется **агентом канала сообщений** (message channel agent). Отправляющий MCA в основном не делает ничего, кроме проверки очередей отправки для сообщения, упаковки его в пакет транспортного уровня и отправки его по соединению к ассоциированному получающему MCA. Аналогичным образом основной задачей принимающего MCA является прослушивание входящего пакета, его развертывание и последующее сохранение развернутого сообщения в соответствующей очереди.

Администраторы очередей могут быть связаны с тем же процессом, что и приложение, для которого оно управляет очередями. В этом случае очереди скрыты от приложения за стандартным интерфейсом, но фактически могут напрямую управляться приложением. Альтернативная организация – это организация, в которой администраторы очередей и приложения работают на разных компьютерах. В этом случае приложению предоставляется тот же интерфейс, что и тогда, когда администратор очередей размещается на одном компьютере. Однако интерфейс реализован в виде прокси-сервера, который обменивается данными с администратором очередей с использо-

ванием традиционной синхронной связи на основе RPC. Таким образом, MQ в основном сохраняет модель, к которой могут быть доступны только очереди, локальные для приложения.

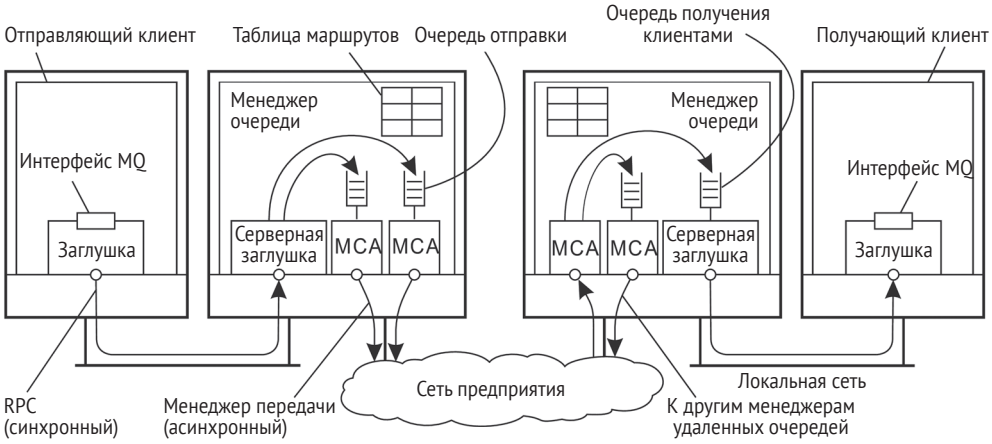


Рис. 4.30 ❖ Общая организация системы очередей сообщений IBM

Каналы

Важным компонентом MQ являются каналы сообщений. Каждый канал сообщений имеет ровно одну связанную очередь отправки, из которой он выбирает сообщения и должен передать их на другой конец. Передача по каналу может осуществляться только в том случае, если и его отправляющая, и принимающая MCA включены и работают. Помимо запуска обоих MCA вручную, есть несколько альтернативных способов запуска канала, некоторые из которых мы обсудим далее.

Одна из них состоит в том, чтобы приложение непосредственно начало свой конец канала путем активации отправляющего или получающего MCA. Однако, с точки зрения прозрачности, это не очень привлекательная альтернатива. Лучшим подходом к запуску MCA для отправки является настройка очереди отправки канала для запуска триггера, когда сообщение впервые помещается в очередь. Этот триггер связан с обработчиком для запуска MCA отправки, чтобы он мог удалять сообщения из очереди отправки.

Другой альтернативой является запуск MCA по сети. В частности, если одна сторона канала уже активна, он может отправить управляющее сообщение, запрашивающее запуск другого MCA. Такое управляющее сообщение отправляется демону, прослушивающему известный адрес на той же машине, где должен быть запущен другой MCA.

Каналы автоматически останавливаются по истечении заданного времени, в течение которого больше сообщений не было сброшено в очередь отправки.

Каждый MCA имеет набор связанных атрибутов, которые определяют общее поведение канала. Некоторые из атрибутов перечислены на рис. 4.31. Значения атрибутов отправляющего и получающего MCA должны быть со-

вместимы и, возможно, согласованы сначала до того, как канал может быть настроен. Например, оба MCA, очевидно, должны поддерживать один и тот же транспортный протокол. Примером необоротного атрибута является то, должны ли сообщения доставляться в том же порядке, в каком они помещены в очередь отправки. Если один MCA хочет доставку FIFO, другой должен этому соответствовать. Примером оборотного значения атрибута является максимальная длина сообщения, которая будет просто выбрана в качестве минимального значения, указанного MCA.

Атрибут	Описание
Тип транспорта	Определяет используемый транспортный протокол
FIFO-доставка	Указывает, что сообщения должны быть доставлены в порядке их отправки
Длина сообщения	Максимальная длина одного сообщения
Количество повторных попыток установки	Максимальное количество повторных попыток запуска удаленной MCA
Попытки доставки	Максимальное количество MCA, которое будет пытаться поставить сообщение в очередь

Рис. 4.31 ❖ Некоторые атрибуты, связанные с агентами канала сообщений

Передача сообщений

Чтобы передать сообщение от одного администратора очередей другому (возможно, удаленному) администратору очередей, необходимо, чтобы каждое сообщение содержало адрес назначения, для которого используется заголовок передачи. Адрес в MQ состоит из двух частей. Первая часть состоит из имени администратора очередей, которому должно быть доставлено сообщение. Вторая часть – это имя целевой очереди, обращающейся к тому администратору, к которому должно быть добавлено сообщение.

Помимо адреса назначения, также необходимо указать маршрут, по которому должно следовать сообщение. Спецификация маршрута выполняется путем предоставления имени локальной очереди отправки, к которой должно быть добавлено сообщение. Таким образом, нет необходимости указывать полный маршрут в сообщении. Напомним, что каждый канал сообщений имеет ровно одну очередь отправки. Указывая, к какой очереди отправки следует добавить сообщение, мы фактически указываем, какому администратору очередей следует переслать сообщение.

В большинстве случаев маршруты явно хранятся у администратора очередей в таблице маршрутизации. Запись в таблице маршрутизации представляет собой пару (`destQM`, `sendQ`), где `destQM` – это имя администратора очередей назначения, а `sendQ` – это имя локальной очереди отправки, к которой следует добавить сообщение для этого администратора очередей. (Запись в таблице маршрутизации называется псевдонимом в MQ.)

Возможно, что сообщение должно быть передано через несколько администраторов очередей, прежде чем оно достигнет пункта назначения. Всякий раз, когда такой промежуточный администратор очередей получает со-

общение, он просто извлекает имя целевого администратора очередей из заголовка сообщения и выполняет поиск в таблице маршрутизации, чтобы найти локальную очередь отправки, к которой следует добавить сообщение.

Важно понимать, что каждый администратор очередей имеет уникальное общесистемное имя, которое эффективно используется в качестве идентификатора для этого администратора очередей.

Проблема с использованием этих имен заключается в том, что замена администратора очередей или изменение его имени повлияет на все приложения, отправляющие ему сообщения.

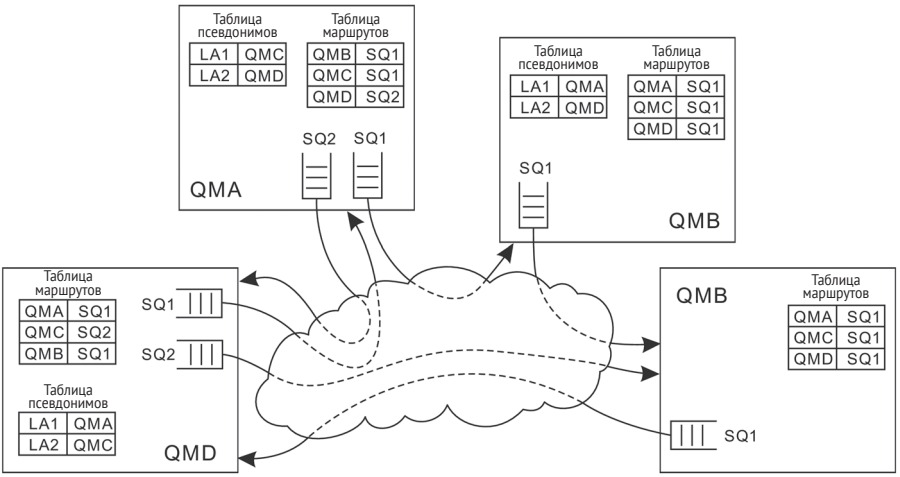


Рис. 4.32 ❖ Общая организация сети массового обслуживания MQ с использованием таблиц маршрутизации и псевдонимов

Проблемы могут быть устранены путем использования локального псевдонима (local alias) для имен администратора очередей. Псевдоним, определенный в администраторе очередей QM_1, является другим именем для администратора очередей QM_2, но он доступен только для приложений, взаимодействующих с QM_1.

Псевдоним позволяет использовать одно и то же (логическое) имя для очереди, даже если администратор очередей этой очереди изменяется. Изменение имени администратора очередей требует, чтобы мы изменили его псевдоним во всех администраторах очередей. Тем не менее приложения могут быть оставлены без изменений.

Принцип использования таблиц маршрутизации и псевдонимов показан на рис. 4.32. Например, приложение, связанное с администратором очередей QMA, может ссылаться на удаленный администратор очередей, используя локальный псевдоним LA_1. Администратор очередей сначала ищет фактическое место назначения в таблице псевдонимов, чтобы определить, является ли он администратором очередей QMC. Маршрут к QMC находится в таблице маршрутизации, в которой говорится, что сообщения для QMC должны быть добавлены к исходящей очереди SQ_1, которая используется для передачи

сообщений администратору очередей QMB. Последний будет использовать свою таблицу маршрутизации для пересылки сообщения в QMC.

Следование этому подходу маршрутизации и создания псевдонимов приводит к относительно простому интерфейсу программирования, который называется **интерфейсом очереди сообщений** (Message Queue Interface, MQI).

Наиболее важные операции MQI приведены на рис. 4.33.

Чтобы поместить сообщения в очередь, приложение вызывает `MQOPEN`, указывая целевую очередь в определенном администраторе очередей. Администратор очередей может быть назван с использованием локально доступного псевдонима. Является очередь назначения действительно удаленной или нет, полностью прозрачно для приложения. `MQOPEN` также должен вызываться, если приложение хочет получать сообщения из своей локальной очереди. Только локальные очереди могут быть открыты для чтения входящих сообщений.

Когда приложение завершает доступ к очереди, оно должно закрыть его, вызывая `MQCLOSE`.

Операция	Описание
`MQOPEN`	Открыть (возможно, удаленную) очередь
`MQCLOSE`	Закрыть очередь
`MQPUT`	Поместить сообщение в открытую очередь
`MQGET`	Получить сообщение из (локальной) очереди

Рис. 4.33 ❖ Операции, доступные в интерфейсе очереди сообщений

Сообщения могут быть записаны или прочитаны из очереди, используя `MQPUT` и `MQGET` соответственно. В принципе, сообщения удаляются из очереди в приоритетном порядке. Сообщения с одинаковым приоритетом удаляются в первую очередь, то есть самое длинное ожидающее сообщение удаляется сначала. Также можно запросить конкретные сообщения. Кроме того, `MQ` предоставляет средства для оповещения приложений о поступлении сообщений, что позволяет избежать того, что приложению постоянно приходится опрашивать очередь сообщений на наличие входящих сообщений.

Интересное наблюдение, характерное для современных систем очередей сообщений, заключается в том, что постоянный обмен сообщениями не получается путем простого сохранения очередей. Вместо этого сообщение помечается как постоянное, и задача администратора очередей – следить за тем, чтобы сообщение могло пережить сбой. Как следствие очередь может одновременно хранить постоянные и непостоянные сообщения.

Управление оверлейными сетями

Из приведенного описания должно быть ясно, что важной частью управления системами MQ является соединение различных администраторов очередей в согласованную оверлейную сеть. Более того, эту сеть необходимо поддерживать с течением времени. Для небольших сетей это обслуживание не потребует намного больше, чем обычная административная работа, но во-

просы усложняются, когда очереди сообщений используются для интеграции и дезинтеграции больших существующих систем.

Основная проблема с MQ заключается в том, что оверлейные сети необходимо контролировать вручную. Это администрирование включает в себя не только создание каналов между администраторами очередей, но и заполнение таблиц маршрутизации. Очевидно, это может перерасти в кошмар. К сожалению, поддержка управления для систем MQ усовершенствована только в том смысле, что администратор может установить практически каждый возможный атрибут и настроить любую мыслимую конфигурацию. Однако суть в том, что каналы и таблицы маршрутизации должны поддерживаться вручную.

В основе управления оверлеями лежит компонент **функции управления каналом** (channel control function), который логически располагается между агентами канала сообщений. Этот компонент позволяет оператору точно отслеживать, что происходит в двух конечных точках канала. Кроме того, он используется для создания каналов и таблиц маршрутизации, а также для управления администраторами очередей, в которых размещаются агенты каналов сообщений. В некотором смысле такой подход к управлению оверлеями сильно напоминает управление серверами кластера, где используется один сервер администрирования. В последнем случае сервер по существу предлагает только удаленную оболочку для каждой машины в кластере, а также несколько коллективных операций для обработки групп машин. Хорошая новость об управлении распределенными системами заключается в том, что оно предлагает множество возможностей, если вы ищете область для поиска новых решений серьезных проблем.

Пример: расширенный протокол очереди сообщений (AMQP)

Интересным наблюдением систем очередей сообщений является то, что они были разработаны частично для обеспечения возможности взаимодействия унаследованных приложений, но в то же время мы видим, что когда речь идет об операциях между различными системами очередей сообщений, мы часто сталкиваемся со стеной. Как следствие когда организация решит использовать систему очередей сообщений от производителя X, ей, возможно, придется согласиться на решения, которые предоставляет только X. Таким образом, решения для очереди сообщений являются в значительной степени частными решениями. Так много для открытости.

В 2006 году была сформирована рабочая группа, чтобы изменить эту ситуацию, что привело к спецификации **протокола расширенной очереди сообщений** (Advanced Message-Queuing Protocol, AMQP). Существуют разные версии AMQP, последняя версия – 1.0. Есть также различные реализации AMQP, в частности версия, предшествующая версии 1.0, которая к моменту создания версии 1.0 приобрела значительную популярность. Поскольку версия, предшествующая версии 1.0, сильно отличается от версии 1.0, но имеет также устойчивую пользовательскую базу, можно увидеть рядом с сервером

1.0 различные серверы, предшествующие серверу 1.0 AMQP (неоспоримо несовместимые с ним). Так много возможностей для открытости.

В этом разделе мы опишем AMQP, но более или менее намеренно смешаем версию, предшествующую 1.0, и версию 1.0, придерживаясь основ и духа AMQP. Подробности можно найти в спецификациях [Group, 2008; ОАЗИС, 2011]. Реализации AMQP включают RabbitMQ [Videla and Williams, 2012] и Qpid Apache.

Основы

Протокол AMQP развивается вокруг приложений, администраторов очередей и очередей. Используя подход, общий для многих сетевых ситуаций, мы различаем AMQP как *службу обмена сообщениями*, реальный *протокол обмена сообщениями* и, наконец, *интерфейс обмена сообщениями*, предлагаемые приложениям. С этой целью проще всего рассмотреть ситуацию, когда имеется только один администратор очередей, работающий как один отдельный сервер, формирующий реализацию AMQP в качестве службы. Приложение связывается с этим администратором очередей через локальный интерфейс. Связь между приложением и администратором очередей происходит в соответствии с протоколом AMQP.

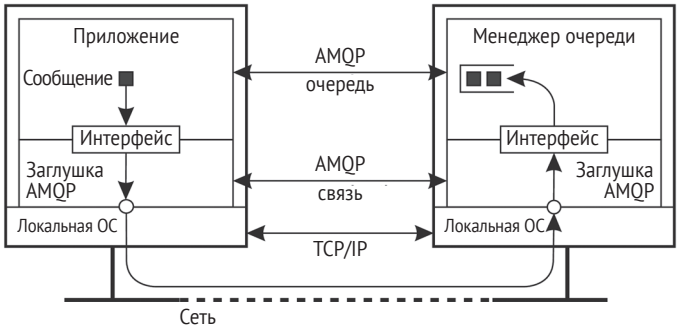


Рис. 4.34 ❖ Обзор экземпляра AMQP с одним сервером

Эта ситуация показана на рис. 4.34 и должна выглядеть знакомо. Заглушка AMQP защищает приложение (а также администратор очередей) от деталей, касающихся передачи сообщений и связи в целом. В то же время она реализует интерфейс очереди сообщений, позволяющий приложению использовать AMQP в качестве службы очереди сообщений. Хотя различие между заглушкой AMQP и администратором очередей явно задано для администраторов очередей, строгость разделения остается за реализацией. Тем не менее если и не строго, то концептуально существует, как мы вскоре поясним, различие между обработкой очередей и обработкой связанных сообщений.

AMQP связи

AMQP позволяет приложению установить **соединение** с администратором очередей; соединение – это контейнер для нескольких односторонних **кана-**

лов. В то время как время жизни канала может быть очень динамичным, соединения предполагаются относительно стабильными. Это различие между соединением и каналом обеспечивает эффективную реализацию, в частности благодаря использованию одного соединения TCP транспортного уровня для мультиплексирования множества различных каналов между приложением и администратором очередей. На практике AMQP предполагает, что TCP используется для установления соединений AMQP.

Двунаправленная связь устанавливается через сеансы: логическая группировка двух каналов. Соединение может иметь несколько сеансов, но надо учитывать, что канал не обязательно должен быть частью сеанса.

Наконец, для фактической передачи сообщений необходима ссылка. Концептуально ссылка или, скорее, ее конечные точки отслеживают состояние передаваемых сообщений. Таким образом, она обеспечивает детальное управление потоком данных между приложением и администратором очередей, и в действительности разные политики управления могут одновременно применяться для разных сообщений, которые передаются через один и тот же сеанс соединения. Управление потоком устанавливается с помощью кредитов: получатель может сообщить отправителю, сколько сообщений ему разрешено отправлять по конкретной ссылке.

Когда сообщение должно быть передано, приложение передает его в свою локальную заглушку AMQP. Как уже упоминалось, каждая передача сообщения связана с одной конкретной ссылкой. Передача сообщений обычно происходит в три этапа.

1. На стороне отправителя сообщению присваивается уникальный идентификатор, и он записывается локально, чтобы находиться в **неустановленном состоянии** (unsettled state). Заглушка впоследствии передает сообщение на сервер, где заглушка AMQP также записывает его как находящееся в неустановленном состоянии. В этот момент заглушка на стороне сервера передает ее администратору очередей.
2. Предполагается, что принимающее приложение (в данном случае администратор очередей) обрабатывает сообщение и, как правило, сообщает своей заглушке, что оно завершено. Заглушка передает эту информацию исходному отправителю, после чего сообщение переходит в заглушке AMQP исходного отправителя в **установленное состояние** (settled state).
3. Заглушка AMQP исходного отправителя теперь сообщает заглушке исходного получателя о том, что передача сообщения завершена (это означает, что исходный отправитель теперь забудет о сообщении). Заглушка получателя теперь может также отклонить что-либо о сообщении, формально записав его как уже установленное.

Отметим, что поскольку принимающее приложение может указывать нижележащему уровню связи AMQP, что оно выполняется с помощью сообщения, AMQP обеспечивает истинную сквозную надежность связи. В частности, приложение, будь то клиентское приложение или фактический администратор очередей, может дать команду коммуникационному уровню AMQP удерживать сообщение (то есть сообщение остается в неустановленном состоянии).

AMQP обмена сообщениями

Обмен сообщениями в протоколе AMQP логически происходит на уровне выше уровня обработки сообщений. Именно здесь приложение может указать, что необходимо сделать с сообщением, но также может увидеть, что произошло до сих пор. Формально обмен сообщениями происходит между двумя **узлами** трех типов: производитель, потребитель или очередь. Обычно узлы производителя и потребителя представляют собой обычные приложения, тогда как очереди используются для хранения и пересылки сообщений. Действительно, администратор очередей обычно состоит из нескольких узлов очереди. Для передачи сообщения двум узлам необходимо установить связь между собой.

Получатель может указать отправителю, было его сообщение принято (что означает, что оно было успешно обработано) или отклонено. Обратите внимание, что это означает, что уведомление возвращается исходному отправителю. AMQP также поддерживает фрагментацию и сборку больших сообщений, для которых отправляются дополнительные уведомления.

Конечно, важным аспектом AMQP является его поддержка *постоянных* сообщений. Достижение постоянства осуществляется с помощью нескольких механизмов. Наиболее важно, во-первых, что сообщение может быть помечено как долговременное, что указывает на то, что источник ожидает восстановления любого промежуточного узла, такого как очередь, в случае сбоя. Промежуточный узел, который не может гарантировать такую долговечность, должен будет отклонить сообщение. Во-вторых, исходный или целевой узел также может указывать свою долговечность: если он будет долговечным, то будет ли он поддерживать свое состояние или он также будет поддерживать (неустановленное) состояние долговечных сообщений? Объединение последних с надежными сообщениями эффективно обеспечивает надежную передачу сообщений и постоянный обмен сообщениями.

Протокол AMQP в действительности является протоколом обмена сообщениями в том смысле, что он сам по себе не поддерживает, например, примитивы публикации-подписки. Ожидается, что такие проблемы решаются более продвинутыми, собственными администраторами очередей, схожими с брокерами сообщений, обсуждаемыми в разделе 4.3.

Наконец, нет причины, по которой администратор очередей не может быть подключен к другому администратору очередей. Фактически довольно часто организаторы очередей объединяются в оверлейную сеть, в которой сообщения направляются от производителей к их потребителям. AMQP не определяет, как должна создаваться и управляться оверлейная сеть, и действительно, разные поставщики систем на основе AMQP предлагают разные решения. Особое значение имеет указание того, как сообщения должны маршрутизироваться через сеть. Суть в том, что администраторам нужно будет сделать бóльшую часть этой спецификации вручную. Только в тех случаях, когда оверлеи имеют регулярные структуры, такие как циклы или деревья, становится проще предоставлять необходимые детали маршрутизации.

4.4. Многоадресная связь

Важной темой связи в распределенных системах является поддержка отправки данных нескольким получателям, также известная как многоадресная связь.

В течение многих лет эта тема относилась к области сетевых протоколов, где были реализованы и оценены многочисленные предложения по решениям сетевого и транспортного уровней [Janic, 2005; Obraczka, 1998]. Основной проблемой во всех решениях была установка путей коммуникации для распространения информации. На практике это потребовало больших управленческих усилий, требующих во многих случаях вмешательства человека. Кроме того, до тех пор, пока нет конвергенции предложений, интернет-провайдеры неохотно поддерживают многоадресную рассылку [Diot et al., 2000].

С появлением одноранговой технологии и, в частности, структурированного оверлейного управления стало проще устанавливать каналы связи. Поскольку одноранговые решения обычно развертываются на уровне приложений, были внедрены различные методы многоадресной передачи на уровне приложений. В этом разделе мы кратко рассмотрим эти методы.

Многоадресная связь также может быть выполнена способами, отличными от настройки явных путей связи. Как мы также увидим в этом разделе, широковещательное распространение информации обеспечивает простые (но зачастую менее эффективные) способы многоадресной рассылки.

Многоадресная рассылка на уровне дерева приложений

Основная идея многоадресной рассылки на уровне приложений состоит в том, что узлы объединяются в оверлейную сеть, которая затем используется для распространения информации среди ее участников. Важным обстоятельством является то, что сетевые маршрутизаторы не участвуют в групповом членстве. Как следствие соединения между узлами в оверлейной сети могут пересекать несколько физических каналов, и как таковые сообщения маршрутизации в оверлейной сети могут быть неоптимальными по сравнению с тем, что могло бы быть достигнуто при маршрутизации на сетевом уровне.

Важнейшей проблемой проектирования является создание оверлейной сети. По сути, существует два подхода [El-Sayed et al., 2003; Hosseini et al., 2007; Allani et al., 2009]. Во-первых, узлы могут организовываться непосредственно в дерево, что означает, что между каждой парой узлов существует уникальный (перекрывающийся) путь. Альтернативный подход состоит в том, что узлы объединяются в ячеистую сеть, в которой каждый узел будет иметь несколько соседей, и, как правило, существует несколько путей между каждой парой узлов. Основное различие между ними состоит в том, что последний обычно обеспечивает более высокую надежность: если разрывается

соединение (например, из-за сбоя узла), все равно сохраняется возможность распространять информацию без необходимости немедленной реорганизации всей оверлейной сети.

Примечание 4.13 (дополнительно: создание многоадресного дерева в Chord)

Чтобы конкретизировать ситуацию, давайте рассмотрим относительно простую схему построения многоадресного дерева в Chord, которую мы описали в примечании 2.5. Эта схема была первоначально предложена для Scribe [Castro et al., 2002b], которая является многоадресной схемой прикладного уровня, построенной поверх Pastry [Rowstron and Druschel, 2001]. Последняя также является одноранговой системой на основе динамического гипертекста (dynamic hypertext, DHT).

Предположим, что узел хочет начать многоадресный сеанс. Для этого он просто генерирует многоадресный идентификатор, скажем `mid`, который является просто случайно выбранным 160-битным ключом. Затем он ищет `succ(mid)`, который является узлом, ответственным за этот ключ, и превращает его в корень дерева многоадресной рассылки, которое будет использоваться для отправки данных заинтересованным узлам. Чтобы присоединиться к дереву, узел P просто выполняет операцию поиска `lookup(mid)`, в результате чего сообщение поиска с запросом на присоединение к группе многоадресной рассылки `mid` будет перенаправлено с P на `succ(mid)`. Сам алгоритм маршрутизации будет подробно объяснен в главе 5.

На пути к корню запрос на присоединение пройдет несколько узлов. Предположим, что он впервые достигает узла Q. Если Q никогда раньше не видел запрос на соединение с `mid`, он станет **перенаправителем** (**forwarder**) для этой группы. В этот момент P станет дочерним для Q, тогда как последний продолжит пересылать запрос на присоединение к корню. Если следующий узел в корне, скажем R, также еще не является сервером пересылки, он станет единым, запишет Q в качестве дочернего узла и продолжит отправку запроса на присоединение.

С другой стороны, если Q (или R) уже является сервером пересылки для `mid`, он также будет записывать предыдущего отправителя как своего дочернего (то есть P или Q соответственно), но у него больше не будет необходимости отправлять запрос на присоединение к корню, так как Q (или R) уже будет членом дерева многоадресной рассылки.

Узлы, такие как P, которые явно запросили присоединение к дереву многоадресной рассылки, по определению также являются серверами пересылки. Результатом этой схемы является то, что мы создаем многоадресное дерево в оверлейной сети с двумя типами узлов: чистые серверы пересылки, которые действуют как помощники, и узлы, которые также являются серверами пересылки, но явно запросили присоединение к дереву. Многоадресная передача теперь проста: узел просто отправляет многоадресное сообщение к корню дерева, снова выполняя операцию поиска `lookup(mid)`, после чего это сообщение может быть отправлено вдоль дерева.

Мы отмечаем, что это высокоуровневое описание многоадресной рассылки в Scribe не полностью соответствует его оригинальному дизайну. Заинтересованному читателю рекомендуется взглянуть на детали, которые можно найти в [Castro et al., 2002b].

Проблемы с производительностью в оверлеях

Из приведенного выше высокоуровневого описания должно быть ясно, что хотя само по себе построение дерева не так уж сложно, после того как мы организовали узлы в оверлей, построение эффективного дерева может стать

другой историей. Обратите внимание, что в нашем описании пока что выбор узлов, которые участвуют в дереве, не учитывает какие-либо метрики производительности: оно основано исключительно на (логической) маршрутизации сообщений через оверлей.

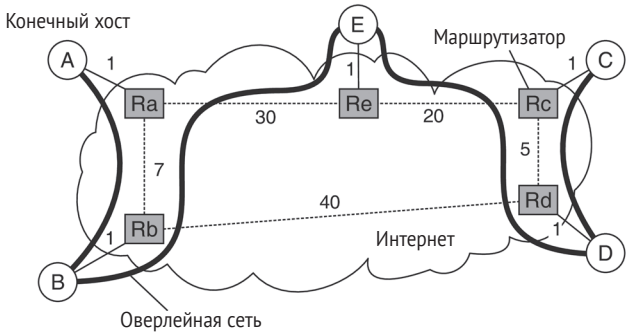


Рис. 4.35 ❖ Соотношение между ссылками в оверлейной сети и фактическими маршрутами сетевого уровня

Чтобы понять возникшую проблему, взгляните на рис. 4.35, на котором показан небольшой набор из пяти узлов, организованных в простой оверлейной сети, причем узел A образует корень дерева многоадресной рассылки. Также показаны затраты на обход физической связи. Теперь, когда A многоадресно передает сообщение на другие узлы, видно, что это сообщение будет проходить дважды по каждой из линий (B,Rb), (Ra,Rb), (E,Re), (Rc,Rd) и (D,Rd). Оверлейная сеть была бы более эффективной, если бы мы не создавали оверлейные ссылки (BE) и (DE), а вместо этого создали ссылки (AE) и (CE). Такая конфигурация спасла бы двойной обход физических связей (Ra,Rb) и (Rc,Rd).

Качество многоадресного дерева на уровне приложения обычно измеряется тремя различными показателями: напряжение линии связи, ее протяжение и стоимость дерева. **Напряжение линии связи** (Link stress) определяется для каждой линии связи и подсчитывает, как часто пакет пересекает одну и ту же линию связи [Chu et al., 2002]. Напряжение линии связи больше 1 происходит из-за того факта, что хотя на логическом уровне пакет может пересылаться по двум разным соединениям, часть этих соединений может фактически соответствовать одному и тому же физическому каналу, как мы показали на рис. 4.35.

Протяженность, или **штраф за относительную задержку** (Relative Delay Penalty, RDP), измеряет отношение задержки между двумя узлами в оверлее и задержку, которую эти два узла будут испытывать в базовой сети. Например, сообщения от B до C будут следовать по маршруту B → Rb → Ra → Re → E → Re → Rc → Rd → D → Rd → Rc → C в оверлейной сети, имеют общую стоимость 73 единицы. Вместе с тем если бы сообщения были направлены в базовой сети по пути B → Rb → Rd → Rc → C, с общей стоимостью 47 единиц, это дало бы протяженность 1,55. Очевидно, что при построении оверлейной сети целью является минимизация агрегированной протяженности, или, аналогично, средней RDP, измеренной по всем парам узлов.

Наконец, **стоимость дерева** (tree cost) является глобальной метрикой, обычно связанной с минимизацией совокупных затрат на связь. Например, если стоимость линии принимается за задержку между двумя ее конечными узлами, то оптимизация стоимости дерева сводится к поиску минимального связующего дерева, в котором общее время распространения информации по всем узлам минимально.

Чтобы немного упростить ситуацию, предположим, что многоадресная группа имеет связанный и хорошо известный узел, который отслеживает узлы, присоединившиеся к дереву. Когда новый узел выдает запрос на присоединение, он связывается с этим **рандеву-узлом** (rendezvous node), чтобы получить (потенциально частичный) список членов. Цель состоит в том, чтобы выбрать лучшего члена, который может работать как родительский узел нового дерева. Кого следует выбрать? Есть много альтернатив, и различные предложения часто следуют за очень разными решениями.

Рассмотрим, например, многоадресную группу только с одним источником. В этом случае выбор лучшего узла очевиден: он должен быть источником (потому что в этом случае мы можем быть уверены, что протяжение будет равно 1). Однако, делая так, мы вводим топологию звезды с источником посередине. Несмотря на простоту, нетрудно понять, что источник может легко перегружаться. Другими словами, выбор узла обычно будет ограничиваться таким образом, что могут быть выбраны только те узлы, которые имеют k или меньше соседей, причем k является параметром проектирования. Это ограничение сильно усложняет алгоритм установления дерева, так как для хорошего решения может потребоваться реконфигурировать эту часть существующего дерева. В работе [Tan et al., 2003] дается обширный обзор и оценка различных решений этой проблемы.

Примечание 4.14 (дополнительно: деревья переключения)

В качестве иллюстрации давайте более подробно рассмотрим одно конкретное семейство, известное как деревья переключателей [Helder and Jamin, 2002]. Основная идея проста. Предположим, у нас уже есть многоадресное дерево с одним источником в качестве корневого. В этом дереве узел P может переключать родителей, отбрасывая ссылку на своего текущего родителя в пользу ссылки на другой узел. Единственное ограничение, накладываемое на переключение ссылок, заключается в том, что новый родитель никогда не может быть членом поддерева с корнем в P (так как это разделит дерево и создаст цикл) и что у нового родителя не будет слишком много непосредственных потомков. Это последнее требование необходимо для ограничения нагрузки пересылки сообщений любым отдельным узлом.

Существуют разные критерии для принятия решения о смене родителей. Простым является оптимизация маршрута к источнику, эффективно минимизируя задержку, когда сообщение должно быть многоадресным. С этой целью каждый узел регулярно получает информацию о других узлах (ниже мы объясним один конкретный способ сделать это). В этот момент узел может оценить, является ли другой узел лучшим родителем с точки зрения задержки на пути к источнику, и, если это так, инициирует переключение.

Другим критерием может быть, является ли задержка для потенциального другого родителя ниже, чем для текущего родителя. Если каждый узел принимает это за критерий, то совокупные задержки результирующего дерева в идеале должны быть

минимальными. Другими словами, это пример оптимизации стоимости дерева, как мы объяснили выше. Однако для построения такого дерева потребуется больше информации, но, как выясняется, эта простая схема является разумной эвристикой, приводящей к хорошему приближению минимального остовного дерева.

В качестве примера рассмотрим случай, когда узел P получает информацию о соседях своего родителя. Обратите внимание, что соседи состоят из прародителя P наряду с другими братьями и сестрами родителя P. Узел P может затем оценить задержки для каждого из этих узлов и потом выбрать один с наименьшей задержкой, скажем Q, в качестве своего нового родителя. С этой целью он отправляет запрос на переключение в Q. Чтобы предотвратить образование петель из-за одновременных запросов на переключение, узел, который имеет невыполненный запрос на переключение, просто откажется обрабатывать любые входящие запросы. Фактически это приводит к ситуации, когда одновременно могут выполняться только полностью независимые переключатели. Кроме того, P предоставит Q достаточно информации, чтобы последние могли сделать вывод, что оба узла имеют одного и того же родителя или что Q является прародителем.

Важной проблемой, которую мы еще не решили, является сбой узла. В случае деревьев переключения предлагается простое решение: всякий раз, когда узел замечает, что его родительский узел вышел из строя, он просто присоединяется к корню. В этот момент протокол оптимизации может работать как обычно, и в конечном итоге узел окажется в хорошей точке дерева многоадресной рассылки. Эксперименты, описанные в [Helder and Jamin, 2002], показывают, что результирующее дерево действительно близко к минимальному остовному дереву.

Многоадресная передача сообщений на основе лавинной маршрутизации

До сих пор мы предполагали, что когда сообщение должно быть многоадресным, оно должно приниматься каждым узлом оверлейной сети. Строго говоря, это соответствует **вещанию** (broadcasting). В общем, групповая адресация относится к отправке сообщения подмножеству всех узлов, то есть **определенной группе узлов** (group of nodes). Ключевой вопрос проектирования, когда дело доходит до многоадресной рассылки, заключается в минимизации использования промежуточных узлов, для которых сообщение не предназначено. Если оверлей организован как многоуровневое дерево, но получать многоадресное сообщение должны только конечные узлы, то, очевидно, могут существовать некоторые узлы, которым необходимо сохранить и затем переслать сообщение, которое для них не предназначено.

Одним из простых способов избежать такой неэффективности является создание оверлейной сети для каждой *многоадресной группы* (*multicast group*). Как следствие многоадресная передача сообщения m в группу G такая же, как и в случае широковещательной передачи m в G. Недостаток этого решения состоит в том, что узлу, принадлежащему нескольким группам, в принципе потребуется вести отдельный список своих соседей для каждой группы, членом которой он является.

Если мы предполагаем, что оверлей соответствует многоадресной группе рассылки, и, следовательно, нам необходимо передать сообщение, простей-

шим способом для этого является применение **лавинной маршрутизации** (flooding). В этом случае каждый узел просто пересылает сообщение m каждому из своих соседей, за исключением того, от которого он получил m. Кроме того, если узел отслеживает полученные и отправленные им сообщения, он может просто игнорировать дубликаты. Мы увидим приблизительно в два раза больше отправляемых сообщений, чем ссылок в оверлейной сети, что делает такое кодирование совершенно неэффективным.

Чтобы оценить производительность потока, мы моделируем оверлейную сеть как связанный граф G с N узлами и M ребрами. Вспомним, что лавинная маршрутизация означает, что нам нужно отправить (как минимум) M сообщений. Только если G является деревом, заполнение будет оптимальным, поскольку в этом случае $M = N - 1$. В худшем случае, когда G полностью подключен, мы должны будем отправить $M = \binom{N}{2} = {}^1/_2 N(N-1)$ сообщений.

Предположим теперь, что у нас нет информации о структуре оверлейной сети и лучшее, что мы можем предположить, – это то, что она может быть представлена в виде **случайного графа** (random graph), который является графом, известным как граф Эрдеша–Реньи [Erdös and Rényi, 1959], в котором вероятность, что две вершины соединены ребром, равна p_{edge}. Обратите внимание, что мы на самом деле считаем нашу оверлейную сеть неструктурированной одноранговой сетью и что у нас нет никакой информации о том, как она создана. При вероятности, что два узла объединены, равной p_{edge}, и в общей сложности количестве ребер $\binom{N}{2}$ нетрудно видеть, что мы можем ожидать, что наш оверлей будет иметь $M = {}^1/_2 p_{edge} N(N-1)$ ребер. Чтобы составить представление о том, с чем мы имеем дело, на рис. 4.36 показана взаимосвязь между количеством узлов и ребер для разных значений p_{edge}.

Для уменьшения количества сообщений мы также можем использовать **вероятностную лавинную маршрутизацию** (probabilistic flooding), представленную [Banaei-Kashani and Shahab, 2003] и формально проанализированную в [Oikonomou and Stavrakakis, 2007].

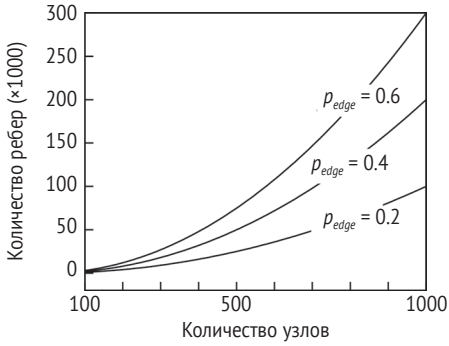


Рис. 4.36 ❖ Размер случайного оверлея как функции количества узлов

Идея очень проста: когда узел запрашивает сообщение `m` и ему нужно переслать `m` конкретному соседу, он будет делать это с вероятностью потока p_{flood}. Эффект может быть драматичным: общее количество отправленных сообщений будет линейно уменьшаться относительно p_{flood}. Однако есть и риск: чем ниже коэффициент, тем выше вероятность того, что не все узлы в сети будут достигнуты. Этот риск вызван тем простым фактом, что все соседи определенного узла `Q` решили не пересылать `m` в `Q`. Если `Q` имеет n соседей, то это может произойти примерно с вероятностью $(1 - p_{flood})^n$. Очевидно, что число соседей играет важную роль в принятии решения о том, следует ли пересылать сообщение, и действительно, мы можем заменить статическую вероятность пересылки на такую, которая учитывает уровень соседа. Эта эвристика получила дальнейшее развитие и анализ в работе [Sereno и Gaeta, 2011]. Чтобы дать представление об эффективности вероятностного широковещания: в случайной сети из 10 000 узлов и $p_{edge} = 0{,}1$ нам нужно всего лишь определить $p_{flood} = 0{,}01$, чтобы установить более чем 50-кратное сокращение числа отправляемых сообщений по сравнению с полным потоком.

При работе со структурированным оверлеем, то есть с более-менее детерминированной топологией, проще разработать эффективные схемы кодирования. В качестве примера рассмотрим n-мерный гиперкуб, показанный на рис. 4.37 для случая $n = 4$, как обсуждалось в главе 2.

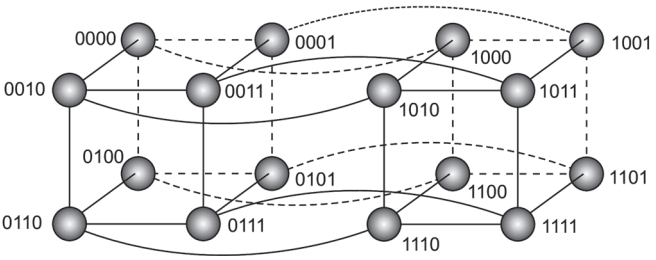


Рис. 4.37 ❖ Простая одноранговая система, организованная в виде четырехмерного гиперкуба

Простая и эффективная схема вещания была разработана в [Schlosser et al., 2002] и опирается на отслеживание соседей по измерению. Это легко понять, если представить каждый узел в n-мерном гиперкубе битовой строкой длины n. Каждый край оверлея помечен своим размером. Для случая $n = 4$ узел 0000 будет иметь в качестве своих соседей набор [0001, 0010, 0100, 1000]. Ребро между 0000 и 0001 помечено как «4», что соответствует изменению 4-го бита при сравнении 0000 с 0001, и *наоборот*. Аналогично ребро [0000, 0100] помечено как «2» и т. д. Узел первоначально передает сообщение m всем своим соседям и помечает `m` меткой ребра, по которому он отправляет сообщение. В нашем примере, если узел 1001 передает сообщение, оно отправит следующее:

- (m,1) к 0001;
- (m,2) к 1101;

- (m,3) к 1011;
- (m,4) к 1000.

Когда узел получает широковещательное сообщение, он будет пересылать его только по ребрам, которые имеют более высокое измерение. Другими словами, в нашем примере узел 1101 будет пересылать m только узлам 1111 (соединенным с 1101 ребром с меткой «3») и 1100 (соединенным ребром с меткой «4»). Используя эту схему, можно показать, что для каждой трансляции требуется ровно $N - 1$ сообщений, где $N = 2^n$, то есть количество узлов в n-мерном гиперкубе. Следовательно, эта схема вещания является оптимальной с точки зрения количества отправленных сообщений.

Примечание 4.15 (дополнительно: кольцевая лавинная маршрутизация)

Гиперкуб является ярким примером того, как мы можем эффективно использовать знания о структуре оверлейной сети для создания эффективной лавинной маршрутизации. В случае Chord мы можем следовать подходу, предложенному Годси [Ghodsi, 2010]. Напомним, что в Chord каждый узел идентифицируется номером p, а каждому ресурсу (обычно файлу) назначается ключ k из того же пространства, что и для идентификаторов узлов. Преемник succ(k) ключа k является узлом с наименьшим идентификатором p ≥ k. Рассмотрим небольшое кольцо Chord, показанное на рис. 4.38, и предположим, что узел 9 хочет передать сообщение всем остальным узлам.

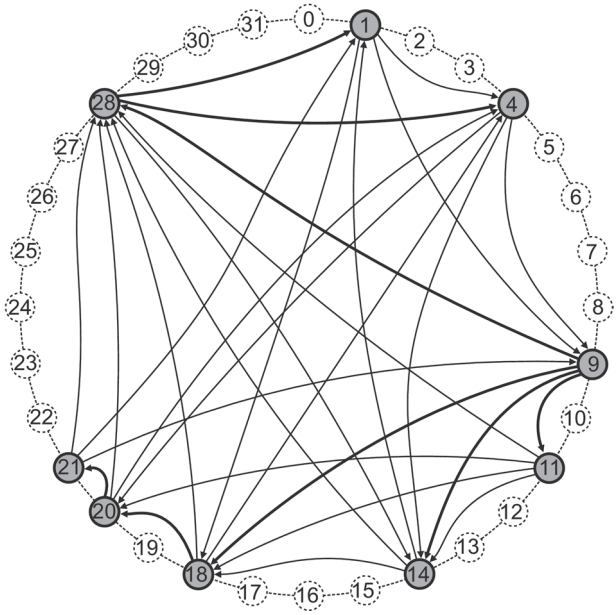


Рис. 4.38 ❖ Кольцо Chord, в котором сообщение передает узел 9

В нашем примере узел 9 делит пространство идентификаторов на четыре сегмента (по одному для каждого из его соседей). Узел 28 запрашивается, чтобы убедиться, что сообщение достигает всех узлов с идентификаторами 28 ≤ k < 9 (напомним, что

мы применяем арифметику по модулю); узел 18 заботится об узлах с идентификаторами 18 ≤ k < 28; узел 14 для 14 ≤ k < 18 и 11 для идентификаторов 11 ≤ k < 14.

Узел 28 впоследствии разделит часть пространства идентификатора, которое он должен обрабатывать, на два подсегмента: один для соседнего узла 1 и другой для 4. Аналогично узел 18, отвечающий за сегмент [18, 28), «разделит» этот сегмент только на одну часть: у него есть только один сосед, которому делегируется поток, и перенаправит сообщение узлу 20, сообщив ему, что он должен обрабатывать сегмент [20, 28).

На последнем шаге должен работать лишь узел 20. Он пересылает сообщение узлу 21, говоря ему переслать его узлам, известным ему в сегменте [21, 28). Поскольку таких узлов больше нет, вещание завершается.

Как и в случае нашего примера с гиперкубом, мы видим, что лавинная маршрутизация выполняется с помощью сообщения $N - 1$, где N – это число узлов в системе.

Распространение данных по принципу сплетни

Важным методом распространения информации является опора на **эпидемическое поведение** (epidemic behavior), также называемое **сплетнями** (gossiping). Наблюдая за тем, как болезни распространяются среди людей, ученые исследовали возможность разработки простых методов распространения информации в очень больших распределенных системах. Основной целью **эпидемических протоколов** (epidemic protocols) является быстрое распространение информации среди большого количества узлов с использованием только локальной информации. Другими словами, отсутствует центральный компонент, с помощью которого координируется распространение информации.

Чтобы объяснить общие принципы этих алгоритмов, мы предполагаем, что все обновления для конкретного элемента данных инициируются на одном узле. Таким образом, мы просто избегаем конфликтов записи-перезаписи. Следующая презентация основана на классической работе [Demers et al., 1987] об эпидемических алгоритмах. Обзор эпидемического распространения информации можно найти в [Eugster et al., 2004].

Модели распространения информации

Как следует из названия, алгоритмы эпидемии основаны на теории эпидемии, которая изучает распространение инфекционных заболеваний. В случае крупномасштабных распределенных систем вместо распространения болезней они распространяют информацию. Исследования в области эпидемий для распределенных систем также направлены на достижение совершенно иной цели: в то время как организации здравоохранения будут делать все возможное для предотвращения распространения инфекционных заболеваний среди больших групп людей, разработчики эпидемических алгоритмов для распределенных систем попытаются «заразить» все узлы новой информацией как можно быстрее.

Используя терминологию эпидемий, узел, являющийся частью распределенной системы, называется **зараженным** (infected), если он содержит

данные, которые готов распространить на другие узлы. Узел, который еще не видел эти данные, называется **восприимчивым** (susceptible). Наконец, обновленный узел, который не хочет или не может распространять свои данные, считается **удаленным** (removed). Обратите внимание, что мы предполагаем, что можем отличить старые данные от новых, например потому, что они были помечены или версифицированы. В этом смысле говорят, что узлы распространяют обновления.

Популярная модель распространения – **антиэнтропия** (anti-entropy). В этой модели узел P выбирает другой узел Q случайным образом и впоследствии обменивается обновлениями с Q. Существует три подхода к обмену обновлениями:

1) P только загружает новые обновления из Q;
2) P только передает свои собственные обновления Q;
3) P и Q отправляют обновления друг другу (т. е. двухтактный подход).

Когда речь идет о быстро распространяющихся обновлениях, лишь принудительное обновление оказывается плохим выбором. Интуитивно это можно понять следующим образом. Во-первых, обратите внимание, что при использовании подхода только передачи обновления могут распространяться лишь зараженными узлами. Однако если много узлов заражены, вероятность того, что каждый из них выберет восприимчивый узел, относительно мала. Следовательно, есть вероятность, что определенный узел остается восприимчивым в течение длительного периода просто потому, что он не выбран зараженным узлом.

Напротив, подход, основанный на загрузке, работает намного лучше, когда многие узлы заражены. В этом случае распространение обновлений по существу инициируется восприимчивыми узлами. Скорее всего, такой узел свяжется с зараженным, чтобы впоследствии получать обновления, и также заразится.

Если заражен только один узел, обновления будут быстро распространяться на все узлы с использованием любой формы антиэнтропии, хотя двухтактный подход (push-pull) остается лучшей стратегией [Jelasity et al., 2007]. Определим **раунд** как остов периода, в котором каждый узел один раз проявил инициативу для обмена обновлениями со случайно выбранным другим узлом. Можно показать, что количество раундов для распространения одного обновления на все узлы составляет $\mathcal{O}(\log(N))$, где N – количество узлов в системе. Это, по существу, означает, что распространение обновлений происходит быстро, но, главное, масштабируется.

Примечание 4.16 (дополнительно: анализ антиэнтропии)

Простой и понятный анализ даст некоторое представление о том, как хорошо работает антиэнтропия. Рассмотрим систему с N узлами. Один из этих узлов инициирует распространение сообщения m на все другие узлы. Обозначим через p_i вероятность того, что узел P еще не получил m после i-го раунда. Мы выделяем три следующих случая:

1) при использовании подхода, основанного исключительно на извлечении, $p_{i+1} = (p_i)^2$: не только P еще не был обновлен в предыдущем раунде, но и контакты узла P еще не получили m;

2) при использовании чисто подхода передачи $p_{i+1} = p_i \cdot \left(1 - \frac{1}{N-1}\right)^{N(1-p_i)}$: опять же, P не должен был обновляться в предыдущем раунде, но также ни один из обновленных узлов не должен связываться с P. Вероятность того, что узел связывается с P, равна $1 - 1/(N - 1)$; мы можем ожидать, что в раунде i будет $N(1 - p_i)$ обновленных узлов;
3) в двухтактном подходе мы можем просто объединить оба: P не должен связываться с обновленным узлом, и не тот должен связываться с ним.

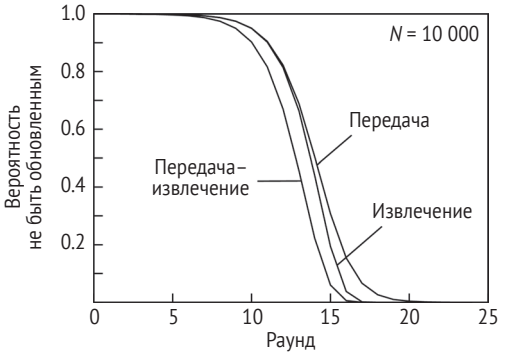


Рис. 4.39 ❖ Вероятность того, что информация еще не была обновлена в зависимости от количества раундов распространения

На рис. 4.39 показано, как быстро падает вероятность того, что узлы еще не обновлены как функция количества раундов. Действительно, если предположить, что узлы постоянно работают, получается, что антиэнтропия является чрезвычайно эффективным протоколом распространения.

Один конкретный вариант эпидемических протоколов называется **распространением слухов** (rumor spreading). Он работает следующим образом: если узел P был только что обновлен для элемента данных x, он связывается с произвольным другим узлом Q и пытается отправить обновление на Q. Однако возможно, что Q уже был обновлен другим узлом. В этом случае P может потерять интерес к дальнейшему распространению обновления, скажем, с вероятностью p_{stop}. Другими словами, тогда он удаляется. Распространение слухов – это сплетни, аналогичные таковым в реальной жизни. Когда у Боба появятся какие-то горячие новости, он может позвонить своей подруге Алисе и рассказать ей об этом. Алиса, как и Боб, будет очень рада распространить слух и своим друзьям. Вместе с тем, позвонив другу, скажем Чаку, она будет разочарована, узнав, что новость уже дошла до него. Скорее всего, она перестанет звонить другим друзьям. Что толку, если они уже знают.

Распространение слухов оказывается отличным способом быстрого распространения новостей. Однако это не может гарантировать, что все узлы будут фактически обновлены [Demers et al., 1987]. Когда в эпидемиях участвует большое количество узлов, доля узлов, которые будут оставаться в не-

ведении об обновлении, то есть оставаться восприимчивыми, удовлетворяет уравнению:

$$s = e^{-(1/p_{stop}+1)(1-s)}.$$

Примечание 4.17 (дополнительно: анализ распространения слухов)

Для того чтобы формально проанализировать ситуацию с распространением слухов, мы обозначим через s долю узлов, которые еще не были обновлены, то есть долю восприимчивых узлов. Аналогично доля зараженных узлов – те, которые были обновлены и все еще связываются с другими узлами для распространения новостей. Наконец, r – это доля узлов, которые были обновлены и отказались, то есть они больше не играют роли в распространении новостей. Очевидно, $s + i + r = 1$. Используя теорию эпидемий, нетрудно установить следующее:

$$\begin{aligned}
(1)\quad & ds/dt &&= -s \cdot i;\\
(2)\quad & di/dt &&= s \cdot i - p_{stop} \cdot (1 - s) \cdot i;\\
\Rightarrow\quad & di/ds &&= -(1 + p_{stop}) + \frac{p_{stop}}{s};\\
\Rightarrow\quad & i(s) &&= -(1 + p_{stop}) \cdot s + p_{stop} \cdot \ln(s) + C,
\end{aligned}$$

где мы используем обозначение $i(s)$, чтобы выразить i как функцию от s. Когда s = 1, ни один узел еще не был заражен, что означает, что $i(1) = 0$. Это позволяет нам получить C = 1 + p_{stop} и, таким образом,

$$i(s) = (1 + p_{stop}) \cdot (1 - s) + p_{stop} \cdot \ln(s).$$

Мы ищем ситуацию, когда больше нет распространения слухов, то есть когда $i(s) = 0$. Наличие закрытого выражения для $i(s)$ приводит к

$$s = e^{-(1/p_{stop}+1)(1-s)}.$$

Чтобы получить представление о том, что это значит, взгляните на рис. 4.40, который показывает s как функцию p_{stop}. Даже для больших значений p_{stop} доля узлов, которые остаются в неведении, относительно мала и всегда меньше, чем приблизительно 0,2. Для $p_{stop} = 0{,}20$ можно показать, что $s = 0{,}0025$. Однако в тех случаях, когда p_{stop} является относительно высоким, потребуются дополнительные меры для обеспечения обновления всех узлов.

Одним из основных преимуществ эпидемических алгоритмов является их масштабируемость благодаря тому, что количество синхронизаций между процессами относительно невелико по сравнению с другими методами распространения. Лин и Марзулло [Lin and Marzullo, 1999] показали, что для глобальных систем имеет смысл принимать во внимание фактическую топологию сети для достижения лучших результатов. В этом случае узлы, которые связаны только с несколькими другими узлами, связываются с относительно высокой вероятностью. Основное предположение состоит в том, что такие узлы образуют мост к другим удаленным частям сети; следовательно, с ними следует связаться как можно скорее. Этот подход называется **направленными сплетнями** (directional gossiping) и существует в разных вариантах.

Данная проблема касается имеющего место у большинства эпидемических решений важного предположения, что узел может случайным образом вы-

бирать для сплетен любой другой узел, и означает, что, в принципе, каждому члену должен быть известен полный набор узлов. В большой системе это предположение никогда не может быть выполнено, и необходимо принять специальные меры для имитации таких свойств. Мы вернемся к этому вопросу в разделе 6.7, когда будем обсуждать сервис одноранговой выборки.

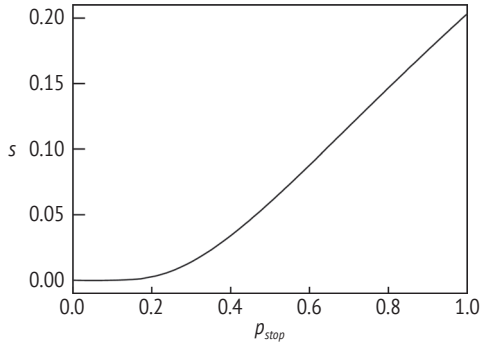


Рис. 4.40 ❖ Соотношение между долей необновленных узлов и вероятностью p_{stop} того, что узел прекратит передачу, как только он свяжется с узлом, который уже был обновлен

Удаление данных

Эпидемические алгоритмы чрезвычайно хороши для распространения обновлений. Однако они имеют довольно странный побочный эффект: сложно распространять удаление элемента данных. Суть проблемы заключается в том, что удаление элемента данных уничтожает всю информацию об этом элементе. Следовательно, когда элемент данных просто удаляется из узла, этот узел в конечном итоге получает старые копии элемента данных и интерпретирует их как обновления того, чего у него раньше не было.

Хитрость заключается в том, чтобы записать удаление элемента данных как очередное обновление и сохранить запись об этом удалении. Таким образом, старые копии не будут интерпретироваться как нечто новое, а просто будут рассматриваться как версии, которые были обновлены операцией удаления. Запись об удалении делается путем распространения **свидетельства о смерти** (death certificates).

Конечно, проблема со свидетельствами о смерти заключается в том, что их в конечном итоге следует удалить, иначе каждый узел будет постепенно создавать огромную локальную базу данных исторической информации об удаленных элементах данных, которая не используется. В работе [Demers et al., 1987] предлагают использовать так называемые спящие сертификаты смерти (dormant death certificates). Каждый сертификат смерти помечается временем, когда он создается. Если можно предположить, что обновления распространяются на все узлы в течение известного конечного времени, то по истечении этого максимального времени распространения могут быть удалены свидетельства о смерти.

Однако, чтобы обеспечить надежные гарантии того, что удаления действительно распространятся на все узлы, только очень немногие узлы поддерживают неактивные, спящие сертификаты смерти, которые никогда не удаляются. Предположим, что узел P имеет такой сертификат для элемента данных *x*. Если по какой-либо причине устаревшее обновление для *x* достигнет P, то P отреагирует, просто снова распространив свидетельство о смерти для *x*.

4.5. Резюме

Наличие мощных и гибких средств связи между процессами крайне важно для любой распределенной системы. В традиционных сетевых приложениях связь часто основана на низкоуровневых примитивах передачи сообщений, предлагаемых транспортным уровнем. Важной проблемой в системах промежуточного программного обеспечения является обеспечение более высокого уровня абстракции, который облегчит взаимодействие между процессами, чем поддержка, предоставляемая интерфейсом на транспортном уровне.

Одна из наиболее широко используемых абстракций – удаленный вызов процедур (RPC). Суть RPC заключается в том, что служба реализуется с помощью процедуры, тело которой выполняется на сервере. Клиенту предлагается только подпись процедуры, то есть название процедуры и ее параметры. Когда клиент вызывает процедуру, реализация на стороне клиента, называемая заглушкой, заботится о том, чтобы упаковать значения параметров в сообщение и отправить их на сервер. Последний вызывает фактическую процедуру и возвращает результаты снова в сообщении. Заглушка клиента извлекает значения результата из ответного сообщения и передает его обратно вызывающему клиентскому приложению.

RPC предлагают синхронные средства связи, с помощью которых клиент блокируется, пока сервер не отправит ответ. Хотя существуют вариации обоих механизмов, с помощью которых эта строгая синхронная модель ослаблена, оказывается, что универсальные высокоуровневые модели, ориентированные на сообщения, часто более удобны.

В моделях, ориентированных на сообщения, проблемы заключаются в том, является ли связь постоянной и синхронной. Суть постоянного общения заключается в том, что сообщение, которое подлежит передаче, сохраняется системой связи столько времени, сколько требуется для его доставки. Другими словами, ни отправитель, ни получатель не должны быть в рабочем состоянии для передачи сообщения. При переходной связи средства хранения не предлагаются, поэтому получатель должен быть готов принять сообщение при его отправке.

При асинхронной связи отправителю разрешается немедленно продолжить работу, после того как сообщение было отправлено для передачи, возможно, даже до того, как оно было отправлено. При синхронной связи отправитель блокируется, по крайней мере до получения сообщения. В качестве альтернативы отправитель может быть заблокирован до тех пор, пока не

произойдет доставка сообщения, или даже до тех пор, пока получатель не ответит, как при RPC.

Модели промежуточного программного обеспечения, ориентированные на сообщения, обычно предлагают постоянную асинхронную связь и используются там, где RPC не подходят. Они часто применяются для содействия интеграции (широко рассредоточенных) коллекций баз данных в крупные информационные системы.

Наконец, важным классом протоколов связи в распределенных системах является многоадресная передача. Основная идея заключается в распространении информации от одного отправителя к нескольким получателям. Мы обсудили два разных подхода. Во-первых, многоадресная рассылка может быть достигнута путем настройки дерева от отправителя к получателям. Учитывая, что теперь хорошо понятно, как узлы могут самостоятельно организовываться в одноранговую систему, также появились решения для динамической настройки деревьев децентрализованным способом. Во-вторых, поток сообщений по сети чрезвычайно устойчив, но требует особого внимания, если мы хотим избежать серьезной траты ресурсов, поскольку узлы могут видеть сообщения несколько раз. Вероятностное кодирование, с помощью которого узел пересылает сообщение с определенной вероятностью, часто сочетает в себе простоту и эффективность, будучи даже высокоэффективным.

Другой важный класс решений по распространению использует эпидемические протоколы. Эти протоколы оказались очень простыми и чрезвычайно надежными. Помимо простого распространения сообщений, эпидемические протоколы также могут быть эффективно развернуты для агрегирования информации в большой распределенной системе.

Глава 5

Присваивание имен

Имена играют важную роль во всех компьютерных системах. Они нужны для совместного использования ресурсов, уникальной идентификации объектов, ссылки на местоположения и т. д. Важной проблемой присваивания имен является то, что имя может быть разрешено только для объекта, к которому оно относится. Разрешение (различие) имен, таким образом, позволяет процессу получить доступ к названной сущности. Для селекции имен необходимо реализовать систему имен. Разница между наименованием в распределенных и нераспределенных системах заключается в том, как реализуются системы наименования.

В распределенной системе реализация системы наименования сама часто распределяется по нескольким машинам. То, как осуществляется это распределение, играет ключевую роль в эффективности и масштабируемости системы наименования. В этой главе мы сосредоточимся на трех разных важных способах использования имен в распределенных системах.

Сначала рассмотрим так называемые **бесструктурные системы имен** (flat-naming systems). В таких системах объекты обозначаются идентификатором, который, в принципе, вообще не имеет никакого значения. Кроме того, такие имена не имеют структуры, что означает, что нам нужны специальные механизмы для отслеживания местоположения таких объектов. Мы обсуждаем различные подходы, начиная от цепочек пересылочных ссылок до распределенных хеш-таблиц и заканчивая службами иерархического определения местоположения.

На практике люди предпочитают использовать читаемые имена. Такие имена структурированы, как хорошо известно, например, из названий вебстраниц. Структурированные имена обеспечивают систематический способ поиска сервера, ответственного за именованный объект, как, например, в системе доменных имен. Мы обсуждаем общие принципы, а также вопросы масштабируемости.

Наконец, люди часто предпочитают описывать объекты с помощью различных характеристик, что приводит к ситуации, в которой нам необходимо различать описание с помощью назначенных объектам атрибутов. Как мы увидим, этот тип различения имен весьма труден, особенно в сочетании с поиском.

5.1. Имена, идентификаторы и адреса

Давайте начнем с более внимательного изучения того, что такое имя на самом деле. Имя в распределенной системе – это строка битов или символов, которая используется для ссылки на объект. Объектом (сущностью) в распределенной системе практически может быть что угодно. Типичные примеры включают такие ресурсы, как хосты, принтеры, диски и файлы. Другими хорошо известными примерами сущностей, которые часто называют явно, являются процессы, пользователи, почтовые ящики, группы новостей, вебстраницы, графические окна, сообщения, сетевые соединения и т. д.

Объектами можно оперировать. Например, такой ресурс, как принтер, предлагает интерфейс, содержащий операции для печати документа, запроса статуса задания на печать и т. п. Такой объект, как сетевое соединение, может предоставлять операции для отправки и приема данных, установки параметров качества обслуживания, запроса статуса и т. д.

Для работы с объектом необходимо получить доступ к нему, для чего нам нужна точка доступа (access point). Точка доступа – это еще один, но особый вид объекта-сущности в распределенной системе. Название точки доступа называется адресом. Адрес точки доступа объекта также называется просто **адресом** (address) этого объекта.

Объект может предложить более одной точки доступа. Для сравнения, телефон можно рассматривать как точку доступа человека, тогда как номер телефона соответствует адресу. Действительно, многие люди в настоящее время имеют несколько телефонных номеров, каждый из которых соответствует точке, в которой они могут быть найдены. В распределенной системе типичным примером точки доступа является хост, на котором запущен конкретный сервер, и его адрес сформирован комбинацией, например, IP-адреса и номера порта (то есть адреса транспортного уровня сервера).

С течением времени объект может изменить свои точки доступа. Например, когда мобильный компьютер перемещается в другое место, ему часто присваивается иной IP-адрес, чем тот, который он имел раньше. Аналогичным образом, когда человек переезжает в другой город или страну, часто также необходимо изменить номера телефонов. Также смена работы или интернет-провайдера означает изменение вашего адреса электронной почты.

Таким образом, адрес – это просто особый вид имени: он относится к точке доступа объекта. Поскольку точка доступа тесно связана с объектом, может показаться удобным использовать адрес точки доступа в качестве обычного имени для ассоциированного объекта. Тем не менее это вряд ли когда-либо делается, так как такое наименование, как правило, очень негибкое и часто недружественное человеку.

Например, неудобно регулярно проводить реорганизацию распределенной системы, чтобы конкретный сервер теперь работал на хосте, отличном от предыдущего. Прежняя машина, на которой работал сервер, может быть переназначена на совершенно иной сервер. Другими словами, объект может легко изменить точку доступа, или точка доступа может быть переназна-

чена другому объекту. Если адрес используется для ссылки на объект, мы будем иметь недействительную ссылку в тот момент, когда точка доступа изменяется или переназначается другому объекту. Поэтому гораздо лучше, чтобы служба была известна под отдельным именем независимо от адреса связанного сервера.

Аналогичным образом, если объект предлагает более одной точки доступа, не ясно, какой адрес использовать в качестве ссылки. Например, многие организации распределяют свои веб-службы по нескольким серверам. Если бы мы использовали адреса этих серверов в качестве ссылки для веб-службы, было бы неочевидно, какой адрес следует выбрать в качестве лучшего. Опять же, гораздо лучшим решением будет иметь одно имя для веб-службы, независимое от адресов различных веб-серверов.

Эти примеры показывают, что имя объекта, которое не зависит от его адресов, часто в использовании намного проще и более гибко. Такое имя называется **независимым от местоположения** (location independent).

Помимо адресов, существуют другие типы имен, которые заслуживают особого отношения, например имена, использующиеся для уникальной идентификации объекта. **Истинный идентификатор** (true identifier) – это имя, обладающее следующими свойствами [Wieringa and de Jonge, 1995]:

- идентификатор относится не более чем к одному объекту;
- на каждый объект ссылается не более одного идентификатора;
- идентификатор всегда относится к одному и тому же объекту (то есть он никогда не используется повторно).

Используя идентификаторы, становится намного проще однозначно ссылаться на объект. Например, предположим, что два процесса каждый ссылаются на объект посредством идентификатора. Чтобы проверить, ссылаются ли процессы на один и тот же объект, достаточно проверить, равны ли два идентификатора. Такой тест не был бы достаточным, если бы два процесса использовали обычные, неуникальные, неидентифицирующие имена. Например, имя «Джон Смит» нельзя воспринимать как уникальную ссылку только на одного человека.

Аналогично, если адрес может быть переназначен другому объекту, мы не можем использовать адрес в качестве идентификатора. Рассмотрим использование телефонных номеров, которые достаточно стабильны в том смысле, что телефонный номер часто в течение некоторого времени относится к одному и тому же лицу или организации. Однако использование номера телефона в качестве идентификатора не будет работать, так как он может быть переназначен с течением времени. Следовательно, в новую пекарню Боба могут поступать телефонные звонки в старый антикварный магазин Алисы. В этом случае было бы лучше использовать истинный идентификатор Алисы вместо ее номера телефона.

Адреса и идентификаторы являются двумя важными типами имен, каждое из которых используется для самых разных целей. Во многих компьютерных системах адреса и идентификаторы представлены только в машиночитаемой форме, то есть в форме битовых строк. Например, адрес сети Ethernet по сути является случайной строкой из 48 бит. Адреса памяти обычно представлены в виде 32-битных или 64-битных строк.

Другим важным типом имени является то, которое специально предназначено для использования людьми, также называемое **именем, дружественными человеку** (human-friendly names). В отличие от адресов и идентификаторов, понятное человеку имя обычно представляется в виде строки символов. Эти имена появляются в разных формах. Например, файлы в системах Unix имеют имена символьных строк, которые обычно могут быть длиной до 255 символов и которые полностью определяются пользователем. DNS-имена представлены в виде относительно простых строк символов без учета регистра.

Наличие имен, идентификаторов и адресов приводит нас к центральной теме этой главы: как мы различаем имена и идентификаторы по адресам? Прежде чем перейти к различным решениям, важно понять, что часто существует тесная связь между различием имен в распределенных системах и маршрутизацией сообщений [Shoch, 1978]. В принципе, система именования поддерживает **привязку имени к адресу** (name-to-address binding), которая в простейшем виде представляет собой просто таблицу пар (имя, адрес). Однако в распределенных системах, которые охватывают большие сети и для которых необходимо назвать много ресурсов, централизованная таблица не будет работать.

Вместо этого имя часто разбивается на несколько частей, таких как ftp.cs.vu.nl, и имена различаются посредством рекурсивного поиска этих частей. Например, клиент, знающий адрес FTP-сервера с именем ftp.cs.vu.nl, сначала разрешил бы nl найти сервер NS(nl), отвечающий за имена, оканчивающиеся на nl, после чего остальная часть имени передается на сервер NS(nl). Этот сервер может затем преобразовать имя vu в сервер NS(vu.nl), отвечающий за имена, оканчивающиеся на vu.nl, который может дополнительно обработать оставшееся имя ftp.cs. В конечном итоге это приводит к маршрутизации запроса разрешения имени как

NS(.) → NS(nl) → NS(vu.nl) → address of ftp.cs.vu.nl,

где NS(.) обозначает сервер, который может вернуть адрес NS(nl), также известный как **корневой сервер** (root server). NS(vu.nl) вернет фактический адрес FTP-сервера. Интересно отметить, что границы между разрешением имен и маршрутизацией сообщений начинают стираться.

Примечание 5.1 (дополнительная информация: информационная сеть)

Разрешение имен и маршрутизация сообщений играют центральную роль в **информационных сетях** (information-centric networking, ICN) [Ahlgren et al., 2012]. Этот тип сетей основан на принципе, согласно которому приложениям на самом деле не интересно знать, *где* хранится объект, а скорее, знать то, что они могут получить, когда это необходимо, копию, имея доступ к его местоположению. С этой целью с 2007 года было проведено много исследований по разработке альтернативы схемам адресации на основе хоста, которые распространены в современном интернете. В частности, основная идея заключается в том, что приложение может извлечь объект из сети, используя имя этого объекта. Сеть принимает это имя в качестве входных данных и впоследствии направляет запрос в соответствующее место, где хранится объект, чтобы вернуть копию запрашивающей стороне.

Ключом к успеху при таком подходе является, конечно, форма маршрутизации на основе имени, которая, по сути, является средством для преобразования имени в адрес, где должен быть найден связанный объект. Как происходит эта маршрутизация, зависит от организации имен. Для бесструктурных (плоских) имен могут потребоваться такие же решения, как, например, для распределенных хеш-таблиц, что мы обсудим в разделе 5.2. Структурированные имена могут быть эффективно разрешены с использованием иерархических решений, что мы обсудим в разделе 5.3.

Примечание 5.2 (дополнительная информация: самоопределяющиеся имена)
Если имя используется для ссылки на объект, как мы узнаем, что на самом деле получаем доступ к предполагаемому объекту? Другими словами, как мы можем гарантировать, что объект и его имя однозначно связаны друг с другом? В случае неизменяемых объектов данных существует простое решение: взять хеш объекта данных и использовать это значение в качестве имени. Когда объект данных извлекается, процесс получения может хешировать объект и посмотреть, соответствует ли значение хеш-функции тому, которое он использовал в качестве имени объекта. Это простая версия **сертификации имени самим объектом** (self-certifying name), которая может локально проверить, действительно ли оно принадлежит объекту, к которому относится.

Мы вернемся к безопасному присвоению имени в разделе 9.4.

В следующих разделах мы рассмотрим три разных класса систем наименования. Сначала мы посмотрим, как идентификаторы могут быть преобразованы в адреса. В этом случае мы также увидим пример, где разрешение имен фактически неотличимо от маршрутизации сообщений. После этого мы рассматриваем понятные для человека имена и описательные имена (то есть объекты, которые описываются совокупностью имен).

5.2. Бесструктурное (плоское) наименование

Выше мы объяснили, что идентификаторы удобны для уникального представления объектов. Во многих случаях идентификаторы – это просто случайные битовые строки, которые мы обычно называем неструктурированными, или **плоскими** (flat), именами. Важным свойством такого имени является то, что оно не содержит никакой информации о том, как найти точку доступа к связанному с ним объекту. Далее мы рассмотрим, как можно разрешить плоские имена, или, что то же самое, как мы можем найти объект, когда дан только его идентификатор.

Простые решения

Сначала рассмотрим два простых решения для определения местонахождения объекта. Оба этих решения в основном применимы только к локальным сетям. В данной среде они часто выполняют свою работу хорошо, делая их

простоту особенно привлекательной. Тем не менее использование широковещательных и прямых указателей создает проблемы масштабируемости. Широковещательную или многоадресную передачу сложно эффективно реализовать в крупных сетях, в то время как длинные цепочки прямых указателей создают проблемы с производительностью и восприимчивы к неработающим связям.

Широковещание

Рассмотрим распределенную систему, построенную на компьютерной сети, которая предлагает эффективные широковещательные средства. Как правило, такие средства предлагаются локальными сетями, в которых все машины подключены к одному кабелю или его логическому эквиваленту. В эту категорию также попадают локальные беспроводные сети.

Найти объект в такой среде просто: сообщение, содержащее идентификатор объекта, передается на каждую машину, и каждой машине предлагается проверить, есть ли у нее этот объект. Только машины, которые могут предложить точку доступа для объекта, отправляют ответное сообщение, содержащее адрес этой точки доступа.

Данный принцип используется в **протоколе разрешения адресов** интернета (Address Resolution Protocol, ARP) для определения адреса канала передачи данных машины, когда ему предоставляется только IP-адрес [Plummer, 1982]. По сути, машина транслирует пакет в локальной сети, спрашивая, кто является владельцем этого IP-адреса. Когда сообщение поступает на компьютер, получатель проверяет, должен ли он прослушивать запрошенный IP-адрес. Если это так, он отправляет ответный пакет, содержащий, например, свой адрес Ethernet.

Широковещание становится неэффективным, когда сеть растет. Сообщения о запросах не только расходуют пропускную способность сети, но, что еще серьезнее, слишком много хостов могут прерываться запросами, на которые они не могут ответить. Одним из возможных решений является переключение на групповое вещание, при котором запрос получает только ограниченная группа хостов. Например, сети Ethernet поддерживают групповую передачу на уровне канала передачи данных непосредственно в аппаратном обеспечении.

Групповое вещание также может использоваться для обнаружения объектов в двухточечных сетях. Например, интернет поддерживает групповое вещание на уровне сети, позволяя узлам присоединяться к определенной группе. Такие группы идентифицируются по групповому адресу. Когда хост отправляет сообщение по групповому адресу, сетевой уровень предоставляет услугу наилучшего качества для доставки этого сообщения всем членам группы. Эффективные реализации группового вещания в интернете обсуждаются в [Deering and Cheriton, 1990] и [Deering et al., 1996].

Адрес групповой рассылки может использоваться как общая служба определения местоположения для нескольких объектов. Например, рассмотрим организацию, в которой у каждого сотрудника есть свой мобильный компьютер. Когда такой компьютер подключается к локальной сети, ему динамиче-

ски назначается IP-адрес. Кроме того, он присоединяется к определенной многоадресной группе. Когда процесс хочет найти компьютер A, он отправляет запрос многоадресной группе «где находится A?». Если A подключен, он отвечает своим текущим IP-адресом.

Другой способ использовать адрес групповой рассылки – связать его с реплицированным объектом и использовать групповую рассылку для определения местоположения ближайшей реплики. При отправке запроса на адрес групповой рассылки каждая реплика отвечает своим текущим (обычным) IP-адресом. Грубый способ выбора ближайшей реплики состоит в том, чтобы выбрать ту, чей ответ приходит первым, но, как оказалось, выбрать ближайшую реплику обычно не просто.

Прямые указатели

Другим популярным подходом к поиску мобильных объектов является использование прямых указателей [Fowler, 1985]. Принцип прост: когда объект перемещается из A в B, он оставляет в A ссылку на свое новое местоположение в B.

Основным преимуществом этого подхода является его простота: как только объект находится, например, с помощью традиционной службы наименования, клиент может искать текущий адрес, следуя цепочке указателей пересылки.

Есть и недостатки. Во-первых, если никакие специальные меры не предпринимаются, цепочка для высокомобильного объекта может стать настолько длинной, что определение местоположения этого объекта окажется непомерно дорогим. Во-вторых, все промежуточные местоположения в цепочке должны поддерживать свою часть цепочки указателей пересылки так долго, как это необходимо. Третий (и связанный с этим) недостаток – это уязвимость к неработающим ссылкам. Как только любой указатель пересылки потерян, объект больше не будет доступен. Следовательно, важной проблемой является сохранение относительно коротких цепочек и обеспечение надежности указателей пересылки.

Примечание 5.3 (дополнительная информация: цепочки SSP)

Чтобы лучше понять, как работают указатели переадресации, рассмотрим их использование по отношению к удаленным объектам – объектам, доступ к которым можно получить с помощью удаленного вызова процедуры. Следуя подходу в **цепочках пунктов коммутации услуг** (service switching point, SSP) [Shapiro et al., 1992], каждый указатель пересылки реализован в виде пары (заглушка клиента, заглушка сервера), как показано на рис. 5.1 (отметим, что в исходной терминологии заглушка сервера называлась наследник (scion), приводящий к парам (заглушка, наследник), что объясняет его имя). Заглушка сервера содержит либо локальную ссылку на фактический объект, либо локальную ссылку на заглушку удаленного клиента для этого объекта.

Всякий раз, когда объект перемещается из адресного пространства A в B, он оставляет клиентскую заглушку на своем месте в A и устанавливает заглушку сервера, которая ссылается на клиентскую заглушку в B. Интересным аспектом этого подхода является то, что миграция полностью прозрачна для клиента. Единственное, что клиент видит в объекте, – это клиентская заглушка. Как и в какое место клиентская

заглушка направляет свои вызовы, скрыто от клиента. Также обратите внимание, что использование указателей переадресации не похоже на поиск адреса. Вместо этого запрос клиента направляется по цепочке к реальному объекту.

Теперь предположим, что процесс P_1 на рис. 5.1 передает свою ссылку процессу P_2. Передача ссылок осуществляется путем установки копии клиентской заглушки p′ в адресное пространство процесса P_2. Клиентская заглушка p′ ссылается на ту же заглушку сервера, что и p, поэтому механизм вызова перенаправления работает так же, как и раньше.

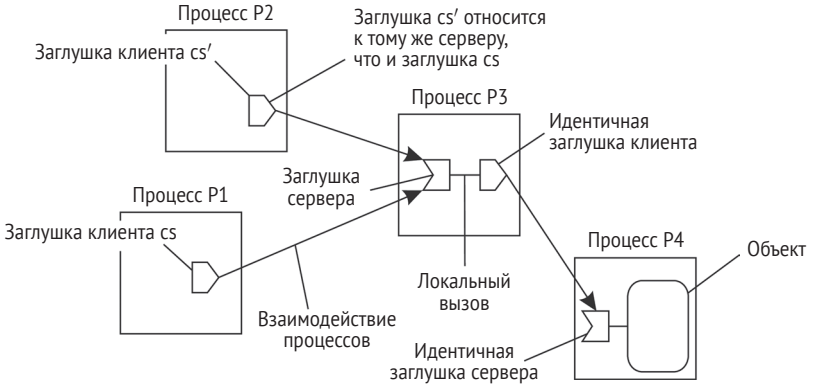


Рис. 5.1 ❖ Принцип пересылки указателей с использованием пары (*клиентская заглушка*, *серверная заглушка*)

Чтобы сократить цепочку пар (клиентская заглушка, серверная заглушка), вызов объекта несет идентификацию клиентской заглушки, с которой был инициирован этот вызов. Идентификация заглушки клиента состоит из адреса транспортного уровня клиента в сочетании с локально сгенерированным номером для идентификации этой заглушки. Когда вызов достигает объекта в его текущем местоположении, ответ отправляется обратно на заглушку клиента, где был инициирован вызов (часто без возврата по цепочке). Текущее местоположение совмещается с этим ответом, и клиентская заглушка настраивает свою заглушку сопутствующего сервера на ту, которая находится в текущем местоположении объекта. Этот принцип показан на рис. 5.2.

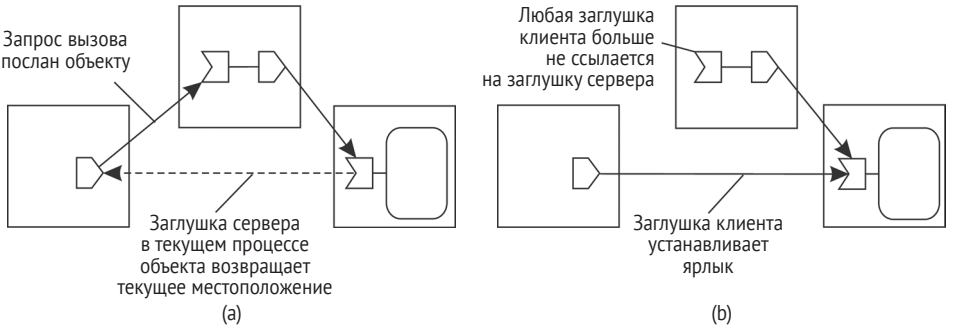


Рис. 5.2 ❖ Перенаправление указателя пересылки путем сохранения ярлыка в заглушке клиента

Существует компромисс между отправкой ответа непосредственно клиенту-заглушке-инициатору или по обратному пути указателей пересылки. В первом случае связь происходит быстрее, потому что может потребоваться пропускать меньше процессов. С другой стороны, можно настроить только исходную заглушку клиента, тогда как отправка ответа по обратному пути позволяет настроить все промежуточные заглушки.

Когда заглушка сервера больше не используется ни одним клиентом, ее можно удалить. Само по себе это тесно связано с распределенной сборкой мусора, которая, как правило, далеко не тривиальная проблема. Мы отсылаем заинтересованного читателя к [Abdullahi and Ringwood, 1998], [Plainfosse and Shapiro, 1995] и [Veiga and Ferreira, 2005].

Проблемы возникают, когда процесс в цепочке пар (*клиентская заглушка, серверная заглушка*) пропадает или становится недоступным по другим причинам. Возможно несколько решений. Одна из возможностей, как в системе Emerald [Jul et al., 1988] и системе LII [Black and Artsy, 1990], состоит в том, чтобы позволить машине, на которой был создан объект, называемой **домашним местоположением объекта**, (object's home location), всегда сохранять ссылку на его текущее местоположение. Эта ссылка хранится и поддерживается отказоустойчивым способом. Когда цепь разорвана, домашнее местоположение объекта спрашивает, где объект находится сейчас. Чтобы разрешить изменение домашнего местоположения объекта, можно использовать традиционную службу наименования для записи текущего домашнего местоположения.

Методы домашнего местоположения

Популярный подход к поддержке мобильных объектов в крупных сетях заключается во введении **домашнего местоположения** (home location), которое отслеживает текущее местоположение объекта. Специальные методы могут применяться для защиты от сбоев сети или процессов. На практике домашнее местоположение часто выбирается как место, где был создан объект.

Метод, основанный на домашнем местоположении, используется в качестве запасного механизма для служб определения местоположения на основе прямых указателей пересылки. Другой пример, в котором применяется этот метод, – «Мобильная IP» [Johnson et al., 2004], которую мы кратко пояснили в примечании 3.9. Каждый мобильный хост использует фиксированный IP-адрес. Вся связь с этим IP-адресом первоначально направляется **домашнему агенту** (home agent) мобильного хоста. Этот домашний агент расположен в локальной сети в соответствии с сетевым адресом, содержащимся в IP-адресе мобильного хоста. В случае IPv6 он реализован как компонент сетевого уровня. Всякий раз, когда мобильный хост перемещается в другую сеть, он запрашивает временный адрес, который может использовать для связи. Этот **временный адрес** (care of address) зарегистрирован у домашнего агента.

Когда домашний агент получает пакет для мобильного хоста, он ищет текущее местоположение хоста. Если хост находится в текущей локальной сети, пакет просто пересылается. В противном случае он туннелируется на текущее местоположение хоста, то есть упаковывается как данные в IP-пакет и отправляется по адресу для передачи. В то же время отправитель пакета

информируется о текущем местоположении хоста. Этот принцип показан на рис. 5.3. Обратите внимание, что IP-адрес эффективно используется в качестве идентификатора для мобильного хоста.

Рис. 5.3 ❖ Принцип мобильного IP

Важным аспектом является то, что весь этот механизм в значительной степени скрыт для приложений. Другими словами, исходный IP-адрес, связанный с мобильным хостом, может использоваться приложением без лишних слов. Клиентское программное обеспечение, которое является частью независимого от приложения уровня связи, будет обрабатывать перенаправление к текущему местоположению цели. Аналогично в расположении цели сообщение, которое было туннелировано, будет распаковано и передано приложению на мобильном хосте, как если бы оно использовало свой первоначальный адрес.

Mobile IP действительно обеспечивает высокую степень прозрачности местоположения.

Рисунок 5.3 также иллюстрирует недостаток домашних подходов в крупных сетях. Для связи с мобильным объектом клиент должен сначала связаться с домом, который может находиться в совершенно ином месте, чем сам объект. Результатом является увеличение задержки связи.

Другим недостатком домашнего подхода является использование фиксированного местоположения дома. Во-первых, необходимо убедиться, что домашнее местоположение всегда существует. В противном случае связаться с объектом станет невозможно. Проблемы усугубляются, когда долгоживущая сущность решает навсегда переместиться в совершенно другую часть сети, чем ее дом. В этом случае было бы лучше, если бы дом мог переехать вместе с хозяином.

Решением этой проблемы является регистрация дома в традиционной службе наименования и разрешение клиенту сначала найти местоположение дома. Поскольку можно предположить, что домашнее местоположение является относительно стабильным, это местоположение может быть эффективно кешировано после того, как оно было найдено.

Распределенные хеш-таблицы

Давайте теперь подробнее рассмотрим, как разрешить идентификатор по адресу ассоциированного объекта. Мы уже несколько раз упоминали о распределенных хеш-таблицах, но отложили обсуждение того, как они на самом деле работают. В данном разделе мы исправляем эту ситуацию, рассматривая систему Chord как простую для объяснения систему на основе **распределенных хеш-таблиц** (Distributed Hash Tables, DHT).

Общий механизм

Многие системы на основе DHT были разработаны в последнее десятилетие, и типичным представителем является система Chord [Stoica et al., 2003]. Chord использует m-битное пространство идентификаторов, чтобы назначать случайно выбранные идентификаторы узлам, а также ключи для конкретных объектов. Последний может быть практически любым: файлы, процессы и т. д. Число m бит обычно составляет 128 или 160, в зависимости от того, какая хеш-функция используется. Объект с ключом k подпадает под юрисдикцию узла с наименьшим идентификатором $id \geq k$. Этот узел называется **преемником** (successor) k и обозначается как $succ(k)$. Чтобы наши обозначения были простыми и непротиворечивыми, ниже мы ссылаемся на узел с идентификатором p как узел p.

Основной проблемой в системах на основе DHT является эффективное преобразование ключа k в адрес $succ(k)$. Очевидный немасштабируемый подход состоит в том, чтобы позволить каждому узлу p отслеживать преемника $succ(p + 1)$, а также его предшественника $pred(p)$. В этом случае всякий раз, когда узел p получает запрос на разрешение ключа k, он просто перенаправляет запрос одному из двух своих соседей – в зависимости от того, какой из них подходит, – пока $pred(p) < k \leq p$, в этом случае узел p должен возвращать свой адрес процесса, который инициировал разрешение ключа k.

Вместо этого линейного подхода к поиску ключа каждый узел Chord поддерживает таблицу указателей, содержащую $s \leq m$ записей. Если FT_p обозначает таблицу поиска узла p, то

$$FT_p[i] = succ(p + 2^{i-1}).$$

Другими словами, i-й вход указывает на первый узел, следующий за p как минимум на 2^{i-1}. Обратите внимание, что эти ссылки на самом деле являются ярлыками для существующих узлов в пространстве идентификаторов, где сокращенное расстояние от узла p увеличивается экспоненциально по мере увеличения индекса в таблице подстановок. Чтобы найти ключ k, узел p

затем немедленно направит запрос узлу q с индексом j в таблице идентификаторов p, где

$$q = \mathrm{FT}_p[j] \le k < \mathrm{FT}_p[j+1],$$

или $q = \mathrm{FT}_p[1]$, когда $p < k < \mathrm{FT}_p[1]$. (Для ясности мы игнорируем арифметику по модулю.) Обратите внимание, что когда размер таблицы указателей s равен 1, поиск Chord соответствует наивному обходу кольца линейно, как мы только что обсуждали.

Чтобы проиллюстрировать этот поиск, рассмотрим разрешение $k = 26$ для узла 1, как показано на рис. 5.4. Сначала узел 1 будет искать в своей таблице кодов $k = 26$, чтобы обнаружить, что это значение больше, чем $\mathrm{FT}_1[5]$, что означает, что запрос будет перенаправлен на узел $18 = \mathrm{FT}_1[5]$. Узел 18, в свою очередь, выберет узел 20 как $\mathrm{FT}_{18}[2] \le k < \mathrm{FT}_{18}[3]$. Наконец, запрос пересылается с узла 20 на узел 21 и оттуда на узел 28, который отвечает за $k = 26$. В этой точке адрес узла 28 возвращается узлу 1, и ключ был разрешен. По тем же причинам, когда узел 28 запрашивается для разрешения ключа $k = 12$, запрос будет маршрутизироваться, как показано пунктирной линией на рис. 5.4. Можно показать, что поиск обычно требует $\mathcal{O}(\log(N))$ шагов, где N – это число узлов в системе.

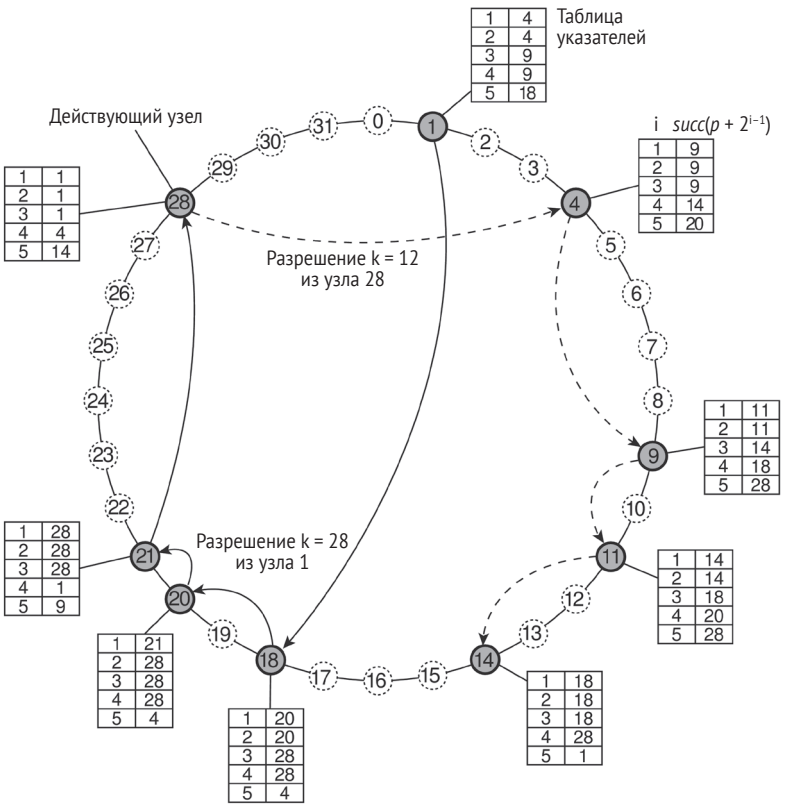


Рис. 5.4 ❖ Разрешение ключа 26 из узла 1 и ключа 12 из узла 28 в системе Chord

В больших распределенных системах можно ожидать, что совокупность участвующих узлов будет постоянно изменяться. Мало того, что узлы присоединяются и уходят добровольно, мы также должны учитывать случай сбоя узлов (и, следовательно, фактического выхода из системы), чтобы впоследствии восстановить их и снова присоединить к системе.

Присоединиться к системе, основанной на DHT, такой как Chord, относительно просто. Предположим, что узел p хочет присоединиться. Он просто связывается с произвольным узлом в существующей системе и запрашивает поиск *succ*(*p* +1). Как только этот узел был идентифицирован, p может вставить себя в кольцо. Аналогичным образом уход может быть таким же простым. Обратите внимание, что узлы также отслеживают своего предшественника.

Очевидно, что сложность заключается в том, что необходимо постоянно обновлять таблицы. Наиболее важным является то, что для каждого узла q $FT_q[1]$ является правильным, поскольку эта запись ссылается на следующий узел в кольце, то есть на преемника $q + 1$. Для достижения этой цели каждый узел q регулярно выполняет простую процедуру, которая связывается с $succ(q + 1)$ и просит вернуть $pred(succ(q + 1))$. Если $q = pred(succ(q + 1))$, то q знает, что его информация соответствует информации его преемника. В противном случае, если преемник q обновил своего предшественника, очевидно, в систему вошел новый узел p с $q < p \leq succ(q + 1)$, так что q настроит $FT_q[1]$ на p.

В этот момент он также проверит, записал ли p как своего предшественника q. Если нет, необходима другая настройка $FT_q[1]$.

Аналогичным образом, чтобы обновить таблицу поиска, узлу q просто нужно найти преемника для $k = q + 2^{i-1}$ для каждой записи i. Опять же, это можно сделать, выполнив запрос на разрешение $succ(k)$. В Chord такие запросы регулярно выдаются посредством фонового процесса.

Аналогично каждый узел q будет регулярно проверять, жив ли его предшественник. Если предшественник потерпел неудачу, единственное, что может сделать q, – это записать факт, установив $pred(q)$ значение «unknown» (неизвестно). С другой стороны, когда узел q обновляет свою ссылку на следующий известный узел в кольце и обнаруживает, что предшественник $succ(q + 1)$ был установлен в «unknown», он просто уведомит $succ(q + 1)$), что он подозревает, что это был предшественник. В целом эти простые процедуры гарантируют, что система Chord, как правило, является согласованной, возможно, за исключением нескольких узлов. Подробности можно найти в [Stoica et al., 2003].

Примечание 5.4 (дополнительно: Cord в Python)

Кодирование Chord в Python удивительно просто. Опять же, опуская многие из несущественных деталей кодирования, ядро поведения узла Chord можно описать, как показано на рис. 5.5. Функция `finger(i)` вычисляет *succ*(*i*) для данного узла. Все узлы, известные конкретному узлу Chord, собираются в локальном наборе nodeSet, который сортируется по идентификатору узла. Узел сначала ищет свою позицию в этом наборе и позицию своего правого соседа. Операция между (k, l, u) вычисляет, является ли $k \in [l, u)$ по модулю. Таким образом, вычисление промежуточных значений `inbetween(k,l+1,u+1)` аналогично проверке, является ли $k \in (l, u]$. Таким образом, мы видим, что `finger(i)` возвращает наибольший существующий идентификатор узла, меньший или равный *i*.

```
class ChordNode:
  def finger(self, i):
    succ = (self.nodeID + pow(2, i-1)) % self.MAXPROC      # succ(p+2^(i-1))
    lwbi = self.nodeSet.index(self.nodeID)                 # self в набор узлов
    upbi = (lwbi + 1) % len(self.nodeSet)                  # следующий сосед
    for k in range(len(self.nodeSet)):                     # обработка сегментов
      if self.inbetween(succ, self.nodeSet[lwbi]+1, self.nodeSet[upbi]+1):
        return self.nodeSet[upbi]                          # найденный преемник
      (lwbi,upbi) = (upbi, (upbi + 1) % len(self.nodeSet)) # следующий сегмент

  def recomputeFingerTable(self):
    self.FT[0] = self.nodeSet[self.nodeSet.index(self.nodeID)-1]  # Pred.
    self.FT[1:] = [self.finger(i) for i in range(1,self.nBits+1)] # Succ.

  def localSuccNode(self, key):
    if self.inbetween(key, self.FT[0]+1, self.nodeID+1): # в (FT[0],self]
      return self.nodeID                                 # ответственный узел
    elif self.inbetween(key, self.nodeID+1, self.FT[1]): # в (self,FT[1]]
      return self.FT[1]                                  # ответственный succ.
    for i in range(1, self.nBits+1):                     # остаток FT
      if self.inbetween(key, self.FT[i], self.FT[(i+1) % self.nBits]):
        return self.FT[i]                                # в [FT[i],FT[i+1])
```

Рис. 5.5 ❖ Существо узла Chord, выраженного в Python

Каждый раз, когда узел узнает о новом узле в системе (или обнаруживает, что он ушел), он просто настраивает локальный nodeSet и повторно вычисляет свою таблицу указателей, вызывая `recomputeFingerTable`. Сама таблица указателей реализована в виде локальной таблицы `FT`, где `FT[0]` указывает на предшественника узла. `nBits` показывает количество битов, используемых для идентификаторов узлов и ключей.

Ядро того, что делает узел во время поиска, закодировано в `localSuccNode(k)`. Когда поступает ключ *k*, он либо возвращает самого себя, своего непосредственного преемника `FT[1]`, либо просматривает таблицу поиска, чтобы найти запись, удовлетворяющую `FT[i]` ≤ *k* < `FT[i+ ]`. Код не показывает, что делается с возвращенным значением (которое является идентификатором узла), но, как правило, в *итерационной схеме* с указанным узлом свяжутся, чтобы продолжить поиск *k*, если только узел не вернул себе ответственности за *k*. В *рекурсивной схеме* сам узел будет связываться с указанным узлом.

Примечание 5.5 (дополнительно: использование близости к сети)

Одна из потенциальных проблем с такими системами, как Chord, заключается в том, что запросы могут беспорядочно маршрутизироваться через интернет. Например, предположим, что узел 1 на рис. 5.5 расположен в Амстердаме (Нидерланды), узел 18 в Сан-Диего (Калифорния), узел 20 снова в Амстердаме и узел 21 в Сан-Диего. В результате разрешения ключа 26 произойдут три глобальные передачи сообщений, которые, возможно, можно было бы уменьшить до одной. Чтобы свести к минимуму эти патологические случаи, при проектировании системы на основе DHT необходимо учитывать базовую сеть.

В работе [Castro et al., 2002a] различают три различных способа информирования системы на основе DHT о базовой сети. В случае **назначения идентификаторов**

узлов на основе топологии (topology-based assignment) идея состоит в том, чтобы назначать идентификаторы так, чтобы два соседних узла имели идентификаторы, которые также близки друг к другу. Нетрудно представить, что этот подход может создать серьезные проблемы даже в случае относительно простых систем, таких как Chord. В случае когда идентификаторы узлов выбираются из одномерного пространства, отображение логического кольца в интернет далеко не тривиально. Более того, такое отображение может легко выявить коррелированные сбои: узлы в одной сети предприятия будут иметь идентификаторы с относительно небольшим интервалом. Когда эта сеть становится недоступной, у нас внезапно возникает разрыв в равномерном распределении идентификаторов.

С помощью **маршрутизации по пространственной близости** (proximity routing) узлы поддерживают список альтернатив, на которые следует переслать запрос. Например, вместо единственного преемника каждый узел в Chord мог бы одинаково хорошо отслеживать r преемников. На самом деле эта избыточность может применяться для каждой записи в таблице подстановок. Для узла p $FT_p[i]$ обычно указывает на первый узел в диапазоне $[p + 2^{i-1}, p + 2^i - 1]$. Всякий раз, когда ему нужно найти ключ k, он пытается предотвратить «перескок», передавая запрос узлу q с $k < q$, не зная точно, существует ли узел q′ с $k \leq q' < q$. По этой причине p передает k в узел, известный как p с *наибольшим* идентификатором, меньшим или равным k.

Однако нет причины, по которой p не может отслеживать r узлов в диапазоне $[p + 2^{i-1}, p + 2^i - 1]$: каждый узел q в этом диапазоне может использоваться для маршрутизации запроса поиска для ключа k до тех пор, пока $q \leq k$. В этом случае при выборе пересылки запроса поиска узел может выбрать одного из r преемников, который находится к нему ближе всего, при этом следя за тем, чтобы не допустить «перескок». Дополнительным преимуществом наличия нескольких преемников для каждой записи таблицы является то, что сбои узла не обязательно должны сразу приводить к сбоям поисков, поскольку можно исследовать несколько маршрутов.

Наконец, при **выборе соседей по близости** (proximity neighbor selection) идея состоит в том, чтобы оптимизировать таблицы маршрутизации таким образом, чтобы в качестве соседа был выбран ближайший узел. Этот выбор работает только тогда, когда есть больше узлов для выбора. В Chord это обычно не так. Однако в других протоколах, таких как Pastry [Rowstron and Druschel, 2001], когда узел присоединяется, он получает информацию о текущем оверлее от множества других узлов. Эта информация используется новым узлом для построения таблицы маршрутизации. Очевидно, что когда есть альтернативные узлы на выбор, выбор близкого соседа позволит присоединяющемуся узлу выбрать лучший.

Обратите внимание, что может быть не так просто провести черту между маршрутизацией по близости и выбором соседей по близости. Фактически, когда Chord модифицируется так, чтобы включать r преемников для каждой записи таблицы поиска, выбор близких соседей превращается в идентификацию ближайших r соседей, что, как мы только что объяснили [Dabek et al., 2004b], очень похоже на маршрутизацию по близости.

Иерархические методы

Теперь мы обсудим общий подход к иерархической схеме местоположения, включая ряд оптимизаций. Подход, который мы представляем, основан на сервисе службы определения местоположения Globe [van Steen et al., 1998]. Подробное описание службы можно найти в [Ballintijn, 2003]. Это универсальная служба определения местоположения, представляющая многие

иерархические сервисы определения местоположения, предлагаемые для так называемых персональных систем связи, общий обзор которых можно найти в [Pitoura and Samaras, 2001].

В иерархической схеме сеть делится на набор **доменов** (domain). Существует один домен верхнего уровня, который охватывает всю сеть. Каждый домен может быть разделен на несколько меньших поддоменов. Домен самого низкого уровня, называемый **конечным (листовым доменом)** (leaf domain), обычно соответствует в компьютерной сети локальной сети или ячейке в сети мобильной телефонной связи. Общее предположение состоит в том, что в меньшем домене среднее время, необходимое для передачи сообщения от одного узла другому, меньше, чем в большом домене.

Каждый домен D имеет связанный каталог узла *dir*(D), который отслеживает объекты в этом домене. Это приводит к дереву узлов каталога. Узел каталога домена верхнего уровня, называемый **корневым узлом** (**каталогом**) (root (directory) node), знает обо всех объектах. Общая организация сети по доменам и узлам каталогов показана на рис. 5.6.

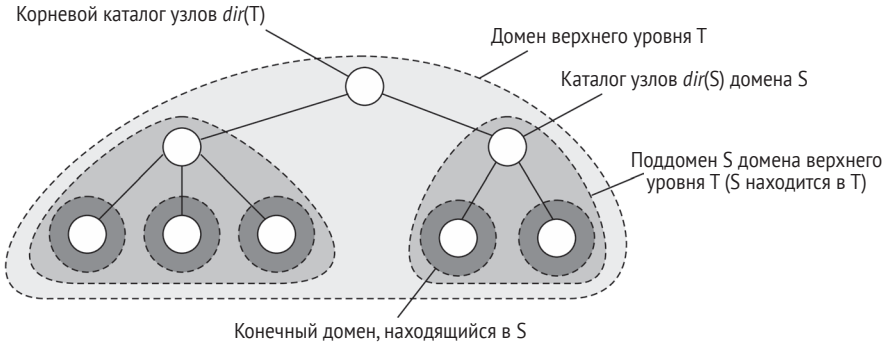


Рис. 5.6 ❖ Иерархическая организация службы определения местоположения по доменам, каждый из которых имеет связанный узел каталога

Чтобы отслеживать местоположение объекта, каждый объект, в настоящее время расположенный в домене D, представлен **записью местоположения** (location record) в узле каталога *dir*(D). Запись местоположения для объекта E в узле N каталога для конечного домена D содержит текущий адрес объекта в этом домене. Напротив, узел каталога N′ для следующего домена более высокого уровня D′, который содержит D, будет иметь запись местоположения для E, содержащую только указатель на N. Аналогично родительский узел N′ будет хранить запись местоположения для E, содержащую только указатель на N′. Следовательно, корневой узел будет иметь запись местоположения для каждого объекта, где каждая запись местоположения хранит указатель на узел каталога следующего поддомена, более низкого уровня, в котором в данный момент находится связанный объект этой записи.

Объект может иметь несколько адресов, например если он реплицируется. Если у объекта есть адрес в конечных доменах D_1 и D_2 соответственно, то узел каталога наименьшего домена, содержащий как D_1, так и D_2, и будет иметь два

указателя, по одному на каждый поддомен, содержащий адрес. Это приводит к общей организации дерева, как показано на рис. 5.7.

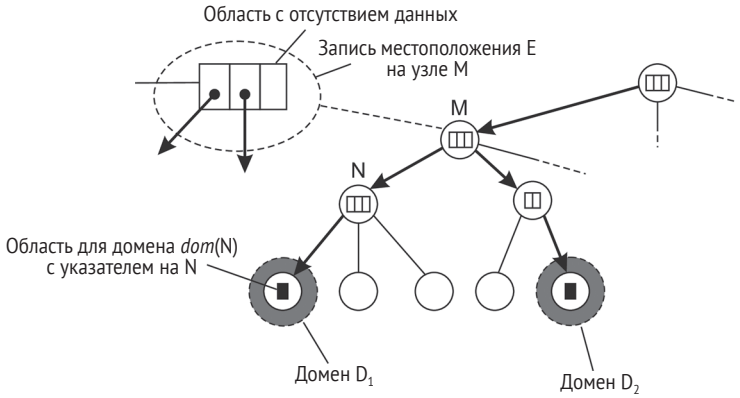


Рис. 5.7 ❖ Пример хранения информации об объекте, имеющем два адреса в разных конечных доменах

Давайте теперь рассмотрим, как происходит операция поиска в такой иерархической службе определения местоположения. Как показано на рис. 5.8, клиент, желающий найти объект E, выдает запрос поиска на узел каталога конечного домена D, в котором находится клиент. Если узел каталога не хранит запись местоположения для объекта, то объект в настоящее время не находится в D. Поэтому узел пересылает запрос своему родителю. Обратите внимание, что родительский узел представляет больший домен, чем его дочерний. Если родительский объект также не имеет записи местоположения для объекта E, запрос поиска пересылается на следующий уровень выше, и т. д.

Как только запрос достигает узла каталога M, который хранит местоположение записи для объекта E, мы знаем, что E находится где-то в домене *dom*(M), представленном узлом M. На рис. 5.8 показано, что M хранит запись местоположения, содержащую указатель на один из его поддоменов. Затем запрос поиска направляется на узел каталога этого поддомена, который, в свою очередь, направляет его дальше вниз по дереву, пока запрос не достигнет конечного узла. Запись местоположения, сохраненная в конечном узле, будет содержать адрес E в этом конечном домене.

Этот адрес затем может быть возвращен клиенту, который первоначально запросил выполнение поиска.

Важным обстоятельством в отношении иерархических служб определения местоположения является то, что операция поиска использует локальность. В принципе, объект ищется в постепенно увеличивающемся кольце, сосредоточенном вокруг запрашивающего клиента. Область поиска расширяется каждый раз, когда запрос поиска пересылается на следующий узел каталога более высокого уровня. В худшем случае поиск продолжается до тех пор, пока запрос достигает корневого узла. Поскольку корневой узел имеет запись местоположения для каждого объекта, запрос может быть затем просто переадресован по нисходящему пути указателей на один из конечных узлов.

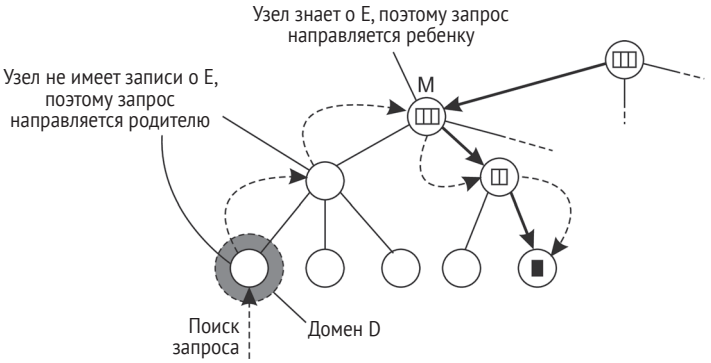


Рис. 5.8 ❖ Поиск местоположения в иерархически организованной службе определения местоположения

Операции обновления используют локальность аналогичным образом, как показано на рис. 5.9. Рассмотрим объект E, который создал реплику в конечном домене D, для которого ему нужно вставить свой адрес. Вставка инициируется в листовом узле *dir*(D) D, который немедленно перенаправляет запрос вставки своему родителю. Родитель также будет пересылать запрос вставки до тех пор, пока он не достигнет узла каталога M, который уже хранит запись местоположения для E.

Затем узел M сохранит указатель в записи местоположения для E, ссылаясь на дочерний узел, с которого был отправлен запрос на вставку. В этот момент дочерний узел создает запись местоположения для E, содержащую указатель на следующий узел нижнего уровня, откуда поступил запрос. Этот процесс продолжается до тех пор, пока мы не достигнем конечного узла, с которого была начата вставка. Конечный узел, наконец, создает запись с адресом объекта в связанном конечном домене.

Вставка только что описанного адреса приводит к установке цепочки указателей сверху вниз, начиная с узла каталога самого низкого уровня, который имеет запись местоположения для объекта E. Альтернативой является создание записи местоположения перед передачей запроса на вставку в родительский узел. Другими словами, цепочка указателей строится снизу вверх. Преимущество последнего состоит в том, что адрес становится доступным для поиска как можно скорее. В результате, если родительский узел временно недоступен, адрес все равно можно искать в домене, представленном текущим узлом.

Операция удаления аналогична операции вставки. Когда необходимо удалить адрес для объекта E в конечном домене D, узел каталога *dir*(D) должен удалить этот адрес из своей записи местоположения для E. Если эта запись местоположения становится пустой, то есть она не содержит других адресов для E в D, запись может быть удалена. В этом случае родительский узел *dir*(D) хочет удалить свой указатель на *dir*(D). Если запись местоположения для E в родительском элементе теперь также становится пустой, эта запись также должна быть удалена, и следующий узел каталога более высокого уровня должен быть проинформирован. Опять же, этот процесс продолжается до

тех пор, пока указатель не будет удален из записи местоположения, которая остается непустой после этого, или пока не будет достигнут корень.

Примечание 5.6 (дополнительно: проблемы масштабируемости)

Один вопрос, который сразу приходит на ум, заключается в том, может ли только что описанный иерархический подход действительно масштабироваться. Очевидная, на первый взгляд, идея заключается в том, что корневому узлу необходимо отслеживать все идентификаторы. Однако важно провести различие между логическим дизайном и его физической реализацией. Давайте проведем здесь это различие и посмотрим, как мы можем реально прийти к хорошо масштабируемой реализации службы иерархического определения местоположения.

С этой целью мы предполагаем, что каждому объекту случайным образом назначается уникальная форма идентификатора из большого пространства m-битных идентификаторов, как в Chord. Более того, допустим, что существует всего N физических хостов $\{H_1, H_2, ..., H_N\}$, которые можно использовать для сервис-поиска, распространяемого через интернет. Каждый хост способен применять один или несколько серверов определения местоположения. Как правило, два сервера, работающих на одном хосте, представляют собой два узла на разных уровнях логического дерева. Пусть $D_k(A)$ обозначает домен на уровне k, который содержит адрес A, с $k = 0$, то есть корневой домен. Аналогично пусть $LS_k(E, A)$ обозначает уникальный сервер местоположения в $D_k(A)$, отвечающий за отслеживание местонахождения объекта E.

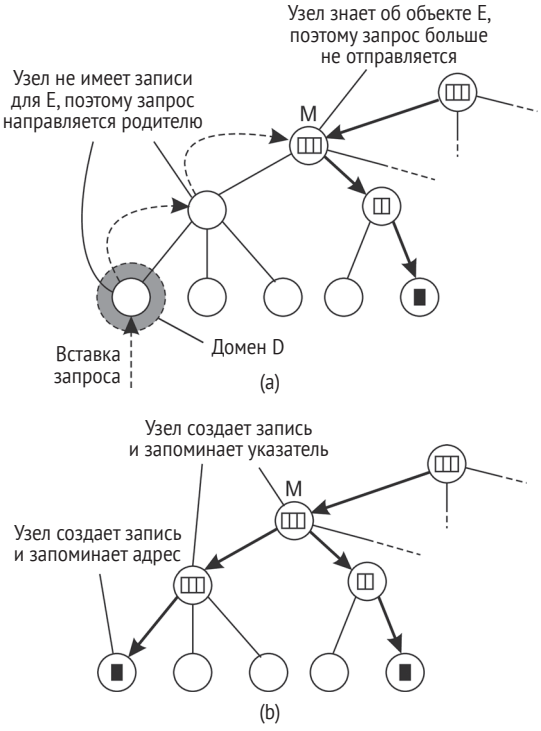


Рис. 5.9 ❖ a) Запрос на вставку пересылается первому узлу, который знает об объекте E; b) создается цепочка указателей пересылки на конечный узел

Теперь мы можем сделать различие между логическим корнем и его реализацией.

Пусть $\mathbf{D}_k = \{D_{k,1}, D_{k,2}, \ldots, D_{k,N_k}\}$ обозначают N_k доменов на уровне k, причем, очевидно, $N_0 = |\mathbf{D}_0| = 1$. Для каждого уровня k множество хостов разбивается на N_k подмножеств, с каждым хостом, на котором работает сервер местоположения, представляющий ровно один из доменов $D_{k,i}$ от $\mathbf{D}_k$. Этот принцип показан на рис. 5.10.

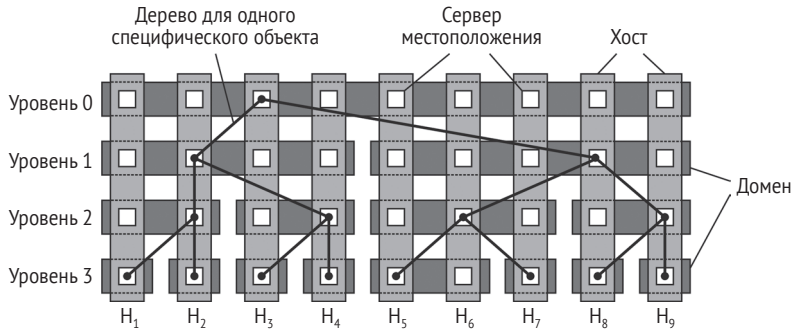


Рис. 5.10 ❖ Принцип распределения серверов логического расположения по физическим хостам

В этом примере мы рассматриваем простое дерево с четырьмя уровнями и девятью хостами. Существует два домена уровня 1, четыре домена уровня 2 и восемь конечных доменов. Мы также показываем дерево для одного конкретного объекта E: любой контактный адрес, связанный с E, будет храниться на одном из восьми серверов определения местоположения уровня 3, в зависимости, конечно, от домена, которому принадлежит этот адрес. Корневой сервер местоположения для E работает на хосте H_3. Обратите внимание, что на этом хосте также работает сервер местоположения конечного уровня для E.

Как показано в работе [van Steen and Ballintijn, 2002], разумно выбирая хост, который должен использовать сервер местоположения для E, мы можем объединить принцип локальных операций (что хорошо для географической масштабируемости) и полное распределение серверов более высокого уровня (что хорошо для масштабируемости размера).

5.3. Структурированное наименование

Плоские названия хороши для машин, но обычно не очень удобны для использования людьми. В качестве альтернативы системы наименования обычно поддерживают структурированные имена, которые составлены из простых, понятных человеку имен. Этот подход следует использовать не только для имен файлов, но и для имен хостов в интернете. В этом разделе мы сосредоточимся на структурированных именах и способах их преобразования в адреса.

Пространства имен

Имена обычно организовываются в пространстве, которое называется **пространством имен** (*name space*). Пространства имен для структурированных имен могут быть представлены в виде помеченного ориентированного графа с двумя типами узлов. **Конечный узел** представляет названный объект и обладает тем свойством, что не имеет исходящих ребер. Чтобы клиент мог получить к нему доступ, конечный узел обычно хранит информацию об объекте, который он представляет, например его адрес. Кроме того, он может хранить состояние этого объекта, например в случае файловых систем, в которых конечный узел фактически содержит полный файл, который он представляет. Мы вернемся к содержанию узлов ниже.

В отличие от конечного узла, **узел каталога** (directory node) имеет несколько исходящих ребер, каждый из которых помечен именем, как показано на рис. 5.11. Каждый узел в графе наименования рассматривается как еще один объект в распределенной системе и, в частности, имеет связанный идентификатор. Узел каталога хранит таблицу, в которой исходящее ребро представлено в виде пары (*идентификатор узла, метка края*). Такая таблица называется **таблицей каталогов** (directory table).

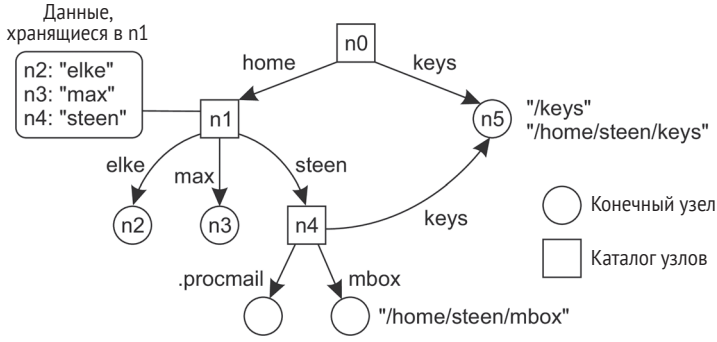


Рис. 5.11 ❖ Общий граф наименования с одним корневым узлом

Схема наименования, показанная на рис. 5.11, имеет один узел, а именно n0, который имеет только исходящие и не имеет входящих ребер. Такой узел называется корневым (узлом) графа наименования. Хотя у графа наименования может быть несколько корневых узлов, для простоты многие системы наименования имеют только один. На каждый путь в графе имен можно ссылаться последовательностью меток, соответствующих ребрам в этом пути, например N:[метка$_1$, метка$_2$, ..., метка$_n$], где N относится к первому узлу пути. Такая последовательность называется **именем пути** (path name). Если первый узел в имени пути является корнем графа наименования, он называется **абсолютным именем пути** (absolute path name). В противном случае он называется **относительным именем пути** (relative path name).

Важно понимать, что имена всегда организованы в пространстве имен. Как следствие имя всегда определяется только относительно узла каталога. В этом смысле термин «абсолютное имя» несколько вводит в заблуждение. Аналогично разница между глобальными и локальными именами часто тоже может сбивать с толку. **Глобальное имя** (global name) – это имя, которое обозначает один и тот же объект, независимо от того, где это имя используется в системе. Другими словами, глобальное имя всегда интерпретируется относительно одного и того же узла каталога. Напротив, **локальное имя** (local name) – это имя, интерпретация которого зависит от того, где это имя используется. Иными словами, локальное имя – это, по сути, относительное имя, каталог, в котором оно содержится, (неявно) известен.

Это описание графа наименования близко к тому, что реализовано во многих файловых системах. Однако вместо написания последовательности меток ребер для представления имени пути имена файлов в файловых системах обычно представляются как одна строка, в которой метки разделены специальным символом-разделителем, таким как косая черта («/»). Этот символ также используется, чтобы указать, является ли имя пути абсолютным. Например, на рис. 5.11 вместо использования n0:[home, steen, mbox], то есть фактического имени пути, обычной практикой является использование его строкового представления /home/steen/mbox. Также обратите внимание, что когда есть несколько путей, ведущих к одному и тому же узлу, этот узел может быть представлен разными именами путей. Например, узел n5 на рис. 5.11 может ссылаться на /home/steen/keys, как /keys. Строковое представление имен путей может одинаково хорошо применяться к графам наименования, отличным от тех, которые используются только для файловых систем. В работе Plan 9 [Pike et al., 1995] все ресурсы, такие как процессы, хосты, устройства ввода-вывода и сетевые интерфейсы, названы так же, как и традиционные файлы. Этот подход аналогичен реализации единого графа имен для всех ресурсов в распределенной системе.

Есть много разных способов организовать пространство имен. Как мы уже упоминали, большинство пространств имен имеют только один корневой узел. Во многих случаях пространство имен также строго иерархично в том смысле, что граф наименования организован в виде дерева. Это означает, что каждый узел, кроме корня, имеет ровно один входящий фронт; корень не имеет входящих ребер. Как следствие каждый узел также имеет ровно одно ассоциированное (абсолютное) имя пути.

Граф имен, показанный на рис. 5.11, является примером **ориентированного ациклического графа** (directed acyclic graph). В такой организации узел может иметь более одного входящего ребра, но у графа не может быть цикла. Существуют также пространства имен, которые не имеют этого ограничения.

Примечание 5.7 (дополнительная информация: реализация графа имен Unix)

Чтобы быть более конкретными, давайте рассмотрим способ именования файлов в традиционной файловой системе Unix. В графе имен для Unix узел каталога представляет файловый каталог, тогда как конечный узел представляет файл. Существу-

ет один корневой каталог, представленный в графе имен с помощью корневого узла. Реализация графа наименования является неотъемлемой частью полной реализации файловой системы. Эта реализация состоит из непрерывной серии блоков логического диска, которые обычно делятся на загрузочный блок, суперблок, серию **индексных узлов** (называемых **инодами**, inode) и файловые блоки данных. Смотрите также [Silberschatz et al., 2012] или [Tanenbaum, 2001]. Такая организация показана на рис. 5.12.

Загрузочный блок – это специальный блок данных и инструкций, которые автоматически загружаются в основную память при загрузке системы. Загрузочный блок используется для загрузки операционной системы в основную память.

Суперблок содержит информацию обо всей файловой системе, такую как ее размер, какие блоки на диске еще не распределены, какие иноды еще не используются и т. д. Иноды обозначаются индексом, начинающимся с нуля, и зарезервированы для индекса, представляющего корневой каталог.

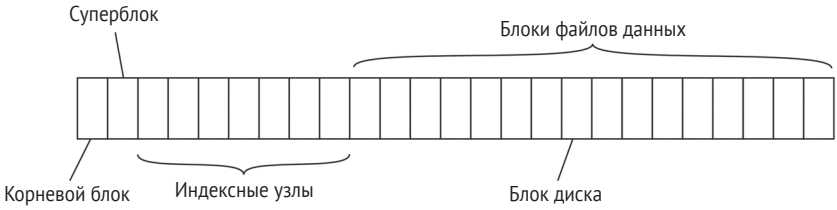


Рис. 5.12 ❖ Общая организация реализации Unix-файловой системы на логическом диске из смежных дисковых блоков

Каждый индекс содержит информацию о том, где на диске находятся данные соответствующего файла. Кроме того, индекс содержит информацию о своем владельце, времени создания и последней модификации, защите и т. п. Следовательно, если задан индексный номер индекса, можно получить доступ к связанному файлу. Каждый каталог также реализован в виде файла. Это также относится и к корневому каталогу, который содержит отображение между именами файлов и индексными номерами инода. Таким образом, видно, что индексный номер инода соответствует идентификатору узла в гра́фе имен.

Разрешение имени

Пространства имен предлагают удобный механизм для хранения и извлечения информации о блоках посредством имен. В более общем смысле, учитывая имя пути, должна быть обеспечена возможность поиска любой информации, хранящейся в узле, на который ссылается это имя. Процесс поиска имени называется **разрешением имени** (name resolution).

Чтобы объяснить, как работает разрешение имен, давайте рассмотрим имя пути, например N:[метка$_1$, метка$_2$, ..., метка$_n$]. Разрешение этого имени начинается с узла N графа имен, где имя метка$_1$ ищется в таблице каталогов и возвращает идентификатор узла, на который ссылается метка$_1$. Затем разрешение продолжается на идентифицированном узле, ища имя метка$_2$ в его таблице каталогов, и т. д. Предполагая, что именованный путь действитель-

но существует, разрешение останавливается на последнем узле, указанном метка$_n$, путем возврата содержимого этого узла.

Примечание 5.8 (дополнительная информация: снова граф имен Unix)

Поиск имени возвращает идентификатор узла, с которого продолжается процесс разрешения имени. В частности, необходимо получить доступ к таблице каталогов идентифицированного узла. Рассмотрим снова граф имен для файловой системы Unix. Как уже упоминалось, идентификатор узла реализован как индексный номер инода. Доступ к таблице каталогов означает, что сначала должен быть прочитан индекс, чтобы узнать, где хранятся фактические данные на диске, а затем надо последовательно прочитать блоки данных, содержащих таблицу каталогов.

Механизм закрытия

Разрешение имени может иметь место, только если мы знаем, как и с чего начать. В нашем примере был задан начальный узел, и мы предположили, что у нас есть доступ к его таблице каталогов. Знание того, как и с чего начать разрешение имен, обычно называется **механизмом закрытия** (closure mechanism). По сути, механизм закрытия имеет дело с выбором начального узла в пространстве имен, с которого должно начинаться разрешение имен [Radia, 1989]. Что иногда затрудняет понимание механизмов закрытия, так это то, что они обязательно являются частично неявными и могут сильно отличаться при сравнении друг с другом.

Рассмотрим, например, строку «00312059837784». Многие люди не будут знать, что делать с этими номерами, если им не скажут, что последовательность является номером телефона. Этой информации достаточно, чтобы начать процесс разрешения, в частности путем набора номера. Телефонная система впоследствии сделает все остальное.

В качестве другого примера рассмотрим использование глобальных и локальных имен в распределенных системах. Типичным примером локального имени является переменная среды. Например, в системах Unix переменная с именем HOME используется для ссылки на домашний каталог пользователя. У каждого пользователя есть собственная копия этой переменной, которая инициализируется глобальным общесистемным именем, соответствующим домашнему каталогу пользователя. Механизм замыкания, связанный с переменными среды, обеспечивает правильное разрешение имени переменной путем поиска его в пользовательской таблице.

Примечание 5.9 (дополнительная информация: граф имен Unix и механизм его закрытия)

Разрешение имен в гра́фе имен для файловой системы Unix использует тот факт, что индекс корневого каталога является первым индексом на логическом диске, представляющем файловую систему. Его фактическое смещение в байтах вычисляется из значений в других полях суперблока вместе с жестко закодированной информацией в самой операционной системе о внутренней организации суперблока.

Чтобы прояснить этот момент, рассмотрим строковое представление имени файла, например /home/steen/mbox. Чтобы разрешить это имя, необходимо иметь доступ к таблице каталогов корневого узла соответствующего графа имен. Корневой узел нельзя было бы найти, если он не был реализован в качестве другого узла в другом графе имен, скажем G. Но в этом случае было бы необходимо уже иметь доступ к корневому узлу G. Следовательно, разрешение имени файла требует, чтобы уже был реализован какой-то механизм, с помощью которого может начаться процесс разрешения.

Связывание и монтаж

С разрешением имен тесно связано использование **псевдонимов** (aliases). Псевдоним – это другое имя для того же объекта. Примером псевдонима является переменная окружения. С точки зрения графов наименования, есть два основных способа реализации псевдонимов. Первый подход заключается в том, чтобы просто позволить нескольким именам абсолютных путей ссылаться на один и тот же узел в гра́фе имен. Этот подход проиллюстрирован на рис. 5.13, в котором на узел n5 могут ссылаться два разных пути. В терминологии Unix оба пути /keys и /home/steen/keys на рис. 5.9 называются **жесткими ссылками** (hard links) на узел n5.

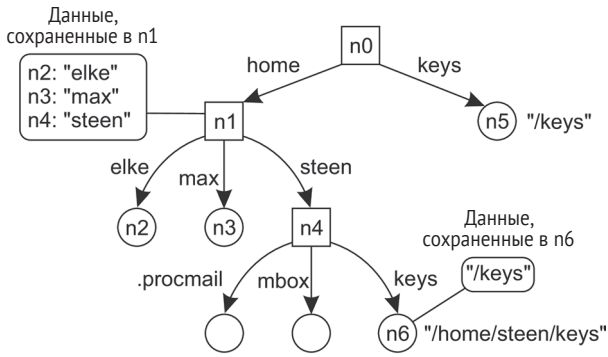


Рис. 5.13 ❖ Концепция символической ссылки, объясненная в гра́фе имен

Второй подход заключается в представлении объекта с помощью конечного узла, скажем N, но вместо сохранения адреса или состояния этого объекта узел сохраняет абсолютное имя пути. При первом разрешении абсолютного имени пути, которое приводит к N, разрешение имени вернет имя пути, сохраненное в N, после чего оно может продолжить разрешение этого нового имени пути. Этот принцип соответствует использованию **символических ссылок** (symbolic links) в файловых системах Unix и проиллюстрирован на рис. 5.13. В данном примере имя пути /home/steen/keys, которое относится к узлу, содержащему абсолютное имя пути /keys, является символической ссылкой на узел n5.

Разрешение имен, как описано выше, происходит полностью в одном пространстве имен. Однако разрешение имен также можно использовать для

прозрачного объединения различных пространств имен. Давайте сначала рассмотрим смонтированную файловую систему.

С точки зрения нашей модели наименования, смонтированная файловая система соответствует разрешению узлу каталога хранить идентификатор узла каталога из другого пространства имен, которое мы называем **пространством внешних имен** (foreign name space). Узел каталога, в котором хранится идентификатор узла, называется **точкой монтирования** (mount point). Соответственно, узел каталога в пространстве внешних имен называется **точкой подключения** (mounting point). Обычно точка монтирования является корнем пространства имен. Во время разрешения имени ищется точка монтирования, и разрешение продолжается, обращаясь к ее таблице каталогов.

Принцип монтажа может быть распространен и на другие пространства имен. В частности, необходим узел каталога, который действует как точка монтирования и хранит всю необходимую информацию для идентификации и доступа к точке монтирования в пространстве внешних имен. Этот подход используется во многих распределенных файловых системах.

Рассмотрим коллекцию пространств имен, которые распределены по разным машинам. В частности, каждое пространство имен реализуется отдельным сервером, и каждый, возможно, работает на отдельной машине. Следовательно, если мы хотим смонтировать пространство имен NS_2 в пространство имен NS_1, может потребоваться обмен данными по сети с сервером NS_2, поскольку этот сервер может работать на другой машине, а не сервере для NS_1. Для монтирования пространства внешних имен в распределенной системе требуется как минимум следующая информация:

- название протокола доступа;
- название сервера;
- название точки монтирования в пространстве внешних имен.

Обратите внимание, что каждое из этих имен должно быть разрешено. Имя протокола доступа должно быть разрешено для реализации протокола, по которому может происходить связь с сервером пространства внешних имен. Имя сервера должно быть преобразовано в адрес, по которому можно связаться с этим сервером. В качестве последней части в разрешении имени имя точки монтирования должно быть разрешено в идентификатор узла в пространстве внешних имен.

В нераспределенных системах ни одна из трех точек на самом деле не нужна. Например, в Unix нет протокола доступа и нет сервера. Кроме того, имя точки монтирования не является обязательным, поскольку это просто корневой каталог внешнего пространства имен.

Имя точки монтирования должно быть разрешено сервером пространства внешних имен. Однако нам также нужны пространства имен и реализации для протокола доступа и имени сервера. Одной из возможностей является представление трех имен, перечисленных выше, в виде URL.

Чтобы конкретизировать ситуацию, рассмотрим ситуацию, в которой пользователь с портативным компьютером хочет получить доступ к файлам, которые хранятся на удаленном файловом сервере. Клиентский компьютер и файловый сервер сконфигурированы с **сетевой файловой системой**

(Network File System, NFS). В частности, чтобы позволить NFS работать через интернет, клиент может точно указать, к какому файлу он хочет получить доступ, с помощью URL-адреса NFS, например nfs://its.cs.vu.nl/home/steen. Этот URL-адрес именует файл (который является каталогом) с именем /home/steen на NFS файловом сервере its.cs.vu.nl, к которому клиент может получить доступ посредством протокола NFS [Shepler et al., 2003].

Имя nfs – это хорошо известное имя в том смысле, что существует всемирное соглашение о том, как интерпретировать это имя. Учитывая, что мы имеем дело с URL, имя nfs будет преобразовано в реализацию протокола NFS. Имя сервера преобразуется в его адрес с использованием DNS, что обсуждается в следующем разделе. Как мы уже говорили, /home/steen разрешается сервером пространства внешних имен.

Организация файловой системы на клиентском компьютере частично показана на рис. 5.14. Корневой каталог содержит несколько пользовательских записей, включая подкаталог /remote. Этот подкаталог предназначен для включения точек монтирования для внешних пространств имен, таких как домашний каталог пользователя в университете VU (Амстердам). С этой целью узел каталога с именем /remote/vu используется для хранения URL nfs://its.cs.vu.nl/home/steen.

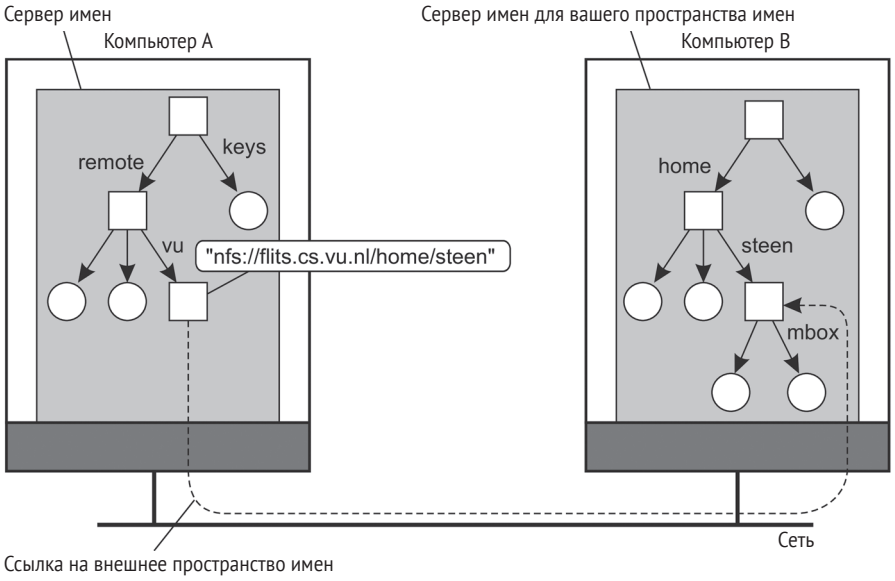


Рис. 5.14 ❖ Монтирование удаленных пространств имен через специальный протокол

Теперь рассмотрим имя /remote/vu/mbox. Это имя разрешается путем запуска в корневом каталоге на компьютере клиента и продолжается до тех пор, пока не будет достигнут узел /remote/vu. Затем процесс разрешения имени продолжается, возвращая URL-адрес nfs://its.cs.vu.nl/home/steen, в свою очередь приводя компьютер клиента к контакту с файловым сервером its.cs.vu.nl

с помощью протокола NFS и для последующего доступа к каталогу /home/steen. Разрешение имени можно затем продолжить, прочитав файл с именем mbox в этом каталоге, после чего процесс разрешения останавливается.

Распределенные системы, которые позволяют монтировать удаленную файловую систему, как описано выше, позволяют компьютеру клиента, например, выполнять следующие команды (предположим, что компьютер клиента называется horton):

```
horton$ cd /remote/vu
horton$ ls -l
```

компьютер впоследствии перечисляет файлы в каталоге /home/steen на удаленном файловом сервере. Прелесть всего этого в том, что пользователь избавлен от деталей фактического доступа к удаленному серверу. В идеале отмечается только некоторая потеря производительности по сравнению с доступом к локально доступным файлам. По сути, клиенту кажется, что пространство имен с корнем на локальном компьютере, а пространство с именем /home/steen на удаленном компьютере образуют единое пространство имен.

Примечание 5.10 (дополнительная информация: монтирование по сети в Unix)

Существует множество способов установки через сеть. Одно из практических решений, которое применяется многими небольшими распределенными системами, заключается в простом назначении фиксированных IP-адресов компьютерам и последующем предложении точек подключения клиентам. Рассмотрим следующий пример. Предположим, у нас есть машина Unix с именем coltrane, использующая частный адрес 192.168.2.3 и хранящая коллекцию музыкальных файлов в локальном каталоге /audio. Этот каталог может быть экспортирован как точка монтирования и, как следствие, может быть импортирован другому компьютеру.

Пусть quandar будет таким компьютером, и предположим, что он хочет смонтировать коллекцию аудиофайлов в локальной точке монтирования /home/maarten/Music. Следующая команда выполнит работу (при условии что были установлены правильные привилегии):

```
quandar$ mount -t nfs 192.168.2.3:/audio /home/maarten/Music
```

С этого момента все файлы, доступные на coltrane в его каталоге /audio, могут быть доступны через quandar в каталоге /home/maarten/Music. Прелесть этой схемы в том, что после установки больше нет необходимости думать об удаленном доступе (конечно, пока что-то не выйдет из строя).

Реализация пространства имен

Пространство имен формирует основу службы имен, то есть службы, которая позволяет пользователям и процессам добавлять, удалять и искать имена. Служба наименования осуществляется серверами имен. Если распределенная система ограничена локальной сетью, часто возможно реализовать службу наименования только с помощью одного сервера имен. Однако в крупномасштабных распределенных системах со многими объектами, возможно,

распределенными в большом географическом ространстве, необходимо распределить реализацию пространства имен по нескольким серверам имен.

Распределение пространства имен

Пространства имен для крупномасштабной, возможно, всемирной распределенной системы обычно организованы иерархически. Как и раньше, предположим, что такое пространство имен имеет только один корневой узел. Чтобы эффективно реализовать такое пространство имен, удобно разбить его на логические уровни. В работе [Cheriton and Mann, 1989] авторы различают следующие три уровня.

Глобальный уровень (global layer) образован узлами самого высокого уровня, то есть корневым узлом и другими узлами каталога, логически близкими к корню, а именно его дочерними элементами. Узлы на глобальном уровне часто характеризуются своей стабильностью в том смысле, что таблицы каталогов редко изменяются. Такие узлы могут представлять организации или группы организаций, имена которых хранятся в пространстве имен.

Административный уровень (administrational layer) состоит из узлов каталогов, которые совместно управляются в рамках одной организации. Характерная особенность каталога узлов на административном уровне состоит в том, что они представляют группы объектов, принадлежащих одной организации или административному подразделению. Например, может существовать узел каталога для каждого отдела в организации или узел каталога, из которого можно найти все узлы. Другой узел каталога может использоваться в качестве отправной точки для именования всех пользователей и т. д. Узлы на административном уровне относительно стабильны, хотя изменения, как правило, происходят чаще, чем для узлов на глобальном уровне.

Наконец, **управленческий уровень** (managerial layer) состоит из узлов, которые обычно могут регулярно меняться. Например, узлы в локальной сети принадлежат этому уровню. По той же причине уровень включает в себя узлы, представляющие общие файлы, такие как файлы для библиотек или двоичные файлы. Другой важный класс узлов включает узлы, которые представляют пользовательские каталоги и файлы. В отличие от глобального и административного уровней, узлы на уровне управления обслуживаются не только системными администраторами, но и отдельными конечными пользователями распределенной системы.

Чтобы конкретизировать ситуацию, на рис. 5.15 показан пример разбиения части пространства имен DNS, включая имена файлов внутри организации, к которым можно получить доступ через интернет, например веб-страницы и передаваемые файлы. Пространство имен разделено на непересекающиеся части, называемые в DNS **зонами** (zones) [Mockapetris, 1987]. Зона является частью имени пространства, которое реализуется отдельным сервером имен. Некоторые из этих зон показаны на рис. 5.15.

Если мы посмотрим на доступность и производительность, серверы имен на каждом уровне должны соответствовать различным требованиям. Высокая доступность особенно важна для серверов имен на глобальном уровне.

В случае сбоя сервера имен большая часть пространства имен будет недоступна, поскольку разрешение имен не может продолжаться после сбоя сервера.

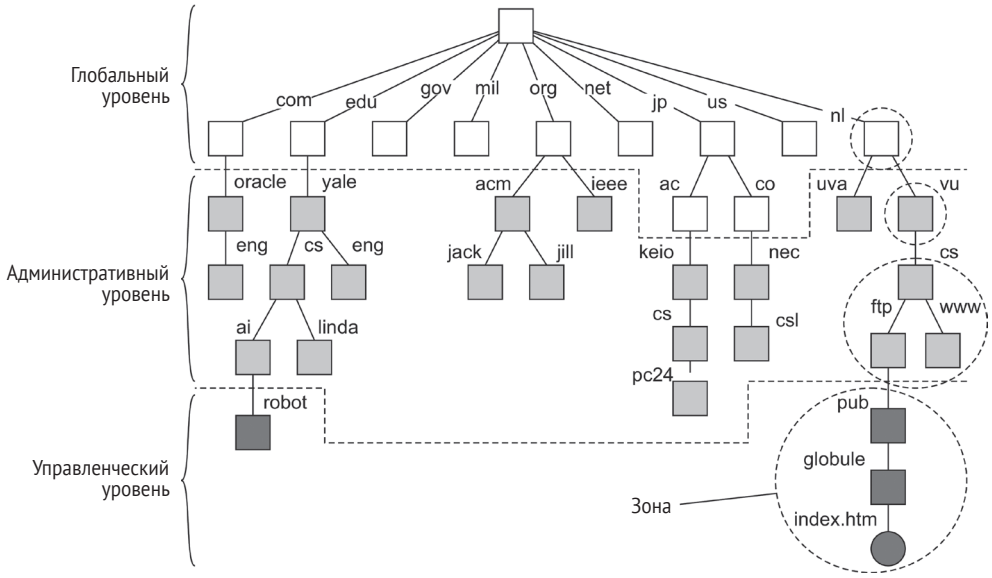


Рис. 5.15 ❖ Пример разбиения пространства имен DNS на три уровня, включая файлы, доступные через интернет

Производительность – довольно тонкая сфера. Из-за низкой скорости смены узлов в глобальном уровне результаты операций поиска обычно остаются действительными в течение длительного времени. Следовательно, эти результаты могут эффективно кешироваться (то есть храниться локально) клиентами. В следующий раз, когда будет выполнена та же операция поиска, результаты, вместо того чтобы позволить серверу имен возвращать результаты, могут быть извлечены из кеша клиента. Поэтому серверы имен на глобальном уровне не должны быстро отвечать на один запрос поиска. С другой стороны, пропускная способность может быть важной, особенно в крупных системах с миллионами пользователей.

Требования к доступности и производительности для серверов имен на глобальном уровне могут быть удовлетворены путем репликации серверов в сочетании с кешированием на стороне клиента. Обновления на этом уровне, как правило, не должны вступать в силу немедленно, что значительно упрощает поддержание согласованности реплик.

Доступность сервера имен на административном уровне в первую очередь важна для клиентов в той же организации, что и сервер имен. В случае сбоя сервера имен многие ресурсы в организации становятся недоступными, поскольку их невозможно найти. С другой стороны, может быть менее важно, чтобы ресурсы в организации были временно недоступны для пользователей за пределами этой организации.

Что касается производительности, серверы имен на административном уровне имеют те же характеристики, что и на глобальном уровне. Поскольку изменения в узлах происходят не так часто, результаты поиска в кеше могут быть очень эффективными, что делает производительность менее критичным параметром. Однако, в отличие от глобального уровня, административный уровень должен следить за тем, чтобы результаты поиска возвращались в течение нескольких миллисекунд либо непосредственно с сервера, либо из локального кеша клиента. Аналогично обновления, как правило, должны обрабатываться быстрее, чем обновления глобального уровня. Например, недопустимо, чтобы учетная запись для нового пользователя вступала в силу часами.

Этим требованиям часто можно соответствовать, используя относительно мощные машины для запуска серверов имен. Кроме того, следует применять кеширование на стороне клиента в сочетании с репликацией для повышения общей доступности.

Требования доступности серверов имен на управленческом уровне, как правило, менее критичны. В частности, нередко достаточно использовать одну машину для запуска серверов имен с риском временной недоступности. Тем не менее производительность имеет решающее значение: операции должны выполняться немедленно. Поскольку обновления происходят регулярно, кеширование на стороне клиента часто менее эффективно.

Проблема	Глобальный	Административный	Управленческий
Географическая шкала	Всемирный	Организационный	Отдел
Количество узлов	Мало	Много	Очень много
Отзывчивость на поиск	Секунды	Миллисекунды	Немедленно
Распространение с обновлением	Медленно	Немедленно	Немедленно
Количество реплик	Много	Нет или мало	Нет
Кеширование на стороне клиента	Да	Да	Иногда

Рис. 5.16 ❖ Сравнение серверов имен для реализации узлов из крупномасштабного пространства имен, разделенного на глобальный уровень, уровень администрирования и уровень управления

Сравнение серверов имен на разных уровнях показано на рис. 5.16. В распределенных системах наиболее сложно реализовать серверы имен на глобальном и административном уровнях. Трудности вызваны репликацией и кешированием, которые необходимы для доступности и производительности, и также создают и проблемы согласованности. Некоторые проблемы усугубляются тем фактом, что кеши и реплики распространяются по глобальной сети, что может привести к длительным задержкам связи во время поисков.

Реализация разрешения имен

Распределение пространства имен по нескольким серверам имен влияет на реализацию разрешения имен. Чтобы объяснить реализацию разрешения

имен в крупных службах имен, предположим, что серверы имен не реплицируются и что на стороне клиента не используются кеши. Каждый клиент имеет доступ к локальному **распознавателю имен** (name resolver), который отвечает за обеспечение выполнения процесса разрешения имен. Ссылаясь на рис. 5.15, предположим, что (абсолютный) путь к имени root:[nl, vu, cs, ftp, pub, globe, index.html] должен быть разрешен. Используя обозначение URL, это имя пути будет соответствовать ftp://ftp.cs.vu.nl/pub/globe/index.html. В настоящее время существует два способа реализации разрешения имен.

При **итеративном разрешении** имен (iterative name resolution) распознаватель имен передает полное имя корневому серверу имен. Предполагается, что адрес, по которому можно связаться с корневым сервером, хорошо известен. Корневой сервер разрешит имя пути, насколько это возможно, и вернет результат клиенту. В нашем примере корневой сервер может разрешить только метку nl, для которой он вернет адрес связанного сервера имен.

В этот момент клиент передает оставшееся имя пути (то есть nl:[vu, cs, ftp, pub, globe, index.html]) этому серверу имен. Этот сервер может разрешить только метку vu и возвращает адрес связанного сервера имен вместе с оставшимся именем пути vu:[cs, ftp, pub, globe, index.html].

Средство распознавания имен клиента затем свяжется с этим следующим сервером имен, который отвечает разрешением метки cs, а потом также ftp, возвращая адрес FTP-сервера вместе с именем пути ftp:[pub, globe, index.html]. Затем клиент связывается с FTP-сервером, запрашивая его разрешение последней части начального пути. Впоследствии FTP-сервер разрешит метки pub, globe и index.html и передаст запрошенный файл (в данном случае с использованием FTP). Этот процесс итеративного разрешения имен показан на рис. 5.17. (Обозначение #[cs] используется для указания адреса сервера, ответственного за обработку узла, на который ссылается [cs].)

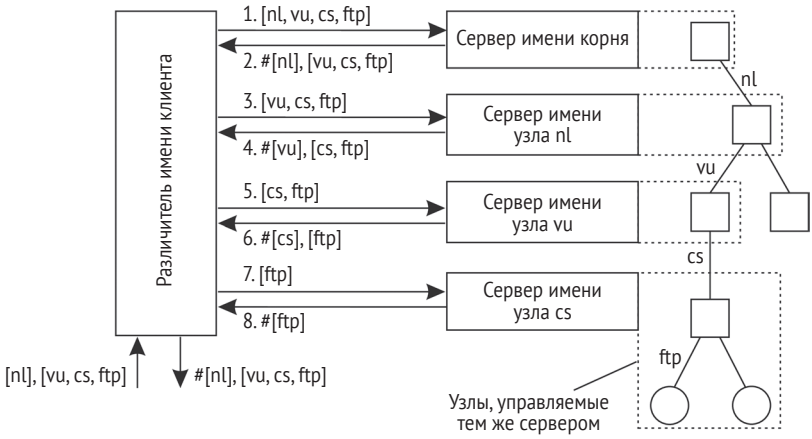


Рис. 5.17 ❖ Принцип итеративного разрешения имен

На практике последний шаг, а именно установление контакта с сервером FTP и запрос его передачи файла с именем пути ftp:[pub, globe, index.html],

выполняется клиентским процессом отдельно. Другими словами, клиент обычно передает обработчику имен только путь с именем root:[nl, vu, cs, ftp], по которому он может ожидать адрес и может связаться с FTP-сервером, как также показано на рис. 5.17.

Альтернативой итеративному разрешению имен является использование рекурсии во время разрешения имен. Вместо того чтобы возвращать каждый промежуточный результат обратно в распознаватель имен клиента, при **рекурсивном разрешении имен** (recursive name resolution) сервер имен передает результат следующему найденному серверу имен. Так, например, когда корневой сервер имен находит адрес сервера имен, реализующего узел с именем nl, он запрашивает у сервера имен разрешение пути nl:[vu, cs, ftp, pub, globe, index.html]. Используя рекурсивное разрешение имен, этот следующий сервер разрешит полный путь и в конечном итоге вернет файл index.html к корневому серверу, который, в свою очередь, передаст этот файл распознавателю имен клиента.

Рекурсивное разрешение имен показано на рис. 5.18. Как и в случае итеративного разрешения имен, последний шаг (обращение к FTP-серверу и запрос его передачи указанного файла) обычно выполняется клиентом как отдельный процесс.

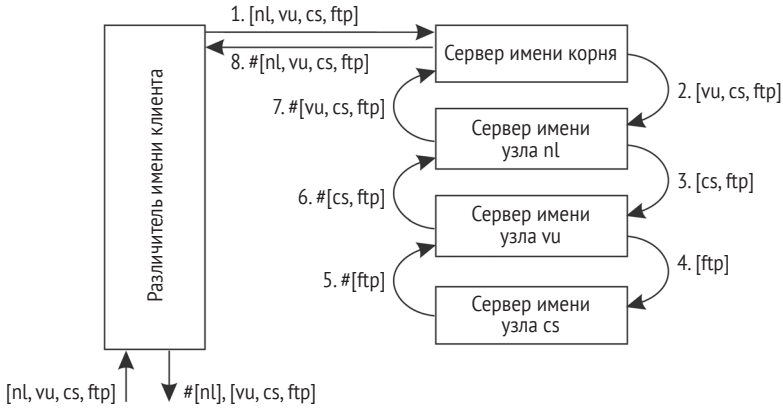


Рис. 5.18 ❖ Принцип рекурсивного разрешения имен

Основным недостатком рекурсивного разрешения имен является то, что оно предъявляет более высокие требования к производительности на каждом сервере имен. По сути, сервер имен требуется для обработки полного разрешения имени пути, хотя это может быть сделано в сотрудничестве с другими серверами имен. Это дополнительное бремя обычно настолько велико, что серверы имен на глобальном уровне пространства имен поддерживают только итеративное разрешение имен.

У рекурсивного разрешения имен есть два важных преимущества. Первое преимущество заключается в том, что результаты кеширования более эффективны по сравнению с итеративным разрешением имен. Второе преимущество заключается в том, что затраты на связь могут быть уменьшены.

Чтобы объяснить эти преимущества, предположим, что распознаватель имен клиента будет принимать имена путей, относящиеся только к узлам на глобальном или административном уровне пространства имен. Чтобы разрешить ту часть имени пути, которая соответствует узлам на уровне управления, клиент будет отдельно связываться с сервером имен, возвращаемым его распознавателем имен, как мы обсуждали выше.

Рекурсивное разрешение имен позволяет каждому серверу имен постепенно узнавать адрес каждого сервера имен, ответственного за реализацию узлов нижнего уровня. В результате кеширование может быть эффективно использовано для повышения производительности. Например, когда от корневого сервера запрашивается разрешение имени пути root:[nl, vu, cs, ftp], он в конечном итоге получит адрес сервера имен, реализующего узел, на который ссылается это имя пути. Чтобы прийти к этому, сервер имен для узла nl должен искать адрес сервера имен для узла vu, тогда как последний должен искать адрес сервера имен, обрабатывающего узел cs.

Поскольку изменения в узлах на глобальном и административном уровнях происходят нечасто, корневой сервер имен может эффективно кешировать возвращаемый адрес. Более того, поскольку адрес также рекурсивно возвращается серверу имен, отвечающему за реализацию узла vu, и серверу, реализующему узел nl, он также может кешироваться на этих серверах.

Аналогично результаты поиска промежуточных имен также могут быть возвращены и кешированы. Например, сервер для узла nl должен будет найти адрес сервера узла vu. Этот адрес может быть возвращен корневому серверу, когда nl-сервер возвращает результат поиска исходного имени. Полный обзор процесса разрешения и результатов, которые могут кешироваться каждым сервером имен, показан на рис. 5.19.

Сервер узла	Должен разрешить	Найти	Отправить ребенку	Получить и кешировать	Вернуть запрашивающему
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]

Рис. 5.19 ❖ Рекурсивное разрешение имен [nl, vu, cs, ftp].
Серверы имен кешируют промежуточные результаты для последующих поисков

Основным преимуществом этого подхода является то, что в конечном итоге операции поиска могут выполняться довольно эффективно. Например, предположим, что другой клиент позже запрашивает разрешение корневого

пути root:[nl, vu, cs, flits]. Это имя передается в корень, который может немедленно переслать его на сервер имен для узла cs и запросить разрешение оставшегося имени пути cs:[flits].

При итеративном разрешении имен кеширование обязательно ограничивается распознавателем имен клиента. Следовательно, если клиент A запрашивает разрешение имени, а другой клиент B позднее запрашивает разрешение того же имени, разрешение имен должно будет проходить через те же серверы имен, что и для клиента A. В качестве компромисса многие организации используют локальный промежуточный сервер имен, общий для всех клиентов. Этот локальный сервер имен обрабатывает все запросы имен и кеширует результаты. Такой промежуточный сервер также удобен с точки зрения управления. Например, только этот сервер должен знать, где расположен корневой сервер имен; другие машины не требуют подобной информации.

Второе преимущество рекурсивного разрешения имен заключается в том, что оно зачастую дешевле в отношении связи. Опять же, рассмотрим разрешение имени пути root:[nl, vu, cs, ftp] и предположим, что клиент находится в Сан-Франциско. Предполагая, что клиент знает адрес сервера для узла nl с рекурсивным разрешением имен, связь идет по маршруту от хоста клиента в Сан-Франциско к серверу nl в Нидерландах (на рис. 5.20 показано как R1). С этого момента необходимо впоследствии установить связь между сервером nl и сервером имен университета VU в кампусе в Амстердаме (Нидерланды). Это сообщение показано как R2. Наконец, требуется связь между сервером vu и сервером имен в факультете компьютерных наук, обозначенном как R3. Маршрут ответа тот же, но в обратном направлении. Очевидно, что расходы на связь диктуются обменом сообщениями между хостом клиента и сервером nl.

Напротив, при итеративном разрешении имен хосту клиента приходится обмениваться данными отдельно с сервером nl, сервером vu и сервером cs, для чего общие затраты могут примерно в три раза превышать рекурсивное разрешение имен. Стрелки на рис. 5.20, обозначенные I1, I2 и I3, показывают путь связи для итеративного разрешения имен.

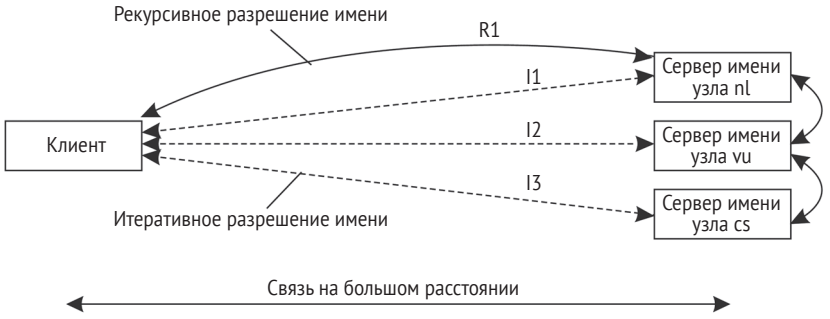


Рис. 5.20 ❖ Сравнение рекурсивного и итеративного разрешений имен с точки зрения затрат на связь

Пример: система доменных имен

Одна из крупнейших служб распределенного именования, используемая сегодня, – это система доменных имен в интернете (DNS). DNS в основном используется для поиска IP-адресов хостов и почтовых серверов. На следующих страницах мы сосредоточимся на организации пространства имен DNS и информации, хранящейся в его узлах. Также мы подробнее рассмотрим фактическую реализацию DNS. Более подробную информацию можно найти в [Mockapetris, 1987] и [Liu and Albitz, 2006]. Оценку DNS, особенно касающуюся того, соответствует ли она потребностям современного интернета, можно найти в [Levien, 2005]. Из этого отчета можно сделать несколько неожиданный вывод о том, что даже после более чем 30 лет DNS не дает никаких указаний на необходимость его замены. Мы бы сказали, что главная причина заключается в глубоком понимании дизайнером того, как все делать просто. Практика в других областях распределенных систем показывает, что не многие одарены таким пониманием.

Пространство имен DNS

Пространство имен DNS иерархически организовано как корневое дерево. Метка – это строка без учета регистра, состоящая из буквенно-цифровых символов. Максимальная длина метки составляет 63 символа; длина полного пути ограничена 255 символами. Строковое представление имени пути состоит из перечисления его меток, начиная с самой правой, и разделения меток точкой («.»). Корень представлен точкой. Так, например, путь к имени root:[nl, vu, cs, flits] представлен строкой «flits.cs.vu.nl.», которая включает в себя крайнюю правую точку для обозначения корневого узла. Мы обычно опускаем эту точку для удобства чтения.

Поскольку каждый узел в пространстве имен DNS имеет ровно одно входящее ребро (за исключением корневого узла, у которого нет входящих ребер), метка, прикрепленная к входящему ребру узла, также используется в качестве имени для этого узла. Поддерево называется **доменом** (domain), а путь к его корневому узлу называется **доменным именем** (domain name). Обратите внимание, что, как и имя пути, имя домена может быть абсолютным или относительным.

Содержимое узла формируется набором записей ресурсов. Существуют разные типы записей ресурсов. Основные из них показаны на рис. 5.21.

Узел в пространстве имен DNS часто представляет несколько объектов одновременно. Например, доменное имя, такое как vu.nl, используется для представления домена и зоны. В этом случае домен реализуется посредством нескольких (неперекрывающихся) зон.

Запись ресурса SOA (start of authority, начало полномочий) содержит такую информацию, как адрес электронной почты системного администратора, ответственного за представленную зону, имя хоста, на котором можно получить данные о зоне, и т. д.

Адресная запись A представляет определенный хост в интернете. Запись A содержит IP-адрес для этого хоста, чтобы разрешить связь. Если у хоста есть

несколько IP-адресов, как в случае с несколькими домашними компьютерами, узел будет содержать запись A для каждого адреса.

Тип	Относится к	Описание
SOA	Зона	Содержит информацию о представленной зоне
A	Хост	Этот узел представляет IP addr. хоста
MX	Домен	Почтовый сервер обработки почты этого узла
SRV	Узел	Сервер домена, обрабатывающий конкретную службу
NS	Зона	Сервер имени для представленной зоны
CNAME	Узел	Символическая ссылка
PTR	Хост	Каноническое имя хоста
HINFO	Хост	Информация на этом узле
TXT	Любая информация	Любая информация считается полезной

Рис. 5.21 ❖ Наиболее важные типы записей ресурсов, формирующих содержание узлов в пространстве имен DNS

Другим типом записи является запись MX (mail exchange), которая похожа на символическую ссылку на узел, представляющий почтовый сервер. Например, узел, представляющий домен cs.vu.nl, имеет запись MX, содержащую имя zephyr.cs.vu.nl, которое относится к почтовому серверу. Этот сервер будет обрабатывать всю входящую почту, адресованную пользователям в домене cs.vu.nl. Может быть несколько записей MX, хранящихся в узле.

С записями MX связаны записи SRV, которые содержат имя сервера для конкретной службы. Сам сервис идентифицируется посредством имени вместе с именем протокола. Например, веб-сервер в домене cs.vu.nl может быть назван с помощью записи SRV, такой как _http_tcp.cs.vu.nl. Эта запись будет затем ссылаться на фактическое имя сервера (которое soling.cs.vu.nl). Важным преимуществом записей SRV является то, что клиентам больше не нужно знать DNS-имя хоста, предоставляющего конкретную услугу. Вместо этого необходимо стандартизировать только имена сервисов, после чего можно найти хост-провайдер.

Узлы, которые представляют зону, содержат одну или несколько записей NS (name server, серверов имен). Как и записи MX, запись NS содержит имя сервера имен, который реализует зону, представленную узлом. В принципе, каждый узел в пространстве имен может хранить запись NS, ссылающуюся на сервер имен, который ее реализует. Однако, как мы обсудим ниже, реализация пространства имен DNS такова, что только узлы, представляющие зоны, должны хранить записи NS.

DNS отличает псевдонимы от так называемых **канонических имен** (canonical names). Предполагается, что каждый хост имеет каноническое или исходное имя. Псевдоним реализован посредством узла, хранящего запись CNAME, содержащую каноническое имя хоста. Таким образом, имя узла, хранящего такую запись, совпадает с символической ссылкой, как показано на рис. 5.13.

DNS поддерживает обратное сопоставление IP-адресов с именами хостов посредством записей PTR (указателя). Чтобы обеспечить поиск имен хостов

при наличии только IP-адреса, DNS поддерживает домен с именем in-addr.arpa, который содержит узлы, представляющие интернет-хосты и которые называются по IP-адресу представленного хоста. Например, хост www.cs.vu.nl имеет IP-адрес 130.37.20.20. DNS создает узел с именем 20.20.37.130.in-addr.arpa, который используется для хранения канонического имени этого хоста (который в записи PTR оказывается soling.cs.vu.nl).

Наконец, запись HINFO (host info, информация о хосте) используется для хранения дополнительной информации о хосте, такой как тип компьютера и операционная система. Аналогичным образом записи TXT используются для любых других типов данных, которые пользователь находит полезными для хранения о блоке, представленном узлом.

Реализация DNS

По сути, пространство имен DNS можно разделить на глобальный уровень и административный уровень, как показано на рис. 5.15. Уровень управления, который обычно формируется локальными файловыми системами, формально не является частью DNS и, следовательно, также им не управляется.

Каждая зона реализуется сервером имен, который практически всегда для доступности реплицируется. Обновления для зоны обычно обрабатываются первичным сервером имен. Обновления происходят путем изменения базы данных DNS, локальной для основного сервера. Вторичные серверы имен не обращаются к базе данных напрямую, а вместо этого запрашивают основной сервер для передачи ее содержимого. Последнее по терминологии DNS называется **зонной передачей** (zone transfer).

База данных DNS реализована в виде (небольшой) коллекции файлов, наиболее важная из которых содержит все записи ресурсов для всех узлов в конкретной зоне. Этот подход позволяет просто идентифицировать узлы посредством их доменного имени, с помощью которого понятие идентификатора узла сводится к (неявному) индексу в файле.

Примечание 5.11 (дополнительная информация: пример базы данных DNS)

Чтобы лучше понять эти проблемы реализации, на рис. 5.22 показана небольшая часть файла, которая содержит бóльшую часть информации для предыдущей организации домена cs.vu.nl. Обратите внимание, что мы сознательно выбрали устаревшую версию из соображений безопасности. Файл был отредактирован для удобства чтения. Он показывает содержимое нескольких узлов, которые являются частью домена cs.vu.nl, где каждый узел идентифицируется с помощью своего доменного имени.

Узел cs.vu.nl представляет домен и зону. Его ресурсная запись SOA содержит конкретную информацию о действительности этого файла, которая нас больше не касается. Для этой зоны существует четыре сервера имен, на которые ссылаются их канонические имена хостов в записях NS. Запись TXT используется для предоставления некоторой дополнительной информации об этой зоне, но не может автоматически обрабатываться никаким сервером имен. Кроме того, существует один почтовый сервер, который может обрабатывать входящую почту, адресованную пользователям в этом домене. Число, предшествующее имени почтового сервера, определяет приоритет выбора. Отправляющий почтовый сервер должен всегда пытаться связаться с почтовым сервером с наименьшим номером.

Имя	Тип записи	Содержание записи
cs.vu.nl.	SOA	star.cs.vu.nl. hostmaster.cs.vu.nl. 2005092900 7200 3600 2419200 3600
cs.vu.nl.	TXT	«VU University – Computer Science»
cs.vu.nl.	MX	1 mail.few.vu.nl.
cs.vu.nl.	NS	ns.vu.nl.
cs.vu.nl.	NS	top.cs.vu.nl.
cs.vu.nl.	NS	solo.cs.vu.nl.
cs.vu.nl.	NS	star.cs.vu.nl.
star.cs.vu.nl.	A	130.37.24.6
star.cs.vu.nl.	A	192.31.231.42
star.cs.vu.nl.	MX	1 star.cs.vu.nl.
star.cs.vu.nl.	MX	666 zephyr.cs.vu.nl.
star.cs.vu.nl.	HINFO	«Sun» «Unix»
zephyr.cs.vu.nl.	A	130.37.20.10
zephyr.cs.vu.nl.	MX	1 zephyr.cs.vu.nl.
zephyr.cs.vu.nl.	MX	2 tornado.cs.vu.nl.
zephyr.cs.vu.nl.	HINFO	«Sun» «Unix»
ftp.cs.vu.nl.	CNAME	soling.cs.vu.nl.
www.cs.vu.nl.	CNAME	soling.cs.vu.nl.
soling.cs.vu.nl.	A	130.37.20.20
soling.cs.vu.nl.	MX	1 soling.cs.vu.nl.
soling.cs.vu.nl.	MX	666 zephyr.cs.vu.nl.
soling.cs.vu.nl.	HINFO	«Sun» «Unix»
vucs-das1.cs.vu.nl.	PTR	0.198.37.130. in-addr.arpa.
vucs-das1.cs.vu.nl.	A	130.37.198.0
inkt.cs.vu.nl.	HINFO	«OCE» «Proprietary»
inkt.cs.vu.nl.	A	192.168.4.3
pen.cs.vu.nl.	HINFO	«OCE» «Proprietary»
pen.cs.vu.nl.	A	192.168.4.2
localhost.cs.vu.nl.	A	127.0.0.1

Рис. 5.22 ❖ Выдержка из базы данных DNS (старой) для зоны cs.vu.nl

Хост star.cs.vu.nl работает как сервер имен для этой зоны. Серверы имен имеют решающее значение для любой службы именования. Что можно видеть об этом сервере имен, так это то, что дополнительная надежность была создана путем предоставления двух отдельных сетевых интерфейсов, каждый из которых представлен отдельной записью ресурса A. Таким образом, последствия разрыва сетевого соединения могут быть несколько смягчены, так как сервер останется доступным.

Следующие четыре строки (для zephyr.cs.vu.nl) содержат необходимую информацию об одном из почтовых серверов отдела. Обратите внимание, что этот почтовый сервер также поддерживается другим почтовым сервером, чей путь tornado.cs.vu.nl.

Следующие шесть строк показывают типичную конфигурацию, в которой веб-сервер отдела, а также FTP-сервер отдела реализуются на одной машине, называемой soling.cs.vu.nl. Благодаря тому что оба сервера выполняются на одной машине (и, по сути, используют эту машину только для интернет-служб, а не для чего-либо

еще), управление системой становится проще. Например, оба сервера будут иметь одинаковое представление файловой системы, и для эффективности часть файловой системы может быть реализована в soling.cs.vu.nl. Этот подход часто применяется в случае служб WWW и FTP.

Следующие две строки показывают информацию об одном из старых кластеров серверов департамента. В этом случае это говорит нам, что адрес 130.37.198.0 связан с именем хоста vucs-das1.cs.vu.nl.

Следующие четыре строки показывают информацию о двух основных принтерах, подключенных к локальной сети. Обратите внимание, что адреса в диапазоне от 192.168.0.0 до 192.168.255.255 являются частными: доступ к ним можно получить только изнутри локальной сети, и они недоступны с произвольного узла интернета.

Поскольку домен cs.vu.nl реализован как единая зона, рис. 5.22 не содержит ссылок на другие зоны. Способ ссылки на узлы в поддомене, реализованные в другой зоне, показан на рис. 5.23. Что нужно сделать, так это указать сервер имен для поддомена, просто указав его доменное имя и IP-адрес. При разрешении имени для узла, находящегося в домене cs.vu.nl, разрешение имен будет продолжаться определенное время путем считывания базы данных DNS, хранящейся на сервере имен для домена cs.vu.nl.

Имя	Тип записи	Значение записи
cs.vu.nl.	NS	solo.cs.vu.nl.
cs.vu.nl.	NS	star.cs.vu.nl.
cs.vu.nl.	NS	ns.vu.nl.
cs.vu.nl.	NS	top.cs.vu.nl.
ns.vu.nl	A	130.37.129.4
top.cs.vu.nl.	A	130.37.20.4
solo.cs.vu.nl.	A	130.37.20.5
star.cs.vu.nl.	A	130.37.24.6
star.cs.vu.nl.	A	192.31.231.42

Рис. 5.23 ❖ Часть описания для домена vu.nl, который содержит домен cs.vu.nl

Примечание 5.12 (дополнительно: децентрализованные и иерархические реализации DNS)

Реализация DNS, которую мы описали, является стандартной. Она следует иерархии серверов с 13 известными корневыми узлами и заканчивается миллионами серверов на конечных узлах (читайте далее). Важно то, что узлы более высокого уровня получают гораздо больше запросов, чем узлы более низкого уровня. Только кешируя привязки имени к адресу этих более высоких уровней, становится возможным избежать отправки запросов к ним и, таким образом, их переполнения.

Этих проблем масштабируемости, в принципе, можно избежать с помощью полностью децентрализованных решений. В частности, мы можем вычислить хеш DNS-имени и впоследствии принять этот хеш в качестве ключевого значения для поиска в распределенной хеш-таблице или службе иерархического определения местоположения с полностью разделенным корневым узлом. Очевидный недостаток этого подхода состоит в том, что мы теряем структуру исходного имени. Эта потеря может помешать эффективной реализации, например, нахождения всех дочерних элементов в определенной области.

Как утверждается в [Walfish et al.2004], когда существует необходимость во многих именах, применение идентификаторов без семантического способа доступа к данным позволит различным системам использовать одну систему наименования. Причина проста: к настоящему времени хорошо известно, как можно эффективно поддерживать огромную коллекцию (плоских) имен. Что нужно сделать, так это сохранить отображение идентификатора информации об имени там, куда в этом случае имя может приходить из пространства DNS, как URL и т. д. Использование идентификаторов может быть упрощено, если позволить пользователям или организациям использовать строгое локальное пространство имен. Последнее полностью аналогично поддержанию приватной установки переменных среды на компьютере.

Вместе с тем заявить, что децентрализованная реализация DNS обойдет многие из проблем масштабируемости, было бы слишком просто. В сравнительном исследовании [Pappas et al., 2006] показано, что есть много компромиссов, которые следует учитывать, и что на самом деле существующая иерархическая структура DNS не так уж плоха, по крайней мере по двум причинам:

- в иерархической структуре не все узлы равны, а в случае DNS, в частности, узлы более высокого уровня проектируются по-разному. Например, несмотря на то что официально существует 13 корневых узлов, каждый из этих узлов имеет высокую степень распределения и реплицируется для обеспечения производительности и доступности. Чтобы проиллюстрировать это, корневой узел, предоставленный RIPE NCC, реализован примерно на 25 разных сайтах (все используют один и тот же IP-адрес), каждый из которых реализован как высоконадежный и реплицируемый кластер серверов. Опять же, мы видим важное различие между логическим и физическим дизайном. Использование этой разницы имеет решающее значение для работы распределенной системы, такой как DNS. Однако практически во всех системах, основанных на DHT, сделать это различие может быть гораздо сложнее при работе с иерархией логических имен, поскольку все имена обязательно рассматриваются как равные. В таких случаях становится намного сложнее спроектировать систему так, чтобы, например, домены верхнего уровня были разделены специальными (физическими) узлами. Конечно, очевидный недостаток отсутствия равенства всех узлов заключается в том, что необходимо принимать специальные меры для защиты наиболее важных частей системы от злоупотреблений. Мы уже упоминали, что узлы верхнего уровня в DNS реализуются как распределенный и реплицируемый серверы (кластеры), но также и то, что связанный сервер не будет обеспечивать рекурсивное разрешение имен. Такие решения по реализации необходимы также и с точки зрения надежности;
- DNS-кеши очень эффективны и почти полностью управляются локальным распределением запросов: если домен D часто запрашивается на сервере S, то ссылки на серверы имен D будут кешироваться на S. Поведение на другом сервере S′ определяетсятем тем, что запрашивается на S′. Эта важная особенность была подтверждена в более недавнем исследовании, которое также показывает, насколько сложно понять эффективность кеширования и принципы локальности преобразователей DNS. В частности, распознаватель DNS провайдера (ISP) может быть очень эффективным при перенаправлении трафика на контент, локализованный в этом провайдере [Ager et al., 2010]. Напротив, кеширование и репликация в системах на основе DHT обычно не демонстрируют таких принципов локальности: результаты просто кешируются в узлах на обратном пути поиска и имеют очень мало общего с тем фактом, что поиск был инициирован локально в конкретный узел в DHT или распознаватель, в котором местный интернет-провайдер может помочь искать контент.

Факт остается фактом: замена DNS децентрализованной реализацией не обязательно является хорошей идеей. DNS в нынешнем виде – это хорошо спроектированная система, которую трудно победить, когда речь идет о производительности и надежности [Vixie, 2009; 2014].

Пример: сетевая файловая система

В качестве другого примера рассмотрим наименование в NFS. Фундаментальная идея, лежащая в основе модели наименования NFS, заключается в предоставлении клиентам полного прозрачного доступа к удаленной (находящейся на расстоянии) файловой системе, поддерживаемой сервером. Эта прозрачность достигается за счет того, что клиент может смонтировать удаленную файловую систему в свою собственную локальную файловую систему, как показано на рис. 5.24.

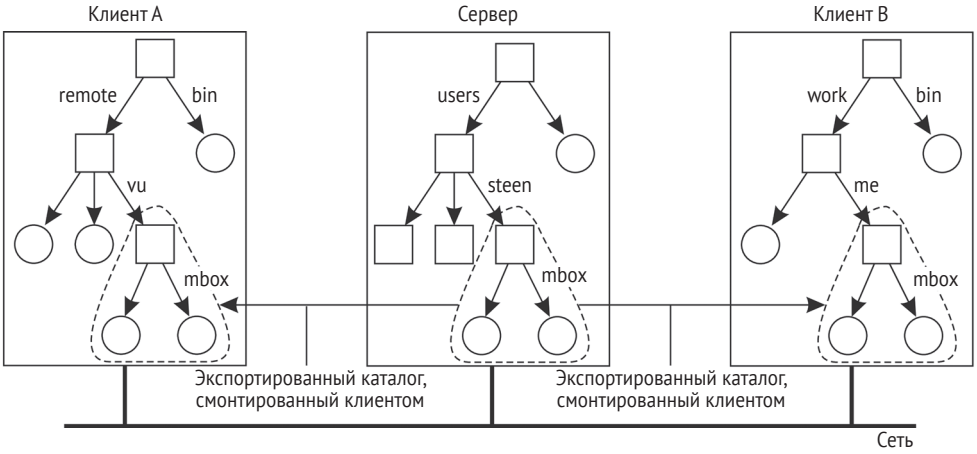


Рис. 5.24 ❖ Монтирование (часть) удаленной файловой системы в NFS

Вместо монтирования всей файловой системы NFS позволяет клиентам монтировать только часть файловой системы, как также показано на рис. 5.24. Говорят, что сервер экспортирует каталог, когда он создает этот каталог и его записи становятся доступными для клиентов. Экспортированный каталог может быть подключен к локальному пространству имен клиента.

Такой подход к проектированию имеет принципиальные последствия: пользователи не разделяют пространства имен. Как показано на рис. 5.24, файл с именем /remote/vu/mbox у клиента A у клиента B называется /work /me/mbox. Следовательно, имя файла зависит от того, как клиенты организуют свое собственное локальное пространство имен и где смонтированы экспортированные каталоги. Недостаток этого подхода в распределенной файловой системе заключается в том, что совместное использование файлов становится намного сложнее. Например, Алиса не может сообщить Бобу

о файле, используя имя, которое она присвоила этому файлу, поскольку это имя может иметь совершенно другое значение в пространстве имен Боба.

Существует несколько способов решения данной проблемы, но наиболее распространенным является предоставление каждому клиенту частично стандартизированного пространства имен. Например, каждый клиент может использовать локальный каталог /usr/bin для монтирования файловой системы, содержащей стандартный набор программ, доступных каждому. Аналогично каталог /local может использоваться в качестве стандарта для монтирования локальной файловой системы, которая расположена на хосте клиента.

Сервер NFS может сам монтировать каталоги, которые экспортируются другими серверами. Однако не разрешается экспортировать эти каталоги своим клиентам. Вместо этого клиент должен будет явно смонтировать такой каталог с сервера, который его обслуживает, как показано на рис. 5.25. Это ограничение частично обусловлено простотой. Если сервер может экспортировать каталог, который он смонтировал с другого сервера, он должен будет возвращать специальные файловые дескрипторы, которые содержат идентификатор для сервера. NFS не поддерживает такие файловые дескрипторы.

Чтобы объяснить это более подробно, предположим, что на сервере A размещен файловый системный FS_A, из которого он экспортирует каталог /packages. Этот каталог содержит подкаталог /draw, который действует как точка монтирования файловой системы FS_B, которая экспортируется сервером B и монтируется A. Пусть A также экспортирует /packages/draw для своих клиентов и предполагает, что клиент смонтировал /packages в его локальный каталог /bin, как показано на рис. 5.25.

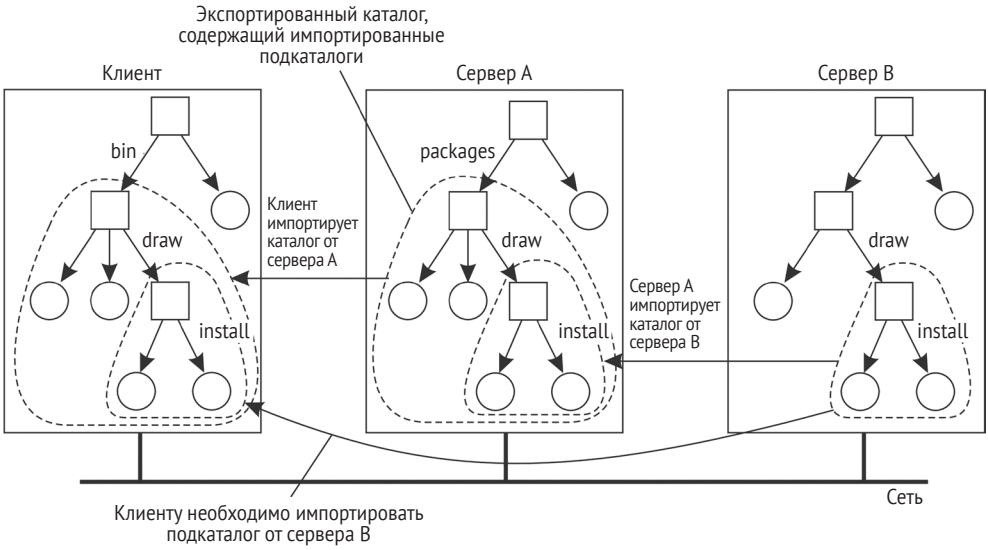


Рис. 5.25 ❖ Монтирование вложенных каталогов от нескольких серверов в NFS

Если разрешение имени является итеративным, то для разрешения имени /bin/draw/install клиент связывается с сервером A, когда он локально разрешил /bin, и просит A вернуть дескриптор файла для directory/draw. В этом случае сервер A должен вернуть дескриптор файла, который содержит идентификатор для сервера B, поскольку только B может разрешить оставшуюся часть пути, в данном случае install. Как мы уже говорили, этот вид разрешения имен не поддерживается NFS.

Разрешение имен в более ранних версиях NFS строго итеративно в том смысле, что можно искать только одно имя файла за один раз. Другими словами, для разрешения имени, такого как /bin/draw/install, требуется три отдельных вызова NFS-сервера. Кроме того, клиент несет полную ответственность за реализацию разрешения имени пути. NFSv4 также поддерживает рекурсивный поиск имен. В этом случае клиент может передать полный путь к серверу и запросить этот сервер для его разрешения.

Есть еще одна особенность поиска имен NFS, которая была решена в самой последней версии (NFSv4). Рассмотрим файловый сервер с несколькими файловыми системами. При строгом итеративном разрешении имен всякий раз, когда выполняется поиск для каталога, в котором смонтирована другая файловая система, поиск будет возвращать дескриптор файла каталога. Последующее чтение этого каталога вернет его исходное содержимое, а не содержимое корневого каталога смонтированной файловой системы.

Для объяснения предположим, что в нашем предыдущем примере обе файловые системы FS_A и FS_B размещены на одном сервере. Если клиент смонтировал /packages в свой локальный каталог /bin, то поиск имени файла draw на сервере вернул бы дескриптор файла для draw. Последующий вызов на сервер для перечисления записей каталога draw с помощью readdir затем вернет список записей каталога, которые изначально были сохранены в FS_A, в подкаталоге /packages/draw. Только если клиент также смонтировал файловую систему FS_B, можно будет правильно разрешить путь к файлу draw/install относительно /bin.

NFSv4 решает эту проблему, позволяя поискам пересекать точки монтирования на сервере. В частности, lookup возвращает дескриптор файла *смонтированного* каталога вместо оригинального каталога. Клиент может обнаружить, что поиск пересек точку монтирования, проверив идентификатор файловой системы искомого файла. При необходимости клиент может также локально смонтировать эту файловую систему.

Дескриптор файла – это ссылка на файл в файловой системе. Она не зависит от имени файла, к которому он относится. Дескриптор файла создается сервером, на котором размещена файловая система, и является уникальным по отношению ко всем файловым системам, экспортируемым сервером. Он создается при создании файла. Клиент остается в неведении относительно фактического содержания дескриптора файла; это совершенно непрозрачно. Файловые дескрипторы были 32 байта в NFS версии 2, но были переменными до 64 байтов в версии 3 и 128 байтов в версии 4. Конечно, длина дескриптора файла не непрозрачна.

В идеале дескриптор файла должен быть реализован как истинный идентификатор файла для файловой системы. Это означает, что пока файл су-

ществует, он должен иметь один и тот же дескриптор файла. Данное требование постоянства позволяет клиенту хранить дескриптор файла локально, как только связанный файл был найден с помощью его имени. Одним из преимуществ этого является производительность: поскольку для большинства файловых операций требуется дескриптор, а не имя, клиент может избежать необходимости повторного поиска имени перед каждой файловой операцией. Еще одним преимуществом этого подхода является то, что клиент теперь может получить доступ к файлу независимо от того, какое (текущее) имя он имеет.

Поскольку дескриптор файла может храниться клиентом локально, также важно, чтобы сервер не использовал повторно дескриптор файла после удаления файла. В противном случае клиент может ошибочно получить доступ к неправильному файлу, когда он использует свой локально сохраненный дескриптор файла.

Обратите внимание, что комбинация итеративного поиска имен, не позволяя операции поиска разрешить пересечение точки монтирования, представляет проблему с получением начального дескриптора файла. Чтобы получить доступ к файлам в удаленной файловой системе, клиенту необходимо предоставить серверу указатель файла каталога, в котором должен выполняться поиск, вместе с именем файла или каталога, который должен быть разрешен. NFSv3 решает эту проблему с помощью отдельного протокола монтирования: клиент фактически монтирует удаленную файловую систему. После монтирования клиенту передается обратно дескриптор корневого файла смонтированной файловой системы, который он впоследствии может использовать в качестве отправной точки для поиска имен.

В NFSv4 эта проблема решается путем предоставления отдельной операции putrootfh, которая указывает серверу разрешать все имена файлов относительно дескриптора корневого файла файловой системы, которой он управляет. Дескриптор корневого файла можно использовать для поиска любого другого дескриптора файла в файловой системе сервера. Этот подход имеет то дополнительное преимущество, что нет необходимости в отдельном протоколе монтирования.

Вместо этого монтирование может быть интегрировано в обычный протокол для поиска файлов. Клиент может просто смонтировать удаленную файловую систему, запросив у сервера разрешение имен, относящихся к дескриптору корневого файла файловой системы, с помощью putrootfh.

Примечание 5.13 (дополнительно: автомонтирование)

Как мы уже упоминали, модель наименования NFS предоставляет пользователям собственное пространство имен. Совместное использование в этой модели может стать затруднительным, если пользователи по-разному называют один и тот же файл. Одним из решений этой проблемы является предоставление каждому пользователю локального пространства имен, которое частично стандартизировано, а последующее монтирование удаленных файловых систем одинаково для каждого пользователя.

Другая проблема, связанная с моделью наименования NFS, связана с принятием решения о необходимости смонтировать удаленную файловую систему. Рассмот-

рим большую систему с тысячами пользователей. Предположим, что у каждого пользователя есть локальный каталог /home, который используется для монтирования домашних каталогов других пользователей. Например, домашний каталог Алисы может быть доступен ей локально как /home/alice, хотя фактические файлы хранятся на удаленном сервере. Этот каталог может быть автоматически смонтирован, когда Алиса заходит на свою рабочую станцию. Кроме того, она может иметь доступ к публичным файлам Боба, открывая каталог Боба через /home/bob.

Вопрос, однако, заключается в том, должен ли домашний каталог Боба также автоматически монтироваться при входе Алисы в систему. Преимущество этого подхода заключается в том, что весь бизнес монтирования файловых систем будет прозрачен для Алисы. Однако если эта политика будет соблюдаться для каждого пользователя, вход в систему может повлечь за собой большие административные и коммуникационные издержки. Кроме того, потребуется, чтобы все пользователи были известны заранее. Гораздо лучший подход заключается в прозрачном монтировании домашнего каталога другого пользователя по требованию, то есть когда это необходимо.

Монтирование по требованию удаленной файловой системы (или фактически экспортированного каталога) выполняется в NFS автомонтировщиком и реализуется как отдельный процесс на компьютере клиента. Принцип, лежащий в основе **автомонтировщика** (automounter), относительно прост. Рассмотрим простой автомонтировщик, реализованный как NFS-сервер уровня пользователя в операционной системе Unix. (Для других реализаций см. [Callaghan, 2000].)

Предположим, что для каждого пользователя домашние каталоги всех пользователей доступны через локальный каталог /home, как описано выше. Когда клиентский компьютер загружается, автомонтирование запускается с монтирования этого каталога. Эффект этого локального монтирования заключается в том, что всякий раз, когда программа пытается получить доступ к /home, ядро Unix будет перенаправлять операцию поиска клиенту NFS, который в этом случае будет перенаправлять запрос на автомонтировщик в роли сервера NFS, как показано на рис. 5.26.

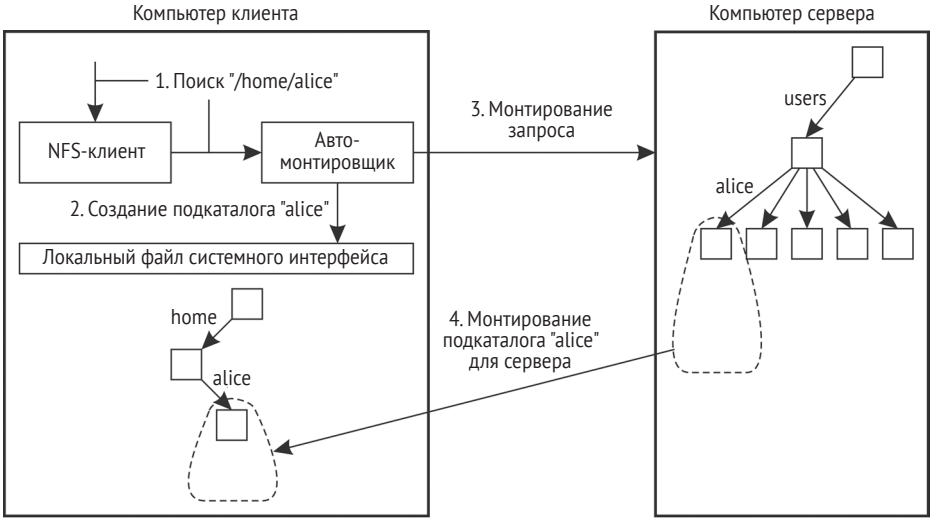


Рис. 5.26 ❖ Простой автомонтировщик для NFS

Предположим, например, что Алиса входит в систему. Программа входа в систему попытается прочитать каталог /home/alice, чтобы найти информацию, такую как сценарии входа. Таким образом, автомонтировщик получит запрос на поиск подкаталога /home/alice, и по этой причине он сначала создает подкаталог /alice в /home. Затем он ищет NFS-сервер, который экспортирует домашний каталог Алисы, чтобы потом смонтировать этот каталог в /home/alice. В этот момент программа входа может продолжиться.

Проблема с этим подходом состоит в том, что автомонтировщик должен участвовать во всех файловых операциях, чтобы гарантировать прозрачность. Если указанный файл не доступен локально, потому что соответствующая файловая система еще не смонтирована, автоустановщик должен знать об этом. В частности, он должен обрабатывать все запросы на чтение и запись, даже для файловых систем, которые уже были смонтированы. При таком подходе может возникнуть большая проблема с производительностью. Было бы лучше, чтобы автомонтировались только каталоги монтирования/размонтирования, но в остальном все оставалось вне цикла.

Простое решение состоит в том, чтобы позволить автомонтированию монтировать каталоги в специальном подкаталоге и устанавливать символическую ссылку на каждый смонтированный каталог. Этот подход показан на рис. 5.27.

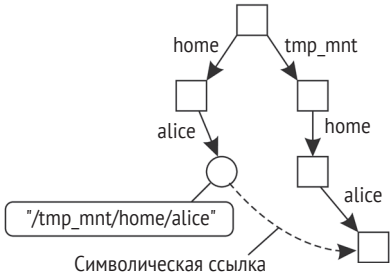


Рис. 5.27 ❖ Использование символических ссылок с автомонтированием

В нашем примере домашние каталоги пользователей монтируются как подкаталоги /tmp_mnt. Когда Алиса входит в систему, автомонтировщик монтирует свой домашний каталог в /tmp_mnt/home/alice и создает символическую ссылку /home/alice, которая ссылается на этот подкаталог. В этом случае всякий раз, когда Алиса выполняет команду, такую как ls--l/home/alice, NFS-сервер, который экспортирует домашний каталог Алисы, связывается напрямую без дальнейшего участия автомонтировщика.

5.4. Наименование на основе атрибутов

Плоские и структурированные имена обычно обеспечивают уникальный и независимый от местоположения способ ссылки на объекты. Кроме того, структурированные имена были частично разработаны для обеспечения удобного для человека способа наименования объектов, чтобы к ним можно было легко получить доступ. В большинстве случаев предполагается, что имя

относится только к одному объекту. Однако независимость от местоположения и удобство для людей – не единственный критерий для наименования объектов. В частности, по мере того как появляется больше информации, становится важно эффективно искать объекты. Этот подход требует, чтобы пользователь мог просто предоставить описание того, что он ищет.

Существует много способов предоставления описаний, но популярным в распределенных системах является описание объекта в терминах пар (атрибут, значение), обычно называемых **именами на основе атрибутов** (attribute-based naming). В этом подходе предполагается, что у объекта есть связанная коллекция атрибутов. Каждый атрибут говорит что-то об этом объекте. Указывая, какие значения должен иметь конкретный атрибут, пользователь по существу ограничивает набор объектов, которые ему интересны. Система наименования должна вернуть один или несколько объектов, которые соответствуют описанию пользователя. В этом разделе мы подробнее рассмотрим системы наименования на основе атрибутов.

Службы каталогов

Системы наименования на основе атрибутов также известны как **службы каталогов** (directory servers), тогда как системы, которые поддерживают структурированное наименование, обычно называют **системами имен** (naming systems). В службе каталогов блоки имеют набор связанных атрибутов, которые могут быть использованы для поиска. В некоторых случаях выбор атрибутов может быть относительно простым. Например, в системе электронной почты сообщения могут быть помечены атрибутами для отправителя, получателя, темы и т. д. Однако даже в случае с электронной почтой проблемы становятся сложными, когда требуются другие типы дескрипторов, что иллюстрируется трудностью разработки фильтров, которые позволяют пропускать только определенные сообщения (на основе их дескрипторов).

Все это приводит к тому, что разработка соответствующего набора атрибутов является нетривиальной задачей. В большинстве случаев дизайн атрибутов должен выполняться вручную. Даже если существует консенсус в отношении набора атрибутов для использования, практика показывает, что установка значений последовательно для разных групп людей сама по себе является проблемой, что многим приходилось испытывать при доступе к музыкальным и видеобазам данных в интернете.

Чтобы облегчить некоторые из этих проблем, были проведены исследования по унификации способов описания ресурсов. В контексте распределенных систем одной особенно важной разработкой является **структура описания ресурсов** (resource description framework, RDF). Основой для модели RDF является то, что ресурсы описываются как триплеты, состоящие из субъекта, предиката и объекта. Например, (Person, name, Alice) описывает ресурс «персона, имя которой Алиса». В RDF каждый субъект, предикат или объект может быть самим ресурсом. Это означает, что Alice можно реализовать как ссылку на файл, который может быть впоследствии извлечен. В случае предиката такой ресурс может содержать текстовое описание этого предиката. Ресурсы,

связанные с предметами и объектами, могут быть чем угодно. Ссылки в RDF по сути являются URL-адресами.

Если описания ресурсов сохраняются, становится возможным запросить это хранилище способом, который является общим для многих систем наименования на основе атрибутов. Например, приложение может запросить информацию, связанную с человеком по имени Алиса. Такой запрос будет возвращать ссылку на ресурс человека, связанный с Алисой. Этот ресурс может впоследствии быть выбран приложением.

В этом примере описания ресурсов хранятся в одном месте. Нет причин, по которым ресурсы должны также находиться в одном месте. Однако отсутствие описаний в одном и том же месте может привести к серьезным проблемам с производительностью. В отличие от структурированных систем наименования, поиск значений в системе именования на основе атрибутов, по существу, требует исчерпывающего поиска по всем дескрипторам. (Чтобы избежать таких исчерпывающих поисков, могут применяться различные методы, одним из которых является индексация.) При рассмотрении производительности исчерпывающий поиск может представлять меньшую проблему в пределах одного, нераспределенного хранилища данных. Просто отправлять поисковый запрос сотням серверов, которые совместно используют распределенное хранилище данных, – как правило, не очень хорошая идея. Далее мы рассмотрим различные подходы к решению этой проблемы в распределенных системах.

Иерархические реализации: протокол LDAP

Распространенным подходом к работе со службами распределенных каталогов является объединение структурированных имен с именами на основе атрибутов. Этот подход получил широкое распространение, например, в службе Microsoft's Active Directory и других системах. Многие из этих систем используют **протокол облегченного доступа к каталогам**, который обычно называют просто LDAP (lightweight directory access protocol). Служба каталогов LDAP была получена из службы каталогов OSI X.500. Как и во многих службах OSI, качество связанных с ними реализаций препятствовало широкому распространению, и для его полезности требовались упрощения. Подробную информацию о LDAP можно найти в [Arkills, 2003].

Концептуально служба каталогов LDAP состоит из ряда записей, обычно называемых записями каталогов. Запись в каталоге сопоставима с записью ресурса в DNS. Каждая запись состоит из набора пар (*attribute, value*), где каждый атрибут имеет связанный тип. Различают однозначные атрибуты и многозначные атрибуты. Последние обычно представляют собой массивы и списки. В качестве примера простая запись каталога, идентифицирующая сетевые адреса некоторых общих серверов из рис. 5.23, показана на рис. 5.28.

В нашем примере мы использовали соглашение об именах, описанное в стандартах LDAP, которое применяется к первым пяти атрибутам. Атрибуты Organization и OrganizationUnit описывают, соответственно, организацию и отдел, связанные с данными, которые хранятся в записи. Аналогично атри-

буты Locality и Country предоставляют дополнительную информацию о том, где хранится запись. Атрибут CommonName часто используется в качестве (неоднозначного) имени для идентификации записи в ограниченной части каталога. Например, имени «Mail_Servers» может быть достаточно, чтобы найти в нашем примере запись с учетом конкретных значений для других четырех атрибутов – Country, Locality, Organization и OrganizationalUnit. В нашем примере только атрибут Mail_Servers имеет несколько значений, связанных с ним. Все остальные атрибуты имеют лишь одно значение.

Атрибут	Аббревиатура	Значение
Country	C	NL
Locality	L	Amsterdam
Organization	O	VU University
OrganizationUnit	OU	Computer Science
CommonName	CN	Main Server
Mail_Servers	–	137.37.20.3, 130.37.24.6, 137.37.20.10
FTP-server	–	130.37.20.20
WWW-server	–	130.37.20.20

Рис. 5.28 ❖ Простой пример записи каталога LDAP с использованием соглашений об именах LDAP

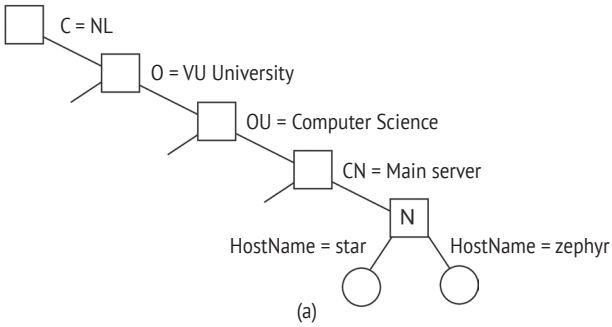
Коллекция всех записей каталога в службе каталогов LDAP называется **справочной информационной базой** (directory information base, DIB). Важным аспектом DIB является то, что каждая запись имеет уникальное имя, чтобы ее можно было найти. Такое глобально уникальное имя появляется в виде последовательности именующих атрибутов в каждой записи. Каждый атрибут наименования называется **относительным отличительным именем** (relative distinguished name, RDN). В нашем примере на рис. 5.28 все пять атрибутов являются атрибутами наименования. Используя обычные сокращения для представления атрибутов именования в LDAP, как показано на рис. 5.28, атрибуты Country, Organization и OrganizationalUnit могут использоваться для формирования глобально уникального наименования /C = NL/O = VU University/OU = Computer Science, аналогично DNS-имени nl.vu.cs.

Как и в DNS, использование глобально уникальных имен путем последовательного перечисления RDN приводит к иерархии коллекции записей каталога, которая называется **деревом информации каталога** (directory information tree, DIT). DIT по существу формирует граф наименования службы каталогов LDAP, в котором каждый узел представляет запись каталога. Кроме того, узел может также действовать как каталог в традиционном смысле, в котором может быть несколько дочерних элементов, для которых узел действует как родительский. Для объяснения рассмотрим график наименования, который частично показан на рис. 5.29. (Напомним, что метки связаны с краями.)

Узел N соответствует записи каталога, показанной на рис. 5.28. В то же время этот узел действует как родительский для ряда других записей каталога,

которые имеют дополнительный атрибут наименования HostName, использующийся в качестве RDN. Например, подобные записи могут использоваться для представления хостов, как показано на рис. 5.29.

Таким образом, узел в графе имен LDAP может одновременно представлять каталог в традиционном смысле, как мы обсуждали ранее, а также запись LDAP. Это различие поддерживается двумя различными операциями поиска. Операция чтения `read` используется для чтения одной записи с указанием ее пути в DIT. Напротив, операция `list` используется для перечисления имен всех исходящих ребер данного узла в DIT. Каждое имя соответствует дочернему узлу данного узла. Обратите внимание, что операция со списком не возвращает никаких записей; она просто возвращает имена.



(a)

Атрибут	Значение	Атрибут	Значение
Locality	Amsterdam	Locality	Amsterdam
Organization	VUUniversity	Organization	VUUniversity
OrganizationalUnit	ComputerScience	OrganizationalUnit	ComputerScience
CommonName	Mainserver	CommonName	Mainserver
HostName	star	HostName	zephyr
HostAddress	192.31.231.42	HostAddress	137.37.20.10

(b)

Рис. 5.29 ❖ a) Часть информационного дерева справочника; b) две записи каталога, имеющие HostName в качестве RDN

Другими словами, вызывая `read` с вводом имени /C = NL/O = VU University/OU = Computer Science/CN = Main server, возвращается запись, показанная на рис. 5.29, тогда как список вызовов вернет имена star и zephyr из записей, показанных на рис. 5.29, а также имена других хостов, которые были зарегистрированы аналогичным образом.

Реализация службы каталогов LDAP происходит во многом так же, как и реализация службы наименования, такой как DNS, за исключением того, что LDAP поддерживает больше операций поиска, что мы вскоре обсудим. При работе с крупномасштабным каталогом DIT обычно разделяется и распределяется по нескольким серверам, известным как **агенты службы каталогов** (directory service agents, DSA). Таким образом, каждая часть разделенного DIT соответствует зоне в DNS. Аналогично каждый DSA ведет себя почти

так же, как обычный сервер имен, за исключением того, что он реализует ряд типичных служб каталогов, таких как расширенные операции поиска.

Клиенты представлены так называемыми **агентами пользователя каталога** (directory service agents, DUA). DUA похож на распознаватель имен в службах структурированных имен. DUA обменивается информацией с DSA в соответствии со стандартизированным протоколом доступа.

Что отличает реализацию LDAP от реализации DNS, так это возможности поиска через DIB. В частности, предоставляются средства для поиска записи каталога с учетом набора критериев, которым должны соответствовать атрибуты искомых записей. Например, предположим, что мы хотим получить список всех основных серверов в университете VU. Используя обозначение, определенное в [Howes, 1997], подобный список можно вернуть с помощью операции поиска, такой как

```
search(''(C=NL)(O=VU University)(OU=*)(CN=Main server)'')
```

В этом примере мы указали, что местом для поиска главных серверов является организация с именем VU_University в стране NL, но мы не заинтересованы в конкретной организационной единице. Однако каждый возвращаемый результат должен иметь атрибут CN, равный Main_server.

Как мы уже упоминали, поиск в службе каталогов обычно является дорогостоящей операцией. Например, чтобы найти все основные серверы в VU University, необходимо выполнить поиск всех записей в каждом отделе и объединить результаты в одном ответе. Другими словами, нам, как правило, нужно получить доступ к нескольким конечным узлам DIT, чтобы получить ответ. На практике это также означает, что необходимо получить доступ к нескольким DSA. Напротив, сервисы именования часто могут быть реализованы таким образом, что операция поиска требует доступа только к одному листовому узлу.

Всю эту настройку LDAP можно продвинуть на один шаг вперед, позволив сосуществовать нескольким деревьям, а также связать их друг с другом. Этот подход используется в Microsoft's Active Directory, ведущей к *лесу* доменов LDAP [Allen and Lowe-Norris, 2003]. Очевидно, что поиск в такой организации может быть чрезвычайно сложным. Чтобы обойти некоторые проблемы масштабируемости, Active Directory обычно предполагает наличие глобального сервера индексирования (называемого глобальным каталогом), который можно искать в первую очередь. Индекс будет указывать, какие домены LDAP необходимо искать в дальнейшем.

Хотя LDAP сам по себе уже использует иерархию для масштабируемости, обычно LDAP объединяют с DNS. Например, каждое дерево в LDAP должно быть доступно в корне (в Active Directory он известен как контроллер домена). Корень часто известен под DNS-именем, которое, в свою очередь, может быть найдено через соответствующую запись SRV, как мы объяснили выше.

Децентрализованные реализации

Примечательно, что с появлением одноранговых систем исследователи также искали решения для децентрализованных систем наименования на осно-

ве атрибутов. В частности, одноранговые системы нередко используются для хранения файлов. Первоначально файлы не могли быть найдены – их можно было найти только по ключу. Однако возможность поиска файла на основе дескрипторов может быть чрезвычайно удобной, когда каждый дескриптор представляет собой не что иное, как пару (атрибут, значение). Очевидно, что запрос каждого узла в одноранговой системе на предмет наличия в нем файла, соответствующего одной или нескольким таким парам, невозможен. Нам нужно сопоставить пары (атрибут, значение) с серверами индекса, которые, в свою очередь, указывают на файлы, соответствующие этим парам.

Использование распределенного индекса

Давайте сначала посмотрим на ситуацию построения (распределенного) индекса. Основная идея заключается в том, что поисковый запрос формулируется как список пар (*атрибут, значение*), как в случае наших примеров LDAP. Результатом должен быть список (ссылки на) объектов, которые соответствуют *всем* парам. В случае одноранговой системы, хранящей файлы, может быть возвращен список ключей для соответствующих файлов, после чего клиент может искать каждый из этих файлов, используя возвращенные ключи.

Прямой подход к распределенному индексу заключается в следующем. Предположим, есть d разных атрибутов. В этом случае мы можем использовать сервер для каждого из атрибутов d, где сервер для атрибута A поддерживает набор пар (E, val) для каждого объекта E, который имеет значение val для атрибута A. Поисковый запрос будет таким:

```
search(''(Country=NL)(Organization=VU University)
       (OrganizationalUnit=*)(CommonName=Main server)'')
```

Он будет отправлен на серверы для Country, Organization и CommonName соответственно, после чего клиент должен будет увидеть, какие объекты встречаются *во всех* трех наборах, возвращенных серверами. Чтобы предотвратить необходимость поддержки сервером очень большого набора объектов, набор для каждого сервера может быть дополнительно разделен и распределен по нескольким субсерверам, каждый субсервер связан с одним и тем же атрибутом.

Более точно: если у нас есть набор атрибутов $\{a^1, \ldots, a^N\}$, то для каждого атрибута a^k мы связываем множество $\mathbf{S^k} = \{S_1^k, \ldots, S_{n_k}^k\}$ n_k серверов. Предполагая, что атрибут a^k принимает значения из множества R^k, мы строим глобальное отображение F так, что

$F(a^k, v) = S_j^k$ при $S_j^k \in \mathbf{S^k}$ и $v \in R^k$.

В этом примере сервер S_j^k будет отслеживать каждый ключ, связанный с файлом, имеющим $a^k = v$. Прелесть данной схемы в ее простоте. Если $L(a^k, v)$ – это набор ключей, возвращаемых сервером $F(a^k, v)$, тогда запрос может быть сформулирован как логическое выражение, такое как

$(F(a^1, v^1) \wedge F(a^2, v^2)) \vee F(a^3, v^3)$,

которое затем может быть обработано на стороне клиента путем построения набора

$(L(a^1, v^1) \cap L(a^2, v^2)) \cup L(a^3, v^3)$.

К сожалению, у этой схемы есть серьезные недостатки. Во-первых, любой запрос, включающий *k* атрибутов, требует обращения к *k* серверам индексов, что может привести к значительным затратам на связь. Более того, клиент должен обрабатывать наборы, возвращаемые серверами индекса. Просто представьте, что у каждого файла есть два атрибута firstName и lastName соответственно и что клиент ищет файл, принадлежащий Pheriby Smith. Теперь, хотя Pheriby может быть совершенно уникальным для первого имени, Smith определенно нет. Однако нашему бедному клиенту придется получить, возможно, миллионы ключей файлов, для которых lastName = Smith, в то время как на самом деле может быть только несколько файлов, для которых firstName = Pheriby. В-третьих, хотя эта схема позволяет оставить некоторые атрибуты неуказанными (просто не упоминая их в запросе), она затрудняет поддержку диапазонных запросов, таких как цена = [1000–2500].

Пространственные кривые

Общий подход к реализации децентрализованных систем наименования на основе атрибутов заключается в использовании так называемых **кривых заполнения пространства** (space-filling curves). Основная идея состоит в том, чтобы отобразить *N*-мерное пространство, охватываемое *N* атрибутами $\{a^1, \ldots, a^N\}$, в одно измерение, а затем использовать, например, простую технику хеширования для распределения результирующего пространства между серверами индекса. Одна из ключевых проблем заключается в том, чтобы пары (*атрибут, значение*), которые были «близки» друг к другу, обрабатывались одним и тем же индексным сервером.

Давайте конкретизируем ситуацию и рассмотрим один популярный случай, а именно кривые заполнения пространства Гильберта (см., например, [Lawder and King, 2000]). Это проще всего объяснить, рассматривая только два измерения, то есть рассматривая только два различных атрибута. Возможные значения, которые может иметь каждый атрибут, соответствуют одной оси в двумерном пространстве. Без ограничения общности мы предполагаем, что каждый атрибут принимает значения в интервале [0, 1). В качестве первого приближения квадрата мы разделим его на четыре квадранта, как показано на рис. 5.30a. Все значения данных (x, y) с $0 \le x, y < 0.5$ связаны с индексом 0. Значения (x, y) с $0.5 \le x, y < 1.0$ связаны с индексом 2.

Мы можем повторить эту процедуру рекурсивно для каждого субквадрата: разделить его на четыре меньших квадрата и соединить меньшие квадраты одной линией. Используя вращение и отражение, мы гарантируем, что эта линия может быть успешно соединена с той, что была в соседнем ранее большом квадрате (который также был разделен на меньшие квадраты). Для иллюстрации на рис. 5.30a показана кривая Гильберта порядка 1, на рис. 5.30b – кривая порядка 4 с 256 индексами. В общем, кривая Гильберта порядка *k* соединяет 2^{2k} субквадратов и, следовательно, имеет также 2^{2k} индексов. Су-

ществуют различные способы, которыми мы можем систематически рисовать кривую в двумерном пространстве, которое было разделено на квадраты одинакового размера. Кроме того, процесс может быть легко расширен до более высоких измерений, как объясняется в [Sagan, 1994] и [Bader, 2013].

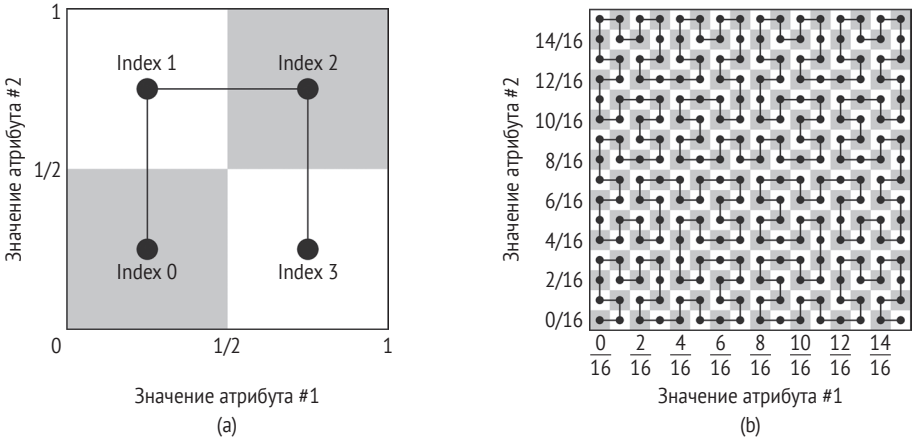


Рис. 5.30 ❖ Сведение двумерного пространства к одному измерению с помощью кривой заполнения пространства Гильберта: a) порядка 1; b) порядка 4

Важное свойство кривых заполнения пространства состоит в том, что они сохраняют локальность: два индекса, которые находятся близко друг к другу на кривой, соответствуют двум точкам, которые также близки друг к другу в многомерном пространстве. (Обратите внимание, что обратное не всегда верно: две точки близко друг к другу в многомерном пространстве не обязательно должны лежать близко друг к другу на кривой.)

Чтобы завершить, необходимо проделать следующее. Во-первых, значения атрибутов нужно проиндексировать. Предположим, что мы имеем дело с N возможными атрибутами $\{a^1, \ldots, a^N\}$ и что каждый объект назначает значение каждому из этих N атрибутов (возможно, эквивалент значения нас «не волнует»). Для простоты мы предполагаем, что каждое значение атрибута нормализуется к значению в интервале [0, 1). Тогда ясно, что объект E, имеющий кортеж значений $(v^1, \ldots, v^N)$, связан с действительной величиной координаты в N-мерном пространстве, в свою очередь уникально связанной с N-мерным квадратом, как мы обсуждали для двумерного случая. Центр такого субквадрата соответствует индексу на соответствующей кривой заполнения пространства Гильберта и теперь является индексом, связанным с объектом E. Конечно, все объекты, чьи связанные координаты попадают в один и тот же квадрат, будут иметь один и тот же индекс. Чтобы максимально избежать таких коллизий, нам нужно использовать кривые заполнения пространства высокого порядка. Порядки 32 или 64 не редкость.

Во-вторых, мы также должны иметь возможность искать объекты. Принцип поиска объектов по значениям их атрибутов теперь должен быть поня-

тен. Предположим, мы искали файлы, у которых два значения атрибутов a^1 и a^2 лежат в интервалах $[v_l^1, v_u^1)$ и $[v_l^2, v_u^2)$ соответственно (с $v_l^i < v_u^i$). Очевидно, что это очерчивает прямоугольную область, через которую проходит кривая, и все файлы, проиндексированные теми сегментами кривой, которые пересекаются с этой областью, соответствуют критерию поиска. Поэтому нам нужна операция, которая возвращает ряд связанных с кривой индексов для заданной области (выраженной в виде квадратов) в соответствующем *N*-мерном пространстве. Такая операция явно зависит от того, какая кривая заполнения пространства использовалась, но, что интересно, не обязательно зависит от реальных объектов.

Наконец, нам нужно поддерживать (ссылки на) объекты, связанные с индексами. Один подход, используемый в системе Squid [Schmidt and Parashar, 2008], заключается в использовании кольца Chord. В Squid индексное пространство выбрано таким же, как и в кольце Chord, то есть оба используют *m*-битные идентификаторы. Тогда ясно, что узел Chord, ответственный за индекс *i*, будет хранить (ссылки на) объекты, индексированные *i*.

Примечание 5.14 (дополнительная информация: другие примеры)

Децентрализованным реализациям систем наименования на основе атрибутов уделяется большое внимание. Те, которые основаны на кривых заполнения пространства, относительно популярны, но также было предложено несколько альтернатив. Давайте посмотрим на некоторые типичные примеры, чтобы пролить свет на пространство дизайна.

Деревья атрибут-значение. Это еще один пример, где пары (*атрибут, значение*) поддерживаются системой на основе DHT. Во-первых, предположим, что запросы состоят из пар, как в LDAP, то есть пользователь задает список атрибутов наряду с уникальным значением, которое он хочет видеть для каждого атрибута. Такие однозначные запросы поддерживаются в системе INS/Twine System [Balazinska et al., 2002]. Предполагается, что каждый объект (называемый ресурсом) описывается с помощью, возможно, иерархически организованных атрибутов, таких как показанные на рис. 5.31. Каждое такое описание преобразуется в **дерево значений атрибутов** (attribute-value tree, AVTree), которое затем используется в качестве основы для кодирования, хорошо отображаемой в системе на основе DHT.

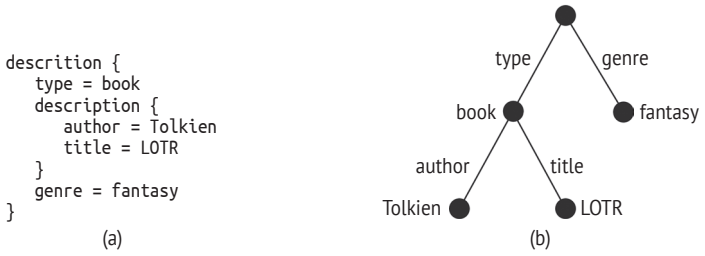


Рис. 5.31 ❖ a) Общее описание ресурса; b) его представление в качестве AVTree

Основная проблема заключается в том, чтобы снова преобразовать AVTrees в набор ключей, которые можно найти в системе DHT. В этом случае каждому пути, начинающемуся в корне, присваивается уникальное хеш-значение, где описание

пути начинается со ссылки (представляющей атрибут) и заканчивается либо узлом (значением), либо другой ссылкой. Взяв рис. 5.31 в качестве нашего примера, рассмотрим следующие ключи всех таких путей:

Ключ	Вычисляется как
h_1:	*хеш*(type-book)
h_2:	*хеш*(type-book-author)
h_3:	*хеш*(type-book-author-Tolkien)
h_4:	*хеш*(type-book-title)
h_5:	*хеш*(type-book-title-LOTR)
h_6:	*хеш*(genre-fantasy)

Узел, отвечающий за хеш-значение h_i, сохранит (ссылку на) фактический ресурс. В нашем примере это может привести к тому, что шесть узлов хранят информацию о Властелине колец Толкиена (Tolkien's Lord of the Ring, LOTR). Однако преимущество подобной избыточности заключается в том, что она позволяет поддерживать частичные запросы. Например, рассмотрим запрос типа «Вернуть книги, написанные Толкиеном». Этот запрос транслируется в AVTree, как показано на рис. 5.32, что приводит к вычислению следующих трех хешей:

h_1: *хеш*(type-book)
h_2: *хеш*(type-book-author)
h_3: *хеш*(type-book-author-Tolkien)

Эти значения будут отправлены на узлы, которые хранят информацию о книгах Толкиена, и, по крайней мере, вернут Властелина колец. Обратите внимание, что хеш, такой как h_1, является довольно общим и будет генерироваться часто. Этот тип хешей может быть отфильтрован из системы. Более того, нетрудно увидеть, что нужно оценивать только самые конкретные хеши.

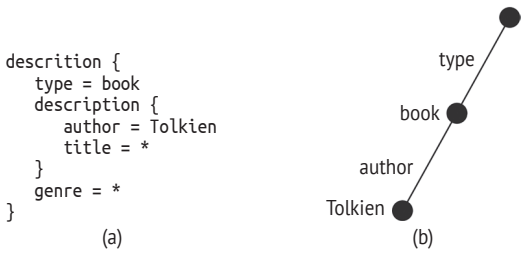


Рис. 5.32 ❖ a) Описание ресурса запроса; b) его представление в качестве AVT

SWORD: включение запросов диапазона. Давайте посмотрим на запросы, которые могут содержать спецификации диапазона для значений атрибутов. Мы обсуждаем решение, принятое в системе обнаружения ресурсов SWORD [Oppenheimer et al., 2005].

В SWORD пары (*атрибут, значение*) сначала преобразуются в ключ для DHT. Эти пары всегда содержат одно значение; только запросы могут содержать диапазоны значений для атрибутов. При вычислении ключа (с помощью хеша) имя атрибута и его значение хранятся отдельно. Определенные биты в ключе идентифицируют имя атрибута, в то время как другие идентифицируют его значение. Кроме того, ключ будет содержать несколько случайных битов, чтобы гарантировать уникальность среди всех ключей, которые должны быть сгенерированы. Таким образом,

пространство атрибутов удобно разбивается: если зарезервировано n бит для имен кодовых атрибутов, можно использовать $2n$ разных групп серверов, по одной группе для каждого имени атрибута. Аналогично, используя m бит для кодирования значений, можно применить дополнительное разбиение на группу серверов для хранения конкретных пар (*атрибут, значение*). DHT используются только для распространения имен атрибутов.

Для каждого имени атрибута возможный диапазон его значений разбивается на поддиапазоны, и каждому поддиапазону назначается отдельный сервер. Рассмотрим в качестве примера описание ресурса с двумя атрибутами: a^1 принимает значения в диапазоне [1..10] и a^2 принимает значения в диапазоне [101..200]. Предположим, есть два сервера для a^1: S^{11} заботится о записи значений a^1 в диапазоне [1..5] и $\mathbf{S^{12}}$ для значений в [6..10]. Аналогично сервер S^{21} записывает значения для a^2 в диапазоне [101..150], и сервер S^{22} для значений в [151..200]. Затем, когда объект E имеет ассоциированные значения атрибутов ($a^1 = 7$, $a^2 = 175$), сервер S^{12} *и* сервер S^{22} сохранят копию или ссылку на E.

Преимущество этой схемы в том, что запросы диапазона могут быть легко поддержаны. Когда выдается запрос на возврат ресурсов, у которых a^2 лежит между 165 и 189, запрос может быть перенаправлен на сервер S^{22}, который затем может вернуть ресурсы, соответствующие диапазону запросов. Недостаток, однако, заключается в том, что обновления необходимо отправлять на несколько серверов. Более того, не сразу понятно, насколько хорошо распределена нагрузка между различными серверами. В частности, если определенные запросы диапазона оказываются очень популярными, определенные серверы будут получать большую часть всех запросов.

Общие замечания. Существует много разных способов поддержки систем наименования децентрализованным способом на основе атрибутов. Суть во всех случаях заключается в том, чтобы назначать атрибуты серверам, дабы клиенты знали, куда направлять свои запросы, и в то же время удостоверились, что в наборе серверов есть баланс нагрузки. В этом свете поддержка запросов диапазона требует особого внимания, хотя бы для решения, как в SWORD, какой сервер будет отвечать за какой поддиапазон.

На практике мы видим, что при работе с N атрибутами многие системы моделируют набор пар (*атрибут, значение*) как N-мерное пространство, в котором каждый объект представлен уникальной точкой в этом пространстве. Концептуально поиск обращается к подпространству и приводит к идентификации серверов, ответственных за это подпространство. В простейшем случае мы присваиваем каждый атрибут одному серверу, ведущему к $\mathcal{O}(N)$ серверам. В этой схеме запрос на k атрибутов должен быть отправлен на k серверов, в то время как запрашивающий клиент должен объединить результаты. Мы обсуждали этот случай ранее. Проблема состоит в том, чтобы разделить диапазоны для каждого атрибута среди субсерверов так, чтобы получить разумный баланс рабочей нагрузки. Решение этой проблемы обсуждается в [Bharambe et al., 2004].

Вместо того чтобы позволить клиенту объединять результаты, мы можем позволить серверам сотрудничать. Для этого N-мерное пространство делится на подпространства путем разбиения каждого измерения d на n_d интервалы. Это разделение приводит в общей сложности к $n_1 \times \cdots \times n_N$ подпространствам, где каждое подпространство назначается отдельному серверу. Даже с $n_d = 2$ для каждого измерения мы столкнемся с $\mathcal{O}(2^N)$ серверами. Используя кривые заполнения пространства, мы можем сократить количество измерений до одного и использовать отдельный метод для определения того, какое N-мерное подпространство обслуживается каким сервером. Практика показывает, что балансировка нагрузки может стать проблемой. В HyperDex [Escriva et al., 2012] было встроено альтернативное решение, в ко-

тором число измерений по-прежнему сокращено, но больше одного, при этом поддерживается балансировка нагрузки. Авторы также решают проблему репликации и согласованности в случае, когда система наименования одновременно сохраняет объекты, которые она индексирует. В этом случае всякий раз, когда сервер индексирует объект E, он должен быть скопирован на соответствующий сервер.

Наконец, укажем на систему наименования на основе атрибутов, которая объединяет именование со службой выбора ресурсов [Stratan et al., 2012]. Опять же, пространство атрибутов моделируется *N*-мерным пространством, в котором каждый ресурс связан с координатой. В этой ситуации каждый ресурс поддерживает ссылку на другой ресурс, но тот, который отвечает за подпространство, экспоненциально увеличивает свой размер. Чистый эффект состоит в том, что каждый ресурс должен иметь только фиксированное число соседей, в то время как для направления запроса в соответствующее подпространство требуется только линейное количество шагов. Организация схожа с использованием финансовых таблиц в Chord.

5.5. Резюме

Имена используются для обозначения объектов. По сути, есть три типа имен. Адрес – это имя точки доступа, связанной с объектом, также называемое просто адресом объекта. Идентификатор – это другой тип имени. Он имеет три свойства: каждый объект ссылается ровно на один идентификатор, идентификатор относится только к одному объекту и никогда не назначается другому объекту. Наконец, понятные человеку имена предназначены для использования людьми и как таковые представлены в виде строк символов. Учитывая эти типы, мы делаем различие между плоским наименованием, структурированным наименованием и наименованием на основе атрибутов.

Системы плоского наименования по существу должны разрешать идентификатор по адресу ассоциированного объекта. Такое обнаружение объекта может быть сделано разными способами. Первый подход заключается в использовании широковещательной или многоадресной рассылки. Идентификатор объекта передается каждому процессу в распределенной системе. Процесс, предлагающий точку доступа для объекта, отвечает, предоставляя адрес для этой точки доступа. Очевидно, что этот подход имеет ограниченную масштабируемость.

Второй подход заключается в использовании указателей пересылки. Каждый раз, когда объект перемещается в следующее место, он оставляет указатель, указывающий, где он будет в следующий раз. Поиск объекта требует прохождения пути пересылки указателей. Чтобы избежать больших цепочек указателей, важно периодически сокращать цепочки.

Третий подход – выделить дом для объекта. Каждый раз, когда объект перемещается в другое место, он сообщает своему дому, где он находится. Поиск объекта начинается с запроса у его дома текущего местоположения.

Четвертый подход состоит в том, чтобы организовать все узлы в структурированную одноранговую систему и систематически назначать узлы объектам с учетом их соответствующих идентификаторов. За счет последующей разработки алгоритма маршрутизации, с помощью которого поисковые за-

просы перемещаются к узлу, ответственному за данный объект, возможно эффективное и надежное разрешение имен.

Пятый подход заключается в построении иерархического дерева поиска. Сеть делится на непересекающиеся домены. Домены могут быть сгруппированы в домены более высокого уровня (неперекрывающиеся) и т. д. Существует один домен верхнего уровня, который охватывает всю сеть. Каждый домен на каждом уровне имеет связанный узел каталога. Если объект находится в домене D, узел каталога следующего домена более высокого уровня будет иметь указатель на D. Узел каталога самого низкого уровня хранит адрес объекта. Узел каталога верхнего уровня знает обо всех объектах.

Структурированные имена легко организовываются в пространстве имен. Пространство имен может быть представлено графом имен, в котором узел представляет именованный объект, а метка на ребре представляет имя, под которым этот объект известен. Узел, имеющий множество исходящих ребер, представляет собой совокупность объектов и также известен как контекстный узел или каталог. Крупномасштабные графы имен часто организованы как корневые ациклические ориентированные графы.

Графы имен удобны для структурирования имен, понятных человеку. Объект может быть указан по имени пути. Разрешение имен – это процесс обхода графа имен путем последовательного поиска компонентов пути. Крупномасштабный граф наименования реализуется путем распределения его узлов по нескольким серверам имен. При разрешении имени пути путем обхода графа имен разрешение имени продолжается на следующем сервере имен, как только этим сервером будет достигнута реализация узла.

Более проблематичными являются схемы наименования на основе атрибутов, в которых объекты описываются набором пар (*атрибут, значение*). Запросы также формулируются как такие пары, по существу требующие исчерпывающего поиска по всем дескрипторам. Такой поиск возможен только тогда, когда дескрипторы хранятся в одной базе данных. Вместе с тем были разработаны альтернативные решения, с помощью которых пары сопоставляются с системами на основе DHT, что, по сути, приводит к распределению набора дескрипторов объектов. Используя кривые заполнения пространства, мы можем затем сделать ответственными за разные значения атрибута разные узлы, что помогает эффективно распределять нагрузку между узлами в случае операций поиска.

Глава 6

Координация

В предыдущих главах мы рассмотрели процессы и связь между процессами. Хотя общение между процессами важно, это еще не все. Тесно связано с этим то, как процессы взаимодействуют и синхронизируются друг с другом. Частично взаимодействие поддерживается посредством наименования, которое позволяет процессам совместно использовать ресурсы или объекты в целом.

В этой главе мы в основном сосредоточимся на том, как процессы могут синхронизировать и координировать свои действия. Например, важно, чтобы несколько процессов одновременно не обращались к общему ресурсу, такому как файл, а вместо этого предоставляли друг другу временный монопольный доступ. Другой пример – нескольким процессам иногда может потребоваться согласовать порядок событий, скажем, было ли отправлено сообщение m_1 из процесса P до или после сообщения m_2 из процесса Q.

Синхронизация и координация – два тесно связанных явления. При синхронизации процессов мы удостоверяемся, что один процесс ожидает, пока другой завершит свою работу. При работе с **синхронизацией данных** (data synchronization) проблема состоит в том, чтобы гарантировать, что два набора данных совпадают. Когда речь идет о **координации** (coordination), целью является управление в распределенной системе взаимодействиями и зависимостями между ними [Malone and Crowston, 1994]. С этой точки зрения можно утверждать, что координация включает в себя синхронизацию.

Очевидно, координация в распределенных системах зачастую намного сложнее, чем в однопроцессорных или многопроцессорных системах. Проблемы и решения, которые обсуждаются в этой главе, по своей природе довольно общие и встречаются в распределенных системах во многих различных ситуациях.

Мы начнем с обсуждения вопроса синхронизации на основе фактического времени, за которым следует синхронизация, в которой важен только относительный порядок, а не порядок в абсолютном времени.

Во многих случаях важно, чтобы группа процессов могла назначить в качестве координатора один процесс, что можно сделать с помощью алгоритмов выбора. Мы рассмотрим различные алгоритмы выбора координации в отдельном разделе. Перед этим мы разберем ряд алгоритмов координации взаимного исключения при использовании взаимных ресурсов. Мы также рассмотрим особый класс проблем координации местоположения систем, при котором процессы размещаются в многомерной плоскости.

Такие размещения пригодятся при работе с очень большими распределенными системами.

Мы уже сталкивались с системами публикации-подписки, но пока не обсуждали подробно, как сопоставляются подписки с публикациями. Есть много способов сделать это, и мы рассматриваем как централизованные, так и децентрализованные реализации.

Наконец, мы рассмотрим три различные проблемы координации на основе сплетен: агрегацию, выборку одноранговых объектов (пиров) и оверлейные конструкции.

Распределенные алгоритмы бывают самых различных типов и разрабатывались для очень разных типов распределенных систем. Многие примеры (и дополнительные ссылки) можно найти в [Andrews, 2000], [Cachin et al., 2011] и [Fokkink, 2013]. Более формальные подходы ко множеству алгоритмов можно найти в учебниках [Attiya and Welch, 2004], [Lynch, 1996], [Santoro, 2007] и [Tel, 2000].

6.1. Синхронизация часов

В централизованной системе время однозначно. Когда процесс хочет узнать время, он просто обращается к операционной системе. Если процесс A запрашивает время, а затем немного позже запрашивает время процесс B, значение, которое получает B, будет выше (или, возможно, равно) полученному значению, и оно, конечно, не будет ниже. В распределенной системе достижение соглашения по времени не является тривиальным.

В качестве простого примера приведем последствия нехватки глобального времени для программы Unix make. Обычно в Unix большие программы разбиваются на многие исходные файлы, так что изменения одного исходного файла требуют перекомпиляции только одного файла, а не всех файлов. Если программа состоит из 100 файлов и один файл был изменен, отсутствует необходимость перекомпилировать все файлы, что значительно увеличивает скорость, с которой могут работать программисты.

Обычно способ работы с программой make прост. Когда программист завершил изменение всех исходных файлов, он запускает программу make, и она проверяет время, когда в последний раз были изменены все исходные и объектные файлы. Если исходный файл input.c имеет время 2151, а соответствующий объектный файл input.o имеет время 2150, make знает, что input.c был изменен с момента создания input.o, и поэтому input.c должен быть перекомпилирован. С другой стороны, если output.c имеет время 2144, а output.o имеет время 2145, компиляция не требуется. Таким образом, make просматривает все исходные файлы, чтобы определить, какие из них необходимо перекомпилировать, и вызывает компилятор для их перекомпиляции.

Теперь представьте, что может произойти в распределенной системе, в которой не было глобального соглашения о времени. Предположим, что output.o имеет время 2144, а вскоре после этого output.c модифицируется, но ему назначается время 2143, потому что часы на его машине немного отстают,

как показано на рис. 6.1. Make не будет вызывать компилятор. В результате исполняемая двоичная программа будет содержать смесь из файлов объектов из старых и новых источников. Вероятно, она перестанет работать, и программист сойдет с ума, пытаясь понять, что не так с кодом.

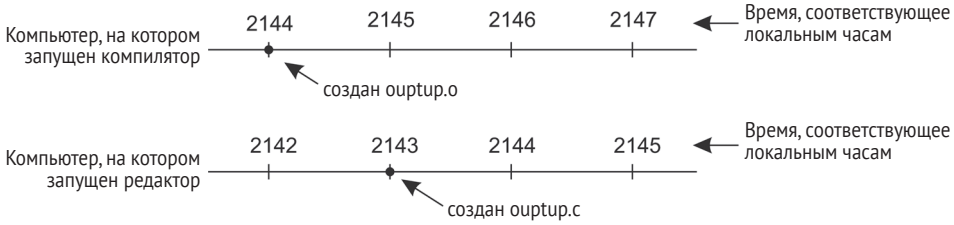


Рис. 6.1 ❖ Когда у каждой машины свои часы, событие, которое произошло после другого события, может быть назначено раньше

Можно привести много других примеров, когда требуется точный учет времени. Приведенный выше пример можно легко переформулировать, чтобы заполнить временные метки в целом. Кроме того, надо помнить о таких прикладных областях, как финансовая брокерская деятельность, аудит безопасности и совместное распознавание, и станет ясно, насколько важны точные сроки. Поскольку время для мышления людей столь важно и эффект от того, что не все часы синхронизированы, может быть настолько драматичным, что мы начнем изучение синхронизации с простого вопроса: можно ли синхронизировать все часы в распределенной системе? Ответ на удивление сложен.

Физические часы

Почти все компьютеры имеют схему для отслеживания времени. Несмотря на широкое использование для обозначения этих устройств слова «часы», на самом деле они не являются часами в обычном смысле. Таймер – пожалуй, лучшее слово. Компьютерный таймер обычно представляет собой точно обработанный кварцевый кристалл. Находясь под напряжением, кристаллы кварца колеблются с четко определенной частотой, которая зависит от типа кристалла, способа его резки и величины напряжения. С каждым кристаллом связаны два регистра, **счетчик** (counter) и **регистр хранения** (holding register). Каждое колебание кристалла уменьшает счетчик на единицу. Когда счетчик достигает нуля, генерируется прерывание, и счетчик перезагружается из регистра хранения.

Таким образом, можно запрограммировать таймер на генерацию прерывания 60 раз в секунду или на любой другой требуемой частоте. Каждое прерывание называется одним **временны́м тактом** (clock tick).

Когда система загружается, она обычно просит пользователя ввести дату и время, которые затем преобразуются в число тактов после некоторого известного времени запуска и сохраняются в памяти. Большинство компью-

теров имеют специальную схему CMOS RAM с резервным питанием от батареи, поэтому нет необходимости вводить дату и время при последующих загрузках. При каждом такте процедура обработки прерывания добавляет единицу ко времени, сохраненному в памяти. Таким образом, (программное обеспечение) часы обновляются.

При наличии одного компьютера и одних часов не имеет большого значения, если часы на некоторое время отключены. Поскольку все процессы на машине используют одни и те же часы, они все равно будут внутренне согласованы. Например, если файл input.c имеет время 2151, а файл input.o имеет время 2150, make перекомпилирует исходный файл, даже если часы выключены на 2, и истинное время равно 2153 и 2152 соответственно. Все, что действительно имеет значение, – это относительное время.

Как только вводится несколько процессоров, каждый со своими часами, ситуация радикально меняется. Хотя частота, на которой работает кварцевый генератор, обычно довольно стабильна, невозможно гарантировать, что кристаллы в разных компьютерах работают с одинаковой частотой. На практике, когда в системе имеется *n* компьютеров, все *n* кристаллов будут работать с несколько разными скоростями, в результате чего (программная) синхронизация часов будет постепенно нарушаться, и они будут давать разные значения при считывании. Эта разница во времени называется **фазовым сдвигом синхронизирующих импульсов** (clock skew). Как следствие этого сдвига тактов, программы, которые ожидают, что время, связанное с файлом, объектом, процессом или сообщением, будет правильным и независимым от машины, на которой оно было сгенерировано (то есть какие часы использовались), могут выполняться неправильно, как мы видели это в приведенном выше примере.

В некоторых системах (например, в системах реального времени) актуально действительное время часов. В этих условиях необходимы внешние физические часы. Из соображений эффективности и избыточности, как правило, желательно, чтобы физических часов было много, что порождает две проблемы: 1) как мы синхронизируем их с реальными часами и 2) как мы синхронизируем часы друг с другом?

Примечание 6.1 (дополнительная информация: определение в реальном времени)

Прежде чем ответить на эти вопросы, давайте немного отвлечемся, чтобы увидеть, как на самом деле измеряется время. Это не так просто, как можно подумать, особенно когда требуется высокая точность. Со времени изобретения механических часов в XVII веке время измерялось астрономически. Каждый день солнце поднимается на восточном горизонте, затем поднимается до максимальной высоты в небе и, наконец, опускается на запад. Событие, когда солнце достигает своей высшей видимой точки на небе, называется **переходом Солнца** (transit of the sun). Это событие происходит около полудня каждый день. Интервал между двумя последовательными переходами Солнца называется **солнечным днем** (solar day). Поскольку день состоит из 24 часов, каждый из которых содержит 3600 секунд, **солнечная секунда** (solar second) определяется как ровно 1/86400-й части солнечного дня. Геометрия расчета среднего солнечного дня показана на рис. 6.2.

В 1940-х годах было установлено, что период вращения Земли не является постоянным. Земля замедляется из-за приливного трения и атмосферного сопротивления. Основываясь на исследованиях закономерностей роста древних кораллов, геологи считают теперь, что 300 млн лет назад было около 400 дней в году. Продолжительность года (время одного оборота вокруг Солнца), как полагают, не изменилась; просто день стал длиннее. В дополнение к этой долгосрочной тенденции также происходят кратковременные изменения в длине дня, вероятно, вызванные турбулентностью расплавленного железа глубоко в ядре Земли. Эти открытия побуждают астрономов вычислять продолжительность дня, измеряя большое количество дней и принимая среднее значение перед делением на 86 400. Полученная величина была названа **средней солнечной секундой** (mean solar second).

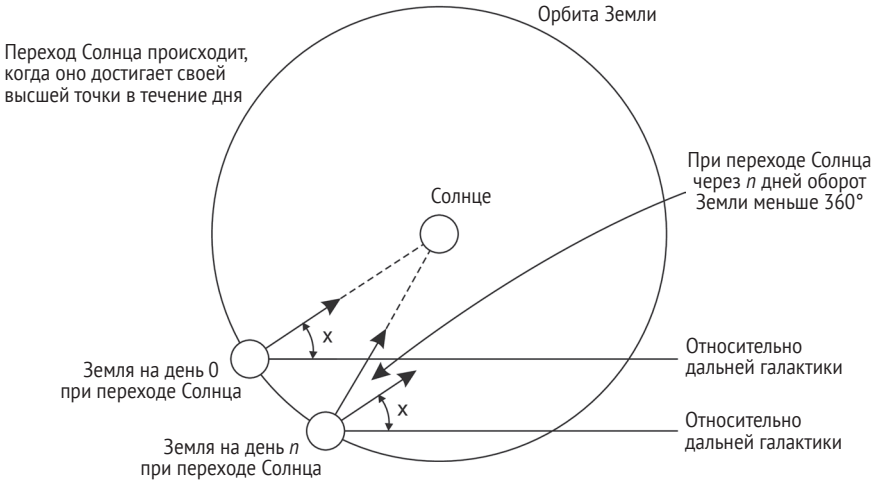


Рис. 6.2 ❖ Расчет среднего солнечного дня

С изобретением атомных часов в 1948 году стало возможным измерять время, подсчитывая переходы атома цезия-133, гораздо более точно и независимо от покачивания и колебания Земли. Физики отобрали работу по хронометражу у астрономов и определили, что секунда – это время, которое требуется атому 133 цезия, чтобы сделать ровно 9 192 631 770 переходов. Выбор 9 192 631 770 был сделан, чтобы сделать атомную секунду равной средней солнечной секунде в год ее появления. В настоящее время в нескольких лабораториях по всему миру установлены цезиевые часы. Периодически каждая лаборатория сообщает Международному бюро в Париже (the Bureau International de l'Heure, BIH), сколько раз тикали часы. Лаборатория BIH усредняет их, чтобы воспроизвести **Международное атомное время**, которое сокращенно обозначается как **TAI** (International Atomic Time). Таким образом, TAI – это просто среднее число тиков цезиевых 133 часов с полуночи 1 января 1958 года (начало времени), поделенное на 9 192 631 770.

Хотя TAI очень стабилен и доступен для всех, кто хочет купить цезиевые часы, существует серьезная проблема: 86 400 секунд TAI теперь примерно на 3 мс меньше, чем средний солнечный день (потому что средний солнечный день все время становится длиннее). Использование TAI для хранения времени будет означать, что с годами полдень наступит все раньше и раньше, пока в конечном итоге это не произойдет в первые утренние часы. Люди могут это заметить, и у нас может сложиться такая же ситуация, как и в 1582 году, когда папа Григорий XIII постановил исклю-

чить из календаря 10 дней. Это событие вызвало беспорядки на улицах, поскольку домовладельцы требовали арендную плату за весь месяц, банкиры – проценты за весь месяц, а работодатели отказывались платить работникам за 10 дней, в течение которых они не работали, и это лишь некоторые из конфликтов. Протестантские страны, в принципе, отказались иметь какое-либо отношение к папским указам и не принимали григорианский календарь в течение 170 лет.

В лаборатории BIH эта проблема решается вводом **високосных секунд** (leap seconds), когда расхождение между TAI и солнечным временем увеличивается до 800 мс. Использование високосных секунд показано на рис. 6.3. Подобная коррекция дает начало системе времени, основанной на постоянных секундах TAI, но которая остается в фазе с видимым движением солнца. Эта система всеобщего скоординированного времени известна как UTC (Universal Coordinated Time).

Рис. 6.3 ❖ Секунды TAI имеют постоянную длину, в отличие от солнечных секунд. Дополнительные секунды вводятся при необходимости, чтобы держаться в фазе с солнцем

Большинство электроэнергетических компаний синхронизируют свои тактовые частоты 60 Гц или 50 Гц с UTC, поэтому, когда BIH объявляет високосную секунду, энергетические компании повышают частоту до 61 Гц или 51 Гц в течение 60 или 50 с, чтобы увеличить время в их зоне распространения. Поскольку 1 с является заметным интервалом для компьютера, операционная система, которая должна сохранять точное время в течение нескольких лет, должна иметь специальное программное обеспечение для учета високосных секунд, когда они объявляются (если они не используют линию питания для времени, которое обычно слишком грубо). Общее количество високосных секунд, введенных до настоящего времени в UTC, составляет около 30.

Основой для хранения глобального времени выступает **универсальное координированное время** (Universal Coordinated Time, UTC). UTC является основой всего современного гражданского хронометража и мировым стандартом. Чтобы предоставить UTC людям, которым необходимо точное время, около 40 коротковолновых радиостанций по всему миру передают короткие импульсы в начале каждой секунды UTC. Точность этих станций составляет около 1 мс, но из-за случайных атмосферных колебаний, которые могут влиять на длину пути прохождения сигнала, на практике точность составляет не лучше 10 мс.

Несколько спутников Земли также предлагают услугу UTC. Оперативный спутник Геостационарной среды может обеспечить точное время UTC до 0,5 мс, а некоторые другие спутники работают даже лучше. Комбинируя приемы от нескольких спутников, можно построить наземные серверы времени,

обеспечивающие точность 50 нс. Приемники UTC имеются в продаже, и многие компьютеры оснащены таковыми.

Алгоритмы синхронизации часов

Если у одного компьютера есть UTC-приемник, то стоит задача – синхронизировать с ним все остальные компьютеры. Если ни на одном из компьютеров нет приемника UTC и каждый компьютер отслеживает свое время, и задача состоит в том, чтобы как можно лучше обеспечить синхронизацию всех компьютеров. Было предложено много алгоритмов для такой синхронизации. Их обзоры предоставлены в [Ramanathan et al., 1990], [Horauer, 2004] и [Shin et al., 2011].

Все часы основаны на каком-то гармоническом осцилляторе: объекте, который резонирует на определенной частоте и из которого мы можем затем извлечь время. Атомные часы основаны на переходах атома цезия-133, частота которых не только очень высока, но и очень стабильна. Аппаратные часы в большинстве компьютеров используют генератор на основе кварца, который также способен генерировать очень высокую стабильную частоту, хотя и не такую стабильную, как у атомных часов. Программные часы в компьютере определяются аппаратными часами этого компьютера. В частности, предполагается, что аппаратные часы вызывают прерывание f раз в секунду. Когда этот таймер отключается, обработчик прерываний добавляет 1 к счетчику, который отслеживает количество тактов (прерываний), начиная с некоторого согласованного времени в прошлом. Этот счетчик действует как программный тактовый генератор C, резонирующий на частоте F.

Обозначим через $C_p(t)$ значение программных часов на компьютере p, когда время UTC равно t. Задача алгоритмов синхронизации часов состоит в том, чтобы удерживать в распределенной системе отклонение между соответствующими часами любых двух машин в пределах определенной границы, известной как **точность** (precision) π:

$$\forall t, \forall \mathrm{p}, \mathrm{q} : |C_p(t) - C_q(t)| \leq \pi.$$

Обратите внимание, что точность относится к отклонению часов только между компьютерами, которые являются частью распределенной системы. При рассмотрении внешней контрольной точки, такой как UTC, мы говорим о **правильности** (accurate), стремясь сохранить ее привязанной к значению α:

$$\forall t, \forall \mathrm{p} : |C_p(t) - t| \leq \alpha.$$

Вся идея синхронизации часов заключается в том, что мы сохраняем *точность* часов применительно к **внутренней синхронизации** (internal synchronization) или *правильность*, известную как **внешняя синхронизация** (external synchronization). Часы, которые являются точными в пределах границы α, будут точными в пределах границы $\pi = 2\alpha$. Вместе с тем, точность не позволяет нам сделать вывод о правильности часов.

В идеальном мире мы имели бы $C_p(t) = t$ для всех p и всех t, и, следовательно, $\alpha = \pi = 0$. К сожалению, аппаратные и, следовательно, программные часы подвержены **смещению тактовой частоты** (clock drift), потому что ввиду их несовершенства и влияния внешних факторов, таких как температура, часы на разных машинах постепенно начнут показывать разные значения времени. Это известно как **скорость дрейфа частоты часов** (clock drift rate): разница в единицу времени от идеальных эталонных часов. Типичные кварцевые аппаратные часы имеют эту частоту около 10^{-6} с в секунду или около 31,5 с в год. Существуют аппаратные часы с гораздо более низким коэффициентом дрейфа.

Спецификации аппаратных часов включают в себя его **максимальную частоту дрейфа часов** (maximum clock drift rate) ρ. Если $F(t)$ обозначает фактическую частоту генератора аппаратных часов в момент времени t, а F – его идеальную (постоянную) частоту, то аппаратные часы работают в соответствии со спецификацией, если

$$\forall t : (1 - \rho) \leq \frac{F(t)}{F} \leq (1 + \rho).$$

Используя аппаратные прерывания, мы напрямую связываем программные часы с аппаратными часами и, следовательно, со скоростью смещения тактовой частоты. В частности, мы имеем

$$C_p(t) = \frac{1}{F}\int_0^t F(t)dt,$$

и отсюда

$$\frac{dC_p(t)}{dt} = \frac{F(t)}{F},$$

что приводит нас к нашей конечной цели – программный тактовый дрейф также ограничивается ρ:

$$\forall t : 1 - \rho \leq \frac{dC_p(t)}{dt} \leq 1 + \rho.$$

Медленные, идеальные и быстрые часы показаны на рис. 6.4.

Если пара часов дрейфует от UTC в противоположных направлениях, то в момент времени Δt после их синхронизации они могут быть на расстоянии до $2\rho\Delta t$. Если разработчики системы хотят гарантировать точность π, то есть то, что никакие два тактовых генератора никогда не будут отличаться более чем на π секунд, тактовые частоты должны быть повторно синхронизированы (в программном обеспечении) по крайней мере каждые $\pi/(2\rho)$ секунд. Различные алгоритмы отличаются по точности в зависимости от того, как эта ресинхронизация выполняется.

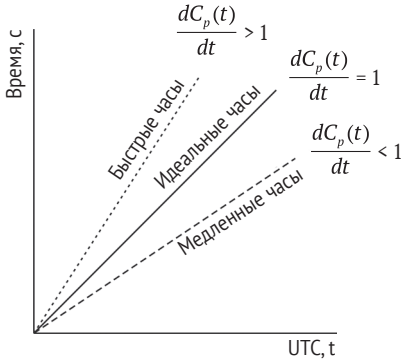


Рис. 6.4 ❖ Соотношение между временем часов и UTC, когда часы идут с разной скоростью

Сетевой временной протокол

Общий подход во многих протоколах, первоначально предложенный в [Cristian, 1989], состоит в том, чтобы позволить клиентам связываться с сервером времени. Последний может точно предоставлять текущее время, например потому, что он оснащен приемником UTC или точными часами. Проблема, конечно, заключается в том, что при обращении к серверу задержки сообщений делают устаревшим и сообщаемое время. Хитрость – найти хорошую оценку для этих задержек. Рассмотрим ситуацию, показанную на рис. 6.5.

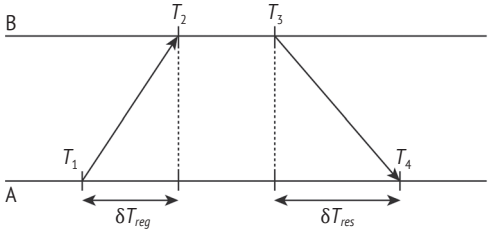


Рис. 6.5 ❖ Получение текущего времени от сервера времени

В этом случае A отправит запрос B, отметка времени которого имеет значение T_1. В свою очередь, B запишет время получения T_2 (взятое из его собственных локальных часов), вернет ответное время, помеченное значением T_3, и совместит ранее записанное значение T_2. Наконец, A записывает время прибытия ответа, T_4. Предположим, что задержки распространения от A до B примерно такие же, как от B до A, что означает, что $\delta T_{req} = T_2 - T_1 \approx T_4 - T_3 = dT_{res}$. В этом случае A может оценить свое смещение относительно B как

$$\theta = T_3 + \frac{(T_2 - T_1) + (T_4 - T_3)}{2} - T_4 = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}.$$

Конечно, время не может бежать назад. Если часы A быстрые, $\theta < 0$, это означает, что A, в принципе, должен отвести свои часы назад. Это недопустимо, так как может вызвать серьезные проблемы, такие как скомпилированный сразу после смены часов файл объекта, имеющий более раннее время, чем источник, который был изменен непосредственно перед сменой часов.

Подобное изменение должно вводиться постепенно. Один из способов заключается в следующем. Предположим, что таймер настроен на генерацию 100 прерываний в секунду. Обычно каждое прерывание добавляет 10 мс ко времени. При замедлении процедура прерывания прибавляет только 9 мс каждый раз, пока не будет выполнена коррекция. Точно так же время можно постепенно увеличивать, добавляя 11 мс при каждом прерывании, вместо того чтобы увеличить сразу.

Протоколом сетевого времени (network time protocol, NTP) время устанавливается попарно между серверами. Другими словами, B также проверяет текущее время A. Смещение θ вычисляется, как указано выше, вместе с оценкой δ для задержки:

$$\delta = \frac{(T_4 - T_1) - (T_3 - T_2)}{2}.$$

Восемь пар (θ, δ) значений буферизируются, в конечном итоге принимая минимальное значение, найденное для δ, в качестве наилучшей оценки задержки между двумя серверами, а затем соответствующее значение θ в качестве наиболее надежной оценки смещения.

Применение симметричного NTP должно, в принципе, также позволить B отрегулировать свои тактовые импульсы в соответствии с часами A. Однако если известно, что тактовые импульсы B более точные, такая настройка была бы неправильной. Чтобы решить эту проблему, NTP делит серверы на уровни (strata). Известно, что сервер с **эталонными часами** (reference clock), такими как приемник UTC или атомные часы, является **сервером stratum-1** (говорят, что сами часы работают на уровне stratum-0). Когда A связывается с B, он будет регулировать только свое время, если уровень его собственного уровня выше, чем уровень B. Более того, после синхронизации уровень A станет на один уровень выше, чем уровень B. Другими словами, если B является сервером на уровне stratum-k, то A станет сервером stratum-(k + 1), если его первоначальный уровень уже был больше k. Из-за симметрии NTP, если уровень страты A был ниже, чем у B, B приспособится к A.

В протоколе NTP есть много важных функций, многие из которых связаны с выявлением и маскированием ошибок, а также с атаками безопасности. Протокол NTP был первоначально описан в [Mills, 1992] и, как известно, обеспечивает (всемирную) точность в диапазоне 1–50 мс. Подробное описание NTP можно найти в [Mills, 2011].

Алгоритм Беркли

Во многих алгоритмах синхронизации часов сервер времени пассивен. Другие машины периодически спрашивают его о времени. Все, что он делает, – это отвечает на их запросы. В программе Berkeley Unix используется

совершенно противоположный подход [Gusella and Zatti, 1989]. Здесь сервер времени (фактически демон времени) активен, время от времени опрашивая каждую машину, чтобы узнать, который час. Основываясь на ответах, он вычисляет среднее время и говорит всем другим машинам перевести свои часы в новое время или замедлить их, пока не будет достигнуто определенное сокращение. Этот метод подходит для системы, в которой ни у одной машины нет приемника UTC. Время демона должно периодически устанавливаться оператором вручную. Этот метод показан на рис. 6.6.

На рис. 6.6a в 3:00 демон времени сообщает другим машинам свое время и запрашивает их. На рис. 6.6b они отвечают, насколько далеко опережают или отстают от демона времени. Вооружившись этими числами, демон времени вычисляет среднее значение и сообщает каждой машине, как настроить часы (см. рис. 6.6c).

Обратите внимание, что для многих целей достаточно, чтобы все машины работали одновременно. Не обязательно, чтобы это время также совпадало с реальным временем, которое объявляется по радио каждый час. Если в нашем примере на рис. 6.6 часы демона времени никогда не будут откалиброваны вручную, то никакого вреда это не принесет, если ни один из других узлов не свяжется с внешними компьютерами. Все просто согласятся на текущее время, не имеющего при этом значения, совпадающего с реальностью. Таким образом, алгоритм Беркли, как правило, является алгоритмом внутренней синхронизации часов.

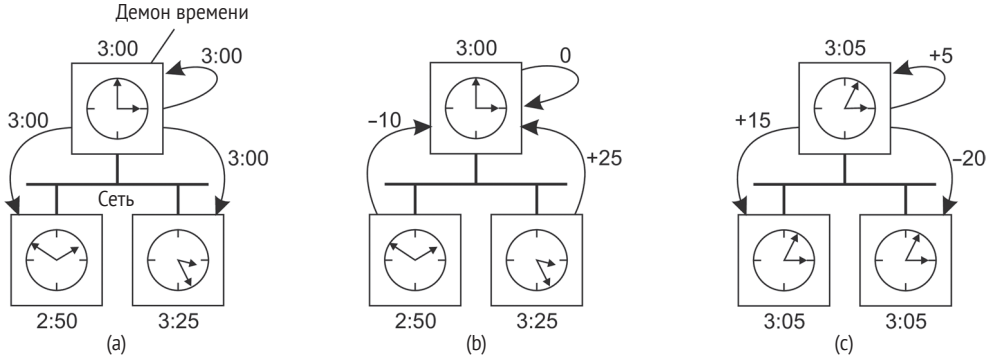


Рис. 6.6 ❖ a) Демон времени запрашивает у всех остальных машин их значения часов; b) компьютеры отвечают; c) демон времени говорит всем, как настроить свои часы

Синхронизация часов в беспроводных сетях

Важным преимуществом более традиционных распределенных систем является то, что мы можем легко и эффективно развертывать серверы времени. Более того, большинство компьютеров могут связываться друг с другом, что позволяет относительно просто распространять информацию. Эти предположения больше не действительны во многих беспроводных сетях, особенно в сенсорных сетях. Узлы ограничены в ресурсах, а маршрутизация с несколькими пересылками обходится дорого. Кроме того, часто важно оптимизи-

ровать алгоритмы энергопотребления. Эти и другие соображения привели к разработке совершенно разных алгоритмов синхронизации часов для беспроводных сетей. Далее мы рассмотрим одно конкретное решение. В работе [Sivrikaya and Yener, 2004] дается краткий обзор других решений. Обширный обзор можно найти в [Sundararaman et al., 2005].

Эталонная широковещательная синхронизация (Reference broadcast synchronization, RBS) представляет собой протокол синхронизации часов, который сильно отличается от других предложений [Elson et al., 2002]. Во-первых, в протоколе не предполагается, что существует один узел с точным отчетом о фактическом доступном времени. Вместо того чтобы стремиться обеспечить время UTC для всех узлов, он стремится просто к внутренней синхронизации часов, как это делает алгоритм Беркли. Во-вторых, решения, которые мы обсуждали до сих пор, предназначены для синхронизации отправителя и получателя, в основном по двустороннему протоколу. RBS отклоняется от этого шаблона, позволяя синхронизироваться только получателям, не давая отправителю выйти из цикла.

В RBS отправитель передает эталонное сообщение, которое позволяет его получателям настроить свои часы. Ключевым обстоятельством является то, что в сети датчиков время распространения сигнала на другие узлы является примерно постоянным, при условии что не предполагается многоперехóдная маршрутизация. Время распространения в этом случае измеряется с момента, когда сообщение покидает сетевой интерфейс отправителя. Как следствие при оценке задержек два важных источника различий в передаче сообщений больше не играют роли: время, затраченное на создание сообщения, и время, затраченное на доступ к сети. Этот принцип показан на рис. 6.7.

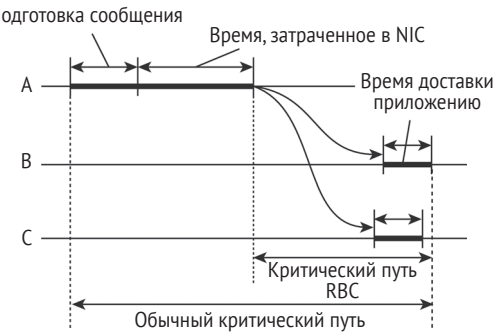


Рис. 6.7 ❖ Обычный критический путь и тот путь, который используется в RBS при определении сетевых задержек

Обратите внимание, что в протоколах, таких как NTP, временная метка добавляется к сообщению перед его передачей в сетевой интерфейс. Кроме того, поскольку беспроводные сети основаны на конкурентном протоколе, как правило, невозможно сказать, сколько времени потребуется, прежде чем сообщение действительно может быть передано. Эти факторы недетерминизма устранены в RBS. Остается только время доставки в приемник, но это время значительно меньше, чем время доступа к сети.

Идея, лежащая в основе RBS, проста: когда узел передает широковещательное сообщение m, каждый узел p просто записывает время $T_{p,m}$, в которое он получал m. Обратите внимание, что $T_{p,m}$ читается из локальных часов p. Игнорируя перекос часов, два узла p и q могут обмениваться временем доставки друг друга, чтобы оценить их взаимное относительное смещение (offset):

$$Offset[p,q] = \frac{\sum_{k=1}^{M}(T_{p,k} - T_{q,k})}{M},$$

где M – общее количество отправленных справочных сообщений. Эта информация важна: узел p будет знать значение часов q относительно своего собственного значения.

Более того, если он просто сохраняет эти смещения, нет необходимости настраивать собственные часы, что экономит энергию.

К сожалению, часы могут разойтись. В результате простое вычисление среднего смещения, как было сделано выше, не сработает: последние отправленные значения просто менее точны, чем первые. Более того, со временем смещение предположительно увеличится. В работе [Elson et al., 2002] используют очень простой алгоритм для компенсации этого: вместо вычисления среднего они применяют стандартную линейную регрессию для вычисления смещения как функции:

$$Offset[p, q](t) = \alpha t + \beta.$$

Константы α и β вычисляются из пар ($T_{p,k}$, $T_{q,k}$). Эта новая форма позволит намного более точно вычислить текущее значение тактового сигнала q по узлу p, и наоборот.

Примечание 6.2 (дополнительная информация: насколько важен точный учет времени?)

Так почему же время так важно для распределенных систем? Достижение консенсуса по глобальному упорядочению событий, как мы обсудим в оставшейся части этой главы, – это то, чего мы действительно хотим, и это может быть достигнуто без какого-либо представления об абсолютном глобальном времени. Однако, как станет ясно, альтернативные методы распределенной координации не так просты.

Жизнь была бы намного проще, если бы процессы в распределенной системе могли метить время своих событий с бесконечной точностью. Хотя бесконечная точность требует слишком многого, мы можем практически приблизиться к этому. Исследователи из Google столкнулись с тем, что их клиенты действительно хотели бы использовать глобально распределенную базу данных, которая поддерживает транзакции. Такая база данных должна обслуживать огромное количество клиентов, что делает невозможным использование, например, центрального монитора обработки транзакций, как мы обсуждали в разделе 1.3. Вместо этого для своей системы Spanner компания Google решила внедрить службу реального времени под названием TrueTime [Corbett et al., 2013]. Этот сервис обеспечивает три операции:

Операция	Результат
`TT.now()`	Временной интервал $[T_{lwb}, T_{upb}]$ при $T_{lwb} < T_{upb}$
`TT.after(t)`	True, если временная метка *t* определенно пришла
`TT.before(t)`	True, если временная метка *t* определенно не пришла

Важнейшей деталью является то, что T_{lwb} и T_{upb} являются *гарантированными* границами. Конечно, если $\varepsilon = T_{upb} - T_{lwb}$ велико, скажем 1 час, внедрение службы было бы относительно простым. Но вполне достаточно ε = 6 мс. Для достижения такой точности служба TrueTime использует *задатчики времени* (*time masters*), которых несколько в каждом центре обработки данных. Ведомые временны́е демоны запускаются на каждом компьютере в центре обработки данных и запрашивают несколько задатчиков времени, в том числе из других центров обработки данных, что очень похоже на то, что мы описали для NTP. Многие задатчики времени оснащены точными приемниками GPS, а многие другие независимо оснащены атомными часами. Результатом является набор источников времени с высокой степенью взаимной независимости (что важно по причинам отказоустойчивости). Используя версию алгоритма, разработанную в [Marzullo and Owicki, 1983], выбросы в вычислениях не учитываются. Между тем работа TrueTime постоянно контролируется, и «плохие» задатчики времени (вручную) удаляются, чтобы дать как минимум высокие гарантии точности обслуживания TrueTime.

С гарантированной точностью 6 мс построение транзакционной системы становится намного проще: транзакции могут на самом деле иметь временную метку даже на разных серверах, с ограничением, что временная метка может быть задержана на ε единиц времени. Точнее, чтобы определенно знать, что транзакция зафиксирована, чтение результирующих данных может привести к ожиданию ε единиц. Это достигается путем пессимистического назначения метки времени транзакции, которая записывает данные в глобальную базу данных и обеспечивает то, что клиенты никогда не видят никаких изменений до назначенной метки времени (что относительно легко реализовать).

Существует много деталей этого подхода, которые можно найти в [Corbett et al., 2013]. Поскольку мы пока имели дело с временными интервалами, применение традиционных механизмов, как показано в [Kulkarni, 2013], может улучшить результаты.

6.2. Логические часы

Синхронизация часов естественным образом связана со временем, хотя, возможно, нет необходимости иметь точный учет реального времени: может быть достаточно, чтобы каждый узел в распределенных системах согласовал *какое-либо* текущее время. Мы можем сделать еще один шаг. Для запуска make достаточно, чтобы два узла согласились с тем, что input.o устарел по сравнению с новой версией input.c. В этом случае им важно отслеживать события друг друга (например, создавать новую версию input.c). Для этих алгоритмов принято говорить о часах как о **логических часах** (logical clocks).

В оригинальной статье Лампорт [Lamport, 1978] показал, что хотя синхронизация часов возможна, она не обязательно должна быть абсолютной. Если два процесса не взаимодействуют, нет необходимости синхронизировать их

часы, потому что отсутствие синхронизации не было бы заметным и, следовательно, не могло вызвать проблем. Кроме того, он указал, что обычно имеет значение не то, что все процессы согласовывают точное время, а скорее то, что они согласовывают порядок, в котором происходят события. В примере make учитывается, является ли input.c старше или новее, чем input.o, а не их абсолютное время создания.

Логические часы Лампорта

Чтобы синхронизировать логические часы, Лампорт определил отношение, называемое «происходит перед» (happens-before). Выражение $a \rightarrow b$ означает, что «событие a происходит до события b» и что все процессы согласны с тем, что сначала происходит событие a, а затем – событие b. Отношение «происходит перед» можно наблюдать непосредственно в двух ситуациях:

1) если a и b являются событиями в одном и том же процессе и a происходит перед b, то $a \rightarrow b$ является истиной;
2) если a – это событие отправки сообщения одним процессом, а b – событие получения сообщения другим процессом, тогда $a \rightarrow b$ также является истиной. Сообщение не может быть получено до того, как оно будет отправлено, или даже в то же время, когда оно отправлено, поскольку для его получения требуется ненулевое количество времени.

Отношение «происходит перед» – это переходное отношение, так что если $a \rightarrow b$ и $b \rightarrow c$, то $a \rightarrow c$. Если два события, x и y, происходят в разных процессах, которые не обмениваются сообщениями (даже косвенно через третьих лиц), то $x \rightarrow y$ не верно, как и $y \rightarrow x$. Эти события называются параллельными, что просто означает, что ничего нельзя сказать (или нужно сказать) о том, когда события произошли или какое событие произошло первым.

Нам нужен такой способ измерения понятия времени, чтобы каждому событию a мы могли присвоить значение времени $C(a)$, с которым согласуются все процессы.

Эти значения времени должны иметь свойство: если $a \rightarrow b$, то $C(a) < C(b)$.

Перефразируя условия, которые мы указали ранее: если a и b являются двумя событиями в одном и том же процессе и a происходит до b, то $C(a) < C(b)$. Точно так же если a – отправка сообщения одним процессом, а b – получение этого сообщения другим процессом, то $C(a)$ и $C(b)$ должны быть назначены таким образом, чтобы каждый согласился относительно значений $C(a)$ и $C(b)$, что $C(a) < C(b)$. Кроме того, время C должно всегда идти вперед (увеличиваться), а не назад (уменьшаться). Поправки ко времени могут быть сделаны путем добавления положительного значения, а не вычитания.

Теперь давайте рассмотрим алгоритм, предложенный Лампортом для назначения времени событиям. Рассмотрим три процесса, изображенных на рис. 6.8. Процессы выполняются на разных компьютерах, каждый со своими часами. Предположим, что часы реализованы в виде программного счетчика: счетчик увеличивается на определенное значение каждые T единиц времени. Однако значение, на которое увеличивается время, отличается для каждого процесса. Синхронизация в процессе P_1 увеличивается на 6 единиц, в про-

цессе P_2 на 8 единиц и на 10 единиц в процессе P_3 соответственно. (Ниже мы объясняем, что часы Лампорта на самом деле являются счетчиками событий, что объясняет, почему их значения могут различаться в разных процессах.)

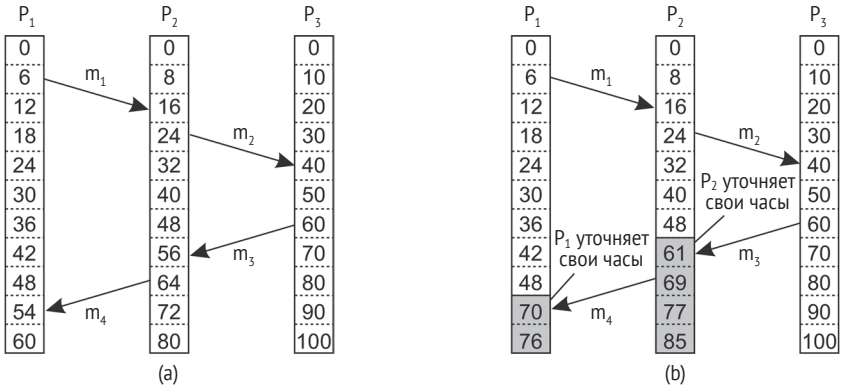


Рис. 6.8 ❖ a) Три процесса, каждый со своими (логическими) часами. Часы работают с разной скоростью; b) алгоритм Лампорта корректирует их значения

В момент 6 процесс P_1 отправляет сообщение m_1 процессу P_2. Сколько времени занимает это сообщение, зависит от того, в чьи часы вы верите. В любом случае, когда он поступает, часы в процессе P_2 читают 16. Если сообщение содержит время начала 6, процесс P_2 заключит, что на передачу потребовалось 10 тиков. Это значение, безусловно, возможно. В соответствии с этим рассуждением сообщение m_2 от P_2 к P_3 занимает 16 тиков, опять же, вероятное значение.

Теперь рассмотрим сообщение m_3. Оно покидает процесс P_3 в 60 и достигает P_2 в 56. Аналогично сообщение m_4 от P_2 до P_1 выходит в 64 и достигает 54. Эти значения явно невозможны. Именно эту ситуацию необходимо предотвратить.

Решение Лампорта следует непосредственно из отношения «до того». Поскольку m_3 осталось в 60, оно должно прибыть в 61 или позже. Поэтому каждое сообщение содержит время отправки в соответствии с часами отправителя. Когда приходит сообщение и часы получателя показывают значение до того, как сообщение было отправлено, получатель быстро пересылает свои часы на единицу больше, чем время отправки. На рис. 6.8 мы видим, что m_3 теперь достигает 61. Аналогично m_4 достигает 70.

Давайте сформулируем эту процедуру более точно. Здесь важно отличать три различных уровня программного обеспечения, с чем мы уже встречались в главе 1: сеть, уровень промежуточного программного обеспечения и уровень приложений, как это показано на рис. 6.9. То, что из этого следует, обычно является частью уровня промежуточного программного обеспечения.

Чтобы реализовать логические часы Лампорта, каждый процесс P_i поддерживает *локальный* счетчик C_i. Эти счетчики обновляются в соответствии со следующими шагами [Ray-nal and Singhal, 1996].

1. Перед выполнением события (т. е. отправки сообщения по сети, доставки сообщения в приложение или другого внутреннего события) P_i увеличивает C_i:

   $C_i \leftarrow C_i + 1$.

2. Когда процесс P_i отправляет сообщение m процессу P_j, он устанавливает метку времени *ts*(m) равной C_i после выполнения предыдущего шага.
3. После получения сообщения m процесс P_j настраивает свой собственный локальный счетчик как $C_j \leftarrow \max\{C_j, ts(m)\}$, после чего выполняет первый шаг и доставляет сообщение приложению.

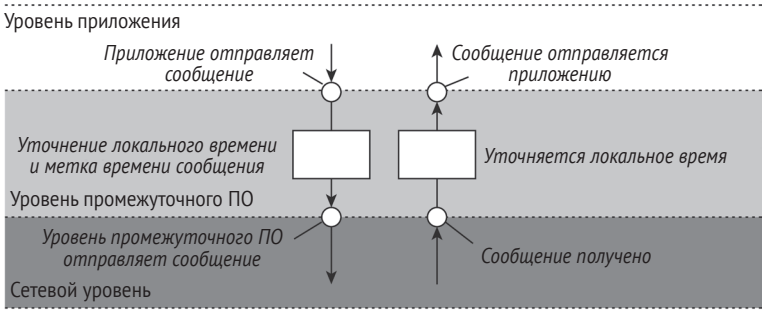


Рис. 6.9 ❖ Расположение логических часов Лампорта в распределенных системах

В некоторых ситуациях желательно дополнительное требование: два события никогда не происходят одновременно. Для достижения этой цели мы также используем уникальный идентификатор процесса для разрыва связей и применения кортежей вместо значений только счетчика. Например, событие в момент времени 40 в процессе P_i будет помечено как {40, i}. Если у нас также есть события {40, j} и $i < j$, то {40, i} <{40, j}.

Обратите внимание, что, назначая время события $C(a) \leftarrow C_i(a)$, если событие произошло в процессе P_i в момент времени $C_i(a)$, мы имеем распределенную реализацию значения глобального времени, которое первоначально искали; таким образом, мы построили **логические часы**.

Пример: полностью упорядоченная многоадресная рассылка

В качестве приложения логических часов Lamport рассмотрим ситуацию, в которой база данных была реплицирована на несколько сайтов. Например, для повышения производительности запросов банк может разместить копии базы данных счетов в двух разных городах, например в Нью-Йорке и Сан-Франциско. Запрос всегда пересылается в ближайшую копию. Стоимость быстрого ответа на запрос частично оплачивается за счет более высоких затрат на обновление, поскольку каждая операция обновления должна выполняться на каждой реплике.

На самом деле существует более жесткое требование в отношении обновлений. Предположим, клиент в Сан-Франциско хочет добавить 100 долларов на свой счет, который в настоящее время содержит 1000 долларов. В то же время сотрудник банка в Нью-Йорке инициирует обновление, по которому счет клиента должен быть увеличен на 1 %. Оба обновления должны выполняться на обеих копиях базы данных. Однако из-за задержек в базовой сети обновления могут поступать в порядке, показанном на рис. 6.10.

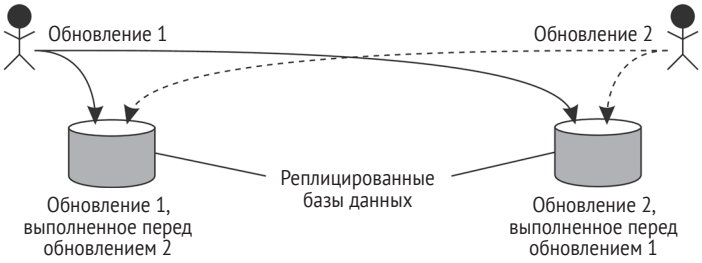


Рис. 6.10 ❖ Обновление реплицируемой базы данных и перевод ее в несогласованное состояние

Операция обновления счета клиента выполняется в Сан-Франциско перед обновлением процентной ставки. Напротив, копия счета в реплике в Нью-Йорке сначала обновляется с процентной ставкой, а после этого – депозитом в 100 долларов.

Следовательно, база данных Сан-Франциско будет записывать общую сумму 1111 долларов, а база данных Нью-Йорка – 1110 долларов. Проблема, с которой мы сталкиваемся, заключается в том, что две операции обновления должны были выполняться в одном и том же порядке для каждой копии. Хотя это не имеет значения, обрабатывается ли депозит до обновления процентов или наоборот, какой порядок следует соблюдать, не имеет значения с точки зрения согласованности. Важным вопросом является то, что обе копии должны быть абсолютно одинаковыми. В общем, в ситуациях, подобных этим, требуется тотально упорядоченная многоадресная группировка, то есть операция многоадресной рассылки, при которой все сообщения доставляются каждому получателю в одинаковом порядке. Логические часы Лампорта могут использоваться для реализации полностью упорядоченных групповых рассылок в полностью распределенном режиме.

Рассмотрим группу процессов многоадресных сообщений друг другу. Каждое сообщение всегда помечается текущим (логическим) временем отправителя. Когда сообщение является многоадресным, оно концептуально также отправляется отправителю. Кроме того, мы предполагаем, что сообщения от одного и того же отправителя принимаются в том порядке, в котором они были отправлены, и что никакие сообщения не теряются.

Когда процесс получает сообщение, оно помещается в локальную очередь, упорядоченную в соответствии с его отметкой времени. Получатель многоадресно передает подтверждение другим процессам. Обратите внимание, что если мы следуем алгоритму Лампорта для настройки локальных часов,

отметка времени полученного сообщения будет меньше отметки времени подтверждения. Интересным аспектом этого подхода является то, что все процессы в конечном итоге будут иметь одну и ту же копию локальной очереди (при условии что сообщения не удаляются).

Процесс может доставить сообщение из очереди приложению, которое он выполняет, только когда это сообщение находится в начале очереди и подтверждено каждым другим процессом. В этот момент сообщение удаляется из очереди и передается приложению; связанные подтверждения могут быть просто удалены. Поскольку каждый процесс имеет одну и ту же копию очереди, все сообщения доставляются везде в одном и том же порядке. Другими словами, мы установили полностью упорядоченную многоадресную рассылку. Мы оставляем читателю в качестве упражнения выяснение того, что нет строгой необходимости подтверждать получение каждого многоадресного сообщения. Достаточно, чтобы процесс реагировал на входящее сообщение, или возвращая подтверждение, или отправляя собственное многоадресное сообщение.

Полностью упорядоченная многоадресная рассылка является важным средством служб репликации, где реплики поддерживаются согласованными, посредством выполнения повсюду в одном и том же порядке одинаковых операций. Поскольку реплики повторяют по существу одни и те же переходы в одном и том же конечном автомате, он также известен как **репликации конечного автомата** (state machine replication) [Schneider, 1990].

Примечание 6.3 (дополнительно: использование часов Lamport для достижения взаимного исключения)

Чтобы дополнительно проиллюстрировать применение часов Лампорта, давайте посмотрим, как мы можем использовать предыдущий алгоритм полностью упорядоченной многоадрессной рассылки, чтобы установить доступ к так называемой **критической секции** (critical section): разделу кода, который может быть выполнен не более чем одним процессом за раз. Этот алгоритм очень похож на алгоритм многоадресной рассылки, поскольку по существу все процессы должны согласовать порядок, в котором процессам разрешено входить в свои критические секции.

На рис. 6.11а показан код, который выполняет каждый процесс, когда запрашивает, освобождает или разрешает доступ к критическому разделу (опять же, опускаем детали). Каждый процесс поддерживает очередь запросов, а также логические часы. Чтобы войти в критическую секцию, выполняется запрос `requestToEnter`, который приводит к вставке сообщения ENTER с меткой времени (`clock`, `procID`) в локальную очередь и отправке этого сообщения другим процессам. Операция `cleanupQ`, по существу, сортирует очередь. Мы вернемся к этому в ближайшее время.

Когда процесс P получает сообщение ENTER от процесса Q, он может просто позволить Q войти в его критическую секцию, даже если P также хочет сделать это. В этом случае запрос P будет иметь более низкую логическую метку времени, чем сообщение ALLOW, отправленное P в Q, что означает, что запрос P будет вставлен в очередь Q *перед* сообщением P ALLOW.

Наконец, когда процесс покидает свою критическую секцию, он вызывает release. Он очищает свою локальную очередь, удаляя все полученные сообщения ALLOW, оставляя только запросы ENTER от других процессов. Затем он отправляет сообщение RELEASE.

```
class Process:
  def __init__(self, chan):
    self.queue      = []                    # Очередь запросов
    self.clock      = 0                     # Текущие логические часы

  def requestToEnter(self):
    self.clock = self.clock + 1                          # Увеличить значение такта
    self.queue.append((self.clock, self.procID, ENTER)) # Добавить запрос к q
    self.cleanupQ()                                      # Сортировка очереди
    self.chan.sendTo(self.otherProcs, (self.clock, self.procID, ENTER))
                                                         # Отправить запрос

  def allowToEnter(self, requestter):
    self.clock = self.clock + 1            # Увеличить значение тактового сигнала
    self.chan.sendTo([requestter], (self.clock,self.procID,ALLOW))
                                           # Разрешить другие

  def release (self):
    tmp = [r for r in self.queue[1:] if r[2] == ENTER] # Удалить все ALLOWs
    self.queue = tmp                                   # и скопировать в новую очередь
    self.clock = self.clock + 1                        # Увеличить значение часов
    self.chan.sendTo(self.otherProcs, (self.clock,self.procID,RELEASE)) # Освободить

  def allowToEnter(self):
    commProcs = set([req[1] for req in self.queue[1:]]) # Посмотрите, кто отправил
                                                        # сообщение
    return(self.queue[0][1]==self.procID and len(self.otherProcs) == len(commProcs))
```

Рис. 6.11 ❖ a) Использование логических часов Лампорта для взаимного исключения

```
  def receive(self):
    msg = self.chan.recvFrom(self.otherProcs)[1]  # Получить любое сообщение
    self.clock = max(self.clock, msg[0])          # Настроить значение такта ...
    self.clock = self.clock + 1                   # ... и увеличить
    if msg[2] == ENTER:
      self.queue.append(msg)                      # Добавить запрос ENTER
      self.allowToEnter(msg[1])                   # и безусловно разрешить
    elif msg[2] == ALLOW:
      self.queue.append(msg)                      # Добавить ALLOW
    elif msg[2] == RELEASE:
      del(self.queue[0])                          # Просто удалить первое сообщение
    self.cleanupQ()                               # и отсортировать, и очистить
```

Рис. 6.11 ❖ b) Использование логических часов Лампорта для взаимного исключения: обработка входящих запросов

Для того чтобы действительно войти в критическую секцию, процесс должен будет повторно вызывать `allowToEnter` и, когда возвращается `False`, блокировать следующее входящее сообщение. Операция `allowToEnter` выполняет ожидаемые действия: она проверяет, находится ли сообщение ENTER вызывающего процесса во главе очереди, и проверяет, все ли процессы также отправили сообщение. По-

следний кодируется через набор `commProcs`, который содержит `procID` всех процессов, отправивших сообщение, проверяя все сообщения в локальной очереди со второй позиции и далее.

Что делать, когда сообщение получено, показано на рис. 6.11b. Во-первых, местные часы настраиваются в соответствии с правилами для логических часов Лампорта, описанными выше. При получении сообщения ENTER или ALLOW это сообщение просто вставляется в очередь. Запрос на вход всегда подтверждается, как мы только что объяснили. Когда получено сообщение RELEASE, исходный запрос ENTER удаляется. Обратите внимание, что этот запрос находится в начале очереди. После этого очередь снова очищается.

На этом этапе обратите внимание, что если мы очистим очередь, только отсортировав ее, у нас могут возникнуть проблемы. Предположим, что процессы P и Q хотят войти в свои соответствующие критические секции примерно в одно и то же время, но что P разрешено идти первым на основании значений логических часов. P может найти запрос Q в своей очереди вместе с сообщениями ENTER или ALLOW от других процессов. Если его собственный запрос находится в начале очереди, P продолжит работу и войдет в критическую секцию. Однако Q также отправит сообщение ALLOW в P в дополнение к его исходному сообщению ENTER. Это сообщение ALLOW может поступить *после того*, как P уже вошел в критическую секцию, но *до* сообщений ENTER от других процессов. Когда Q в конце концов входит и покидает свою критическую секцию, сообщение Q RELEASE приведет к удалению исходного сообщения ENTER Q, но не к сообщению ALLOW, которое оно ранее отправляло в P. К настоящему времени это сообщение находится в начале очереди P, эффективно блокируя вход в критическую секцию других процессов в очереди P. Таким образом, очистка очереди также включает удаление старых разрешенных сообщений.

Векторные часы

Логические часы Лампорта приводят к ситуации, в которой все события в распределенной системе полностью упорядочены в том смысле, что если событие a произошло до события b, то a будет располагаться в этом порядке перед b, то есть $C(a) < C(b)$.

Однако с часами Лампорта ничего нельзя сказать о взаимоотношении между двумя событиями a и b, просто сравнивая их значения времени $C(a)$ и $C(b)$ соответственно. Другими словами, если $C(a) < C(b)$, то это необязательно подразумевает, что a действительно произошло раньше, чем b. Иногда для этого нужно нечто большее.

Для объяснения рассмотрим сообщения, отправленные тремя процессами, показанными на рис. 6.12. Обозначим через $T_{snd}(m_i)$ логическое время, когда сообщение m_i было отправлено, а также через $T_{rcv}(m_i)$ время его получения. По построению мы знаем, что для каждого сообщения $T_{snd}(m_i) < T_{rcv}(m_i)$. Но какой мы можем сделать вывод из $T_{rcv}(m_i) < T_{rcv}(m_j)$ для разных сообщений m_i и m_j?

В случае когда $m_i = m_1$ и $m_j = m_3$, мы знаем, что эти значения соответствуют событиям, произошедшим в процессе P_2, что означает, что m_3 действительно было отправлено после получения сообщения m_1. Это может указывать на то, что отправка сообщения m_3 зависела от того, что было получено через сообщение m_1. В то же время мы также знаем, что $T_{rcv}(m_1) < T_{snd}(m_2)$. Однако, как мы можем видеть по рис. 6.12, отправка m_2 не имеет ничего общего с получением m_1.

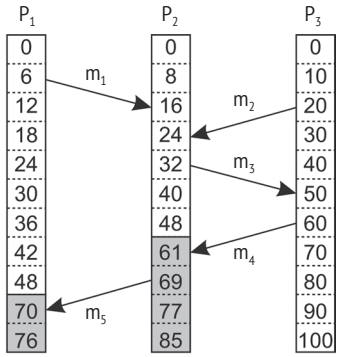


Рис. 6.12 ❖ Параллельная передача сообщений с использованием логических часов

Проблема в том, что часы Лампорта не фиксируют причинно-следственную связь. На практике причинность фиксируется с помощью **векторных часов** (vector clock). Чтобы лучше понять, откуда они берутся, мы следуем объяснениям, данным в [Baquero and Preguic, 2016]. Фактически отслеживание причинно-следственной связи является простым, если мы назначаем каждому событию уникальное имя, такое как сочетание идентификатора процесса и локально увеличивающегося счетчика: p_k – это *k*-е событие, которое произошло в процессе P. Затем проблема сводится к отслеживанию **истории причинно-следственной связи** (casual histories). Например, если два локальных события произошли в процессе P, то причинной историей *H* (p_2) события p_2 является $\{p_1, p_2\}$.

Теперь предположим, что процесс P отправляет сообщение процессу Q (который является событием в P и, таким образом, записывается как p_k из некоторого *k*) и что во время прибытия (и события для Q) самая последняя причинная история Q была $\{q_1\}$. Чтобы отследить причинность, P также отправляет свою самую раннюю причинную историю (предположим, что это была $\{p_1, p_2\}$, расширенная с p_3, выражающим отправку сообщения). По прибытии Q записывает событие (q_2) и объединяет две причинные истории в новую: $\{p_1, p_2, p_3, q_1, q_2\}$.

Проверка того, предшествует ли событие p событию q, может быть выполнена путем проверки, является ли $H(p) \subset H(q)$ (то есть оно должно быть *надлежащим* подмножеством). На самом деле с нашими обозначениями даже достаточно проверить, действительно ли $p \in H(q)$, предполагая, что q всегда является последним локальным событием в $H(q)$.

Проблема с историями причинно-следственных связей заключается в том, что их представление не очень эффективно. Тем не менее нет необходимости отслеживать все последовательные события из одного и того же процесса: это будет делать последний. Если впоследствии мы назначим индекс каждому процессу, то можем представить историю причин как вектор, в котором *j*-я запись представляет количество событий, которые произошли в процессе P_j. Причинность может быть затем зафиксирована с помощью **векторных**

часов, которые строятся так, что каждый процесс P_i поддерживает вектор VC_i со следующими двумя свойствами:

1) $VC_i[i]$ – количество событий, которые произошли до сих пор в P_i. Другими словами, $VC_i[i]$ – это локальные логические часы у процесса P_i;
2) если $VC_i[j] = k$, то P_i знает, что в P_j произошло k событий. Таким образом, P_i знает местное время в P_j.

Первое свойство поддерживается путем увеличения $VC_i[i]$ при возникновении каждого нового события, которое происходит в процессе P_i. Второе свойство поддерживается за счет использования векторов с отправляемыми сообщениями. В частности, выполняются следующие шаги:

1) перед выполнением события (т. е. отправки сообщения по сети, доставки сообщения в приложение или какого-либо другого внутреннего события) P_i выполняет $VC_i[i] \leftarrow VC_i[i] + 1$. Это эквивалентно записи нового события, которое произошло в P_i;
2) когда процесс P_i отправляет сообщение m в P_j, он устанавливает метку времени m (вектора) $ts(m)$, равную VC_i, после выполнения предыдущего шага (т. е. он также записывает отправку сообщения как событие, которое происходит в P_i);
3) после получения сообщения m процесс P_i корректирует свой собственный вектор, устанавливая $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$ для каждого k (что эквивалентно объединению причинно-следственных связей), после чего он выполняет первый шаг (запись получения сообщения), а затем доставляет сообщение в приложение.

Обратите внимание, что если событие a имеет метку времени $ts(a)$, то $ts(a)[i] - 1$ обозначает количество обработанных в P_i событий, которые предшествуют a. Как следствие, когда P_j получает сообщение m от P_i с отметкой времени $ts(m)$, он знает о количестве событий, произошедших в P_i, которые предшествовали отправке m. Однако более важно то, что P_j также сообщают, сколько событий произошло *в других* процессах, известных P_i, до того, как P_i отправил сообщение m. Другими словами, отметка времени $ts(m)$ сообщает получателю, сколько событий в других процессах предшествовало отправке m, от которых m может зависеть.

Чтобы увидеть, что это значит, рассмотрим рис. 6.13, на котором показаны три процесса. На рис. 6.13a P_2 отправляет сообщение m_1 в логическое время $VC_2 = (0, 1, 0)$ для обработки P_1. Таким образом, сообщение m_1 получает метку времени $ts(m_1) = (0, 1, 0)$. После получения P_1 устанавливает логическое время на $VC_1 \leftarrow (1, 1, 0)$ и доставляет его. Сообщение m_2 отправляется P_1 в P_3 с отметкой времени $ts(m_2) = (2, 1, 0)$. Перед тем как P_1 отправит другое сообщение, m_3, в P_1 происходит событие, которое в конечном итоге приводит к метке времени m_3 со значением $(4, 1, 0)$. После получения m_3 процесс P_2 отправляет сообщение m_4 в P_3 с отметкой времени $ts(m_4) = (4, 3, 0)$.

Теперь рассмотрим ситуацию, показанную на рис. 6.13b. Здесь мы задержали отправку сообщения m_2 до тех пор, пока сообщение m_3 не было отправлено, и после того, как событие произошло. Нетрудно видеть, что $ts(m_2) = (4, 1, 0)$, а $ts(m_4) = (2, 3, 0)$. По сравнению с рис. 6.13a, мы имеем следующую ситуацию:

Ситуация	$ts(m_2)$	$ts(m_4)$	$ts(m_2) < ts(m_4)$	$ts(m_2) > ts(m_4)$	Заключение
Рис. 6.13a	(2, 1, 0)	(4, 3, 0)	Да	Нет	m_2 может предшествовать m_4
Рис. 6.13b	(4, 1, 0)	(2, 3, 0)	Нет	Нет	m_2 и m_4 могут конфликтовать

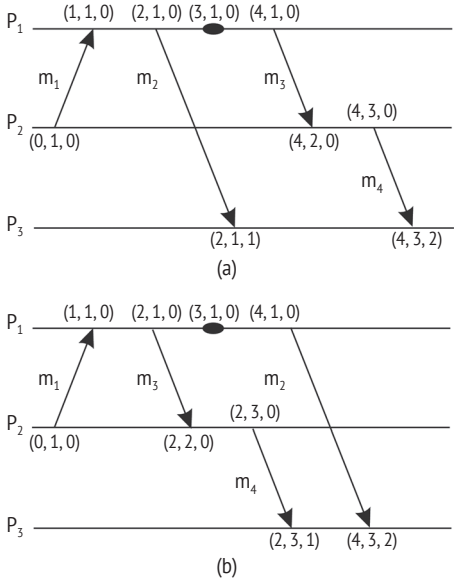


Рис. 6.13 ❖ Захват потенциальной причинности при обмене сообщениями

Мы используем обозначение $ts(a) < ts(b)$ тогда и только тогда, когда для всех k существует $ts(a)[k] \leq ts(b)[k]$ и существует хотя бы один индекс k', для которого $ts(a)[k'] < ts(b)[k']$. Таким образом, используя векторные тактовые импульсы, процесс P_3 может обнаружить, может ли m_4 быть причинно-зависимой от m_2 или возможен конфликт. Обратите внимание, кстати, что, не имея фактической информации, содержащейся в сообщениях, невозможно с уверенностью утверждать, что действительно существует причинно-следственная связь или возможен конфликт.

Примечание 6.4 (дополнительно: обеспечение причинно-следственной связи)

Используя векторные часы, теперь возможно гарантировать, что сообщение доставлено, только если все сообщения, которые могли иметь предшествующие ему, были также получены. Чтобы сделать возможной такую схему, мы будем предполагать, что сообщения являются многоадресными в группе процессов. Обратите внимание, что эта **причинно-упорядоченная групповая адресация** (causal-ordered multicasting) слабее, чем общая упорядоченная групповая передача. В частности, если два сообщения никак не связаны друг с другом, нам все равно, в каком порядке они доставляются приложениям. Они могут даже быть доставлены в другом порядке в разных местах.

Для обеспечения причинной доставки сообщений мы предполагаем, что часы корректируются только при отправке и доставке сообщений (обратите внимание, опять же, что сообщения не корректируются, когда они получены процессом, а только когда они доставляются в приложение). В частности, после отправки сообщения процесс P_i будет увеличивать только $VC_i[i]$ на 1. Когда он доставляет сообщение m с отметкой времени $ts(m)$, он только настраивает $VC_i[k]$ на $\max\{VC_i[k], ts(m)[k]\}$ для каждого k.

Теперь предположим, что P_j получает сообщение m от P_i с (векторной) меткой времени $ts(m)$. Доставка сообщения на прикладной уровень будет отложена до тех пор, пока не будут выполнены следующие два условия:

1) $ts(m)[i] = VC_j[i] + 1$;
2) $ts(m)[k] = VC_j[k]$ для всех $k \neq i$.

Первое условие гласит, что m – это следующее сообщение, которого P_j ожидал от процесса P_i. Второе условие гласит, что P_j доставил все сообщения, которые были доставлены P_i, когда он отправил сообщение m. Обратите внимание, что процессу P_j нет необходимости задерживать доставку своих собственных сообщений.

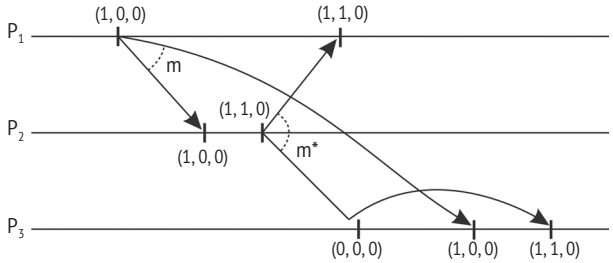


Рис. 6.14 ❖ Обеспечение причинно-следственной связи

В качестве примера рассмотрим три процесса P_1, P_2 и P_3, как показано на рис. 6.14. В местное время (1, 0, 0) P_1 отправляет сообщение m двум другим процессам. Обратите внимание, что $ts(m) = (1, 0, 0)$. Его получение и последующая доставка P_2 приведут логические часы в P_2 к (1, 0, 0), эффективно указывая, что он получил одно сообщение от P_1, сам пока не отправил никакого сообщения и еще не получил сообщение из P_3.

Затем P_2 решает отправить m в обновленное время (1, 1, 0), которое прибывает в P_3 раньше, чем m.

Сравнивая временную метку m с ее текущим временем (0, 0, 0), P_3 приходит к выводу, что все еще отсутствует сообщение от P_1, которое P_2, по-видимому, доставило до отправки m. Поэтому P_3 решает отложить доставку m (и также не будет корректировать свои локальные логические часы). Позже, после того как m было получено и доставлено P_3, который переводит свои локальные часы в (1, 0, 0), P_3 может доставить сообщение m*, а также обновить свои часы.

Примечание о доставке заказанного сообщения. Некоторые системы промежуточного программного обеспечения, в частности ISIS и его преемник Horus [Birman and van Renesse, 1994], обеспечивают поддержку многоадресной и упорядоченной причинно-следственной (надежной) многоадресной рассылки. Существуют некоторые противоречия относительно того, должна ли такая поддержка предоставляться как часть уровня обмена сообщениями или приложения должны обрабатывать упорядочение (см., например, [Cheriton and Skeen, 1993]; [Birman, 1994]). Вопросы не урегулированы, но важнее то, что аргументы все еще актуальны.

Есть две основные проблемы, связанные с разрешением промежуточному программному обеспечению упорядочивать сообщения. Во-первых, поскольку промежуточное ПО не может определить, что на самом деле содержит сообщение, фиксируется только потенциальная причинность. Например, два сообщения от одного и того же отправителя, которые полностью независимы, всегда будут помечены как причинно связанные уровнем промежуточного программного обеспечения. Этот подход чрезмерно ограничен и может привести к проблемам с эффективностью.

Вторая проблема заключается в том, что не вся причинность может быть уловлена. Рассмотрим электронную доску объявлений. Предположим, Алиса опубликовала статью. Если она тогда позвонит Бобу, рассказывая о том, что она только что написала, Боб может опубликовать еще одну статью в качестве реакции, даже не увидев сообщения Алисы на доске. Другими словами, существует связь между постами Боба и Алисы из-за *внешнего* общения. Эта причинно-следственная связь не фиксируется системой досок объявлений.

По сути, проблемы с упорядочением, как и многие другие проблемы связи, специфичные для приложения, могут быть адекватно решены путем рассмотрения приложения, для которого осуществляется связь. Это также известно как **сквозной аргумент** (end-to-end argument) в проектировании систем [Saltzer et al., 1984]. Недостаток наличия только решений уровня приложения состоит в том, что разработчик вынужден концентрироваться на проблемах, которые не имеют непосредственного отношения к основным функциям приложения. Например, заказ может быть не самой важной проблемой при разработке системы обмена сообщениями, такой как электронная доска объявлений. В этом случае упорядочение дескриптора нижележащего уровня связи может оказаться удобным. Мы будем сталкиваться со сквозным аргументом несколько раз.

6.3. Взаимное исключение

Основой распределенных систем являются параллелизм и совместная работа нескольких процессов. Во многих случаях это также означает, что процессам необходимо одновременно получать доступ к одним и тем же ресурсам. Чтобы предотвратить такой одновременный доступ, который может повредить ресурс или сделать его несовместимым, необходимы решения для предоставления взаимоисключающего доступа процессам. В этом разделе мы рассмотрим некоторые важные и типичные распределенные алгоритмы, которые были предложены. Обзоры распределенных алгоритмов взаимного исключения предоставлены в [Saxena and Rai, 2003] и [Velazquez, 1993]. Различные алгоритмы также представлены в [Kshemkalyani and Singhal, 2008].

Обзор

Распределенные алгоритмы взаимного исключения могут быть классифицированы по двум различным категориям. В **решениях на основе токенов** (token-based solutions) взаимное исключение достигается путем передачи специального сообщения между процессами, известного как токен. Доступен только один токен, и тот, у кого он есть, может получить доступ к общему ре-

сурсу. После завершения токен передается следующему процессу. Если процесс с токеном не заинтересован в доступе к ресурсу, он передает его дальше.

Решения на основе токенов имеют несколько важных свойств. Во-первых, в зависимости от того, как организованы процессы, они могут довольно легко гарантировать, что каждый процесс получит возможность доступа к ресурсу. Другими словами, они избегают голода. Во-вторых, благодаря его простоте можно легко избежать **тупиковых ситуаций** (deadlocks), в результате которых несколько процессов бесконечно ждут продолжения друг друга. Основным довольно серьезным недостатком решений на основе токенов является то, что когда токен теряется (например, из-за сбоя удерживающего его процесса), необходимо запустить сложную распределенную процедуру, чтобы гарантировать создание нового токена, но прежде всего это должен быть также единственный токен.

В качестве альтернативы многие распределенные алгоритмы взаимного исключения следуют **подходу на основе разрешений** (permission-based approach). В этом случае процесс, который хочет получить доступ к ресурсу, сначала требует разрешения от других процессов. Существует много разных способов предоставления такого разрешения, и в следующих разделах мы рассмотрим некоторые из них.

Централизованный алгоритм

Простой способ добиться взаимного исключения в распределенной системе – моделировать, как это делается в однопроцессорной системе. Один процесс избран координатором. Всякий раз, когда процесс хочет получить доступ к общему ресурсу, он отправляет сообщение запроса координатору, в котором указывается, к какому ресурсу он хочет получить доступ, и запрашивает разрешение. Если никакой другой процесс в настоящее время не обращается к этому ресурсу, координатор отправляет обратно ответ, предоставляющий разрешение, как показано на рис. 6.15a. Когда приходит ответ, заявитель получает доступ.

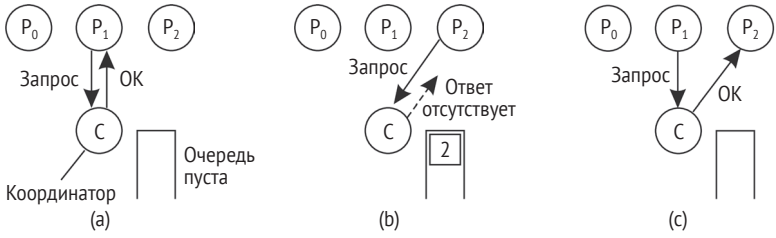


Рис. 6.15 ❖ a) Процесс P_1 запрашивает разрешение на доступ к общему ресурсу. Разрешение предоставляется; b) процесс P_2 запрашивает разрешение на доступ к тому же ресурсу, но не получает ответа; c) когда P_1 освобождает ресурс, координатор отвечает P_2

Теперь предположим, что другой процесс, P_2 на рис. 6.15b, запрашивает разрешение на доступ к ресурсу. Координатор знает, что другой процесс

уже использует ресурс, поэтому он не может предоставить разрешение. Точный метод, используемый для отказа в разрешении, зависит от системы. На рис. 6.15b координатор просто воздерживается от ответа, блокируя таким образом процесс P_2, который ожидает ответа.

В качестве альтернативы он может отправить ответ, говорящий «отказано в разрешении». В любом случае, он ставит в очередь запрос от P_2 и ожидает новых сообщений.

Когда процесс P_1 завершает работу с ресурсом, он отправляет сообщение координатору, освобождая свой эксклюзивный доступ, как показано на рис. 6.15c.

Координатор берет первый элемент из очереди отложенных запросов и отправляет этому процессу сообщение о предоставлении доступа. Если процесс все еще был заблокирован (то есть это первое сообщение для него), он разблокируется и получает доступ к ресурсу. Если явное сообщение уже было отправлено с отказом в разрешении, процесс должен будет опросить входящий трафик или заблокировать его позже. В любом случае, когда он видит разрешение, он также может начинать работу.

Легко видеть, что алгоритм гарантирует взаимное исключение: координатор позволяет только одному процессу одновременно обращаться к ресурсу. Это также справедливо, поскольку запросы удовлетворяются в том порядке, в котором они были получены. Никакой процесс никогда не ждет вечно (не голодает). Схема также проста в реализации и требует только трех сообщений на использование ресурса (запрос, предоставление, выпуск). Его простота делает его привлекательным решением для многих практических ситуаций.

Централизованный подход имеет недостатки. Координатор является единственной точкой отказа, поэтому в случае сбоя вся система может выйти из строя. Если процессы обычно блокируются после выполнения запроса, они не могут отличить вышедшего из строя координатора от ответа «отказано в разрешении», поскольку в обоих случаях сообщение не возвращается. Кроме того, в большой системе один координатор может стать узким местом в производительности. Тем не менее выгоды от его простоты во многих случаях перевешивают потенциальные недостатки. Более того, распределенные решения не обязательно лучше, как показано в разделе 6.3.

Распределенный алгоритм

В работе [Ricart and Agrawala, 1981] был представлен алгоритм с использованием оригинального решения логических часов Лампорта для распределенного взаимного исключения, который мы обсуждали в разделе 6.3. Этот алгоритм требует полного упорядочения всех событий в системе. То есть для любой пары событий, таких как сообщения, должно быть однозначно определено, какое событие произошло в первую очередь.

Алгоритм работает следующим образом. Когда процесс хочет получить доступ к общему ресурсу, он создает сообщение, содержащее имя ресурса, номер его процесса и текущее (логическое) время. Затем отправляет сообщение

всем другим процессам, концептуально включая себя. Отправка сообщений считается надежной; то есть ни одно сообщение не потеряно.

Когда процесс получает сообщение запроса от другого процесса, выполняемое им действие зависит от его собственного состояния в отношении ресурса, указанного в сообщении. Необходимо четко различать три различных случая:

1) если получатель не обращается к ресурсу и не хочет получать к нему доступ, он отправляет обратно сообщение OK отправителю;
2) если получатель уже имеет доступ к ресурсу, он просто не отвечает. Вместо этого он ставит запрос в очередь;
3) если получатель также хочет получить доступ к ресурсу, но еще не сделал этого, он сравнивает временную метку входящего сообщения с временной меткой, содержащейся в сообщении, которое он отправил всем. Запрос с самой ранней отметкой выигрывает. Если входящее сообщение имеет более раннюю отметку времени, получатель отправляет обратно сообщение OK. Если его собственное сообщение имеет более раннюю отметку времени, получатель ставит в очередь входящий запрос и ничего не отправляет.

После отправки запросов, запрашивающих разрешение, процесс бездействует и ждет, пока все остальные не дадут разрешения. Как только все разрешения будут введены, этот процесс может продолжаться. Когда он завершен, он отправляет сообщения OK всем процессам в своей очереди и удаляет их всех из очереди. Если нет никакого конфликта, это очевидным образом работает. Однако предположим, что два процесса пытаются одновременно получить доступ к ресурсу, как показано на рис. 6.16a.

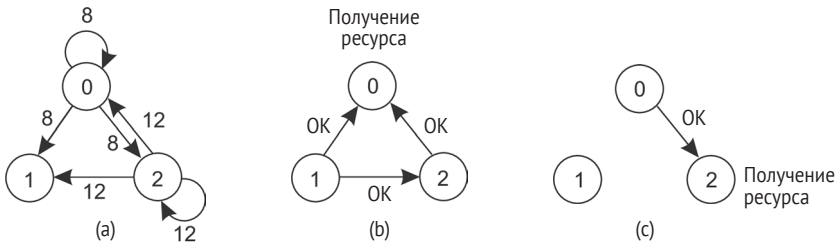


Рис. 6.16 ❖ a) Два процесса хотят получить доступ к общему ресурсу одновременно; b) P_0 имеет самую низкую отметку времени, поэтому он выигрывает; c) когда процесс P_0 завершен, он также отправляет OK, поэтому P_2 теперь может продолжать работу

Процесс P_0 отправляет всем запрос с отметкой времени 8, в то же время процесс P_2 отправляет всем запрос с отметкой времени 12. P_1 не заинтересован в ресурсе, поэтому отправляет OK обоим отправителям. Процессы P_0 и P_2 видят конфликт и сравнивают временные метки. P_2 видит, что проиграл, поэтому дает разрешение P_0, отправив OK. Процесс P_0 теперь ставит в очередь запрос от P_2 для последующей обработки и обращается к ресурсу, как показано на рис. 6.16b. Когда он завершается, он удаляет запрос P_2 из

своей очереди и отправляет сообщение OK в P_2, позволяя последнему идти дальше, как показано на рис. 6.16с. Алгоритм работает, потому что в случае конфликта выигрывает самая низкая временная метка, и все соглашаются с порядком временных меток.

С помощью этого алгоритма взаимное исключение гарантирует отсутствие тупика или голода. Если общее число процессов равно *N*, то количество сообщений, которые процессу необходимо отправить и получить, прежде чем он сможет войти в критическую секцию, равно $2(N - 1)$: $N - 1$ запросов сообщения ко всем другим процессам и впоследствии $N - 1$ сообщений OK, одно от каждого другого процесса.

К сожалению, этот алгоритм имеет *N* точек сбоя. В случае сбоя какого-либо процесса он не сможет отвечать на запросы. Это молчание будет неверно истолковано как отказ в разрешении, что блокирует все последующие попытки всех процессов войти в любую из своих критических областей. Алгоритм может быть исправлен следующим образом. Когда приходит запрос, получатель всегда отправляет ответ, предоставляя или отказывая в разрешении. Каждый раз, когда запрос или ответ теряется, отправитель останавливается и продолжает пытаться, пока либо ответ не вернется, либо отправитель не решит, что адресат мертв. После того как запрос отклонен, отправитель должен заблокировать ожидание последующего сообщения OK.

Другая проблема, связанная с этим алгоритмом, заключается в том, что необходимо использовать либо примитив многоадресной связи, либо каждый процесс должен поддерживать сам список членства в группе, включая процессы, входящие в группу, покидающие группу и аварийно завершающие работу. Этот метод лучше всего работает с небольшими группами процессов, которые никогда не меняют членство в группах. Наконец, обратите внимание, что все процессы участвуют во всех решениях, касающихся доступа к общему ресурсу, что может увеличить нагрузку на процессы, работающие на компьютерах с ограниченными ресурсами.

Возможны различные незначительные улучшения для этого алгоритма. Например, получать разрешение от всех – это лишнее. Все, что нужно, – это метод, предотвращающий одновременный доступ двух процессов к ресурсу. Алгоритм может быть изменен, чтобы дать разрешение, когда он собирает разрешения от простого большинства других процессов, а не ото всех них.

Алгоритм кольцо токенов

Совершенно другой подход к детерминированному достижению взаимного исключения в распределенной системе показан на рис. 6.17. В программном обеспечении мы строим оверлейную сеть в форме логического кольца, в котором каждому процессу назначается позиция в кольце. Все, что имеет значение, – это то, что каждый процесс знает, кто следующий в очереди после него.

Когда кольцо инициализируется, процессу P_0 присваивается **токен**. Маркер циркулирует по кольцу. Предполагая, что имеется *N* процессов, токен передается от процесса P_k к процессу $P_{(k+1) \bmod N}$ в сообщениях точка-точка.

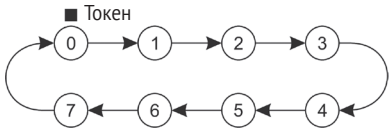


Рис. 6.17 ❖ Оверлейная сеть, построенная как логическое кольцо с токеном, циркулирующим между его членами

Когда процесс получает токен от своего соседа, он проверяет, нужен ли ему доступ к общему ресурсу. Если это так, процесс продолжается, выполняет всю необходимую работу и освобождает ресурсы. После того как он закончен, он передает токен вдоль кольца. Не допускается немедленный повторный вход в ресурс с использованием того же токена.

Если процесс получает токен от своего соседа и не заинтересован в ресурсе, он просто передает токен. Как следствие, когда ресурс не нужен ни одному процессу, токен просто циркулирует по кольцу.

Легко видеть корректность этого алгоритма. Только один процесс может иметь токен в любой момент, поэтому лишь один процесс может получить доступ к ресурсу. Поскольку токен циркулирует среди процессов в четко определенном порядке, голодания произойти не может. Как только процесс решит, что хочет получить доступ к ресурсу, в худшем случае ему придется ждать, пока все остальные процессы будут использовать этот ресурс.

У этого алгоритма есть свои проблемы. Если токен когда-либо будет потерян, например из-за сбоя его владельца или из-за потерянного сообщения, содержащего токен, он должен быть восстановлен. Фактически обнаружение, что это потеряно, может быть трудным, так как промежуток времени между последовательными появлениями токена в сети не ограничен. Тот факт, что токен не был обнаружен в течение часа, не означает, что он был потерян; кто-то может все еще использовать его.

Алгоритм также сталкивается с проблемами в случае сбоя процесса, но восстановление осуществляется сравнительно легко. Если мы требуем, чтобы процесс, получающий токен, подтвердил получение, мертвый процесс будет обнаружен, когда его сосед попытается выдать ему токен и произойдет сбой. В этот момент мертвый процесс может быть удален из группы, и владелец токена может перебросить токен мимо мертвого процесса следующему члену по линии или, если необходимо, какому-нибудь после него. Конечно, это требует, чтобы все сохранили текущую конфигурацию кольца.

Децентрализованный алгоритм

Давайте посмотрим на полностью децентрализованное решение. В работе [Lin et al., 2004] предлагается использовать алгоритм голосования. Предполагается также, что каждый ресурс реплицируется N раз. Каждая реплика имеет своего собственного координатора для контроля доступа параллельными процессами.

Однако когда процесс хочет получить доступ к ресурсу, ему просто нужно получить большинство голосов от $m > N/2$ координаторов. Мы предполагаем, что когда координатор не дает разрешения на доступ к ресурсу (что он будет делать, когда предоставил разрешение другому процессу), он сообщит об этом запрашивающей стороне.

Предполагается, что когда координатор выходит из строя, он быстро восстанавливается, но забывает о любом голосовании, которое дал до того, как вышел из строя. Другой способ просмотра состоит в том, что координатор сбрасывает себя в произвольные моменты времени. Риск, который мы принимаем, состоит в том, что сброс приведет к тому, что координатор забудет, что ранее предоставил разрешение на доступ к ресурсу для какого-либо процесса. Как следствие он может неправильно предоставить это разрешение другому процессу после восстановления.

Пусть $p = \Delta t/T$ будет вероятностью того, что координатор сбрасывается в течение интервала времени Δt, имея время жизни T. Тогда вероятность $\mathbb{P}[k]$, что k из m координаторов сбрасывается в течение того же интервала, равна

$$\mathbb{P}[k] = \binom{m}{k} p^k (1-p)^{m-k}.$$

Если f координаторов сбрасываются, то правильность механизма голосования будет нарушена, когда у нас будет только меньшинство работающих координаторов, то есть когда $m - f \leq N/2$, или, другими словами, когда $f \geq m - N/2$. Вероятность такого нарушения тогда равна $\sum_{k=m-N/2}^{m} \mathbb{P}[k]$. Чтобы составить представление о том, что это может означать, на рис. 6.18 показана вероятность нарушения правильности для различных значений N, m и p. Обратите внимание, что мы вычисляем p, считая количество секунд в час, которое координатор переустанавливается, а также принимая это значение за среднее время, необходимое для доступа к ресурсу. Наши значения для p считаются (очень) консервативными. Вывод состоит в том, что, как правило, вероятность нарушения правильности может быть настолько низкой, что ею можно пренебречь по сравнению с другими типами ошибок.

N	m	p	Нарушение
8	5	3 с/ч	$< 10^{-15}$
8	6	3 с/ч	$< 10^{-18}$
16	9	3 с/ч	$< 10^{-27}$
16	12	3 с/ч	$< 10^{-36}$
32	17	3 с/ч	$< 10^{-52}$
32	24	3 с/ч	$< 10^{-73}$

N	m	p	Нарушение
8	5	30 с/ч	$< 10^{-10}$
8	6	30 с/ч	$< 10^{-11}$
16	9	30 с/ч	$< 10^{-18}$
16	12	30 с/ч	$< 10^{-24}$
32	17	30 с/ч	$< 10^{-35}$
32	24	30 с/ч	$< 10^{-49}$

Рис. 6.18 ❖ Вероятности нарушения для различных значений параметров децентрализованного взаимного исключения

Для реализации этой схемы мы можем использовать систему, в которой ресурс реплицируется N раз. Предположим, что ресурс известен под его уникальным именем rname. Затем мы можем предположить, что i-я реплика

называется $rname_i$, которая затем используется для вычисления уникального ключа с использованием известной хеш-функции. Как следствие каждый процесс может генерировать N ключей с учетом имени ресурса и впоследствии искать каждый узел, отвечающий за реплику (и контролирующий доступ к этой реплике), с использованием некоторой обычно используемой системы наименования.

Если в доступе к ресурсу отказано (то есть процесс получает меньше m голосов), предполагается, что он откатится на какое-то случайное время и сделает следующую попытку позже. Проблема с этой схемой состоит в том, что если много узлов хотят получить доступ к одному и тому же ресурсу, оказывается, что использование ресурса быстро падает. В этом случае существует так много узлов, конкурирующих за получение доступа, что в конечном итоге никто не может набрать достаточное количество голосов, оставляя ресурс неиспользованным. Решение этой проблемы можно найти в [Lin et al., 2004].

Примечание 6.5 (дополнительная информация: сравнение алгоритмов взаимного исключения)

Краткое сравнение алгоритмов взаимного исключения, которые мы рассмотрели, поучительно. На рис. 6.19 мы перечислили алгоритмы и два свойства производительности: количество сообщений, необходимых для процесса, чтобы получить доступ и освободить общий ресурс, и задержка, прежде чем доступ может произойти (при условии что сообщения передаются последовательно по сети).

Алгоритм	Сообщений за вход/выход	Задержка перед входом (в количестве сообщений)
Централизованный	3	2
Распределенный	$2(N-1)$	$2(N-1)$
Кольцо токенов	$1, \ldots, \infty$	$0, \ldots, N-1$
Децентрализованный	$2mk + m, k = 1, 2$	$2mk$

Рис. 6.19 ❖ Сравнение четырех алгоритмов взаимного исключения

Далее мы предполагаем только сообщения точка-точка (или, что то же самое, считаем многоадресную рассылку для N процессов как N сообщений).

- Централизованный алгоритм является самым простым, а также наиболее эффективным. Для входа и выхода из критической области требуется всего три сообщения: запрос, разрешение на вход и выпуск для выхода.
- Для распределенного алгоритма требуется $N - 1$ сообщение с запросами, по одному для каждого из других процессов, и дополнительно $N - 1$ сообщение о предоставлении, всего $2(N - 1)$.
- При использовании алгоритма кольцо токенов число является переменным. Если каждый процесс постоянно хочет войти в критическую область, то каждый проход токена приведет к одному входу и выходу, в среднем по одному сообщению на каждую введенную критическую область. С другой стороны, токен может иногда циркулировать часами, и в нем никто не заинтересован. В этом случае количество сообщений на вход в критическую область не ограничено.
- В децентрализованном случае мы видим, что сообщение с запросом необходимо отправить m координаторов, а затем отправить ответное сообщение. Тем не

менее возможно, что будет предпринято несколько попыток (для которых мы вводим переменную k). Для выхода требуется отправить сообщение каждому из m координаторов.

Задержка с момента, когда процесс должен войти в критическую область, до фактического входа также изменяется. Для анализа наихудшего случая мы предполагаем, что сообщения отправляются одно за другим (т. е. никогда не бывает двух или более сообщений в пути одновременно) и что время передачи сообщений везде примерно одинаково. Задержка может быть выражена в **единицах времени передачи сообщений** (message transfer time units, MTTU).

При этих предположениях, когда время использования ресурса мало, доминирующий фактор задержки определяется общим количеством сообщений, отправленных через систему, прежде чем доступ может быть предоставлен. Когда ресурсы используются в течение длительного периода времени, доминирующий фактор ждет, когда все остальные займут свою очередь. На рис. 6.19 мы показываем первый случай.

- Для входа в критическую область в централизованном случае требуется всего две MTTU, вызванные сообщением запроса и последующим сообщением о предоставлении, отправленным координатором.
- Распределенный алгоритм требует отправки $N - 1$ сообщений с запросами и получения еще $N - 1$ сообщений о предоставлении, увеличивая MTTU до $2(N - 1)$.
- Для кольца токенов задержка варьируется от 0 MTTU (в случае если токен только что прибыл) до $N - 1$ (для того момента, когда токен только что вышел).
- Децентрализованный случай требует отправки m сообщений координаторам и еще m ответов, но процессу может потребоваться выполнить $k \geq 1$ попыток, увеличивая MTTU до $2mk$.

Практически все алгоритмы сильно страдают в случае сбоев. Специальные меры и дополнительные сложности должны быть введены, чтобы избежать сбоя всей системы. Несколько иронично, что распределенные алгоритмы, как правило, более чувствительны к сбоям, чем централизованные. В этом смысле не должен вызывать удивления тот факт, что действительно широко применяется централизованное взаимное исключение: оно простое для понимания поведения и относительно легко повышает отказоустойчивость централизованного сервера. Однако централизованные решения могут страдать от проблем масштабируемости.

6.4. Алгоритмы выбора

Многие распределенные алгоритмы требуют, чтобы один процесс выполнял функции координатора, инициатора или иным образом выполнял какую-то особую роль. В общем, не имеет значения, какой процесс берет на себя эту особую ответственность, но один из них должен это сделать. В этом разделе мы рассмотрим алгоритмы выбора координатора (используя его как общее имя для специального процесса).

Если все процессы одинаковы, без отличительных характеристик, нет способа выбрать один из них, который будет особенным. Следовательно, мы будем предполагать, что каждый процесс P имеет уникальный идентификатор $id(\mathrm{P})$. В общем, алгоритмы выборов пытаются найти процесс с самым высоким идентификатором и назначить его в качестве координатора. Алгоритмы отличаются тем, как они находят координатора.

Кроме того, мы также предполагаем, что каждый процесс знает идентификатор каждого другого процесса. Другими словами, каждый процесс обладает полным знанием группы процессов, в которой должен быть избран координатор. Чего процессы не знают, так это какие из них в настоящее время работают, а какие не работают. Цель алгоритма выбора состоит в том, чтобы гарантировать, что когда выборы начинаются, он завершается всеми процессами, согласовывающими, кем должен быть новый координатор. Существует много алгоритмов и вариантов, некоторые из которых обсуждаются в учебниках [Tel, 2000] и [Lynch, 1996].

Алгоритм хулигана

Хорошо известным решением для выбора координатора является **алгоритм хулигана** (bully algorithm), разработанный в [Garcia-Molina, 1982]. Далее мы рассмотрим N процессов $\{P_0, \ldots, P_{N-1}\}$, и пусть $id(P_k) = k$. Когда какой-либо процесс замечает, что координатор больше не отвечает на запросы, он инициирует выборы. Процесс P_k проводит выборы следующим образом:

1) P_k отправляет сообщение ELECTION всем процессам с более высокими идентификаторами: P_{k+1}, P_{k+2}, ..., P_{N-1};
2) если никто не отвечает, P_k побеждает на выборах и становится координатором;
3) если один с более высоким идентификатором отвечает, он вступает во владение, и работа P_k выполнена.

В любой момент процесс может получить сообщение ELECTION (ВЫБОР) от одного из своих коллег с меньшим номером. Когда такое сообщение приходит, получатель отправляет сообщение OK обратно отправителю, чтобы указать, что он жив и вступит во владение. Получатель затем проводит выборы, если только они уже не проведены. В конце концов, все процессы отказываются, кроме одного, и он становится новым координатором. Он объявляет о своей победе, отправляя всем процессам сообщение, что становится новым координатором.

Если процесс, который ранее был остановлен, возвращается, он проводит выборы.

Если это будет процесс с наибольшим номером в настоящее время, он победит на выборах и возьмет на себя работу координатора. Таким образом, всегда побеждает самый большой парень на деревне, отсюда и название «алгоритм хулигана».

На рис. 6.20 мы видим пример работы алгоритма хулигана. Группа состоит из восьми процессов с номерами от 0 до 7. Раньше координатором был процесс P_7, но он только что потерпел крах. Процесс P_4 является первым, кто это заметил, поэтому он отправляет сообщения ELECTION всем процессам выше его, а именно P_5, P_6 и P_7, как показано на рис. 6.20a. Процессы P_5 и P_6 оба отвечают OK, как показано на рис. 6.20b. Получив первый из этих ответов, P_4 знает, что его работа закончена и что P_5 или P_6 вступит во владение и станет координатором. Процесс P_4 просто бездельничает и ждет, чтобы увидеть, кто станет победителем (хотя в этот момент можно сделать довольно хорошее предположение).

На рис. 6.20c P_5 и P_6 проводят выборы, каждый из которых отправляет сообщения только процессам с более высокими идентификаторами, чем он сам. На рис. 6.20d P_6 говорит P_5, что он вступит во владение. В этот момент P_6 знает, что P_7 мертв и что он (P_6) является победителем. Если есть информация о состоянии, которая должна быть считана с диска или в другом месте, чтобы узнать, где остановился старый координатор, P_6 должен теперь сделать то, что нужно. Когда он готов к передаче, он объявляет о передаче, отправляя сообщение COORDINATOR (КООРДИНАТОР) всем запущенным процессам. Когда P_4 получает это сообщение, он может продолжить операцию, которую пытался выполнить, обнаружив, что P_7 мертв, но на этот раз используя P_6 в качестве координатора. Таким образом обрабатывается сбой P_7, и работа может продолжаться.

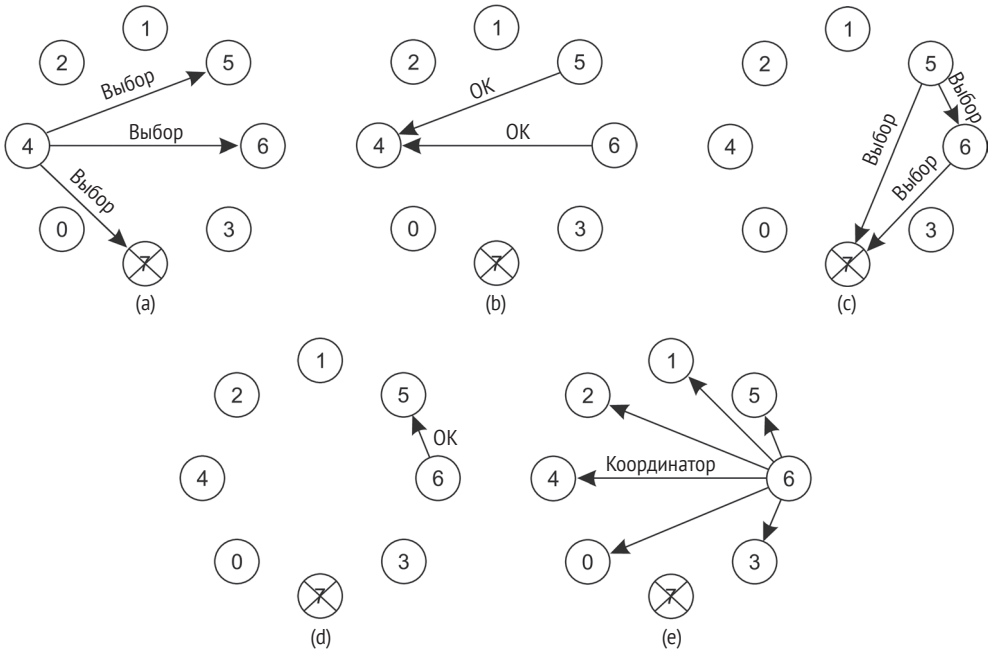


Рис. 6.20 ❖ Алгоритм хулигана: a) процесс 4 проводит выборы; b) процессы 5 и 6 отвечают, сообщая 4, что ему нужно остановиться; c) теперь и 5, и 6 проводят выборы; d) процесс 6 приказывает 5 остановиться; d) процесс 6 – победитель и сообщает об этом всем остальным

Если процесс P_7 когда-либо будет перезапущен, он отправит всем остальным сообщение COORDINATOR, и они станут подчиненными.

Кольцевой алгоритм

Рассмотрим следующий алгоритм выбора, основанный на использовании (логического) кольца. В отличие от некоторых кольцевых алгоритмов, он не

использует токен. Мы предполагаем, что каждый процесс знает, кто является его преемником. Когда какой-либо процесс замечает, что координатор не функционирует, он создает сообщение ELECTION, содержащее свой собственный идентификатор процесса, и отправляет сообщение своему преемнику. Если преемник не работает, отправитель пропускает преемника и переходит к следующему участнику по кольцу или дальше до тех пор, пока не будет найден запущенный процесс. На каждом этапе пути отправитель добавляет свой собственный идентификатор в список в сообщении, фактически превращая себя в кандидата, который будет выбран в качестве координатора.

В конце концов, сообщение возвращается к процессу, который все это начал. Этот процесс распознает событие, когда получает входящее сообщение, содержащее его собственный идентификатор. В этот момент тип сообщения изменяется на COORDINATOR и распространяется еще раз, на этот раз чтобы проинформировать всех остальных о том, кто является координатором (член списка с самым высоким идентификатором), кто – членами нового кольца. Когда это сообщение было распространено один раз, оно удаляется, и все возвращаются к работе.

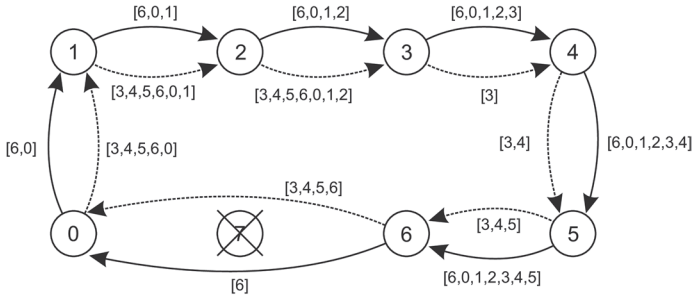


Рис. 6.21 ❖ Алгоритм выбора с использованием кольца. Сплошной линией показаны сообщения о выборах, инициированные P6; пунктирная – сообщения, инициированные P_3

На рис. 6.21 мы видим, что произойдет, если два процесса, P_3 и P_6, обнаружат одновременно, что предыдущий координатор, процесс P_7, потерпел крах. Каждый из них создает сообщение ELECTION, и каждый из них начинает распространять свое сообщение независимо от другого. В конце концов, оба сообщения будут проходить по всем направлениям, и P_3 и P_6 преобразуют их в сообщения COORDINATOR с абсолютно одинаковыми членами и в том же порядке. Когда оба снова обойдут сеть, оба будут удалены. Это не вредно для распространения дополнительных сообщений; в худшем случае будет потребляться немного больше пропускной способности, но это не будет расточительным.

Выборы в беспроводной среде

Традиционные алгоритмы выборов, как правило, основаны на предположениях, которые нереалистичны в беспроводной среде. Например, они предпо-

лагают, что передача сообщений является надежной и что топология сети не изменяется. Эти предположения являются необоснованными в большинстве беспроводных сред, особенно в мобильных одноранговых сетях.

Было разработано только несколько протоколов для выборов, которые работают в специальных сетях. В работе [Vasudevan et al., 2004] предлагается решение, которое может обрабатывать отказавшие узлы и разделять сети. Важным свойством этого решения является то, что лучший лидер может быть избран не просто случайным образом, как это было в той или иной степени в решениях, обсуждавшихся нами ранее. Их протокол работает следующим образом. Чтобы упростить наше обсуждение, мы концентрируемся только на специальных сетях и игнорируем то, что узлы могут перемещаться.

Рассмотрим специальную беспроводную сеть. Чтобы выбрать лидера, любой узел в сети, называемый источником, может инициировать выборы, отправив сообщение ELECTION своим непосредственным соседям (то есть узлам в своем диапазоне). Когда узел получает выбор ELECTION в первый раз, он назначает отправителя своим родителем, а затем отправляет сообщение ELECTION всем своим непосредственным соседям, кроме родителя. Когда узел получает сообщение ELECTION от узла, отличного от его родителя, он просто подтверждает получение.

Когда узел R назначил узел Q в качестве своего родителя, он пересылает сообщение ELECTION своим непосредственным соседям (исключая Q) и ожидает поступления подтверждений, прежде чем подтвердить сообщение ELECTION от Q. Это ожидание имеет важное последствие. Во-первых, обратите внимание, что соседи, которые уже выбрали родителя, будут немедленно отвечать на R. Более конкретно, если у всех соседей уже есть родитель, R является листовым (конечным) узлом и сможет быстро сообщить Q. При этом он также будет сообщать такую информацию, как срок службы батареи и другие содержательные ресурсы.

Эта информация позже позволит Q сравнивать пропускную способность R с другими нисходящими узлами и выбирать наиболее подходящий узел для лидерства. Конечно, Q послал сообщение ELECTION только потому, что его собственный родитель P также сделал это. В свою очередь, когда Q в конечном итоге подтверждает сообщение ELECTION, ранее отправленное P, он также передает P наиболее подходящий узел. Таким образом, источник в конечном итоге узнает, какой узел лучше всего выбрать в качестве лидера, после чего он передаст эту информацию всем другим узлам.

Этот процесс показан на рис. 6.22. Узлы были помечены от a до j вместе с их емкостью. Узел a инициирует выборы, передавая сообщение на узлы b и j, как показано на рис. 6.22b. После этого шага сообщения ELECTION распространяются на все узлы, заканчивая ситуацией, показанной на рис. 6.22d, где мы пропустили последнюю трансляцию по узлам f и i. С этого момента каждый узел сообщает своему родителю узел с наилучшей пропускной способностью, как показано на рис. 6.22f. Например, когда узел g получает подтверждения от своих дочерних элементов e и h, он замечает, что h – лучший узел, распространяющий [h, 8] к своему собственному родителю, узлу b. В конце концов, источник отметит, что h является лучшим лидером, и передаст эту информацию всем остальным узлам.

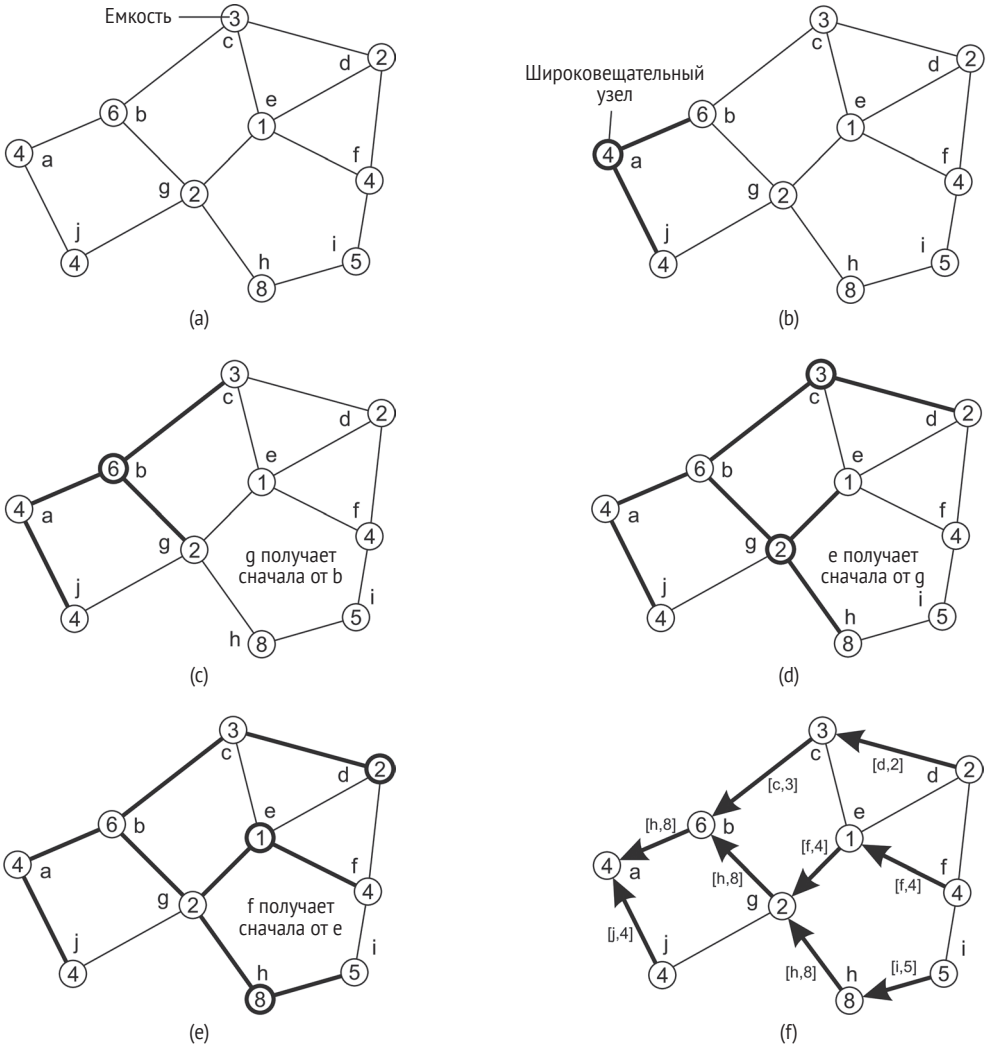


Рис. 6.22 ❖ Алгоритм выбора в беспроводной сети с узлом a в качестве источника: a) начальная сеть; b–e) фаза дерева сборки (последний шаг широковещания по узлам f и i не показан; f) сообщение о лучшем узле источнику

Когда инициировано несколько выборов, каждый узел решит присоединиться только к одним выборам. С этой целью каждый источник помечает свое сообщение ELECTION уникальным идентификатором. Узлы будут участвовать только в выборах с самым высоким идентификатором, что прекращает любое участие в других выборах.

С некоторыми незначительными корректировками можно показать, что протокол работает тогда, и когда сетевые разделы, и когда узлы присоединяются и уходят. Подробности можно найти в [Vasudevan et al., 2004].

Выборы в масштабных системах

Многие алгоритмы выбора лидера применимы только к относительно небольшим распределенным системам. Более того, алгоритмы часто концентрируются на выборе лишь одного узла. Существуют ситуации, когда на самом деле следует выбрать несколько узлов, например в случае **суперпиров** (super peer) в **одноранговых** (peer-to-peer) сетях, которые мы обсуждали в разделе 2.3. В этом разделе мы сконцентрируемся конкретно на проблеме выбора суперпиров.

Следующие требования должны быть выполнены для выбора суперпира (см. также [Lo et al., 2005]):

1) нормальные узлы должны иметь доступ с минимальными задержками к суперпирам;
2) суперпиры должны быть равномерно распределены по оверлейной сети;
3) должна быть заранее определенная часть суперпиров относительно общего количества узлов в оверлейной сети;
4) каждый суперпир не должен обслуживать больше, чем фиксированное количество нормальных узлов.

К счастью, эти требования относительно легко удовлетворяются в большинстве одноранговых систем, учитывая тот факт, что оверлейная сеть является либо структурированной (как в системах на основе DHT), либо случайно неструктурированной (как, например, можно реализовать с помощью сплетни на основе решений). Давайте посмотрим на решения, предложенные в [Lo et al., 2005].

В случае систем, основанных на DHT, основная идея состоит в том, чтобы зарезервировать часть пространства идентификаторов для суперпиров. В системе на основе DHT каждый узел получает случайный и равномерно назначенный m-битный идентификатор. Теперь предположим, что мы резервируем первые (то есть самые левые) k бит для идентификации суперпиров. Например, если нам нужно N суперуровней, то первые биты $\{\log_2(N)\}$ любого ключа могут быть использованы для идентификации этих узлов.

Для объяснения предположим, что у нас есть (маленькая) система Cord с $m = 8$ и $k = 3$. При поиске узла, ответственного за конкретный ключ K, мы можем сначала решить направить запрос поиска на узел, отвечающий за шаблон $K \wedge 11100000$, который затем обрабатывается как супероператор[1]. Обратите внимание, что каждый узел с идентификатором ID может проверить, является ли он суперузлом, просмотрев $ID \wedge 11100000$, чтобы проверить, направлен ли этот запрос на него. При условии что идентификаторы узлов равномерно назначаются узлам, можно видеть, что при общем количестве N узлов число суперпиров в среднем равно $2^{k-m}N$.

Совершенно другой подход основан на позиционировании узлов в m-мерном геометрическом пространстве. В этом случае предположим, что нам нужно разместить N суперпиров равномерно по всему оверлею. Основная

[1] Мы используем бинарный оператор $\wedge$ для обозначения побитового and.

идея проста: всего N токенов распределены по N случайно выбранным узлам. Ни один узел не может содержать более одного токена. Каждый токен представляет собой отказ, который склоняет другой токен отойти. Общий эффект состоит в том, что если все токены будут иметь одинаковую силу отталкивания, они будут отходить друг от друга и равномерно распределяться в геометрическом пространстве.

Этот подход требует, чтобы узлы, содержащие токен, изучали другие токены. Для этого мы можем использовать протокол сплетен, по которому сила токена распространяется по всей сети. Если узел обнаруживает, что суммарные силы, действующие на него, превышают пороговое значение, он будет перемещать токен в направлении объединенных сил, как показано на рис. 6.23. Когда токен удерживается узлом в течение заданного промежутка времени, этот узел продвигается до уровня суперпира.

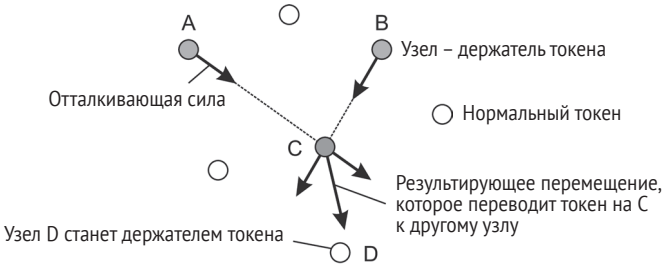


Рис. 6.23 ❖ Перемещение токенов в двумерном пространстве с использованием сил отталкивания

6.5. Системы локации

При рассмотрении очень больших распределенных систем, которые распределены по глобальной сети, часто необходимо принимать во внимание близость. Представьте себе распределенную систему, организованную в виде оверлейной сети, в которой два процесса являются соседями в оверлейной сети, но фактически расположены далеко друг от друга в базовой сети. Если эти два процесса много общаются, возможно, было бы лучше убедиться, что они также физически размещены поблизости друг от друга. В этом разделе мы рассмотрим методы определения местоположения процессов и их связь.

GPS: система глобального позиционирования

Давайте начнем с рассмотрения того, как определить ваше географическое положение в любой точке Земли. Эта проблема позиционирования сама по себе решается с помощью специальной специализированной распределенной системы, а именно **системы глобального позиционирования** (Global Positioning System, GPS). Это спутниковая распределенная система, которая была запущена в 1978 году. Хотя первоначально она использовалась в ос-

новном для военных целей, к настоящему времени она нашла применение во многих гражданских приложениях, особенно для дорожной навигации. Однако существует еще много доменов приложений. Например, современные смартфоны теперь позволяют владельцам отслеживать положение друг друга. Этот принцип может быть легко применен для отслеживания других объектов, включая домашних животных, детей, автомобили, лодки и т. д.

GPS использует до 72 спутников, каждый из которых вращается по орбите на высоте около 20 000 км. Каждый спутник имеет до четырех атомных часов, которые регулярно калибруются со специальных станций на Земле. Спутник непрерывно транслирует свою позицию и в каждом сообщении указывает свое местное время. Это позволяет каждому приемнику на Земле точно вычислять свою собственную позицию, используя, в принципе, только четыре спутника. Для пояснения давайте сначала предположим, что все часы, включая часы получателя, синхронизированы.

Чтобы вычислить позицию, рассмотрим первый двумерный случай, как показано на рис. 6.24, в котором нарисованы три спутника, а также круги, представляющие точки на одинаковом расстоянии от каждого соответствующего спутника. Мы видим, что пересечение трех окружностей является уникальной точкой.

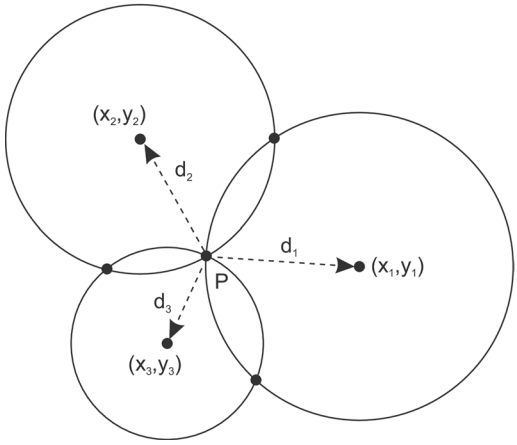


Рис. 6.24 ❖ Вычисление положения узла в двумерном пространстве

Этот принцип пересекающихся кругов можно расширить до трех измерений. Это означает, что нам нужно знать расстояние до четырех спутников, чтобы определить долготу, широту и высоту приемника на Земле. Такое расположение довольно просто, но определение расстояния до спутника становится сложным, когда мы переходим от теории к практике. Есть, по крайней мере, два важных конкретных факта, которые мы должны принять во внимание:

1) требуется некоторое время, прежде чем данные о местоположении спутника достигнут приемника;

2) часы приемника, как правило, не синхронизированы с часами спутника.

Предположим, что отметка времени от спутника абсолютно точна. Обозначим через Δ_r отклонение часов получателя от фактического времени. Когда сообщение принимается от спутника S_i с отметкой времени T_i, измеренная задержка получателем Δ_i состоит из двух компонентов – фактической задержки и ее собственного отклонения:

$$\Delta_i = (T_{now} - T_i) + \Delta_r,$$

где T_{now} – фактическое текущее время. Поскольку сигналы распространяются со скоростью света c, *измеренное* расстояние $\tilde{d}_i$ до спутника S_i равно $c\Delta_i$. Так как

$$d_i = c \cdot (T_{now} - T_i)$$

является реальным расстоянием между приемником и спутником S_i, измеренное расстояние можно переписать как $\tilde{d}_i = d_i + c\Delta_r$. Реальное расстояние теперь вычисляется как

$$\tilde{d}_i - c \cdot \Delta_r = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2},$$

где x_i, y_i и z_i обозначают координаты спутника S_i. Теперь мы видим систему квадратичных уравнений с четырьмя неизвестными (x_r, y_r, z_r), а также Δ_r.

Таким образом, нам нужны четыре опорные точки (то есть спутника), чтобы найти уникальное решение, которое даст нам Δ_r. Измерение GPS также даст отчет о реальном времени.

До сих пор мы предполагали, что измерения абсолютно точны. Конечно, нет. Существует множество источников ошибок, начиная с того, что атомные часы на спутниках не всегда находятся в идеальной синхронизации, положение спутника точно неизвестно, часы приемника имеют конечную точность, скорость распространения сигнала не постоянна (поскольку при входе сигналы замедляются, например в ионосфере) и т. д. В среднем это приводит к погрешности порядка 5–10 м. Для повышения точности необходимы специальные методы модуляции, а также специальные приемники. Используя так называемый дифференциальный GPS, с помощью которого корректирующая информация передается по глобальным каналам связи, точность может быть дополнительно улучшена. Более подробную информацию можно найти в [LaMarca and de Lara, 2008], а также в превосходном обзоре [Zogg, 2002].

Когда GPS не выбор

Основным недостатком GPS является то, что его обычно нельзя использовать в помещении. Для этого необходимы другие методы. Все более популярным методом является использование многочисленных точек доступа Wi-Fi. Основная идея проста: если у нас есть база данных известных точек доступа

вместе с их координатами и мы можем оценить наше расстояние до точки доступа, то, имея только три обнаруженные точки доступа, мы сможем вычислить наше положение. Конечно, это на самом деле не так просто.

Основной проблемой является определение координат точки доступа. Популярный подход заключается в том, чтобы сделать это **в военных целях при перемещении** (war driving): используя устройство с поддержкой Wi-Fi вместе с приемником GPS, кто-то проезжает или ходит и регистрирует наблюдаемые точки доступа. Точка доступа может быть идентифицирована через ее SSID или сетевой адрес MAC-уровня (medium access control). Точка доступа AP (access point) должна быть обнаружена в нескольких местах, прежде чем ее координаты могут быть оценены. Простой метод состоит в том, чтобы вычислить центроид: предположим, что мы обнаружили AP в N разных местах $\{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_N\}$, где каждое местоположение $\vec{x}_i$ состоит из пары (широта, долгота), предоставленной приемником GPS. Затем мы просто оцениваем местоположение AP $\vec{x}_{AP}$ как

$$\vec{x}_{AP} = \frac{\sum_{i=1}^{N} \vec{x}_i}{N}.$$

Точность может быть улучшена путем учета наблюдаемой силы сигнала и придания большего веса местоположению с относительно высокой наблюдаемой силой сигнала, чем местоположению, где был обнаружен только слабый сигнал. В итоге получаем оценку координат точки доступа. На точность этой оценки сильно влияют:

- точность каждой точки обнаружения GPS $\vec{x}_i$;
- тот факт, что точка доступа имеет неоднородный диапазон передачи;
- количество выборочных точек обнаружения N.

Исследования показывают, что оценки координат точки доступа могут находиться на расстоянии десятков метров от фактического местоположения (см., например, [Kim et al., 2006a] или [Tsui et al., 2010]). Более того, точки доступа приходят и уходят с относительно высокой скоростью. Тем не менее поиск и позиционирование точек доступа широко популярны, примером чему служит база данных Wigle с открытым доступом, которая заполняется с помощью краудсорсинга (crowd sourcing)[1].

Логическое позиционирование узлов

Альтернативой в попытке найти абсолютное местоположение узла в распределенной системе может быть использование логического местоположения на основе близости.

В **геометрических оверлейных сетях** (geometric overlay networks) каждому узлу присваивается положение в m-мерном геометрическом пространстве, так что расстояние между двумя узлами в этом пространстве отражает реальную метрику производительности. Вычисление такой позиции является основной задачей **системы сетевых координат NCS** (Network Coordinates

[1] См. wiggle.net.

System), обзор которой дан в [Donnet et al., 2010]. Самый простой пример – это когда расстояние соответствует задержке между узлами. Другими словами, если заданы два узла P и Q, то расстояние $\hat{d}$(P, Q) отражает то, сколько времени потребуется, чтобы сообщение перешло от P к Q, и наоборот. Мы используем $\hat{d}$ для обозначения расстояния в системе, где узлам были назначены координаты.

Существует множество применений геометрических оверлейных сетей. Рассмотрим ситуацию, когда веб-сайт на сервере O был реплицирован в интернете на несколько серверов S_1, ..., S_N. Когда клиент C запрашивает страницу у O, последний может решить перенаправить этот запрос на сервер, ближайший к C, то есть тот, который даст лучшее время ответа. Если известно геометрическое местоположение C, а также расположение каждого сервера реплики, O может просто выбрать сервер S_i, для которого $\hat{d}$(C, S_i) минимально. Обратите внимание, что такой выбор требует только локальной обработки в O. Другими словами, нет, например, необходимости выбирать все задержки между C и каждым из серверов реплики.

Другой пример – оптимальное размещение реплик. Рассмотрим снова веб-сайт, который собрал позиции своих клиентов. Если сайт должен был реплицировать свое содержимое на N серверов, он может вычислить N лучших позиций, где размещать реплики так, чтобы среднее время отклика клиента на реплику было минимальным. Выполнение подобных вычислений почти тривиально, если клиенты и серверы имеют геометрические положения, отражающие задержки между узлами.

В качестве еще одного примера рассмотрим **маршрутизацию на основе местоположения** (position-based routing) (см., например, [Popescu et al., 2012] или [Bilal et al., 2013]). В таких схемах сообщение пересылается к месту назначения с использованием только информации о местоположении. Например, простой алгоритм маршрутизации, позволяющий каждому узлу переслать сообщение соседу, ближайшему к месту назначения. Хотя легко показать, что этот конкретный алгоритм не должен сходиться, он показывает, что для принятия решения используется только локальная информация. Нет необходимости распространять информацию о соединении или подобную информацию на все узлы в сети, как в случае с обычными алгоритмами маршрутизации.

Централизованное позиционирование

Позиционирование узла в m-мерном геометрическом пространстве требует измерения расстояния $m + 1$ до узлов с известными позициями. Предполагая, что узел P хочет вычислить свою собственную позицию, он связывается с тремя другими узлами с известными позициями и измеряет свое расстояние до каждого из них. Связь только с одним узлом скажет P о круге, на котором он расположен; контакт только с двумя узлами говорит о положении пересечения двух окружностей (которое обычно состоит из двух точек); третий узел впоследствии позволил бы P вычислить его фактическое местоположение.

Узел P может вычислять свои собственные координаты (x_P, y_P), решая три квадратных уравнения с двумя неизвестными x_P, y_P:

$$\tilde{d}_i = \sqrt{(x_i - x_P)^2 + (y_i - y_P)^2} \quad (i = 1, 2, 3).$$

Здесь мы используем $\tilde{d}_i$ для обозначения измеренного или расчетного расстояния. Как уже говорилось, $\tilde{d}_i$ обычно соответствует измерению задержки между P и узлом в точке (x_i, y_i). Эта задержка может быть оценена как половина двусторонней задержки, но должно быть ясно, что ее значение будет изменяться со временем. Эффект – это другое позиционирование, когда P захочет пересчитать свою позицию. Более того, если другие узлы будут использовать текущее положение P для вычисления своих собственных координат, то очевидно, что ошибка в позиционировании P повлияет на точность позиционирования других узлов.

Также очевидно, что измеренные расстояния между множеством узлов, как правило, даже не будут согласованными. Например, предположим, что мы вычисляем расстояния в одномерном пространстве, как показано на рис. 6.25. В этом примере мы видим, что хотя R измеряет свое расстояние до Q как 2,0, а $\tilde{d}$(P, Q) было измерено равным 1,0, когда R измеряет $\tilde{d}$(P, R), оно находит 2.8, что явно противоречит двум другим измерениям.

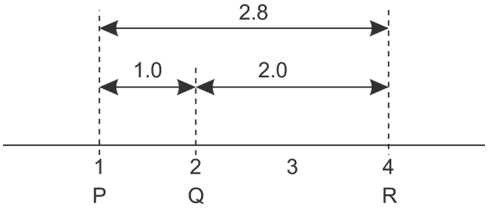


Рис. 6.25 ❖ Несогласованные измерения расстояний в одномерном пространстве

На рис. 6.25 также показано, как можно улучшить эту ситуацию. В нашем простом примере мы могли бы разрешить несоответствия, просто вычислив положения в двумерном пространстве. Это само по себе, однако, не является общим решением при работе со многими измерениями. Фактически, учитывая, что измерения задержки в интернете могут нарушать **неравенство треугольника** (triangle inequality), как правило, невозможно полностью устранить несоответствия. Неравенство треугольника утверждает, что в геометрическом пространстве для любых произвольных трех узлов P, Q и R всегда должно быть верно:

d(P, R) ≤ d(P, Q) + d(Q, R).

Существуют различные способы решения этих проблем. Один общий подход, предложенный в [Ng и Zhang, 2002], заключается в использовании N специальных узлов $L_1, \ldots, L_N$, известных как **ориентиры** (landmarks). Ориентиры измеряют свои попарные задержки $\tilde{d}(L_i, L_j)$ и впоследствии позволяют центральному узлу вычислять координаты для каждого ориентира.

С этой целью центральный узел стремится минимизировать следующую агрегированную ошибку функции:

$$\sum_{i=1}^{N}\sum_{j=i+1}^{N}\left(\frac{\tilde{d}(\mathsf{L}_i,\mathsf{L}_j)-\hat{d}(\mathsf{L}_i,\mathsf{L}_j)}{\tilde{d}(\mathsf{L}_i,\mathsf{L}_j)}\right)^2,$$

где, опять же, $\tilde{d}(\mathsf{L}_i, \mathsf{L}_j)$ соответствует расстоянию после позиционирования узлов L_i и L_j.

Скрытый параметр минимизации функции агрегированной ошибки – это размерность m. Очевидно, что у нас есть $N > m$, но ничто не мешает нам выбрать значение для m, которое намного меньше N. В этом случае узел P измеряет свое расстояние до каждого из N ориентиров и вычисляет свои координаты путем минимизации

$$\sum_{i=1}^{N}\left(\frac{\tilde{d}(\mathsf{L}_i,\mathsf{P})-\hat{d}(\mathsf{L}_i,\mathsf{P})}{\tilde{d}(\mathsf{L}_i,\mathsf{P})}\right)^2.$$

Как выясняется, при правильно выбранных ориентирах m может быть всего 6 или 7, причем $\tilde{d}(\mathsf{P}, \mathsf{Q})$ не более чем в 2 раза отличается от фактической задержки $d(\mathsf{P}, \mathsf{Q})$ для произвольных узлов P и Q [Szymaniak et al., 2004; 2008].

Децентрализованное позиционирование

Другим способом решения этой проблемы является просмотр совокупности узлов в виде большой системы, в которой узлы соединены друг с другом с помощью пружин. В этом случае $|\tilde{d}(\mathsf{P}, \mathsf{Q}) - \hat{d}(\mathsf{C}, \mathsf{S}_i)|$ указывает, насколько узлы P и Q смещены относительно ситуации, в которой система пружин будет в покое. Позволяя каждому узлу (слегка) изменить свое положение, можно показать, что система в конечном итоге будет сведена к оптимальной организации, в которой совокупная ошибка минимальна. Этот подход используется в системе Vivaldi, описанной в [Dabek et al., 2004a].

В системе с N узлами P_1, ..., P_N в Vivaldi стремятся минимизировать следующую агрегированную ошибку:

$$\sum_{i=1}^{N}\sum_{j=1}^{N}|\tilde{d}(\mathsf{P}_i,\mathsf{P}_j)-\hat{d}(\mathsf{P}_i,\mathsf{P}_j)|^2,$$

где $\tilde{d}(\mathsf{P}_i, \mathsf{P}_i)$ представляет собой измеренное расстояние (то есть задержку) между узлами P_i и P_j и $\hat{d}(\mathsf{P}_i, \mathsf{P}_j)$, расстоянием, вычисленным из сетевых координат каждого узла. Обозначим через $\vec{x}_i$ координаты узла P_i. В ситуации, когда каждый узел помещается в геометрическое пространство, сила, которую узел P_i оказывает на узел P_j, вычисляется как

$$\vec{F}_{ij} = (\tilde{d}(\mathsf{P}_i,\mathsf{P}_j) - \hat{d}(\mathsf{P}_i,\mathsf{P}_j)) \times u(\vec{x}_i - \vec{x}_j),$$

где $u(\vec{x}_i - \vec{x}_j)$ обозначает единичный вектор в направлении $\vec{x}_i - \vec{x}_j$. Другими словами, если $F_{ij} > 0$, узел P_i оттолкнет P_j и в противном случае потянет его к себе. Node P_i теперь неоднократно выполняет следующие шаги:

1) измерить задержку $\tilde{d}_{ij}$ до узла P_j, а также получить P_j координаты $\vec{x}_j$;
2) вычислить ошибку $e = \tilde{d}(\mathsf{P}_i, \mathsf{P}_j) - \hat{d}(\mathsf{P}_i, \mathsf{P}_j)$;

3) вычислить направление $\vec{u} = u(\vec{x}_i - \vec{x}_j)$;
4) вычислить вектор силы $F_{ij} = e\vec{u}$;
5) отрегулировать собственную позицию, перемещаясь вдоль вектора силы: $\vec{x}_i \leftarrow \vec{x}_i + \delta\vec{u}$.

Важным элементом является выбор δ: при слишком большом значении система будет колебаться; слишком маленьком – сближение со стабильной ситуацией займет много времени. Хитрость заключается в том, чтобы иметь адаптивное значение, которое является большим, когда ошибка также велика, но маленьким, когда требуются только небольшие корректировки. Подробности можно найти в [Dabek et al., 2004a].

6.6. Сопоставление распределенных событий

В качестве последнего аспекта координации между процессами рассмотрим сопоставление распределенных событий. Сопоставление событий, или, более точно, **фильтрация уведомлений** (notification filtering), лежит в основе систем публикации-подписки. Проблема сводится к следующему:

- процесс указывает посредством подписки S событие, в котором он заинтересован;
- когда процесс публикует уведомление N о возникновении события, система должна проверить, *соответствует* ли S N;
- в случае совпадения система должна отправить уведомление N подписчику, возможно, включая данные, связанные с событием, которое состоялось.

Как следствие нам нужно облегчить выполнение как минимум двух вещей: 1) сопоставление подписок с событиями и 2) уведомление подписчика в случае совпадения. Их можно разделить, но это получается не всегда. Далее мы предполагаем существование функции *match*(S, N), которая возвращает истину, когда подписка S соответствует уведомлению N, и ложь в противном случае.

Централизованные реализации

Простая, наивная реализация сопоставления событий должна иметь полностью централизованный сервер, который обрабатывает все подписки и уведомления. В такой схеме подписчик просто представляет подписку, которая впоследствии сохраняется. Когда издатель отправляет уведомление, это уведомление проверяется для каждой подписки, а когда совпадение найдено, уведомление копируется и пересылается соответствующему подписчику.

Очевидно, что это не очень масштабируемое решение. Тем не менее при условии, что сопоставление может быть выполнено эффективно и сам сервер обладает достаточной вычислительной мощностью, решение во многих случаях выполнимо. Например, при использовании централизованного сервера это – каноническое решение для реализации пространств кортежей Linda или

Java Spaces. Аналогично многие системы публикации-подписки, работающие в пределах одного отдела или организации, могут быть реализованы через центральный сервер. Важным в этих случаях является то, что функция сопоставления может быть реализована эффективно. На практике это часто имеет место при работе с фильтрацией по темам: сопоставление затем приводит к проверке на равенство значений атрибутов.

Обратите внимание, что простой способ масштабирования централизованной реализации состоит в том, чтобы детально разделить работу между несколькими серверами. Стандартный подход заключается в использовании двух функций, как объяснено в [Baldoni et al., 2009]:

- функция *sub2node*(S), которая берет подписку S и отображает ее на непустое подмножество серверов;
- функция *not2node*(N), которая принимает уведомление N и отображает его на непустое подмножество серверов.

Серверы, на которые отображается *sub2node*(S), называются **узлами рандеву** (rendezvous nodes) для S. Аналогично *sub2node*(N) являются узлами рандеву для N. Единственное ограничение, которое должно быть удовлетворено, – это то, что для любой подписки S и соответствующего уведомления N *sub2node*(S) ∩ *not2node*(N) ≠ Ø. Другими словами, должен существовать хотя бы один сервер, который может обрабатывать подписки при наличии соответствующего уведомления. На практике это ограничение удовлетворяется основанными на темах системами публикации-подписки с помощью функции хеширования названий тем.

Идея наличия центрального сервера может быть расширена путем распределения соответствия между несколькими серверами и разделения их работы. Серверы, обычно называемые **брокерами** (brokers), организованы в оверлейную сеть. При этом возникает вопрос: как направить уведомления соответствующему набору подписчиков. Следуя [Baldoni et al., 2009], мы различаем три разных класса: 1) потоки, 2) выборочная маршрутизация и 3) распространение на основе сплетен. Обширный обзор по объединению одноранговых сетей и систем публикации-подписки представлен в [Kermarrec и Triantafillou, 2013].

Простой способ убедиться, что уведомления достигают своих подписчиков, – это применить вещание. Есть два основных подхода. Первый: мы храним каждую подписку у каждого брокера, при этом публикуя уведомления только для одного брокера. Последний будет обрабатывать идентификацию соответствующих подписок, а затем копировать и пересылать уведомление. Альтернатива – хранить подписку только у одного брокера, передавая уведомления всем брокерам. В этом случае сопоставление распределяется между брокерами, что может привести к более сбалансированной рабочей нагрузке среди брокеров.

Примечание 6.6 (пример: TIB/Rendezvous)

В **TIB/Rendezvous**, основная архитектура которой показана на рис. 6.26 [TIBCO], используются лавинные уведомления. В этом подходе уведомлением является сообщение, помеченное составным ключевым словом, описывающим его содержание, например news.comp.os.books. Подписчик предоставляет (часть) ключевое слово или

указывает на сообщения, которые он хочет получить, например news.comp.*.books. Говорят, что эти ключевые слова указывают на **тему** сообщения.

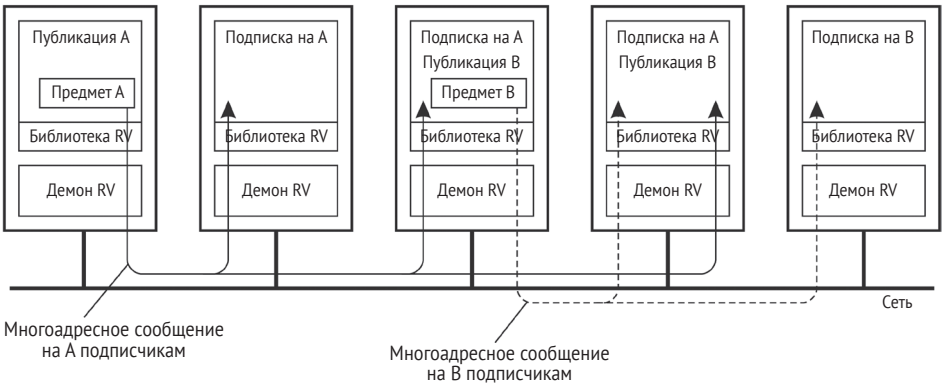


Рис. 6.26 ❖ Принцип системы публикации-подписки, реализованный в TIB/Rendezvous

Основополагающим для его реализации является применение вещания, распространенного в локальных сетях, хотя оно также использует, по возможности, более эффективные средства связи. Например, если точно известно, где находится абонент, обычно используются двухточечные сообщения. Каждый хост в такой сети будет запускать **демон рандеву** (rendezvous daemon, RV), который заботится о том, чтобы сообщения отправлялись и доставлялись в соответствии с их темой. Всякий раз, когда сообщение публикуется, оно направляется на каждый хост в сети, на котором запущен демон рандеву. Как правило, многоадресная рассылка реализуется с использованием средств, предлагаемых базовой сетью, таких как IP-рассылка или аппаратное вещание.

Процессы, которые подписываются на тему, передают свою подписку своему локальному демону. Демон создает таблицу (процесс, субъект) записей и всякий раз, когда приходит сообщение на тему S, просто проверяет в своей таблице локальных подписчиков и пересылает сообщение каждому из них. Если подписчиков на S нет, сообщение немедленно сбрасывается.

При использовании многоадресной рассылки, как это делается в TIB/Rendezvous, нет причин, по которым подписки не могут быть сложными и могут быть чем-то бóльшим, чем просто сравнение строк, как в данном случае. Ключевым аспектом здесь является то, что, поскольку сообщения в любом случае пересылаются на каждый узел, потенциально сложное сопоставление опубликованных данных с подписками может быть выполнено полностью локально без дальнейшей необходимости в сетевой связи.

Когда системы становятся большими, лавинное уведомление – не лучший путь, если он вообще возможен. Вместо этого могут потребоваться уведомления о маршрутизации через оверлейную сеть брокеров. Обычно это путь к информационным сетям (information-centric-networking), в которых используется маршрутизация на основе имен (name-based routing) [Ahlgren et al., 2012; Xylomenos et al., 2014]. Маршрутизация на основе имен – это особый случай выборочной маршрутизации уведомлений. Важным в этом слу-

чае является то, что брокеры могут принимать решения о маршрутизации, рассматривая содержимое сообщения уведомления. Точнее, предполагается, что каждое уведомление содержит достаточно информации, которая может использоваться для отключения маршрутов, для которых известно, что они не ведут к своим подписчикам.

Практический подход к избирательной маршрутизации предложен в [Carzaniga et al., 2004]. Рассмотрим систему публикации-подписки, состоящую из N брокеров, которым клиенты (то есть приложения) могут отправлять подписки и получать уведомления. В [Carzaniga et al., 2004] предлагается двухуровневая схема маршрутизации, в которой самый нижний уровень состоит из общего широковещательного дерева, соединяющего N брокеров. Существуют различные способы настройки такого дерева, начиная от многоадресной поддержки сетевого уровня до многоадресных деревьев уровня приложения, как мы обсуждали в главе 4. Здесь мы также предполагаем, что такое дерево было настроено с N посредниками в качестве концевых узлов вместе с набором промежуточных узлов, образующих маршрутизаторы. Обратите внимание, что различие между сервером и маршрутизатором является только логическим: на одной машине могут размещаться оба вида процессов.

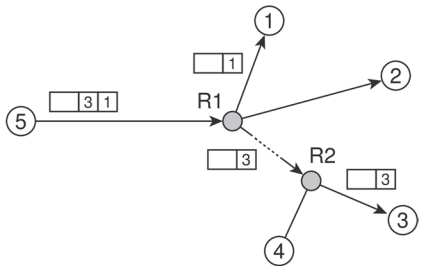


Рис. 6.27 ❖ Наивная маршрутизация, основанная на контенте

Каждый брокер передает свои подписки всем другим брокерам. В результате каждый брокер сможет составить список пар (*субъект, пункт назначения*). Затем всякий раз, когда процесс публикует уведомление N, связанный с ним брокер добавляет к этому сообщению брокеров-адресатов. Когда сообщение достигает маршрутизатора, последний может использовать список для определения путей, которым должно следовать сообщение, как показано на рис. 6.27.

Мы можем уточнить возможности маршрутизаторов для принятия решения, куда направлять уведомления. Для этого каждый брокер передает свою подписку по сети, чтобы маршрутизаторы могли составлять фильтры маршрутизации. Например, предположим, что узел 3 на рис. 6.27 подписывается на уведомления, для которых атрибут a лежит в диапазоне [0, 3], но этот узел 4 хочет получать сообщения от $a \in [2, 5]$. В этом случае маршрутизатор R_2 создаст фильтр маршрутизации в виде таблицы с записью для каждого из своих исходящих каналов (в этом случае три: один к узлу 3, один к узлу 4 и один к маршрутизатору R_1), как показано на рис. 6.28.

Более интересно то, что происходит на маршрутизаторе R_1. В этом примере подписки от узлов 3 и 4 диктуют, что любое уведомление, лежащее в интервале $[0, 3] \cup [2, 5] = [0, 5]$, должно быть перенаправлено по пути к маршрутизатору R_2, и это именно та информация, которую R_1 будет хранить в своей таблице. Нетрудно предположить, что могут поддерживаться и более сложные композиции подписки.

Интерфейс	Фильтр
Для узла 3	$a \in [0, 3]$
Для узла 4	$a \in [2, 5]$
На пути к маршрутизатору R_1	(не определено)

Рис. 6.28 ❖ Частично заполненная таблица маршрутизации

Этот простой пример также показывает, что всякий раз, когда узел покидает систему или когда он больше не заинтересован в конкретных уведомлениях, он должен отменить свою подписку и, по существу, передать эту информацию всем маршрутизаторам. Эта отмена, в свою очередь, может привести к настройке различных фильтров маршрутизации. Поздние корректировки в худшем случае приведут к ненужному трафику, так как уведомления могут пересылаться по маршрутам, для которых больше нет абонентов. Тем не менее своевременные корректировки необходимы для поддержания производительности на приемлемом уровне.

Последний тип сопоставления распределенных событий основан на сплетнях. Основная идея заключается в том, что подписчики, заинтересованные в одних и тех же уведомлениях, формируют свою собственную оверлейную сеть (которая создается с помощью сплетен), поэтому после публикации уведомления его просто необходимо направить на соответствующий оверлей. Для последнего можно использовать случайную прогулку. Этот подход применяется в TERA [Baldoni et al., 2007]. В качестве альтернативы в PolderCast [Setty et al., 2012] издатель сначала присоединяется к оверлею подписчиков, а затем передает свое уведомление. Оверлей подписчика строится по темам и представляет собой кольцо с ярлыками для облегчения эффективного распространения уведомления.

Примечание 6.7 (дополнительно: сплетни для сопоставления событий на основе содержимого)

Более сложный подход к объединению сплетен и совпадений событий используется в Sub-2-Sub [Voulgaris et al., 2006]. Рассмотрим систему публикации-подписки, в которой элементы данных могут быть описаны с помощью N атрибутов $a_1, \ldots, a_N$, значение которого может быть напрямую отображено как число с плавающей точкой. К таким значениям относятся, например, числа с плавающей точкой, целые числа, перечисления, логические значения и строки. Подписка S принимает форму кортежа пар (*атрибут, значение/диапазон*), таких как

$S = \langle a_1 \rightarrow 3.0, a_4 \rightarrow [0.0, 0.5) \rangle$.

В этом примере S указывает, что a_1 должно быть равно 3.0, а a_4 должно находиться в интервале [0.0, 0.5). Другие атрибуты могут принимать любое значение. Для ясности предположим, что каждый узел i входит только в одну подписку S_i.

Заметим, что каждая подписка S_i фактически определяет подмножество $\mathbf{S_i}$ в *N*-мерном пространстве чисел с плавающей точкой. Такое подмножество называется также гиперпространством. Для системы в целом интерес представляют только уведомления, которые попадают в объединение $\overline{\mathbf{S}} = \cup\mathbf{S_i}$ этих гиперпространств. Вся идея состоит в том, чтобы автоматически разбить $\overline{\mathbf{S}}$ на M непересекающихся гиперпространств $\overline{\mathbf{S}}_1 \ldots \overline{\mathbf{S}}_м$ – таких, что каждое полностью попадает в одно из гиперпространств подписки $\mathbf{S_i}$ и вместе они охватывают все подписки. Более формально, мы имеем:

$$(\overline{\mathbf{S}_m} \cap \mathbf{S_i} \neq \varnothing) \Rightarrow (\overline{\mathbf{S}_m} \subseteq \mathbf{S_i}).$$

Sub-2-Sub сохраняет *M* минимальным в том смысле, что нет разделения на меньшее количество частей $\overline{\mathbf{S}}_m$. С этой целью для каждого гиперпространства $\overline{\mathbf{S}}_m$ регистрируются именно те узлы i, для которых $\mathbf{S}_m \subseteq S_i$. В этом случае, когда уведомление публикуется, система должна просто найти $\overline{\mathbf{S}}_m$, которому принадлежит соответствующее событие, и с данного момента она может переслать уведомление соответствующим узлам.

С этой целью узлы регулярно обмениваются подписками через сплетни. Если два узла i и j заметят, что их соответствующие подписки пересекаются, то есть $\mathbf{S_{ij}} \equiv \mathbf{S_i} \cap \mathbf{S_j} \neq \emptyset$, они запишут этот факт и сохранят ссылки друг на друга. Если они обнаружат третий узел k с $\mathbf{S_{ijk}} \equiv \mathbf{S_{ij}} \cap \mathbf{S_k} \neq \emptyset$, три из них соединятся друг с другом, так что уведомление *N* от $\mathbf{S_{ijk}}$ может быть эффективно распространено. Обратите внимание, что если $\mathbf{S_{ij}} - \mathbf{S_{ijk}} \neq \emptyset$, узлы i и j сохранят свои взаимные ссылки, но теперь связывают его строго с $\mathbf{S_{ij}} - \mathbf{S_{ijk}}$.

По сути, то, что мы ищем, – это средство кластеризации узлов в *M* различных групп, так что узлы i и j принадлежат одной и той же группе тогда и только тогда, когда их подписки S_i и S_j пересекаются. Более того, узлы в той же группе должны быть организованы в оверлейную сеть, которая позволит эффективно распространять элемент данных в гиперпространстве, связанном с этой группой. Эта ситуация для одного атрибута показана на рис. 6.29.

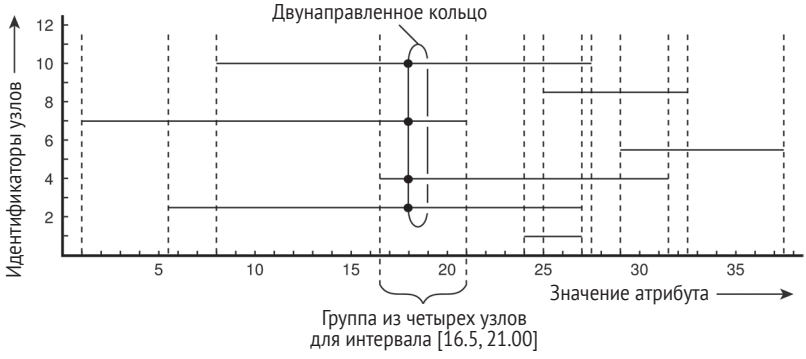


Рис. 6.29 ❖ Группировка узлов для поддержки запросов в основанной на сплетне системе публикация-подписка

Здесь мы видим всего семь узлов, в которых горизонтальная линия для узла i указывает диапазон его интересов для значения одного атрибута. Также показа-

но группирование узлов в непересекающиеся диапазоны интересов для значений атрибута. Например, узлы 3, 4, 7 и 10 будут сгруппированы вместе, представляя интервал [16.5, 21.0]. Любой элемент данных со значением в этом диапазоне должен распространяться только на эти четыре узла.

Чтобы построить эти группы, узлы организованы в неструктурированную сеть на основе сплетен. Каждый узел поддерживает список ссылок на других соседей (то есть **частичное представление**, partial view), которым он периодически обменивается с одним из своих соседей. Такой обмен позволит узлу узнавать о случайных других узлах в системе. Каждый узел отслеживает обнаруженные узлы с пересекающимися интересами (то есть с пересекающейся подпиской).

В определенный момент каждый узел i обычно будет иметь ссылки на другие узлы с перекрывающимися интересами. В рамках обмена информацией с узлом j узел i упорядочивает эти узлы по их идентификаторам и выбирает тот, у которого самый низкий идентификатор $i_1 > j$, так что его подписка перекрывается с подпиской узла j, то есть $\mathbf{S}_{j,i_1} \equiv \mathbf{S}_{i_1} \cap \mathbf{S}_j \neq \emptyset$.

Следующим выбирается $i_2 > i_1$, так что его подписка также перекрывается с подпиской j, но только если она содержит элементы, еще не покрытые узлом i_1. Другими словами, мы должны иметь $\mathbf{S}_{j,i_1,i_2} \equiv (\mathbf{S}_{i_2} - \mathbf{S}_{i_1}) \cap \mathbf{S}_i \neq \emptyset$. Этот процесс повторяется до тех пор, пока не будут проверены все узлы, которые имеют перекрывающийся интерес с узлом i, что приводит к упорядоченному списку $i_1 < i_2 < \ldots < i_n$. Обратите внимание, что узел i_k находится в этом списке, поскольку он охватывает область **R** общего интереса для узлов i и j, еще не покрытых совместно узлами с более низким идентификатором, чем i_k. По сути, узел i_k является *первым* узлом, которому узел j должен переслать уведомление, т. е. тому, кто попадает в эту уникальную область **R**. Эту процедуру можно расширить, чтобы позволить узлу i построить двунаправленное кольцо. Такое кольцо показано на рис. 6.29.

Всякий раз, когда уведомление N публикуется, оно распространяется как можно быстрее на *любой* узел, который заинтересован в нем. Как выясняется, с информацией, доступной на каждом узле, нахождение узла i, которого интересует N, просто. С этого момента узлу i нужно просто переслать N по кольцу подписчиков для определенного диапазона, в который попадает N. Чтобы ускорить распространение, ярлыки также поддерживаются для каждого кольца.

6.7. Координация на основе сплетен

В заключение темы координации мы рассмотрим несколько важных примеров использования сплетен. Далее разберем агрегацию, крупномасштабную одноранговую выборку и построение оверлея соответственно.

Объединение

Давайте посмотрим на некоторые интересные применения эпидемических протоколов. Мы уже упоминали о распространении обновлений, которое, пожалуй, является наиболее распространенным приложением. В том же свете сплетни могут использоваться для обнаружения узлов, которые имеют несколько исходящих глобальных ссылок, чтобы впоследствии применять направленные сплетни.

Другой интересной областью применения является просто сбор или фактическая агрегация информации [Jelasity et al., 2005]. Рассмотрим следующий обмен информацией. Каждый узел P_i изначально выбирает произвольное число, скажем v_i. Когда узел P_i связывается с узлом P_j, каждый из них обновляет свое значение как

$$v_i, v_j \leftarrow (v_i + v_j)/2.$$

Очевидно, что после этого обмена и P_i, и P_j будут иметь одинаковое значение. На самом деле нетрудно увидеть, что в конечном итоге все узлы будут иметь одинаковое значение, а именно среднее значение всех начальных значений. Скорость распространения снова экспоненциальная.

Какая польза от вычисления среднего? Рассмотрим ситуацию, когда все узлы P_i установили v_i в ноль, кроме P_1, который установил v_1 в 1:

$$v_i \leftarrow \begin{cases} 1, & \text{если } i = 1 \\ 0 & \text{в противном случае} \end{cases}.$$

Если существует N узлов, то в конечном итоге каждый узел будет вычислять среднее значение, которое равно $1/N$. Как следствие каждый узел P_i может оценить размер системы как $1/v_i$.

Вычисление среднего может оказаться трудным, когда узлы регулярно присоединяются и покидают систему. Одним из практических решений этой проблемы является введение эпох. Предполагая, что узел P_1 стабилен, он просто время от времени начинает новую эпоху. Когда узел P_i впервые видит новую эпоху, он сбрасывает свою собственную переменную v_i в ноль и снова начинает вычислять среднее значение.

Конечно, другие результаты также могут быть рассчитаны. Например, вместо того чтобы фиксированный узел, такой как P_1, начал вычисление среднего значения, мы можем легко выбрать случайный узел следующим образом. Каждый узел P_i первоначально устанавливает для v_i случайное число из того же интервала, скажем (0, 1], а еще постоянно сохраняет его как m_i. При обмене между узлами P_i и P_j каждый из них меняет свое значение на

$$v_i, v_j \leftarrow \max\{v_i, v_j\}.$$

Каждый узел P_i, для которого $m_i < v_i$, проиграет соревнование за то, что является инициатором в начале вычисления среднего. В конце концов, будет один победитель. Конечно, хотя легко сделать вывод о том, что узел проиграл, гораздо сложнее определить, что он выиграл, так как остается неопределенным, все ли результаты получены. При решении этой проблемы следует исходить из того, что узел всегда предполагает, что это победитель, пока не доказано обратное. В этот момент он просто сбрасывает переменную, которую использует для вычисления среднего на ноль. Обратите внимание, что несколько разных вычислений (в нашем примере вычисление максимума и вычисление среднего) могут выполняться одновременно.

Служба одноранговой выборки

Важным аспектом эпидемических протоколов является способность узла P *случайным образом* выбирать другой узел Q из всех доступных узлов в сети. Если задуматься над этим вопросом, то можно обнаружить серьезную проблему: если сеть состоит из тысяч узлов, как P может выбрать один из этих узлов, не имея полного обзора сети? Для небольших сетей часто можно прибегнуть к центральной службе, которая регистрирует каждый участвующий узел. Очевидно, что этот подход никогда не позволит получить масштабирование для больших сетей.

Решение состоит в том, чтобы создать полностью **децентрализованную службу одноранговой выборки** (peer-sampling service, PSS). Как оказывается, и кажется несколько нелогичным, PSS может быть построена с использованием эпидемического протокола. Как показано в [Jelasity et al., 2007], каждый узел поддерживает список соседей, в котором в идеале каждый из этих соседей представляет случайно выбранный живой узел из текущего набора узлов. Этот список соседей также называется **частичным представлением** (partial view). Есть много способов построить такое частичное представление. В решении авторов предполагается, что узлы регулярно обмениваются записями их частичного просмотра. Каждая запись идентифицирует другой узел в сети и имеет связанный возраст, который указывает, сколько времени ссылке на этот узел. Используются два потока, как показано на рис. 6.30.

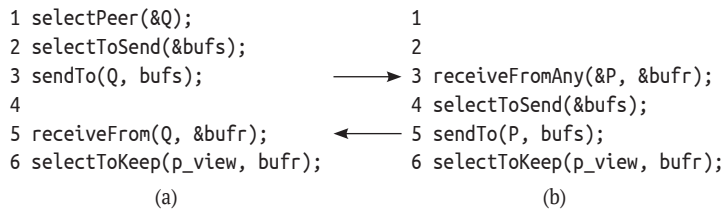


Рис. 6.30 ❖ Связь между (a) активным и (b) пассивным потоками в службе одноранговой выборки

Различные операции выбора определяются следующим образом:

- выбрать пир (select peer): случайный выбор соседа из локального частичного представления;
- выбрать послать (select to send): выбор некоторых других записей из частичного просмотра и добавление в список, предназначенный для выбранного соседа;
- выбрать сохранить (select to keep): добавление полученных записей в частичное представление, удаление повторяющихся элементов и сокращение представления до c элементов.

Активный поток берет на себя инициативу для связи с другим узлом. Он выбирает этот узел из его текущего частичного представления. Продолжается создание списка, содержащего записи $c/2 + 1$, включая запись, идентифици-

рующую себя. Другие записи взяты из текущего частичного представления. После отправки списка выбранному соседу он ожидает ответа.

Тем временем этот сосед также построит список с помощью пассивного потока, показанного на рис. 6.30b, чьи действия сильно напоминают действия активного потока.

Важнейшим моментом является построение нового частичного представления. Это представление как для связи, так и для контактирующего партнера, будет содержать ровно *c* записей, часть из которых будет из полученного списка. По сути, есть два способа построить новый обзор. Первый: два узла могут решить отказаться от записей, которые они отправили друг другу. Фактически это означает, что они обменяются частью их первоначальных обзоров. Второй подход заключается в отбрасывании как можно большего количества старых записей (на практике это означает, что после каждого раунда сплетен возраст каждой записи в каждом частичном представлении увеличивается на единицу).

Как оказалось, до тех пор, пока одноранговые узлы (пиры) регулярно запускают только что описанный алгоритм обмена, выбор случайного одноранговой узла из таким образом динамически изменяющегося частичного представления неотличим от случайного выбора одноранговой узла из всей сети. Конечно, выбор одноранговой узла в частичном представлении должен происходить примерно на той же частоте, что и обновление частичных представлений. Таким образом, мы создали полностью децентрализованную службу, основанную на сплетнях. Простая и часто используемая реализация службы одноранговой выборки – Cyclon [Voulgaris et al., 2005].

Структура оверлея, основанная на сплетнях

Хотя может показаться, что структурированные и неструктурированные одноранговые системы образуют строгие независимые классы, в действительности это не обязательно (см. также [Castro et al., 2005]). Одним из ключевых аспектов является то, что путем тщательного обмена и выбора записей из частичных представлений можно создавать и поддерживать конкретные топологии оверлейных сетей. Такое управление топологией достигается путем применения двухуровневого подхода, как показано на рис. 6.31.

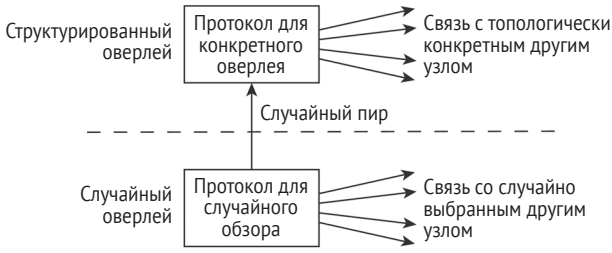


Рис. 6.31 ❖ Двухуровневый подход для построения и поддержки определенных оверлейных топологий с использованием методов неструктурированных одноранговых систем

Самый нижний уровень представляет собой неструктурированную одноранговую систему, в которой узлы периодически обмениваются записями своих частичных обзоров с целью предоставления услуги одноранговой выборки. Точность в этом случае соотносится с тем фактом, что частичный обзор должен быть заполнен записями, относящимися к случайно выбранным живым узлам.

Самый нижний уровень передает свой частичный обзор на верхний уровень, где происходит дополнительный выбор записей. Это приводит ко второму списку соседей, соответствующему желаемой топологии. В работе [Jelasity и Kermarrec, 2006] предлагается использовать функцию ранжирования, с помощью которой узлы упорядочиваются по некоторому критерию относительно данного конкретного узла. Простая функция ранжирования заключается в упорядочении набора узлов путем увеличения расстояния до заданного узла P. В этом случае узел P будет постепенно составлять список своих ближайших соседей, при условии что нижний уровень продолжает проходить случайно выбранные узлы.

В качестве иллюстрации рассмотрим логическую сетку размером $N \times N$ с узлом, размещенным в каждой точке сетки. Каждый узел должен поддерживать список c ближайших соседей, где расстояние между узлами в (a_1, a_2) и (b_1, b_2) определяется как $d_1 + d_2$, с $d_i = \min(N - |a_i - b_i|, |a_i - b_i|)$. Если самый нижний уровень периодически выполняет протокол, как показано на рис. 6.31, то топология, которая будет развиваться, становится тором, показанным на рис. 6.32.

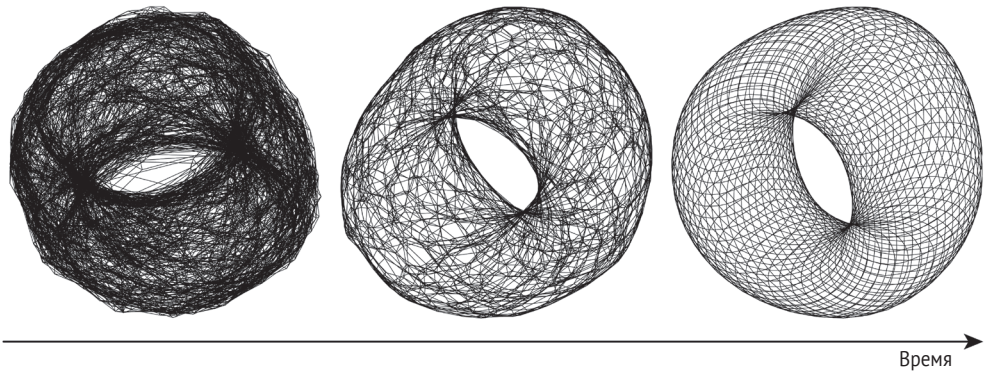


Рис. 6.32 ❖ Создание конкретной оверлейной сети с использованием двухслойной неструктурированной одноранговой системы (адаптировано с разрешения [Jelasity and Babaoglu, 2006])

6.8. Резюме

С коммуникацией между процессами тесно связана проблема синхронизации процессов в распределенных системах. Синхронизация – это правильные действия в нужное время. Проблема в распределенных системах

и компьютерных сетях в целом заключается в том, что отсутствует понятие глобально распределенных часов. Другими словами, процессы на разных машинах имеют свое представление о том, который час.

Существуют различные способы синхронизации часов в распределенной системе, но все методы по существу основаны на обмене значениями часов, принимая во внимание время, которое требуется для отправки и получения сообщений. Различия в задержках связи и способах их устранения во многом определяют точность алгоритмов синхронизации часов.

Во многих случаях знание абсолютного времени необязательно. Считается, что связанные события в разных процессах происходят в правильном порядке. Лампорт показал, что, введя понятие логических часов, можно собрать совокупность процессов для достижения глобального соглашения о правильном порядке событий. По сути, каждому событию e, такому как отправка или получение сообщения, назначается глобально уникальная логическая временная метка $C(e)$, так что когда событие a произошло до события b, $C(a) < C(b)$. Временные метки Лампорта могут быть распространены на векторные временные метки: если $C(a) < C(b)$, мы даже знаем, что событие a причинно предшествовало b.

Важным классом алгоритмов синхронизации является распределенное взаимное исключение. Эти алгоритмы гарантируют, что в распределенной коллекции процессов не более одного процесса одновременно имеет доступ к общему ресурсу. Распределенное взаимное исключение может быть легко достигнуто, если мы используем координатора, который отслеживает, чья это очередь. Также существуют полностью распределенные алгоритмы, но их недостатком является то, что они обычно более восприимчивы к сбоям связи и процессов.

Синхронизация между процессами часто требует, чтобы один процесс действовал как координатор. В тех случаях, когда координатор не зафиксирован, необходимо, чтобы процессы в распределенных вычислениях принимали решение о том, кто будет этим координатором. Такое решение принимается с помощью алгоритмов выборов. Алгоритмы выборов в основном используются в тех случаях, когда координатор может потерпеть крах. Тем не менее они также могут быть применены для выбора суперпиров в одноранговых системах.

С этими проблемами синхронизации связано позиционирование узлов в геометрических овелеях. Основная идея заключается в назначении координат каждого узла из m-мерного пространства таким образом, чтобы геометрическое расстояние можно было использовать в качестве точной меры для задержки между двумя узлами. Метод назначения координат сильно напоминает метод, используемый при определении местоположения и времени в GPS.

Особенно сложной задачей при координации является сопоставление распределенных событий, которое лежит в основе систем публикации и подписки. Относительно прост тот случай, когда у нас есть централизованные реализации, где сопоставление подписок с уведомлениями может быть выполнено, по существу, путем сравнения один к одному. Однако как только мы стремимся распределить нагрузку, мы сталкиваемся с проблемой необходимости

заранее определить, какой узел отвечает за какую часть подписок, не зная, каких уведомлений ожидать. Это особенно проблематично для сопоставления на основе контента, которое в конечном итоге требует продвинутых методов фильтрации для направления уведомлений соответствующим подписчикам.

Наконец, наиболее важным аспектом в координации на основе сплетен является возможность случайного выбора другого пира из целого оверлея. Как выясняется, мы можем реализовать такую службу выборки, используя сплетни, гарантируя, что частичный обзор обновляется регулярно и случайным образом. Объединение одноранговой выборки с выборочной заменой записей в частичном обзоре позволяет нам эффективно строить структурированные оверлейные сети.

Глава 7

Согласованность и репликация

Важной проблемой в распределенных системах является репликация (копирование и тиражирование) данных. Данные обычно копируются для повышения надежности или производительности. Одной из основных задач является поддержание согласованности реплик. Неформально это означает, что когда обновляется одна копия, мы должны гарантировать, что обновляются и другие копии; в противном случае реплики больше не будут теми же. В этой главе мы подробно рассмотрим, что на самом деле означает согласованность копируемых данных, а также различные способы достижения этой согласованности.

Мы начнем с общего введения и обсудим, почему репликация полезна и как она связана с масштабируемостью. В продолжение сосредоточимся на том, что на самом деле означает согласованность. Важный класс так называемых моделей согласованности предполагает, что несколько процессов одновременно получают доступ к общим данным. В такой ситуации согласованность можно определить как отношение процессов при чтении и обновлении общих данных в условиях, когда другие процессы также имеют доступ к этим данным.

В крупномасштабных распределенных системах модели согласованности для совместно используемых данных зачастую сложно эффективно реализовать. Во многих случаях могут использоваться более простые и легко реализуемые модели. Конкретный класс составляют модели согласованности, ориентированные на перспективного (возможно, мобильного) клиента. Ориентированные на клиента модели согласованности обсуждаются в отдельном разделе.

Согласованность – это только половина проблемы. Нам необходимо также рассмотреть, как реализуется согласованность на практике. По сути, следует рассмотреть две более или менее независимые проблемы. Мы начнем с управления репликами, которое учитывает не только размещение серверов реплики, но и то, как контент распределяется по этим серверам.

Вторая проблема заключается в поддержке реплик согласованными. В большинстве случаев приложения требуют строгой согласованности. Неформально это означает, что обновления должны распространяться между репликами как можно быстрее. Существуют различные альтернативы для

реализации строгой согласованности, которые обсуждаются в отдельном разделе. Также уделено внимание протоколам кеширования, которые образуют особый случай протоколов согласованности.

Мы уделяем отдельное внимание кешированию и репликации в веб-системах, пожалуй, самых крупных распределенных системах, особенно сетям доставки контента, а также методам кеширования на пограничных серверах.

7.1. Введение

Этот раздел мы начнем с обсуждения причин нашего желания реплицировать данные. Мы сосредоточимся на репликации как методе достижения масштабируемости и мотивации необходимости обеспечения согласованности.

Причины репликации

Есть две основные причины для репликации данных. Во-первых, данные реплицируются для повышения надежности системы. Если файловая система была реплицирована, можно продолжить работу после сбоя одной реплики, просто переключившись на одну из других реплик. Кроме того, благодаря поддержке нескольких копий становится возможным обеспечить лучшую защиту от поврежденных данных. Например, представим, что существует три копии файла, и каждая операция чтения и записи выполняется для каждой отдельной копии. Мы можем защитить себя от одной неудачной операции записи, считая правильным значение, возвращаемое как минимум двумя копиями.

Другая причина репликации данных – производительность. Репликация для производительности важна, когда распределенная система должна масштабироваться с точки зрения размера или с точки зрения географической области, которую она охватывает. Масштабирование по размеру происходит, например, когда все большему числу процессов требуется доступ к данным, которые управляются одним сервером. В этом случае производительность может быть улучшена путем репликации сервера и последующего распределения рабочей нагрузки между процессами, обращающимися к данным.

Масштабирование относительно географической области может также потребовать репликации. Основная идея состоит в том, что, помещая копию данных в непосредственной близости от процесса, использующего ее, уменьшает время доступа к данным. Как следствие производительность этого процесса увеличивается. Преимущества репликации для повышения производительности в связи с этим трудно переоценить. Хотя клиентский процесс получает более высокую производительность, это также может потребовать больше пропускной способности сети для поддержания актуальности всех реплик.

Коль репликация помогает повысить надежность и производительность, кто может быть против? К сожалению, за репликацию данных приходится платить. Проблема может возникнуть в связи с тем, что наличие нескольких копий может привести к проблемам согласованности. Всякий раз, ког-

да копия модифицируется, эта копия становится отличной от остальных. Следовательно, модификации должны распространяться на все копии для обеспечения согласованности. Цена репликации определяется в полном соответствии с тем, когда и как она выполняется.

Чтобы пояснить проблему, посмотрим на улучшение времени доступа к веб-страницам. Если не предпринимается никаких специальных мер, загрузка страницы с удаленного веб-сервера иногда может занять несколько секунд. Для повышения производительности веб-браузеры часто локально хранят копию ранее извлеченной веб-страницы (то есть они кешируют веб-страницу). Если пользователю снова требуется эта страница, браузер автоматически возвращает локальную копию. Это воспринимается пользователем как отличное время доступа. Однако если пользователь всегда хочет иметь последнюю версию страницы, ему может не повезти, потому что, если страница была изменена за это время, модификации не будут распространяться на кешированные копии, что делает эти копии устаревшими.

Одним из решений проблемы возврата устаревшей копии пользователю является запрет браузеру сохранять локальные копии, что позволяет серверу полностью отвечать за репликацию. Однако это решение может по-прежнему приводить к увеличению времени доступа, если реплика не размещена рядом с пользователем. Другое решение состоит в том, чтобы позволить веб-серверу сделать недействительной или обновить каждую кешированную копию, но для этого необходимо, чтобы сервер отслеживал все кеши и отправлял им сообщения. Это, в свою очередь, может ухудшить общую производительность сервера. Мы вернемся к проблемам производительности и масштабируемости ниже.

В дальнейшем мы в основном сосредоточимся на репликации как средстве повышения производительности. Репликация как средство повышения надежности обсуждается в главе 8.

Репликация как метод масштабирования

Репликация и кеширование для повышения производительности широко применяются в качестве методов масштабирования. Проблемы масштабируемости обычно появляются в форме проблем производительности. Размещение копий данных рядом с использующими их процессами может повысить производительность за счет сокращения времени доступа и, таким образом, решать проблемы масштабируемости.

Возможный компромисс, который приходится делать для поддержания копий в актуальном состоянии, может потребовать большей пропускной способности сети. Рассмотрим процесс P, который обращается к локальной реплике N раз в секунду, тогда как сама реплика обновляется M раз в секунду. Предположим, что обновление полностью обновляет предыдущую версию локальной реплики. Если N много меньше M, то есть отношение доступа к обновлению очень низкое, мы имеем ситуацию, когда P никогда не будет получать доступ ко многим обновленным версиям локальной реплики, что делает сетевое взаимодействие для этих версий бесполезным. В этом случае,

возможно, было бы лучше не устанавливать локальную реплику рядом с P или применять другую стратегию для обновления реплики.

Однако более серьезная проблема заключается в том, что сохранение согласованности нескольких копий само по себе может привести к серьезным проблемам с масштабируемостью. Интуитивно понятно, что коллекция копий согласована, когда копии всегда одинаковы. Это означает, что операция чтения, выполненная для любой копии, всегда будет возвращать тот же результат. Следовательно, когда операция обновления выполняется для одной копии, обновление должно распространяться на все копии до того, как будет выполнена последующая операция, независимо от того, с какой копии эта операция была инициирована или выполнена.

Этот тип согласованности иногда неофициально (и неточно) относится к так называемой жесткой согласованности, которая обеспечивается синхронной репликацией. (В разделе 7.2 мы предоставим точные определения согласованности и представим ряд моделей согласованности.) Основная идея заключается в том, что обновление выполняется для всех копий как отдельная атомарная операция или транзакция. К сожалению, реализация атомарности, включающей большое количество реплик, которые могут быть широко рассредоточены по крупномасштабной сети, затруднительна по своей сути, когда требуется также и быстро завершить операции.

Трудности связаны с тем, что нам нужно синхронизировать все реплики. По сути, это означает, что в первую очередь все реплики сначала должны прийти к соглашению о том, когда именно локальное обновление должно быть выполнено. Например, репликам может потребоваться принять решение о глобальном упорядочении операций с использованием меток времени Лампорта или позволить назначить такой порядок координатору. Глобальная синхронизация просто занимает много времени на общение, особенно когда реплики распространяются по глобальной сети.

Мы сейчас стоим перед дилеммой. С одной стороны, проблемы масштабируемости можно решить, применяя репликацию и кеширование, что приводит к повышению производительности. С другой – для обеспечения согласованности всех копий, как правило, требуется глобальная синхронизация, которая по своей природе сопряжена с большими затратами. Лечение может оказаться хуже, чем сама болезнь.

Во многих случаях единственным реальным решением является ослабление ограничений согласованности. Другими словами, если мы можем ослабить требование о том, что обновления должны выполняться как атомарные операции, мы можем избежать (немедленных) глобальных синхронизаций и, таким образом, повысить производительность. Цена за это будет определяться тем, насколько копии всегда и везде будут одинаковыми. Как выясняется, степень ослабления согласованности в значительной мере зависит от шаблонов доступа и обновления реплицируемых данных, а также от цели, для которой эти данные используются.

Существует целый ряд моделей согласованности и множество различных способов реализации моделей с помощью так называемых протоколов распределения и согласованности. Подходы к классификации согласованности и репликации можно найти в [Gray et al., 1996; Wiesmann et al., 2000] и [Aguilera and Terry, 2016].

7.2. Модели согласованности, ориентированные на данные

Традиционно согласованность обсуждалась в контексте операций чтения и записи общих данных, доступных с помощью (распределенной) общей памяти, (распределенной) общей базы данных или (распределенной) файловой системы. Здесь мы используем более широкий термин **хранилище данных** (data store). Хранилище данных может быть физически распределено по нескольким компьютерам. В частности, предполагается, что каждый процесс, который может получить доступ к данным из хранилища, имеет локальную (или близлежащую) копию всего хранилища. Операции записи распространяются на другие копии, как показано на рис. 7.1. Операция с данными классифицируется как операция записи, когда она изменяет данные, в противном случае классифицируется как операция чтения.

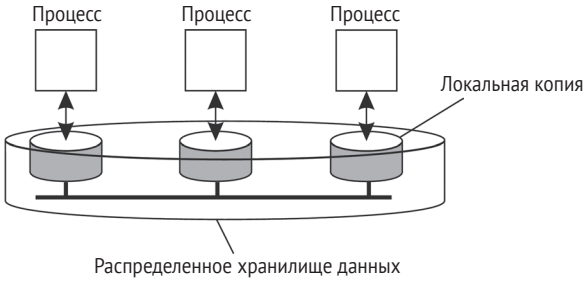


Рис. 7.1 ❖ Общая организация логического хранилища данных, физически распределенного и реплицированного по нескольким процессам

Модель согласованности (consistency model) – это, по сути, контракт между процессами и хранилищем данных. В ней говорится, что если процессы соглашаются подчиняться определенным правилам, хранилище обещает работать правильно. Обычно процесс, который выполняет операцию чтения над элементом данных, ожидает, что операция вернет значение, которое показывает результаты последней операции записи в этих данных.

В отсутствие глобальных часов трудно точно определить, какая операция записи является последней. В качестве альтернативы нам нужно дать другие определения, которые приведут к ряду моделей согласованности. Каждая модель эффективно ограничивает значения элемента данных, которые операция чтения может возвращать. Как и следовало ожидать, приложения с серьезными ограничениями просты в использовании, например при разработке приложений, тогда как приложения с небольшими ограничениями, как правило, считаются на практике трудными для использования. Разумеется, компромисс заключается в том, что простые в использовании модели работают не так хорошо, как сложные. Такова жизнь.

Непрерывное согласование

Лучшее решение для репликации данных как понятие отсутствует. Репликация данных создает проблемы согласованности, которые не могут быть эффективно решены в общем виде. Можно надеяться на достижение эффективных решений, только если мы ослабим согласованность. К сожалению, также нет общих правил для ослабления согласованности: то, что можно допустить, сильно зависит от приложений. У приложений есть разные способы указать, какие несоответствия они могут выдерживать. Авторы работы [Yu and Vahdat, 2002] придерживаются общего подхода, выделяя три независимые оси для определения несогласованности: отклонения между репликами в числовых значениях, отклонения между репликами в устаревании и отклонение относительно порядка операций обновления. Они относятся к этим отклонениям как к формированию непрерывных диапазонов согласованности.

Измерение несоответствия в терминах числовых отклонений может использоваться приложениями, для которых данные имеют числовую семантику. Одним из очевидных примеров является тиражирование записей, содержащих цены на фондовом рынке. В этом случае приложение может указать, что две копии не должны отклоняться более чем на 0,02 доллара США, что будет абсолютным числовым отклонением. В качестве альтернативы можно указать *относительное числовое отклонение*, указав, что две копии должны отличаться не более, например, чем на 0,5 %. В обоих случаях мы увидим, что если запас увеличивается (и одна из реплик немедленно обновляется) без нарушения указанных числовых отклонений, реплики будут считаться взаимно непротиворечивыми.

Числовое отклонение также можно воспринимать с точки зрения количества обновлений, которые были применены к данной реплике, но еще не были замечены другими. Например, веб-кеш, возможно, не видел пакет операций, выполняемых веб-сервером. В этом случае соответствующее отклонение в *значении* также называется его *весом*.

Отклонения устаревания относятся к последнему обновлению реплики. Для некоторых приложений допустимо, чтобы реплика предоставляла старые данные, если они не слишком старые. Например, отчеты о погоде обычно остаются достаточно точными в течение некоторого времени, например нескольких часов. В таких случаях основной сервер может получать своевременные обновления, но может принять решение распространять обновления на реплики только время от времени.

Наконец, существуют классы приложений, в которых порядок обновлений в разных репликах может быть различным, если различия остаются ограниченными. Один из способов просмотра этих обновлений заключается в том, что они предварительно применяются к локальной копии, ожидая глобального соглашения со всеми репликами. Как следствие от некоторых обновлений, возможно, придется отказаться и применить их в другом порядке, прежде чем они станут постоянными. Интуитивно понятно, что упорядочить отклонения гораздо сложнее, чем две другие вышеуказанные метрики согласованности.

О понятии конит

Чтобы определить несогласованность, Ю (Yu) и Вахдат (Vahdat) вводят **единицу согласованности** (consistency unit), сокращенно **конит** (conit). Конит определяет единицу измерения, которой должна измеряться согласованность. Например, в нашем примере с фондовой биржей конит может быть определен как запись, представляющая одну акцию. Другой пример – индивидуальный прогноз погоды.

Чтобы привести пример конита и одновременно проиллюстрировать числовые и порядковые отклонения, рассмотрим ситуацию с отслеживанием парка автомобилей. Владелец сайта, в частности, интересуется, сколько в среднем он платит за бензин. С этой целью всякий раз, когда он заправляет бензин, он сообщает количество бензина, которое было заправлено (обозначено как g), заплаченную сумму (обозначена как p) и общее расстояние с момента последней заправки (обозначено как переменная d). Технически три переменные g, p и d образуют конит. Это условие реплицируется на два сервера, как показано на рис. 7.2, и водитель регулярно сообщает о своем потреблении бензина одному из серверов, отдельно обновляя каждую переменную (без дальнейшего упоминания самого автомобиля).

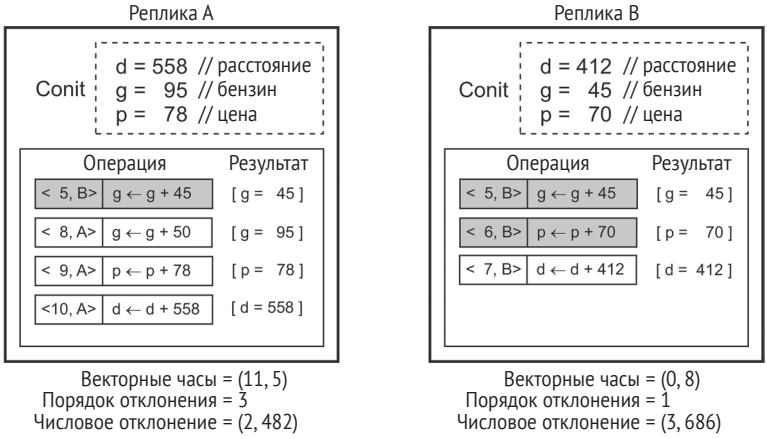


Рис. 7.2 ❖ Пример отслеживания отклонений согласованности

Задача серверов состоит в том, чтобы обеспечить постоянную репликацию конита. Для этого каждый сервер реплики поддерживает двумерные векторные часы. Мы используем обозначения (T, R), чтобы выразить операцию, которая была выполнена репликой R в (ее) логическое время T.

В этом примере мы видим две реплики, которые работают на конит, содержащий элементы данных g, p и d из нашего примера. Предполагается, что все переменные были инициализированы равными 0. Реплика A получила операцию

$\langle 5, B\rangle : g \leftarrow g + 45$

из реплики B. Мы заштриховали эту операцию серым цветом, чтобы указать, что A передал эту операцию своему локальному хранилищу. Другими словами, оно стало постоянным и не может быть отменено. Реплика A также имеет три *предварительные* операции обновления: (8, A), (9, A) и (10, A) соответственно. С точки зрения непрерывной согласованности, тот факт, что A имеет три предварительные операции, ожидающие выполнения, называется **отклонением порядка** (order deviation), в данном случае со значением 3. Аналогично, если всего три операции, две из которых были совершены, B имеет отклонение порядка 1.

Из этого примера мы заключаем, что логическое значение часов A теперь равно 11. Поскольку последняя операция, которую получил A из B, имела временную метку 5, векторные часы в A будут (11, 5), где мы принимаем, что первый компонент вектора используется для A, а второй для B. В тех же строках логические часы в B равны (0, 8).

Числовое отклонение (numerical deviation) в реплике R состоит из двух компонентов: числа операций во всех *других* репликах, которые еще не были замечены R, и суммы соответствующих пропущенных значений (более сложные схемы, конечно, также возможны). В нашем примере A еще не видел операций (6, B) и (7, B) с общим значением 70 + 412 единиц, что привело к числовому отклонению (2, 482). Аналогичным образом B по-прежнему не хватает трех предварительных операций в A с общим суммированным значением 686, что приводит к числовому отклонению B (3, 686).

Используя эти понятия, становится возможным указывать конкретные схемы согласованности. Например, мы можем ограничить отклонение заказа, указав допустимое максимальное значение. Аналогично мы можем захотеть, чтобы две реплики никогда не отклонялись численно более чем на 1000 единиц. Наличие таких схем согласованности требует, чтобы реплика знала, насколько она отклоняется от других реплик, что подразумевает, что необходимо отдельное общение для поддержки реплик в курсе. Лежащее в основе предположение состоит в том, что такое общение намного менее затратно, чем общение для поддержания синхронизации реплик. Правда, сомнительно, что это предположение подтверждается в нашем примере.

Примечание 7.1 (дополнительно: о степени детализации конитов)

Существует компромисс между поддержанием более и менее детализированных конитов. Если конит представляет много данных, таких как полная база данных, то в конит объединяются обновления для всех данных. Как следствие это может привести реплики в несогласованное состояние. Например, предположим, что на рис. 7.3 две реплики могут отличаться не более чем одним ожидающим обновлением. В этом случае каждый раз, когда элементы данных на рис. 7.3 были обновлены в первой реплике, во второй их также необходимо обновить. Это не тот случай, когда выбирается меньший конит, как показано на рис. 7.3. Там реплики все еще считаются актуальными. Эта проблема особенно важна, когда элементы данных, содержащиеся в коните, используются совершенно независимо, и в этом случае говорят, что они **ложно разделяют** (falsely share) конит.

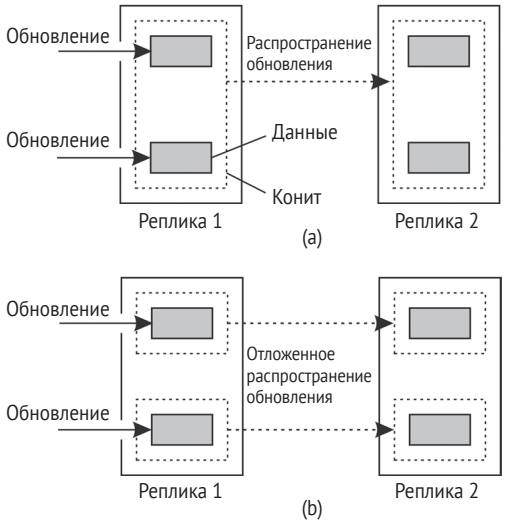


Рис. 7.3 ❖ Выбор подходящей гранулярности для конита:
а) два обновления приводят к распространению обновлений;
б) обновление не требуется

К сожалению, создание очень маленьких конитов не является хорошей идеей по той простой причине, что общее количество конитов, которыми необходимо управлять, также растет. Другими словами, существующие затраты, связанные с управлением, должны быть приняты во внимание. Эти затраты, в свою очередь, могут негативно повлиять на общую производительность, и это необходимо учитывать.

Хотя с концептуальной точки зрения кониты являются привлекательным средством для сбора требований согласованности, есть две важные проблемы, которые необходимо решить, прежде чем они могут быть использованы на практике. Во-первых, для обеспечения согласованности нам нужны протоколы. Протоколы непрерывной согласованности обсуждаются далее в этой главе.

Вторая проблема заключается в том, что разработчики программ должны определить требования согласованности для своих приложений. Практика показывает, что получение таких требований может быть чрезвычайно сложным. Программисты, как правило, не привыкли к репликации, не говоря уже о том, чтобы предоставлять подробную информацию о согласованности. Поэтому обязательно наличие простых и понятных интерфейсов программирования.

Примечание 7.2 (дополнительно: программирование)

Непрерывная согласованность может быть реализована в виде инструментария, кажущегося программистам просто еще одной библиотекой, которую они связывают со своими приложениями. Конит просто объявляется вместе с обновлением элемента данных. Например, фрагмент псевдокода

```
AffectsConit(ConitQ, 1, 1);
append message m to queue Q;
```

заявляет, что добавление сообщения в очередь `Q` принадлежит кониту с именем `ConitQ`.

Аналогично операции теперь могут быть также объявлены как зависимые от конитов:

```
DependsOmConit(ConitQ, 4, 0, 60);
read message m from head of queue Q;
```

В этом случае вызов `DependsOnConit()` указывает, что числовое отклонение, отклонение порядка и устаревание должны быть ограничены значениями 4, 0 и 60 (секунд) соответственно. Это может быть истолковано как то, что в других репликах должно быть не более 4 невидимых операций обновления, не должно быть никаких предварительных локальных обновлений, и локальная копия `Q` должна проверяться на предмет устаревания не более 60 секунд назад. Если эти требования не выполнены, базовое промежуточное программное обеспечение будет пытаться привести локальную копию `Q` в такое состояние, чтобы можно было выполнить операцию чтения.

Вопрос, конечно, в том, как система узнает, что `Q` связан с `ConitQ`? По практическим соображениям мы можем избежать явного объявления конитов и сосредоточиться только на группировке операций. Данные, подлежащие репликации, в совокупности считаются принадлежащими друг другу. Впоследствии, связывая операцию записи с именованным конитом и аналогично операции чтения, мы сообщаем промежуточному программному уровню, когда начинать синхронизацию всей реплики. Действительно, в таком случае может быть значительное количество ложных сообщений. Если необходимо избежать ложного совместного использования, мы должны были бы ввести отдельную программную конструкцию для явного объявления конитов.

Согласованный порядок операций

В последние десятилетия ведется большая работа над моделями согласованности, ориентированными на данные. Важный класс моделей проистекает из области параллельного программирования. Столкнувшись с тем фактом, что при параллельных и распределенных вычислениях нескольким процессам потребуется совместно использовать ресурсы и получать доступ к этим ресурсам одновременно, исследователи стремились выразить семантику одновременного доступа при репликации общих ресурсов. Все модели, которые мы здесь обсуждаем, имеют дело с последовательным упорядочением операций над общими, реплицированными данными.

В принципе, эти модели дополняют модели непрерывной согласованности в том смысле, что когда необходимо зафиксировать предварительные обновления в репликах, реплики должны будут достичь соглашения о глобальном, то есть согласованном, порядке этих обновлений.

Последовательная согласованность

В дальнейшем мы будем использовать специальные обозначения, в которых отражаем операции процесса вдоль оси времени. Ось времени всегда рису-

ется горизонтально, а время увеличивается слева вправо. Мы используем обозначение $W_i(x)a$, чтобы отметить, что процесс P_i записывает значение a в элемент данных x. Аналогично $R_i(x)b$ представляет тот факт, что процесс P_i читает x и возвращает значение b. Мы предполагаем, что каждый элемент данных имеет начальное значение NIL. Когда нет путаницы относительно того, какой процесс обращается к данным, мы опускаем индекс из символов W и R.

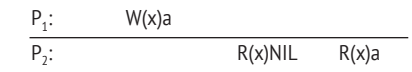


Рис. 7.4 ❖ Поведение двух процессов, работающих с одним и тем же элементом данных. Горизонтальная ось – время

В качестве примера на рис. 7.4 P_1 выполняет запись в элемент данных x, изменяя его значение на a. Обратите внимание, что в соответствии с нашей моделью системы операция $W_1(x)a$ сначала выполняется над копией хранилища данных, которая является локальной для P_1, и только затем она распространяется на другие локальные копии. В нашем примере P_2 позже считывает значение NIL, а через некоторое время спустя это a (из своей локальной копии хранилища). Здесь мы видим, что на распространение обновления x на P_2 потребовалось некоторое время, что вполне приемлемо.

Последовательная согласованность (sequential consistency) является важной моделью ориентированной на данных согласованности, которая была впервые определена Лампортом [1979] в контексте совместной памяти для многопроцессорных систем. Говорят, что хранилище данных последовательно согласовано, когда оно удовлетворяет следующему условию:

> *Результат любого выполнения такой же, как если бы операции (чтение и запись) выполнялись всеми процессами в хранилище данных в том же порядке последовательности, и операции каждого отдельного процесса появляются в этой последовательности в порядке, указанном его программой.*

Это определение означает, что когда процессы выполняются одновременно (возможно) на разных компьютерах, любое чередование операций чтения и записи является приемлемым поведением, но все процессы видят *одно и то же чередование операций*. Обратите внимание, что ничего не сказано о времени; то есть нет ссылки на «самую последнюю» операцию записи для элемента данных. Кроме того, процесс «видит» записи всех процессов, но только посредством собственного чтения.

То, что время не играет роли, видно на рис. 7.5. Рассмотрим четыре процесса, работающих с одним и тем же элементом данных x. На рис. 7.5a процесс P_1 сначала выполняет $W_1(x)a$ на x. Позже (в абсолютном времени) процесс P_2 также выполняет операцию записи $W_2(x)b$, устанавливая значение x равным b. Однако оба процесса P_3 и P_4 сначала считывают значение b, а затем значение a. Другими словами, операция записи $W_2(x)b$ процесса P_2, по-видимому, имела место до операции $W_1(x)a$ из P_1.

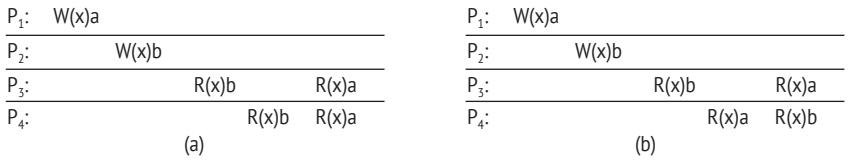


Рис. 7.5 ❖ a) Хранилище последовательного согласования данных; b) хранилище данных, которые не являются последовательными

Напротив, рис. 7.5b нарушает последовательную согласованность, потому что не все процессы видят одинаковое чередование операций записи. В частности, для обработки P_3 создается впечатление, что элемент данных сначала был изменен на b, а затем на a. С другой стороны, P_4 заключит, что конечное значение равно b.

Процесс P_1	Процесс P_2	Процесс P_3
x ← 1 print(y, z)	y ← 1 print(x, z)	z ← 1 print(x, y)

Рис. 7.6 ❖ Три одновременно выполняющихся процесса

Чтобы сделать понятие последовательной согласованности более конкретным, рассмотрим три одновременно выполняющихся процесса P_1, P_2 и P_3, показанных на рис. 7.6 (взят из [Dubois et al., 1988]). Элементы данных в этом примере образованы тремя целочисленными переменными x, y и z, которые хранятся в (возможно, распределенном) совместном последовательно согласованном хранилище данных. Мы предполагаем, что каждая переменная инициализируется с 0. В этом примере присваивание соответствует операции записи, тогда как оператор печати (`print`) соответствует одновременной операции чтения двух аргументов. Все операторы предполагаются неделимыми.

Возможны различные чередующиеся последовательности выполнения. С шестью независимыми операторами возможно потенциально 720 (6!) возможных последовательностей выполнения, хотя некоторые из них нарушают порядок программы. Рассмотрим 120 (5!) последовательностей, начинающихся x ← 1. Половина из них имеет `print(x, z)` перед y ← 1 и, таким образом, нарушает программный порядок. Половина также имеет `print(x, y)` перед z ← 1 и также нарушает порядок программы. Только 1/4 из 120 последовательностей, или 30, действительны. Возможны еще 30 действительных последовательностей, начиная с y ← 1, и еще 30 могут начинаться с z ← 1, всего 90 действительных последовательностей выполнения. Четыре из них показаны на рис. 7.7.

На рис. 7.7a три процесса выполняются по порядку: сначала P_1, затем P_2, потом P_3. Другие три примера демонстрируют различное, но одинаково правильное чередование утверждений во времени. Каждый из трех процессов печатает две переменные. Поскольку единственной величиной, которую может принимать каждая переменная, является начальное значение (0) или присвоенное значение (1), каждый процесс создает 2-битную строку. Числа после распечатки *Prints* являются фактическими выходами, которые появляются на устройстве вывода.

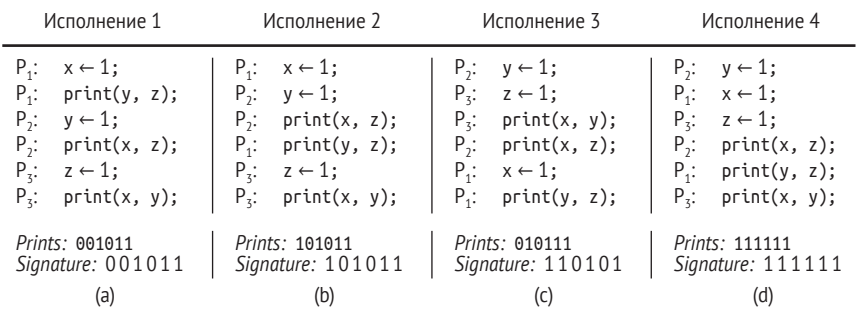


Рис. 7.7 ❖ Четыре допустимые последовательности выполнения для процессов, показанных на рис. 7.6. Вертикальная ось – время

Если мы объединяем выходные данные P_1, P_2 и P_3 в этом порядке, то получаем 6-битную строку, которая характеризует конкретное чередование операторов. Эта строка указана в качестве подписи (*Signature*) на рис. 7.7. Ниже мы будем характеризовать каждый порядок по его подписи, а не по распечатке.

Не все 64 шаблона подписи разрешены. В качестве тривиального примера 00 00 00 недопустимо, поскольку будет означать, что операторы печати выполняются перед операторами присваивания, что нарушает требование выполнения операторов в программном порядке. Более тонкий пример – 00 10 01. Первые два бита, 00, означают, что y и z оба были 0, когда P_1 выполнял печать. Эта ситуация возникает, только когда P_1 выполняет оба оператора до запуска P_2 или P_3. Следующие два бита, 10, означают, что P_2 должен работать после запуска P_1, но до начала P_3. Последние два бита, 01, означают, что P_3 должен завершиться до начала P_1, но мы уже видели, что P_1 должен идти первым. Поэтому 00 10 01 не допускается.

Короче говоря, 90 различных правильных порядков операторов приводят к различным результатам программы (но не более 64), которые допускаются при условии последовательной согласованности. Контракт между процессами и распределенным общим хранилищем данных заключается в том, что процессы должны принимать все эти данные как действительные результаты. Другими словами, процессы должны принять эти четыре результата, показанных на рис. 7.7, и все другие действительные результаты как правильные ответы и должны работать правильно, если произойдет какой-либо из них. Программа, которая работает только для некоторых из этих результатов и не работает для других, нарушает договор с хранилищем данных и является неправильной.

Примечание 7.3 (дополнительно: важность и тонкости последовательной согласованности)

Нет сомнений в том, что последовательная согласованность является важной моделью. По сути, из всех моделей согласованности, которые были разработаны и существуют, ее проще всего понять при разработке общих и параллельных приложений. Это связано с тем, что модель лучше всего соответствует нашим ожиданиям, когда мы позволяем нескольким программам работать с общими данными одно-

временно. В то же время реализация последовательной согласованности далеко не тривиальна [Adve and Boehm, 2010]. Для иллюстрации рассмотрим пример с двумя переменными x и y, показанный на рис. 7.8.

P_1:	W(x)a		W(y)b	R(x)b
P_2:		W(y)b	W(x)b	R(y)a

Рис. 7.8 ❖ Как x, так и y обрабатываются последовательно согласованным образом, но если взяты вместе, последовательная согласованность нарушается

Если мы просто рассмотрим операции записи и чтения на x, то факт, что P_1 считывает значение a, вполне согласован. То же самое относится к операции $R_2(y)b$ процессом P_2. Однако они берутся вместе, мы не можем упорядочить операции записи по x и y так, чтобы у нас были $R_1(x)a$ и $R_2(y)b$ (обратите внимание, что нам нужно сохранить порядок, выполняемый каждым обработать):

Порядок операций	Результат	
$W_1(x)a$; $W_2(y)b$; $W_1(y)a$; $W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_1(x)a$; $W_2(y)b$; $W_2(x)b$; $W_1(y)a$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b$; $W_1(x)a$; $W_1(y)a$; $W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b$; $W_1(x)a$; $W_2(x)b$; $W_1(y)a$	$R_1(x)b$	$R_2(y)a$

С точки зрения транзакций операции, выполняемые P_1 и P_2, **не упорядочены** (not serializable). Наш пример показывает, что последовательная согласованность не является **композиционной** (compositional): при наличии элементов данных, каждый из которых поддерживает последовательную согласованность, их композиция как набор не должна быть такой [Herlihy and Shavit, 2008]. Проблема некомпозиционной согласованности может быть решена посредством предположения **линеаризуемости** (linearizability). Это лучше всего объяснить, делая различие между началом и завершением операции и предполагая, что это может занять некоторое время. Линеаризуемость утверждает:

> *Каждая операция вступает в силу мгновенно в какой-то момент между ее началом и завершением.*

Возвращаясь к нашему примеру, на рис. 7.9 показан тот же набор операций записи, но теперь мы также указали, когда они выполняются: заштрихованная область обозначает время выполнения операции. Линеаризуемость утверждает, что эффект от операции должен иметь место где-то в интервале, указанном заштрихованной областью. В принципе, это означает, что во время завершения операции записи результаты следует распространять в другие хранилища данных.

P_1:	W(x)a		W(y)a	R(x)b
P_2:		W(y)b	W(x)b	R(y)a

Рис. 7.9 ❖ Пример учета линеаризуемой последовательной согласованности только с одним возможным исходом для x и y

Имея это в виду, возможности для правильного заказа становятся ограниченными:

Порядок операций	Результат	
$W_1(x)a; W_2(y)b; W_1(y)a; W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_1(x)a; W_2(y)b; W_2(x)b; W_1(y)a$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b; W_1(x)a; W_1(y)a; W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b; W_1(x)a; W_2(x)b; W_1(y)a$	$R_1(x)b$	$R_2(y)a$

В частности, $W_2(y)b$ завершается до запуска $W_1(y)a$, так что y будет иметь значение a. Аналогично $W_1(x)a$ завершается до начала $W_2(x)b$, так что x будет иметь значение b. Неудивительно, что реализация линеаризуемости на многоядерной архитектуре может вызвать серьезные проблемы с производительностью. Тем не менее это в то же время значительно облегчает программирование, поэтому следует искать компромисс.

Причинная согласованность

Модель причинно-следственной согласованности [Hutto and Ahamad, 1990] представляет собой ослабление последовательной согласованности в том смысле, что она делает различие между событиями, которые потенциально связаны с причинно-следственными связями, и событиями, которые не являются таковыми. Мы уже сталкивались в предыдущей главе с причинно-следственной связью при обсуждении векторных временных меток. Если событие b является причиной или находится под влиянием более раннего события a, причинно-следственная связь требует, чтобы все остальные сначала увидели a, а затем b.

Рассмотрим простое взаимодействие с помощью распределенной общей базы данных. Предположим, что процесс P_1 записывает элемент данных x. Затем P_2 читает x и записывает y. Здесь чтение x и запись y потенциально связаны причинно-следственной связью, потому что вычисление y могло зависеть от значения x, прочитанного P_2 (то есть значения, записанного P_1).

С другой стороны, если два процесса спонтанно и одновременно записывают два разных элемента данных, для них не существует причинно-следственной связи. Операции, которые не имеют причинно-следственной связи, называются **параллельными** (concurrent).

Чтобы хранилище данных считалось причинно согласованным, необходимо, чтобы хранилище выполняло следующее условие:

> *Записи, которые потенциально связаны причинно-следственной связью, должны просматриваться всеми процессами в одном и том же порядке. Одновременные записи могут быть просмотрены на разных компьютерах в другом порядке.*

В качестве примера причинной последовательности рассмотрим рис. 7.10. Здесь у нас есть последовательность событий, которая разрешена с причинно-согласованным хранилищем, но запрещена с последовательным согласованным хранилищем или строго согласованным хранилищем. Следует отметить, что записи $W_2(x)b$ и $W_1(x)c$ параллельны, и поэтому не требуется, чтобы все процессы видели их в том же порядке.

P_1:	W(x)a				W(x)c		
P_2:		R(x)a	W(x)b				
P_3:		R(x)a				R(x)c	R(x)b
P_4:		R(x)a				R(x)b	R(x)c

Рис. 7.10 ❖ Эта последовательность допускается с причинно-согласованным хранилищем, но не с последовательно-согласованным хранилищем

Теперь рассмотрим второй пример. На рис. 7.11a мы имеем $W_2(x)$ b, потенциально зависящий от $W_1(x)$a, потому что запись значения b в x может быть результатом вычисления, включающего ранее прочитанное значение с помощью $R_2(x)$a. Две записи причинно связаны, поэтому все процессы должны видеть их в одном и том же порядке. Следовательно, рис. 7.11a неверен. С другой стороны, на рис. 7.11b чтение было удалено, поэтому $W_1(x)$a и $W_2(x)$b являются теперь одновременными записями. Причинно-согласованное хранилище не требует, чтобы одновременные записи были глобально упорядочены, поэтому рис. 7.11b правильный. Обратите внимание, что рис. 7.11b отражает ситуацию, которая была бы неприемлемой для последовательно-согласованного хранилища.

P_1:	W(x)a				W(x)c
P_2:		R(x)a	W(x)b		
P_3:				R(x)b	R(x)a
P_4:				R(x)a	R(x)b

(a)

P_1:	W(x)a				W(x)c
P_2:			W(x)b		
P_3:				R(x)b	R(x)a
P_4:				R(x)a	R(x)b

(b)

Рис. 7.11 ❖ (a) Нарушение причинно-согласованного хранилища; b) правильная последовательность событий в причинно-следственном хранилище

Реализация причинно-следственной согласованности требует отслеживания того, какие процессы видели какие записи. Есть много тонких вопросов, которые необходимо принять во внимание. Для иллюстрации предположим, что мы заменим $W_2(x)$b на рис. 7.11a на $W_2(y)$b и аналогично $R_3(x)$b на $R_3(y)$b соответственно. Эта ситуация показана на рис. 7.12.

P_1:	W(x)a				W(x)c
P_2:		R(x)a	W(y)b		
P_3:				R(y)b	R(x)?
P_4:				R(x)a	R(y)?

Рис. 7.12 ❖ Небольшая модификация рис. 7.11a. Что должны вернуть R_3 (x) или R_4 (y)?

Давайте сначала посмотрим на операцию $R_3(x)$. Процесс P_3 выполняет эту операцию после $R_3(y)$b. В этот момент мы точно знаем, что W(x)a *произошло до* W(y)b.

В частности, W(x)a → R(x)a → W(y)b, что означает, что если мы хотим сохранить причинность, чтение x после чтения b из y может вернуть только a. Если система вернет NIL к P_3, это нарушит сохранение причинно-следственных связей.

А как насчет R_4(y)? Может ли он вернуть начальное значение y, а именно NIL? Ответ положительный: хотя у нас есть формальное отношение W(x)a → W(y)b, не прочитав b из y, процесс P_4 все еще может справедливо наблюдать, что W(x)a имел место независимо от инициализации y.

Внедрение, сохранение причинно-следственной связи ставит некоторые интересные вопросы. Рассмотрим, например, процесс промежуточного программного обеспечения P_3 из рис. 7.12. В тот момент, когда это промежуточное ПО возвращает значение b из чтения y, оно должно знать о соотношении W(x)a → W(y)b. Другими словами, когда последнее значение y было передано в промежуточное ПО P_3, по крайней мере, метаданные о зависимости y также должны были распространяться. В качестве альтернативы распространение также может быть выполнено вместе с обновлением x в узле P_3. В целом суть в том, что нам нужен график зависимости того, какая операция зависит от каких других операций. Такой граф может быть удален в тот момент, когда зависимые данные также локально сохранены.

Группирование операций

Многие модели согласованности определяются на уровне элементарных операций чтения и записи. Этот уровень детализации обусловлен историческими причинами: эти модели изначально были разработаны для многопроцессорных систем с общей памятью и фактически были реализованы на аппаратном уровне.

Большая детализация (гранулярность) этих моделей согласованности во многих случаях не соответствует детализации, обеспечиваемой приложениями. Мы видим, что параллелизм между программами обмена данными обычно находится под контролем посредством механизмов синхронизации для взаимного исключения и транзакций.

По сути, происходит то, что на программном уровне операции чтения и записи заключаются в скобки парой операций `ENTER_CS` и `LEAVE_CS`. Процесс, который успешно выполнил `ENTER_CS`, будет гарантировать, что все данные в его локальном хранилище обновлены. В этот момент он может безопасно выполнить серию операций чтения и записи в этом хранилище, а затем завершить работу, вызвав `LEAVE_CS`. Данные и инструкции между `ENTER_CS` и `LEAVE_CS` обозначаются как **критический раздел** (critical section).

По сути, происходит то, что в программе данные, которые обрабатываются серией операций чтения и записи, защищены от одновременного доступа, который может привести к появлению чего-то другого, кроме результата выполнения ряда в целом. Иными словами, скобки превращают серию операций чтения и записи в атомарно выполненную единицу, что повышает уровень детализации.

Чтобы достичь этой точки, нам нужно иметь точную семантику, касающуюся операций ENTER_CS и LEAVE_CS. Эта семантика может быть сформулирована в терминах общих **переменных синхронизации** (synchronization variables), или просто **блокировок** (blocks). Блокировка имеет общие элементы данных, связанные с ней, и каждый общий элемент данных связан максимум с одной блокировкой. В случае конечно-детальной синхронизации все общие элементы данных будут связаны только с одной блокировкой. Детальная синхронизация достигается, когда каждый элемент данных общего пользования имеет свою уникальную блокировку. Конечно, это всего лишь две крайности привязки общих данных к блокировке. Когда процесс входит в критический раздел, он должен получить соответствующие блокировки, а также и когда он покидает критический раздел, то снимает эти блокировки.

У каждой блокировки есть текущий владелец, а именно процесс, который последний раз его получил. Процесс, в настоящее время не владеющий блокировкой, но желающий ее получить, должен отправить текущему владельцу сообщение с просьбой о владении и текущих значениях данных, связанных с этой блокировкой. Имея *эксклюзивный доступ* (exclusive access) к блокировке, процессу разрешено выполнять операции чтения и записи. Кроме того, несколько процессов могут одновременно иметь неэксклюзивный доступ к блокировке, что означает, что они могут читать, но не записывать связанные данные. Конечно, неэксклюзивный доступ может быть предоставлен тогда и только тогда, когда нет другого процесса, имеющего эксклюзивный доступ.

Теперь мы требуем соблюдения следующих критериев [Bershad et al., 1993]:

- получение блокировки может быть успешным только после завершения всех обновлений связанных с ним общих данных;
- эксклюзивный доступ к блокировке может быть успешным, только если ни один другой процесс не имеет эксклюзивного или неэксклюзивного доступа к этой блокировке;
- неэксклюзивный доступ к блокировке разрешается только в том случае, если был завершен какой-либо предыдущий эксклюзивный доступ, включая обновления связанных данных блокировки.

Обратите внимание, что мы фактически требуем, чтобы использование блокировок было линеаризованным, придерживаясь последовательной согласованности. На рис. 7.13 показан пример того, что известно как **согласованность ввода** (entry consistency). Мы связываем блокировку с каждым элементом данных отдельно. Мы используем обозначение L(x) как сокращение для получения блокировки для x, то есть *блокирования* x. Аналогично U(x) означает снятие блокировки x, или *разблокирование* x. В этом случае P_1 блокирует x, меняет x один раз, после чего блокирует y. Процесс P_2 также получает блокировку для x, но не для y, так что он будет считывать значение a для x, но может считывать NIL для y. Однако, поскольку процесс P_3 сначала получает блокировку для y, он будет считывать значение b, когда y был разблокирован P_1. Здесь важно отметить, что у каждого процесса есть копия переменной, но эта копия не требует немедленного или автоматического обновления. Когда блокируется или разблокируется переменная, процесс явно сообщает базовой распределенной системе, что копии этой переменной

должны быть синхронизированы. Таким образом, простая операция чтения без блокировки может привести к считыванию локального значения, которое фактически устарело.

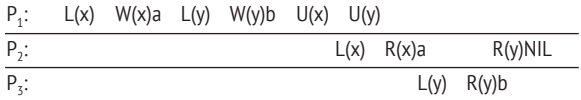


Рис. 7.13 ❖ Действительная последовательность событий для согласованности ввода

Одной из проблем программирования с согласованностью ввода является правильное связывание данных с блокировками. Один простой подход заключается в явном указании промежуточному программному обеспечению, к каким данным будет осуществляться доступ, как это обычно делается путем объявления, какие таблицы базы данных будут затронуты транзакцией. В объектно-ориентированном подходе мы могли бы связать уникальную блокировку с каждым объявлением объекта, эффективно устанавливая порядок обращения к таким объектам.

Согласованность и когерентность

На этом этапе полезно прояснить разницу между двумя тесно связанными понятиями. Все модели, которые мы обсуждали до сих пор, имеют дело с тем фактом, что ряд процессов выполняют операции чтения и записи для набора элементов данных.

Модель согласованности (consistency model) описывает, что можно ожидать от этого набора, когда несколько процессов одновременно работают с этими данными. Тогда говорят, что набор согласован, если он соответствует правилам, описанным моделью.

В то время как согласованность данных связана с набором элементов данных, **когерентные модели** (coherence models) описывают то, что можно ожидать только для одного элемента данных [Cantin et al., 2005]. В этом случае мы предполагаем, что элемент данных реплицируется; говорят, что он согласован, когда различные копии соответствуют правилам, определенным соответствующей моделью согласованности.

Популярной моделью остается последовательная согласованность, но теперь она применяется только к одному элементу данных. По сути, это означает, что в случае одновременной записи все процессы в конечном итоге будут иметь одинаковый порядок обновления.

Конечная согласованность

Степень, в которой процессы на самом деле работают одновременно, и степень необходимой гарантии согласованности могут варьироваться. Есть много примеров, в которых параллелизм появляется только в ограниченной форме. Например, во многих системах баз данных большинство процессов

практически не выполняют операций обновления; они в основном читают данные из базы данных. Только один или очень мало процессов выполняют операции обновления. Тогда возникает вопрос, как быстро обновления должны быть доступны только для процессов чтения. С появлением глобально работающих сетей доставки контента разработчики часто предпочитают распространять обновления медленно, неявно предполагая, что большинство клиентов всегда перенаправляются на одну и ту же реплику и поэтому никогда не будут испытывать несоответствия.

Другой пример – веб-сеть. Практически во всех случаях веб-страницы обновляются одним органом, таким как веб-мастер или фактический владелец страницы. Обычно конфликтов запись-запись для разрешения нет. С другой стороны, для повышения эффективности браузеры и веб-прокси часто конфигурируются так, чтобы хранить выбранную страницу в локальном кеше и возвращать эту страницу при следующем запросе.

Важным аспектом обоих типов веб-кешей является то, что они могут возвращать устаревшие веб-страницы. Другими словами, кешированная страница, которая возвращается запрашивающему клиенту, является более старой версией по сравнению с версией, доступной на реальном веб-сервере. Как выясняется, многие пользователи находят это несоответствие приемлемым (в определенной степени), если они имеют доступ только к одному и тому же кешу. По сути, они не знают о том, что обновление имело место, как и в предыдущем случае сетей доставки контента.

Еще один пример – всемирная система именования, такая как DNS. Пространство имен DNS разделено на домены, где каждый домен назначен органу именования, который действует как владелец этого домена. Только этому органу разрешено обновлять свою часть пространства имен. Следовательно, конфликты, возникающие в результате двух операций, которые обе хотят выполнить обновление для одних и тех же данных (то есть **конфликты запись-запись** (write-write conflicts)), никогда не возникают. Единственной ситуацией, которая должна быть обработана, являются **конфликты чтение-запись** (read-write conflict), в которых один процесс хочет обновить элемент данных, в то время как одновременно другой пытается прочитать этот элемент. Как оказалось, и в этом случае часто допустимо распространять обновление ленивым образом, что означает, что процесс чтения увидит обновление только через некоторое время, прошедшее с момента обновления.

Эти примеры можно рассматривать как случаи (крупномасштабных) распределенных и реплицированных баз данных, которые допускают относительно высокую степень несогласованности. Общим для них является то, что если в течение длительного времени не происходит никаких обновлений, все реплики постепенно станут согласованными, то есть будут храниться одни и те же данные. Эта форма согласованности называется **конечной согласованностью** (eventual consistency) [Vogels, 2009].

Хранилища данных, которые в конечном итоге становятся согласованными, обладают тем свойством, что при отсутствии конфликтов запись-запись все реплики будут сходиться к идентичным копиям друг друга. Последовательная согласованность, по сути, требует только того, чтобы обновления гарантированно распространялись на все реплики. Конфликты запись-запись

часто относительно легко разрешить, если предположить, что только небольшая группа процессов может выполнять обновления. На практике мы также часто видим, что в случае конфликтов одна конкретная операция записи (глобально) объявляется как «победитель», перезаписывая эффекты любой другой конфликтующей операции записи. Согласованность в конечном счете, следовательно, часто дешева в реализации.

Примечание 7.4 (дополнительно: усиление конечной согласованности)

Конечная согласованность является относительно простой для понимания моделью, но не менее важным является тот факт, что ее также относительно легко реализовать. Тем не менее эта модель слабой согласованности имеет свои особенности. Рассмотрим календарь, которым поделились Алиса, Боб и Чак. Собрание M имеет два атрибута: предполагаемое время начала и группа людей, которые подтвердили свое присутствие. Когда Алиса предлагает начать встречу M в момент времени T, предполагая, что еще никто не подтвердил посещаемость, она выполняет операцию $W_A(M)[T, \{A\}]$. Когда Боб подтвердит свое присутствие, он прочтет кортеж [T, {A}] и обновит M соответственно: $W_B(M)[T, \{A, B\}]$. В нашем примере необходимо запланировать две встречи M_1 и M_2.

Предположим следующую последовательность событий:

$$W_A(M_1)[T_1, \{A\}] \rightarrow R_B(M_1)[T_1, \{A\}] \rightarrow W_B(M_1)[T_1, \{A, B\}] \rightarrow W_B(M_2)[T_2, \{B\}].$$

Другими словами, Боб подтверждает свое присутствие на M_1, а затем сразу же предлагает запланировать M_2 на T_2. К сожалению, Чак одновременно предлагает запланировать M_1 в T_3, когда Боб подтверждает, что он может посетить M_1 в T_1. Формально, используя символ «||» для обозначения одновременных операций, мы имеем:

$$W_B(M_1)[T_1, \{A, B\}] \,||\, W_C(M_1)[T_3, \{C\}].$$

Используя наши обычные обозначения, эти операции можно проиллюстрировать, как показано на рис. 7.14.

A:	$W(M_1)[T_1, \{A\}]$				$R(M_1)?$
B:		$R(M_1)[T_1, \{A\}]$	$W(M_1)[T_1, \{A,B\}]$	$W(M_2)[T_2, \{B\}]$	$R(M_1)?$
C:			$W(M_1)[T_2, \{C\}]$		$R(M_1)?$

Рис. 7.14 ❖ Ситуация обновления двух встреч M_1 и M_2

Конечная согласованность может привести к очень разным сценариям. Существует ряд конфликтов запись-запись, но в любом случае в конечном итоге $[T_2, (B)]$ будет сохранено для встречи M_2 в результате связанной операции записи Бобом. Для встречи M_1 есть разные варианты. В принципе, у нас есть три возможных результата: $[T_1, \{A\}]$, $[T_1, \{A, B\}]$ и $[T_3, \{C\}]$. Предполагая, что мы можем поддерживать некоторое представление о глобальных часах, маловероятно, что $W_A(M_1)[T1, \{A\}]$ будет преобладать.

Однако две операции записи $W_B(M_1)[T_1, \{A, B\}]$ и $W_C(M_1)[T_1, \{C\}]$ действительно находятся в конфликте. На практике один из них победит, предположительно по решению центрального координатора.

Исследователи стремились объединить возможную последовательность с более строгими гарантиями по очередности. В [Bailis et al. 2013] предлагается использовать отдельный уровень, который работает поверх конечного последовательного,

распределенного хранилища. Этот уровень реализует причинно-следственную согласованность, относительно которой ранее было доказано, что это наилучшая достижимая согласованность при наличии разбиения сети [Mahajan et al., 2011]. В нашем примере у нас есть только одна цепочка зависимостей:

$W_A(M_1)[T_1, \{A\}] \rightarrow R_B(M_1)[T_1, \{A\}] \rightarrow W_B(M_1)[T_1, \{A, B\}] \rightarrow W_B(M_2)[T_2, \{B\}]$.

Важное наблюдение заключается в том, что при наличии причинно-следственной согласованности, когда процесс читает $[T_2, \{B\}]$ для встречи M_2, получение значения для M_1 возвращает либо $[T_1, \{A, B\}]$, либо $[T_3, \{C\}]$, но, конечно, не $[T_1, \{A\}]$. Причина в том, что $W(M_1)[T_1, \{A, B\}]$ непосредственно предшествует $W(M_2)[T_2, \{B\}]$ и, что еще хуже, могло быть перезаписано $W(M_1)[T_3, \{C\}]$. Причинная согласованность исключает возможность возврата системой $[T_1, \{A\}]$.

Однако конечная согласованность может перезаписать ранее сохраненные элементы данных. При этом зависимости могут быть потеряны. Чтобы прояснить этот момент, важно понимать, что на практике операция в лучшем случае отслеживает непосредственно предшествующую операцию, от которой она зависит. Как только $W_c(M_1)[T_3, \{C\}]$ перезаписывает $W_B(M_1)[T_1, \{A, B\}]$ (и распространяется на все реплики), мы также разрываем цепочку зависимостей.

$W_A(M_1)[T_1, \{A\}] \rightarrow R_B(M_1)[T_1, \{A\}] \rightarrow \ldots \rightarrow W_B(M_2)[T_2, \{B\}]$,

что обычно предотвращает $W_A(M_1)[T_1, \{A\}]$ когда-либо обойти $W_B(M_1)[T_1, \{A, B\}]$ и любые операции, зависящие от него. Как следствие поддержание причинной согласованности требует, чтобы мы сохраняли историю зависимостей, а не просто отслеживали непосредственно предшествующие операции.

7.3. Модели согласованности, ориентированные на клиента

Модели согласованности, ориентированные на данные, нацелены на обеспечение единообразного представления всей системы в хранилище данных. Важным предположением является то, что параллельные процессы могут одновременно обновлять хранилище данных и что необходимо обеспечить согласованность перед лицом такого параллелизма. Например, в случае согласованности записей на основе объектов хранилище данных гарантирует, что при вызове объекта вызывающему процессу предоставляется копия объекта, которая отражает все изменения в объекте, которые были сделаны до сих пор, возможно, другими процессами. Во время вызова также гарантируется, что никакой другой процесс не может вмешиваться, то есть взаимоисключающий доступ предоставляется вызывающему процессу.

Возможность обрабатывать параллельные операции над общими данными при сохранении строгой согласованности является фундаментальной задачей для распределенных систем. По соображениям производительности строгая согласованность может быть гарантирована только тогда, когда процессы используют такие механизмы, как транзакции или переменные синхронизации. По этой же причине может быть невозможно гарантировать сильную

согласованность и должны быть приняты более слабые формы, такие как причинная согласованность в сочетании с конечной согласованностью.

В этом разделе мы рассмотрим специальный класс распределенных хранилищ данных. Хранилища данных, которые мы рассматриваем, характеризуются отсутствием одновременных обновлений, или, когда такие обновления происходят, предполагается, что их можно относительно легко разрешить. Большинство операций включают чтение данных. Эти хранилища данных предлагают модель слабой согласованности, например конечную согласованность. Внедрив специальные ориентированные на клиента модели согласованности, можно, оказывается, относительно дешевым способом скрыть многие несоответствия.

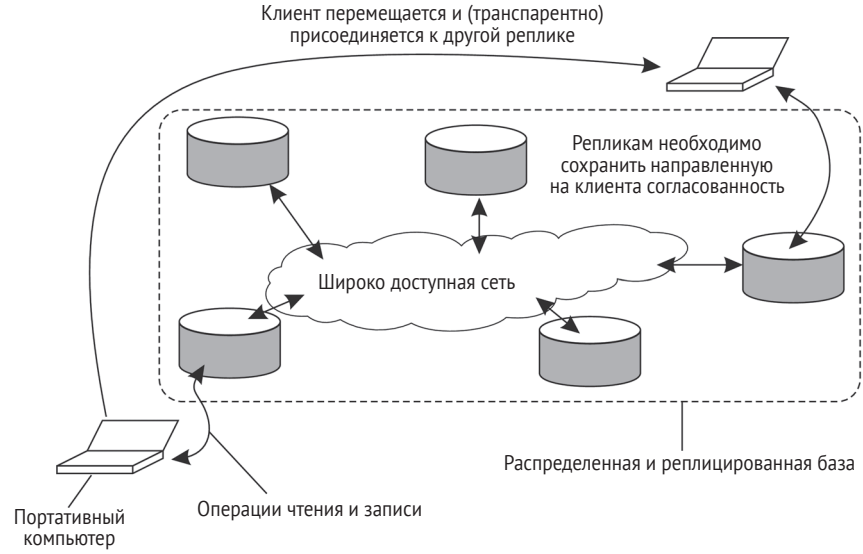


Рис. 7.15 ❖ Принцип доступа мобильного пользователя к различным репликам распределенной базы данных

В конечном итоге согласованные хранилища данных обычно работают нормально, если клиенты всегда имеют доступ к одной и той же реплике. Однако проблемы возникают при доступе к различным репликам в течение короткого периода времени. Это лучше всего проиллюстрировано на примере мобильного пользователя, обращающегося к распределенной базе данных, как показано на рис. 7.15.

Мобильный пользователь, скажем Алиса, получает доступ к базе данных, прозрачно подключаясь к одной из реплик. Другими словами, приложение, работающее на мобильном устройстве Алисы, не знает, на какой реплике оно фактически работает. Предположим, что Алиса выполняет несколько операций обновления, а затем снова отключается. Позже она снова обращается к базе данных, возможно, после перемещения в другое место или с помощью иного устройства доступа. В этот момент она может быть подключена к другой реплике, как показано на рис. 7.15. Однако если ранее выполненные

обновления еще не были распространены, Алиса заметит несогласованное поведение. В частности, она ожидала увидеть все ранее сделанные изменения, но вместо этого кажется, что ничего не изменилось.

Этот пример типичен для в конечном итоге согласованных хранилищ данных и вызван тем фактом, что пользователи могут иногда работать с разными репликами, пока обновления не были полностью распространены. Эту проблему можно решить, введя согласованность, ориентированную на клиента. По сути, согласованность, ориентированная на клиента, обеспечивает гарантии для одного клиента в отношении согласованности доступа к хранилищу данных этим клиентом. Нет никаких гарантий относительно одновременного доступа различных клиентов. Если Боб изменяет данные, которые передаются Алисе, но хранятся в другом месте, мы можем легко создать конфликты запись-запись. Более того, если ни Алиса, ни Боб не получат доступа к одному и тому же месту в течение некоторого времени, такие конфликты могут занять много времени, прежде чем они будут обнаружены.

Модели согласованности, ориентированные на клиента, берут свое начало из работы над системой Bayou и, в более общем плане, из систем мобильной передачи данных (см., например, [Terry et al., 1994], [Terry et al., 1998] или [Terry, 2008]). Bayou – это система баз данных, разработанная для мобильных компьютеров, где предполагается, что сетевое подключение ненадежно и подвержено различным проблемам с производительностью. Беспроводные сети и сети, охватывающие большие территории, такие как интернет, попадают в эту категорию.

Система Bayou, по сути, различает четыре разные модели согласованности. Чтобы объяснить эти модели, мы снова рассмотрим хранилище данных, которое физически распределено по нескольким машинам. Когда процесс получает доступ к хранилищу данных, он обычно подключается к локальной (или ближайшей) доступной копии, хотя, в принципе, любая копия будет отлично работать. Все операции чтения и записи выполняются для этой локальной копии. Обновления в конечном итоге распространяются на другие копии.

Ориентированные на клиента модели согласованности описаны с использованием следующих обозначений. Пусть x_i обозначает версию элемента данных x. Версия x_i является результатом серии операций записи, которые имели место после инициализации ее **набора записи** (write set) $WS(x_i)$. Добавляя операции записи к этому ряду, мы получаем другую версию x_j и говорим, что x_j следует из x_i. Мы используем обозначение $WS(x_i; x_j)$, чтобы указать, что x_j следует из x_i. Если мы не знаем, следует ли x_j из x_i, то используем обозначение $WS(x_i|x_j)$.

Монотонные чтения

Первая ориентированная на клиента модель согласованности – это модель монотонного чтения. Говорят, что (распределенное) хранилище данных обеспечивает **согласованность монотонного чтения** (monotonic-read consistency), если выполняется следующее условие:

Если процесс считывает значение элемента данных x, любая последующая операция чтения для x этим процессом всегда будет возвращать то же значение или более новое значение.

Другими словами, согласованность монотонного чтения гарантирует, что как только процесс увидит значение x, он никогда не увидит более старую версию x.

В качестве примера, где полезны монотонные чтения, рассмотрим распределенную базу данных электронной почты. В такой базе данных почтовый ящик каждого пользователя может быть распределен и реплицируется на нескольких машинах. Почту можно вставить в почтовый ящик в любом месте. Однако обновления распространяются ленивым (то есть по запросу) способом. Только когда копия требует определенных данных для согласованности, эти данные распространяются на эту копию. Предположим, что пользователь читает свою почту в Сан-Франциско и что только чтение почты не влияет на почтовый ящик, то есть сообщения не удаляются, не сохраняются в подкаталогах и даже не помечаются как уже прочитанные и т. д. Когда пользователь позже летит в Нью-Йорк и снова открывает свой почтовый ящик, согласованность монотонного чтения гарантирует, что сообщения, которые были в почтовом ящике в Сан-Франциско, также будут в почтовом ящике при его открытии в Нью-Йорке.

Используя обозначения, аналогичные обозначениям для моделей согласованности, ориентированных на данные, согласованность монотонного чтения можно графически представить, как показано на рис. 7.16. Вместо того чтобы показывать процессы вдоль вертикальной оси, мы теперь показываем *локальные хранилища данных*, в нашем примере L_1 и L_2. Операция записи или чтения индексируется процессом, который выполнил операцию, то есть $W_1(x)a$ обозначает, что процесс P_1 записал значение a в x. Поскольку нас не интересуют конкретные значения общих элементов данных, а скорее их версии, мы используем обозначение $W_1(x_2)$, чтобы указать, что процесс P_1 создает версию x_2, ничего не зная о других версиях. $W_2(x_1; x_2)$ указывает, что процесс P_2 отвечает за создание версии x_2, которая следует из x_1. Аналогично $W_2(x_1; x_2)$ обозначает, что процесс P_2 создает версию x_2 одновременно с версией x_1 (и, таким образом, потенциально вводит конфликт запись-запись). $R_1(x_2)$ просто означает, что P_1 читает версию x_2.

Рис. 7.16 ❖ Операции чтения, выполняемые одним процессом P в двух разных локальных копиях одного и того же хранилища данных: a) монотонное чтение последовательного хранилища данных; b) хранилище данных, которое не обеспечивает монотонное чтение

На рис. 7.16a процесс P_1 сначала выполняет операцию записи в x в L_1, создавая версию x_1, а затем читает эту версию. Но L_2-процесс P_2 сначала выдает версию x_2, следующую из x_1. Когда процесс P_1 переходит на L_2 и снова читает

x, он находит более свежее значение, но, по крайней мере, принявшее во внимание предыдущую запись.

Рисунок 7.16b показывает ситуацию, в которой нарушается согласованность монотонного чтения. После того как процесс P_1 прочитал x_1 в L_1, он позже выполняет операцию $R_1(x_2)$ в L_2. Однако предшествующая операция записи $W_2(x_1|x_2)$ процессом P_2 в L_2 создает версию, которая не следует из x_1. Как следствие операция чтения P_1 в L_2, как известно, не включает в себя влияние операций записи, когда она выполняла $R_1(x_1)$ в местоположении L_1.

Монотонные записи

Во многих ситуациях важно, чтобы операции записи распространялись в правильном порядке на все копии хранилища данных. Это свойство выражается в согласованности **монотонной записи** (monotonis-write consistence). В согласованном хранилище с монотонной записью выполняется следующее условие:

> *Операция записи процессом в элементе данных* x *завершается перед любой последовательной операцией записи в* x *тем же процессом.*

Более формально, если у нас есть две последовательные операции $W_k(x_i)$ и $W_k(x_j)$ процесса P_k, то независимо от того, где $W_k(x_j)$ имеет место, у нас также есть $WS(x_i; x_j)$. Таким образом, завершение операции записи означает, что копия, над которой выполняется последовательная операция, отражает эффект предыдущей операции записи тем же процессом, независимо от того, где эта операция была инициирована. Другими словами, операция записи для копии элемента x выполняется только в том случае, если эта копия была обновлена с помощью какой-либо предшествующей операции записи тем же процессом, который мог иметь место в других копиях x. В случае необходимости новая запись должна ждать завершения старых.

Обратите внимание, что согласованность монотонной записи напоминает согласованность данных FIFO с ориентацией на данные. Суть согласованности FIFO заключается в том, что операции записи одним и тем же процессом выполняются везде в правильном порядке. Это ограничение порядка также применимо к монотонной записи, за исключением того, что мы сейчас рассматриваем согласованность только для одного процесса, а не для совокупности параллельных процессов.

Обновление копии x в актуальном состоянии не требуется, когда каждая операция записи полностью перезаписывает текущее значение x. Однако операции записи часто выполняются только для части состояния элемента данных. Рассмотрим, например, библиотеку программного обеспечения. Во многих случаях обновление такой библиотеки выполняется путем замены одной или нескольких функций, что приводит к следующей версии. При согласованности монотонной записи даются гарантии, что если обновление выполняется для копии библиотеки, все предыдущие обновления будут выполняться в первую очередь. Полученная библиотека действительно станет самой последней версией и будет включать все обновления, которые привели к предыдущим версиям библиотеки.

Согласованность монотонной записи показана на рис. 7.17. На рис. 7.17a процесс P_1 выполняет операцию записи на x в L_1, представленную как операция $W_1(x_1)$. Позже P_1 выполняет еще одну операцию записи на x, но на этот раз на L_2, обозначенную как $W_1(x_2; x_3)$. Версия, созданная P_1 на L_2, следует из обновления процессом P_2, в свою очередь основанного на версии x_1. Последнее выражается операцией $W_2(x_1; x_2)$. Для обеспечения согласованности монотонной записи необходимо, чтобы предыдущая операция записи в L_1 уже была распространена в L_2 и, возможно, обновлена.

Напротив, на рис. 7.17b показана ситуация, в которой согласованность монотонной записи не гарантируется. По сравнению с рис. 7.17a не хватает распространения x_1 до L_2 до того, как будет произведена другая версия x, выраженная операцией $W_2(x_1|x_2)$.

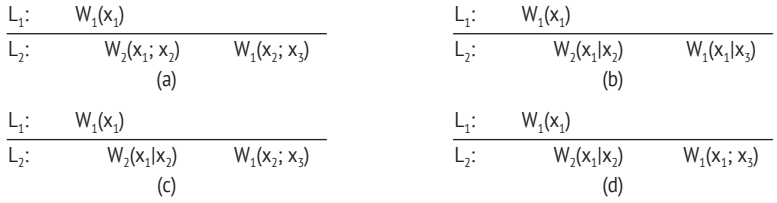


Рис. 7.17 ❖ Операции записи выполняются в двух разных локальных копиях одного и того же хранилища данных: a) хранилище данных с монотонной записью; b) хранилище данных, которое не обеспечивает согласованность монотонной записи; c) опять же, нет согласованности как $WS(x_1|x_2)$ и, следовательно, также $WS(x_1|x_3)$; d) последовательный как $WS(x_1; x_3)$, хотя x_1 явно перезаписал x_2

В этом случае процесс P_2 создает параллельную версию для x_1, после чего процесс P_1 просто создает версию x_3, но опять-таки одновременно для x_1. Ситуация, изображенная на рис. 7.17c, лишь немного более тонкая, но все еще нарушающая согласованность монотонной записи. Процесс P_1 теперь производит версию x_3, которая следует из x_2. Однако, поскольку x_2 не включает в себя операции записи, которые привели к x_1, то есть $WS(x_1|x_2)$, мы также имеем $WS(x_1|x_3)$.

Интересный случай показан на рис. 7.17d. Операция $W_2(x_1|x_2)$ создает версию x_2 одновременно с x_1. Тем не менее более поздний процесс P_1 производит версию x_3, но, по-видимому, основанную на том факте, что версия x_1 стала доступной в L_2. Как и когда x_1 был перенесен в L_2, остается неопределенным, но в любом случае конфликт запись-запись был создан с версией x_2 и разрешен в пользу x_1. Как следствие ситуация, показанная на рис. 7.17d, следует правилам согласованности монотонной записи. Обратите внимание, однако, что любая последующая запись процессом P_2 в L_2 (без чтения версии x_1) немедленно снова нарушит согласованность. Как такое нарушение можно предотвратить, оставлено читателю в качестве упражнения.

Обратите внимание, что благодаря определению согласованности монотонной записи операции записи с помощью одного и того же процесса выполняются в том же порядке, в котором они были инициированы. Несколько более слабой формой монотонной записи является та, в которой эффекты

операции записи видны только в том случае, если все предыдущие записи также были выполнены, но, возможно, не в том порядке, в котором они были первоначально инициированы.

Эта согласованность применима в тех случаях, когда операции записи являются коммутативными, поэтому реально в этом порядке нет необходимости. Детали можно найти в [Terry et al., 1994].

Чтение собственных записей

Говорят, что хранилище данных обеспечивает согласованность чтения собственных записей, если выполняется следующее условие:

> *Влияние операции записи процесса на элемент данных* x *всегда будет наблюдаться последовательной операцией чтения на* x *тем же процессом.*

Другими словами, операция записи всегда завершается перед последовательной операцией чтения одним и тем же процессом, независимо от того, где происходит эта операция чтения.

Отсутствие согласованности чтения собственной записи иногда возникает при обновлении веб-документов и последующем просмотре эффектов. Операции обновления часто выполняются с помощью стандартного редактора или текстового процессора, возможно, встроенного как часть системы управления контентом, который затем сохраняет новую версию в файловой системе, совместно используемой веб-сервером. Веб-браузер пользователя обращается к тому же файлу, возможно, после запроса его с локального веб-сервера. Однако после извлечения файла либо сервер, либо браузер часто кешируют локальную копию для последующих обращений. Следовательно, когда веб-страница обновляется, пользователь не увидит эффектов, если браузер или сервер вернет кешированную копию вместо исходного файла. Согласованность чтения собственной записи может гарантировать, что если редактор и браузер интегрированы в одну программу, кеш-память становится недействительной при обновлении страницы, поэтому обновленный файл выбирается и отображается.

Подобные эффекты возникают при обновлении паролей. Например, чтобы войти в цифровую библиотеку в интернете, часто необходимо иметь учетную запись с сопроводительным паролем. Однако для вступления в силу изменения пароля может потребоваться некоторое время, в результате чего библиотека может быть недоступна для пользователя в течение нескольких минут. Задержка может быть вызвана тем, что для управления паролями используется отдельный сервер, и может потребоваться некоторое время для последующей передачи (шифрования) паролей на различные серверы, которые составляют библиотеку. На рис. 7.18а показано хранилище данных, обеспечивающее согласованность операций чтения и записи. Обратите внимание, что рис. 7.18а очень похож на рис. 7.16а, за исключением того, что согласованность теперь определяется последней операцией записи процессом P_1 вместо его последнего чтения.

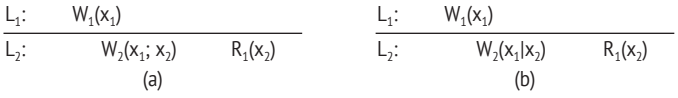


Рис. 7.18 ❖ Хранилище данных:
a) обеспечивающее согласованность операций чтения и записи;
b) хранилище данных, которое этого не делает

На рис. 7.18a процесс P_1 выполнил операцию записи $W_1(x_1)$, а затем операцию чтения в другой локальной копии. Последовательность чтения собственной записи гарантирует, что последующая операция чтения позволяет увидеть последствия операции записи. Это выражается $W_2(x_1; x_2)$, который утверждает, что процесс P_2 создал новую версию x, но основанную на x_1. Напротив, на рис. 7.18b процесс P_2 создает версию одновременно с x_1, выраженную как $W_2(x_1|x_2)$. Это означает, что эффекты предыдущей операции записи процессом P_1 не были распространены на L_2 во время исполнения x_2. Когда P_1 читает x_2, он не увидит эффектов своей собственной операции записи в L_1.

Запись следует за чтением

Последняя ориентированная на клиента модель согласованности – это модель, в которой обновления распространяются в результате предыдущих операций чтения. Говорят, что хранилище данных обеспечивает согласованность операций записи-следования-чтения (writes-follow-reads), если выполняется следующее.

Операция записи, выполняемая процессом в элементе данных x после предыдущей операции чтения в x тем же процессом, гарантированно будет выполняться для того же или более позднего значения x, которое было прочитано.

Другими словами, любая последовательная операция записи, выполняемая процессом в элементе данных x, будет выполняться для копии x, которая соответствует значению, наиболее недавно прочитанному этим процессом. Последовательность «записывает, следует, читает» можно использовать, чтобы гарантировать, что пользователи сетевой группы новостей увидят публикацию реакции на статью только после того, как увидят оригинальную статью [Terry et al., 1994]. Чтобы понять проблему, предположим, что пользователь сначала читает статью A. Затем он реагирует, публикуя ответ B. Требуя согласованности записи-следования-чтения, B будет записываться в любую копию группы новостей только после того, как A также было записано. Обратите внимание, что пользователи, которые только читают статьи, не должны требовать какой-либо конкретной ориентированной на клиента модели согласованности. Согласованность записи-следования-чтения гарантирует, что реакции на статьи хранятся в локальной копии, только если там также хранится оригинал.

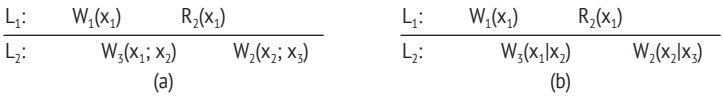


Рис. 7.19 ❖ a) Непротиворечивое хранилище данных записи-следования-чтения; b) хранилище данных, которое не обеспечивает согласованность операций запись-следование-чтение

Эта модель согласованности показана на рис. 7.19. На рис. 7.19a процесс P_2 читает версию x_1 в локальной копии L_1. Эта версия x была ранее произведена в L_1 процессом P_1 посредством операции $W_1(x_1)$, впоследствии распространена на L_2 и использована другим процессом P_3 для создания новой версии x_2, выраженной как $W_3(x_1; x_2)$. Когда процесс P_2 позже обновляет свою версию x после перехода к L_2, известно, что он будет работать с версией, следующей из x_1, выраженной как $W_2(x_2; x_3)$. Поскольку у нас также есть $W_3(x_1; x_2)$, мы знаем $WS(x_1; x_3)$.

Ситуация, показанная на рис. 7.19b, отличается. Процесс P_3 создает версию x_2 одновременно с версией x_1. Как следствие, когда P_2 обновляет x после прочтения x_1, будет обновлена версия, которую он ранее не читал. Последовательность «запись-следование-чтение» при этом нарушается.

7.4. Управление репликами

Ключевой вопрос для любой распределенной системы, которая поддерживает репликацию, состоит в том, чтобы решить, где, когда и кем реплика должна размещаться, а затем какие механизмы использовать для обеспечения согласованности реплик. Сама проблема размещения должна быть разбита на две подзадачи: задачу размещения серверов реплики и задачу размещения контента. Разница тонкая, и эти две задачи часто нечетко разделены. Размещение сервера-реплики связано с поиском лучших мест для размещения сервера, на котором можно разместить (частично) хранилище данных. Размещение контента связано с поиском лучших серверов для размещения контента. Обратите внимание, что это часто означает, что мы ищем оптимальное размещение только одного элемента данных. Очевидно, что перед размещением контента серверы реплик должны быть размещены в первую очередь.

Поиск лучшего местоположения сервера

Там, где более десяти лет назад можно было беспокоиться о том, где разместить отдельный сервер, с появлением множества крупномасштабных центров обработки данных, расположенных в интернете, ситуация значительно изменилась. Кроме того, связь продолжает улучшаться, делая точное расположение серверов менее критичным.

Примечание 7.5 (дополнительно: размещение сервера реплики)

Размещение сервера реплики не является интенсивно изучаемой проблемой по той простой причине, что зачастую это скорее проблема управления и коммерции, чем проблема оптимизации. Тем не менее анализ свойств клиента и сети полезен для принятия обоснованных решений.

Существуют различные способы вычисления наилучшего размещения сервера реплики, но все сводятся к задаче оптимизации, при которой необходимо выбрать лучшие K местоположений из N ($K < N$). Эти проблемы, как известно, сложны в вычислительном отношении и могут быть решены только с помощью эвристики. В работе [Qiu et al., 2001] в качестве отправной точки принято расстояние между клиентами и местоположениями. Расстояние может быть измерено с точки зрения задержки или пропускной способности. Их решение выбирает один сервер за раз так, чтобы среднее расстояние между этим сервером и его клиентами было минимальным, учитывая, что уже было размещено k серверов (это означает, что осталось $N - k$ расположений).

В качестве альтернативы в работе [Radoslavov et al., 2001] предлагается игнорировать положение клиентов и использовать только топологию интернета, сформированную автономными системами. **Автономную систему** (autonomous system, AS) лучше всего рассматривать как сеть, в которой все узлы работают по одному и тому же протоколу маршрутизации и которой управляет одна организация. По состоянию на 2015 год насчитывалось около 30 000 AS [Radoslavov et al., 2001]. Сначала рассмотрите самую большую AS и поместите сервер на маршрутизаторе с наибольшим числом сетевых интерфейсов (т. е. ссылок). Этот алгоритм затем повторяется со вторым по величине AS и т. д.

Как выясняется, размещение сервера, не зависящее от клиента, дает результаты, аналогичные расположению с учетом клиента, при условии что клиенты распределены по интернету равномерно (относительно существующей топологии). Насколько это предположение верно, не ясно. Это не было хорошо изучено.

Одна из проблем с этими алгоритмами состоит в том, что они вычислительно до́роги. Например, оба предыдущих алгоритма имеют сложность, которая выше, чем $O(N^2)$, где N – количество мест для проверки. На практике это означает, что даже для нескольких тысяч местоположений вычисление может потребоваться в течение десятков минут. Это может быть неприемлемо.

В работе [Szymaniak et al., 2006] описывается метод, с помощью которого можно быстро определить область для размещения реплик. Идентифицируется регион как набор узлов, обращающихся к тому же контенту, но для которых задержка междоузлия низкая. Цель алгоритма – сначала выбрать наиболее требовательные области, то есть область с наибольшим количеством узлов, а затем позволить одному из узлов в такой области действовать в качестве сервера реплики.

Для этого предполагается, что узлы располагаются в m-мерном геометрическом пространстве, как мы обсуждали в предыдущей главе. Основная идея состоит в том, чтобы определить K крупнейших кластеров и назначить узел из каждого кластера для размещения реплицируемого контента. Чтобы идентифицировать эти кластеры, все пространство разделено на ячейки. K самых плотных ячеек затем выбираются для размещения сервера реплики. Клетка – это не что иное, как мерный гиперкуб. Для двумерного пространства это соответствует прямоугольнику.

Очевидно, что размер ячейки важен, как показано на рис. 7.20. Если ячейки выбраны слишком большими, то несколько кластеров узлов могут содержаться в одной ячейке. В этом случае будет выбрано слишком мало серверов реплики для этих кластеров. С другой стороны, выбор небольших ячеек может привести к тому, что

один кластер будет распределен по нескольким ячейкам, что приведет к выбору слишком большого количества серверов реплики.

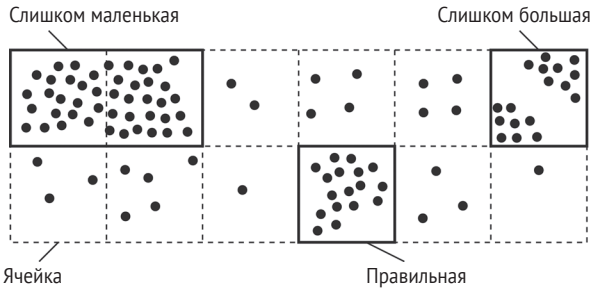


Рис. 7.20 ❖ Выбор правильного размера ячейки для размещения на сервере

Как выясняется, подходящий размер ячейки может быть вычислен как простая функция среднего расстояния между двумя узлами и количества требуемых реплик. С этим размером ячейки можно показать, что алгоритм работает так же, как и алгоритм, близкий к оптимальному, описанный в [Qiu et al., 2001], но с гораздо меньшей сложностью: $O(N \times \max\{\log(N), K\})$. Чтобы понять, что означает этот результат: эксперименты показывают, что вычисление 20 лучших размещений реплик для 64 000 узлов происходит примерно в 50 000 раз быстрее. Как следствие размещение сервера реплики теперь может выполняться в режиме реального времени.

Репликация и размещение контента

Когда речь идет о репликации и размещении контента, можно различить логически организованные три различных типа реплик, как показано на рис. 7.21.

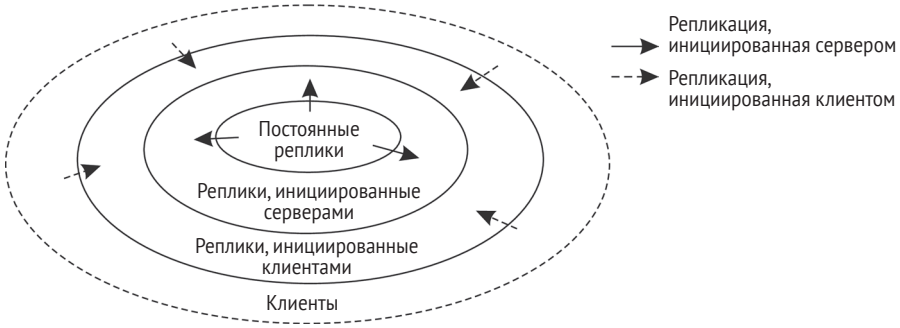


Рис. 7.21 ❖ Логическая организация различных видов копий хранилища данных в три концентрических кольца

Постоянные реплики

Постоянные реплики можно рассматривать как начальный набор реплик, составляющих распределенное хранилище данных. Во многих случаях количество постоянных реплик невелико. Рассмотрим, например, веб-сайт. Распространение веб-сайта обычно происходит в одной из двух форм. Первый тип распространения – это тот, в котором файлы, составляющие сайт, реплицируются на ограниченное количество серверов в одном месте. Всякий раз, когда приходит запрос, он пересылается на один из серверов, например с использованием стратегии циклического перебора.

Вторая форма распределенных веб-сайтов – это то, что называется **зеркалами** (mirroring). В этом случае веб-сайт копируется на ограниченное количество серверов, называемых **зеркальными сайтами** (mirror sites), которые географически распределены по интернету. В большинстве случаев клиенты просто выбирают один из различных зеркальных сайтов из предложенного им списка. Зеркальные веб-сайты имеют то же содержание, что и веб-сайты на основе кластеров, и лишь несколько реплик, которые более или менее статически сконфигурированы.

Подобные статические организации также существуют с распределенными базами данных [Kemme et al., 2010; Özsu and Valduriez, 2011]. Опять же, база данных может быть распределена и реплицирована по ряду серверов, которые вместе образуют кластер серверов, часто называемый архитектурой без общего доступа, подчеркивая, что ни диски, ни основная память не используются процессорами совместно. В качестве альтернативы – база данных распределена и, возможно, реплицирована по ряду географически распределенных сайтов. Эта архитектура обычно используется в федеративных базах данных [Sheth and Larson, 1990].

Реплики, инициированные сервером

В отличие от постоянных реплик, инициируемые сервером реплики являются копиями хранилища данных, которые существуют для повышения производительности и создаются по инициативе (владельца) хранилища данных. Рассмотрим, например, веб-сервер, расположенный в Нью-Йорке. Обычно этот сервер может обрабатывать входящие запросы довольно легко, но может случиться так, что в течение пары дней внезапный поток запросов поступит из неожиданного места, далеко от сервера. В этом случае может оказаться целесообразным установить несколько временных реплик в регионах, откуда поступают запросы.

Примечание 7.6 (дополнительно:
пример динамического размещения веб-контента)

Проблема динамического размещения реплик уже давно решается в службах веб-хостинга. Эти службы предлагают часто относительно статичную коллекцию серверов, распределенную по интернету, что может поддерживать и предоставлять доступ к веб-файлам, принадлежащим третьим сторонам. Чтобы обеспечить оптимальные возможности, такие хостинговые службы могут динамически реплициро-

вать файлы на серверы, где эти файлы необходимы для повышения производительности, то есть близко к требовательным (группам) клиентов.

Учитывая, что серверы реплики уже установлены, решить, где разместить контент, не так уж сложно. Ранний случай динамической репликации файлов в случае службы веб-хостинга описан в [Rabinovich et al., 1999]. Алгоритм предназначен для поддержки веб-страниц, поэтому он предполагает, что обновления относительно редки по сравнению с запросами на чтение. Используя файлы в качестве единицы данных, алгоритм работает следующим образом.

Этот алгоритм динамической репликации учитывает две проблемы. Во-первых, репликация может иметь место, чтобы уменьшить нагрузку на сервер. Во-вторых, файлы на сервере могут быть перенесены или скопированы на серверы, расположенные рядом с клиентами, которые выдают много запросов на эти файлы. Далее мы сосредоточимся только на этом втором случае. Мы также опускаем ряд деталей, которые можно найти в [Rabinovich et al., 1999].

Каждый сервер отслеживает количество обращений к файлу и откуда поступают запросы доступа. В частности, когда клиент C входит в сервис, он делает это через близкий к нему сервер. Если клиент C_1 и клиент C_2 совместно используют один и тот же ближайший сервер P, все запросы доступа к файлу F на сервере Q от C_1 и C_2 совместно регистрируются в Q как единый счетчик доступа $cnt_Q(P, F)$. Эта ситуация показана на рис. 7.22.

Когда количество запросов для конкретного файла F на сервере S падает ниже порога удаления del(S, F), этот файл может быть удален из S. Как следствие количество реплик этого файла уменьшается, что может привести к более высоким нагрузкам на других серверах. Особые меры принимаются для обеспечения того, чтобы по крайней мере одна копия каждого файла продолжала существовать.

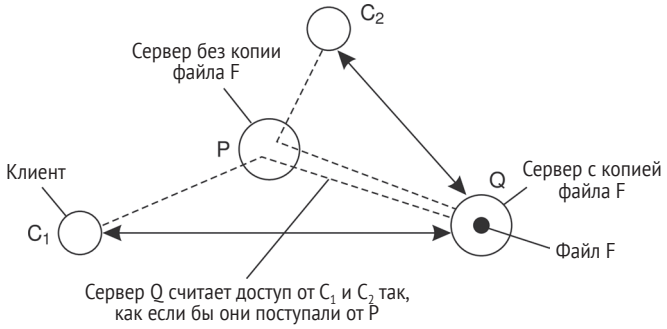


Рис. 7.22 ❖ Подсчет запросов на доступ от разных клиентов

Порог репликации rep(S, F), который всегда выбирается выше порога удаления, указывает на то, что количество запросов для конкретного файла настолько велико, что, может быть, стоит реплицировать его на другой сервер. Если число запросов лежит где-то между порогом удаления и репликации, файл разрешается только мигрировать. Другими словами, в этом случае важно, как минимум, сохранить количество реплик для этого файла одинаковым.

Когда сервер Q решает изменить оценку размещения файлов, которые он хранит, то проверяет счетчик доступа для каждого файла. Если общее количество запросов доступа для файла F в Q падает ниже порога удаления del(Q, F), он удалит F, если это не последняя копия. Кроме того, если для некоторого сервера P счетчик $cnt_Q(P, F)$ превышает более половины от общего числа запросов F в Q, серверу P предлагается принять копию F. Другими словами, сервер Q попытается перенести F в P.

Миграция файла F на сервер P не всегда может быть успешной, например из-за того, что P уже сильно загружен или места на диске не хватает. В этом случае Q попытается реплицировать F на других серверах. Разумеется, репликация может иметь место только в том случае, если общее количество запросов доступа для F в Q превышает порог репликации rep(Q, F). Сервер Q проверяет все остальные серверы в веб-хостинге, начиная с самого дальнего. Если для некоторого сервера R $cnt_Q(R,F)$ превышает определенную долю всех запросов для F в Q, делается попытка реплицировать F в R.

Обратите внимание, что если можно дать гарантии, что каждый элемент данных размещен хотя бы на одном сервере, может быть достаточно использовать только репликацию, инициированную сервером, и не иметь постоянных реплик. Тем не менее постоянные реплики часто полезны в качестве средства резервного копирования или для использования в качестве единственных реплик, которые могут быть изменены для обеспечения согласованности. Инициированные сервером реплики затем используются для размещения копий только для чтения рядом с клиентами.

Реплики, инициированные клиентом

Важным видом реплики является тот, который инициирован клиентом. Инициированные клиентом реплики более известны как (клиентские) кеши. По сути, кеш – это локальное хранилище, которое используется клиентом для временного хранения копии только что запрошенных данных. В принципе, управление кешем полностью остается за клиентом. Хранилище данных, из которого были получены данные, не имеет никакого отношения к сохранению кешированных данных согласованными. Однако есть много случаев, когда клиент может рассчитывать на участие хранилища данных, чтобы сообщить ему, когда кешированные данные стали устаревшими.

Кеши клиентов используются только для улучшения времени доступа к данным. Обычно, когда клиент хочет получить доступ к некоторым данным, он подключается к ближайшей копии хранилища данных, откуда выбирает данные, которые хочет прочитать, или туда, где он хранит данные, которые он только что изменил. Когда большинство операций связаны только с чтением данных, производительность можно улучшить, позволяя клиенту сохранять запрошенные данные в ближайшем кеше. Такой кеш может быть расположен на компьютере клиента или на отдельной машине в той же локальной сети, что и клиент. В следующий раз, когда те же данные должны быть прочитаны, клиент может просто извлечь их из этого локального кеша. Эта схема работает до тех пор, пока извлеченные данные не были изменены в это время.

Данные, как правило, хранятся в кеше в течение ограниченного периода времени, например чтобы предотвратить использование крайне устаревших данных или просто освободить место для других данных. Всякий раз, когда запрашиваемые данные могут быть получены из локального кеша, считается, что произошло попадание в кеш. Чтобы увеличить количество обращений к кешу, кеши могут быть разделены между клиентами. Основное предпо-

ложение состоит в том, что запрос данных от клиента C_1 также может быть полезен для запроса от другого соседнего клиента C_2.

Правильно ли это предположение, во многом зависит от типа хранилища данных. Например, в традиционных файловых системах файлы данных вообще редко используются совместно (см., например, [Muntz and Honeyman, 1992] и [Blaze, 1993]), что делает общий кеш бесполезным. Оказывается, что использование веб-кешей для обмена данными теряет свои позиции, в том числе из-за улучшения производительности сети и серверов. Вместо этого инициируемые сервером схемы репликации становятся более эффективными.

Размещение клиентских кешей относительно просто: кеш обычно располагается на той же машине, что и его клиент, или иным образом на компьютере, совместно используемом клиентами в той же локальной сети. Однако в некоторых случаях дополнительные уровни кеширования вводятся системными администраторами путем размещения общего кеша между несколькими отделами или организациями или даже размещения общего кеша для всего региона, такого как провинция или страна.

Еще один подход заключается в размещении (кешировании) серверов в определенных точках глобальной сети и предоставлении клиенту возможности найти ближайший сервер. Когда сервер находится, его можно запросить для хранения копий данных, которые клиент ранее извлекал откуда-то еще [Noble et al., 1999].

Распространение контента

Управление репликами также касается распространения (обновленного) контента на соответствующие серверы реплик. Есть различные компромиссы выполнения этого.

Состояние против операции

Важный вопрос дизайна касается того, что на самом деле должно распространяться. По сути, есть три возможности:

1) распространять только уведомление об обновлении;
2) переносить данные из одной копии в другую;
3) распространять операцию обновления на другие копии.

Распространение уведомлений – это то, что делают **протоколы аннулирования** (invalidation protocols). В протоколе о признании недействительными другие копии информируются о том, что произошло обновление и что содержащиеся в них данные больше не действительны. Аннулирование может указывать, какая часть хранилища данных была обновлена, так что фактически только часть копии становится недействительной. Важным вопросом является то, что распространяется не более чем уведомление. Всякий раз, когда запрашивается операция с недействительной копией, эта копия обычно должна обновляться в первую очередь, в зависимости от конкретной модели согласованности, которая должна поддерживаться.

Основным преимуществом протоколов аннулирования является то, что им необходима небольшая пропускная способность сети. Единственная информация, которую нужно передать, – это указание, какие данные больше не действительны. Такие протоколы обычно работают лучше всего, когда имеется много операций обновления по сравнению с операциями чтения, то есть отношение чтение-запись относительно мало.

Рассмотрим, например, хранилище данных, в котором обновления распространяются путем отправки измененных данных во все реплики. Если размер измененных данных велик, а обновления происходят часто по сравнению с операциями чтения, мы можем столкнуться с ситуацией, когда два обновления происходят одно за другим без какой-либо операции чтения между ними. Следовательно, распространение первого обновления на все реплики фактически бесполезно, так как оно будет перезаписано вторым обновлением. Вместо этого отправка уведомления о том, что данные были изменены, была бы более эффективной.

Перенос модифицированных данных между репликами является второй альтернативой и полезен, когда отношение чтение-запись относительно велико. В этом случае вероятность того, что обновление будет эффективным в том смысле, что измененные данные будут считаны до следующего обновления, высока. Вместо распространения измененных данных также возможно регистрировать изменения и передавать только эти изменения для экономии полосы пропускания. Кроме того, передачи часто объединяются в том смысле, что несколько модификаций упаковываются в одно сообщение, тем самым экономя затраты на связь.

Третий подход заключается не в передаче каких-либо модификаций данных, а в том, чтобы сообщать каждой реплике, какую операцию обновления она должна выполнить (и отправлять только те значения параметров, которые необходимы этим операциям). Этот подход, также называемый **активной репликацией** (active replication), предполагает, что каждая реплика представлена процессом, способным «активно» обновлять связанные с ней данные путем выполнения операций [Schneider, 1990]. Основное преимущество активной репликации заключается в том, что обновления часто могут распространяться с минимальными затратами на пропускную способность, при условии что размер параметров, связанных с операцией, является относительно небольшим. Кроме того, операции могут быть произвольной сложности, что может позволить дальнейшие улучшения в поддержании согласованности реплик. С другой стороны, каждой реплике может потребоваться больше вычислительной мощности, особенно в тех случаях, когда операции являются относительно сложными.

Протоколы извлечения и проталкивания

Другая проблема дизайна заключается в том, извлекать или проталкивать обновления. В подходе **на основе проталкивания** (push-based approach), также называемом **серверные протоколы** (server-based protocols), обновления распространяются на другие реплики, даже если эти реплики не запрашивают обновления. Этот подход часто используется между постоянными

и инициируемыми сервером репликами, но может использоваться и для принудительного обновления клиентских кешей. Серверные протоколы обычно применяются, когда требуется строгая согласованность.

Эта потребность в строгой согласованности связана с тем, что постоянные и инициируемые сервером реплики, а также большие общие кеши часто используются многими клиентами, которые, в свою очередь, в основном выполняют операции чтения. Следовательно, отношение чтение-обновление в каждой реплике является относительно высоким. В этих случаях протоколы на основе проталкивания эффективны в том смысле, что каждое ожидаемое обновление может быть полезным как минимум для одного, но, возможно, и большего количества читателей. Кроме того, протоколы, основанные на проталкивании, делают непротиворечивые данные немедленно доступными по запросу.

В подходе на **основе извлечения** (pull-based approach), напротив, сервер или клиент запрашивает другой сервер отправить ему любые обновления, которые тот имеет на данный момент. Протоколы на основе извлечения, также называемые **клиентские протоколы** (client-based protocols), часто используются кешами клиентов. Например, общая стратегия, применяемая к веб-кешам, в первую очередь проверяет актуальность кешированных элементов данных. Когда кеш получает запрос на элементы, которые по-прежнему доступны локально, он проверяет с помощью исходного веб-сервера, были ли эти элементы данных изменены с момента их кеширования. В случае модификации модифицированные данные сначала передаются в кеш, а затем возвращаются запрашивающему клиенту. Если никаких модификаций не было, кешированные данные возвращаются. Другими словами, клиент опрашивает сервер, чтобы узнать, нужно ли обновление.

Подход, основанный на извлечении, эффективен, когда отношение чтения к обновлению относительно низкое. Это часто имеет место с (неразделенными) клиентскими кешами, у которых есть только один клиент. Тем не менее даже когда кеш совместно используется многими клиентами, подход, основанный на извлечении, также может оказаться эффективным, когда кешированные элементы данных редко используются совместно. Основным недостатком стратегии на основе извлечения по сравнению с подходом на основе выталкивания является то, что время отклика увеличивается в случае пропуска кеша.

При сравнении решений, основанных на извлечении и проталкивании, необходимо учитывать необходимость ряда компромиссов, как показано на рис. 7.23. Для простоты рассмотрим клиент-серверную систему, состоящую из одного нераспределенного сервера и нескольких клиентских процессов, каждый из которых имеет свой собственный кеш.

Важной проблемой является то, что в протоколах проталкивания сервер должен отслеживать все клиентские кеши. Помимо того что серверы с состоянием часто менее отказоустойчивы, отслеживание всех клиентских кешей может привести к значительным затратам на сервере. Например, при использовании подхода проталкивания веб-серверу может понадобиться отслеживать десятки тысяч клиентских кешей.

Источник	Проталкивание	Извлечение
Состояние на сервере	Список клиентских реплик и кешей	Нет
Сообщения отправлены	Обновление (и, возможно, получение обновления позже)	Опрос и обновление
Время ответа клиента	Немедленное (или время выборки-обновления)	Время обновления-извлечения

Рис. 7.23 ❖ Сравнение протоколов на основе проталкивания и на основе извлечения в случае систем с несколькими клиентами и одним сервером

Каждый раз, когда веб-страница обновляется, серверу нужно будет просмотреть свой список клиентских кешей, содержащих копию этой страницы, и затем распространить обновление. Более того, если клиент очищает страницу из-за недостатка места, он должен сообщить об этом серверу, что приводит к еще большему объему обмена данными.

Сообщения, которыми обмениваются клиент и сервер, также различаются. В подходе, основанном на проталкивании, единственное сообщение состоит в том, что сервер отправляет обновления каждому клиенту. Когда обновления фактически являются только аннулированием, клиенту требуется дополнительная связь для извлечения измененных данных. В подходе, основанном на извлечении, клиент должен будет опросить сервер и, если необходимо, получить модифицированные данные.

Наконец, время отклика клиента также отличается. Когда сервер проталкивает модифицированные данные в клиентские кеши, время отклика на стороне клиента равно нулю. Когда проталкиваются аннулирования, время ответа такое же, как и в подходе на основе извлечения, и определяется временем, которое требуется для извлечения измененных данных с сервера.

Эти компромиссы привели к гибридной форме распространения обновлений на основе аренды. В случае управления репликами аренда – это обещание сервера, что оно будет отправлять обновления клиенту в течение определенного времени. Когда срок аренды истекает, клиент вынужден опрашивать сервер на наличие обновлений и извлекать измененные данные, если это необходимо. Альтернативой является то, что клиент запрашивает новую аренду для отправки обновлений, когда истекает предыдущая аренда.

Аренда, первоначально введенная в работе [Gray and Cheriton, 1989], обеспечивает удобный механизм для динамического переключения между стратегиями, основанными на проталкивании и извлечении. Рассмотрим следующую систему аренды, которая позволяет динамически адаптировать время истечения в зависимости от различных критериев аренды, описанных в [Duvvuri et al., 2003]. Мы различаем следующие три вида аренды. (Обратите внимание, что во всех случаях обновления отправляются сервером до тех пор, пока срок аренды не истек.)

Во-первых, для элементов данных в зависимости от того, когда последний раз был изменен элемент, выдаются аренды по возрасту. Основное предположение заключается в том, что данные, которые не были изменены в течение длительного времени, могут оставаться неизменными в течение еще некоторого времени. Это предположение оказалось разумным в случае, например,

веб-данных и регулярных файлов. Предоставляя долгосрочные договоры аренды для элементов данных, которые, как ожидается, останутся неизменными, количество сообщений об обновлении может быть значительно уменьшено по сравнению со случаем, когда все договоры аренды имеют одинаковое время истечения.

Другим критерием аренды является то, как часто конкретный клиент запрашивает обновление своей кешированной копии. В случае **аренды на основе частоты обновления** (renewal-frequency-based lease) сервер будет выдавать долгосрочную аренду клиенту, чей кеш необходимо часто обновлять. С другой стороны, клиенту, который только изредка запрашивает конкретный элемент данных, будет передана краткосрочная аренда для этого элемента. Результатом данной стратегии является то, что сервер отслеживает только тех клиентов, у которых его данные популярны; кроме того, этим клиентам предлагается высокая степень согласованности.

Последний критерий – это издержки пространства состояний на сервере. Когда сервер осознает, что он начинает постепенно перегружаться, он уменьшает срок действия новых договоров аренды, которые раздает клиентам. Результатом этой стратегии **аренды на основе состояния** (state-based lease strategy) является то, что серверу необходимо отслеживать меньше клиентов, поскольку срок аренды истекает быстрее. Другими словами, сервер динамически переключается в режим работы без состояния, ожидая, таким образом, разгрузить себя и иметь возможность обрабатывать запросы более эффективно. Очевидным недостатком является то, что при высоком отношении чтение-обновление может потребоваться дополнительная работа.

Одноадресная и многоадресная рассылка

Решение о том, следует ли использовать одноадресную или многоадресную рассылку, связано с проталкиванием или извлечением обновлений. При одноадресной связи, когда сервер, который является частью хранилища данных, отправляет свое обновление *N* другим серверам, он делает это, отправляя *N* отдельных сообщений по одному на каждый сервер. При многоадресной передаче базовая сеть обеспечивает эффективную отправку сообщения нескольким получателям.

Во многих случаях дешевле использовать доступные средства многоадресной рассылки. Экстремальная ситуация – когда все реплики расположены в одной локальной сети и аппаратное вещание доступно. В этом случае широковещательная или многоадресная рассылка сообщения обходится не дороже, чем отдельное сообщение «точка-точка», и одноадресные обновления становятся менее эффективными.

Многоадресную рассылку часто можно эффективно сочетать с основанным на подходе распространением обновлений проталкиванием. Когда они тщательно интегрированы, сервер, который решает отправить свои обновления ряду других серверов, просто использует одну многоадресную группу для отправки своих обновлений. Напротив, при использовании подхода на основе извлечения, как правило, только один клиент или сервер запрашивает

обновление своей копии. В этом случае наиболее эффективным решением может быть одноадресная рассылка.

Управление реплицированными объектами

Как мы уже упоминали, согласованность, ориентированная на данные для распределенных объектов, естественно возникает в форме согласованности ввода. Напомним, что в этом случае цель состоит в том, чтобы группировать операции над общими данными с использованием переменных синхронизации (например, в форме блокировок). Поскольку объекты естественным образом объединяют данные и операции с этими данными, блокировка объектов во время вызова устанавливает последовательность доступов и поддерживает их согласованность.

Хотя концептуально связать блокировку с объектом просто, она не обязательно обеспечивает правильное решение при репликации объекта. Есть две проблемы, которые необходимо решить для обеспечения согласованности ввода. Во-первых, нам нужны средства для предотвращения одновременного выполнения нескольких вызовов одного и того же объекта. Другими словами, когда выполняется какой-либо метод объекта, никакие другие методы не могут быть выполнены. Это требование гарантирует, что доступ к внутренним данным объекта действительно упорядочен. Простое использование локальных механизмов блокировки обеспечит эту упорядоченную последовательность.

Вторая проблема заключается в том, что в случае реплицируемого объекта мы должны убедиться, что все изменения в реплицированном состоянии объекта одинаковы. Другими словами, нам нужно убедиться, что в разных репликах не происходит двух независимых вызовов методов одновременно. Это требование подразумевает, что нам нужно упорядочить вызовы так, чтобы каждая реплика видела все вызовы в одном и том же порядке. Мы опишем несколько общих решений этого в разделе 7.5.

Во многих случаях проектирование реплицируемых объектов выполняется сначала путем проектирования одного объекта, возможно, защищая его от одновременного доступа через локальную блокировку, а затем реплицируя его. Роль промежуточного программного обеспечения состоит в том, чтобы гарантировать, что, если клиент вызывает реплицируемый объект, вызов передается репликам и передается на соответствующие серверы объектов везде в одинаковом порядке. Однако нам также необходимо убедиться, что все потоки на этих серверах также обрабатывают эти запросы в правильном порядке. Проблема показана на рис. 7.24.

Многопоточные (объектные) серверы просто принимают входящий запрос, передают его доступному потоку и ждут поступления следующего запроса. Планировщик потоков сервера впоследствии для запускаемых потоков выделяет центральный процессор. Конечно, если промежуточное программное обеспечение сделало все возможное, чтобы обеспечить полный порядок доставки запросов, планировщики потоков должны работать детерминированно, чтобы не смешивать порядок вызовов методов для одного и того же

объекта. Другими словами, если потоки T_1^1 и T_2^1 на рис. 7.24 обрабатывают один и тот же входящий (реплицированный) запрос вызова, они должны быть запланированы до T_1^2 и T_2^2 соответственно.

Конечно, просто планировать *все* потоки не обязательно. В принципе, если у нас уже есть полностью заказанная доставка запросов, нам нужно только обеспечить, чтобы все запросы для одного и того же реплицированного объекта обрабатывались в том порядке, в котором они были доставлены. Такой подход позволил бы обрабатывать вызовы для разных объектов одновременно и без дополнительных ограничений от планировщика потоков. К сожалению, существует только несколько систем, поддерживающих подобный параллелизм.

Подход, описанный в [Basile et al., 2002], гарантирует, что потоки, совместно использующие одну и ту же (локальную) блокировку, планируются в одном и том же порядке по каждой реплике. В его основе лежит первичная схема, в которой один из серверов реплики играет ведущую роль в определении, для конкретной блокировки, какой поток идет первым. Улучшение, позволяющее избежать частой связи между серверами, описано в [Basile et al., 2003]. Обратите внимание, что потоки, которые не разделяют блокировку, могут работать одновременно на каждом сервере.

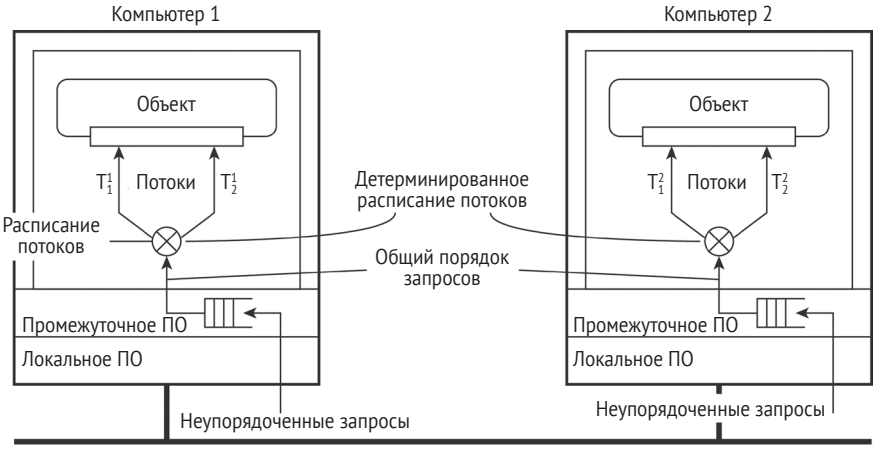


Рис. 7.24 ❖ Детерминированное планирование потоков для серверов реплицированных объектов

Одним из недостатков этой схемы является то, что она работает на уровне базовой операционной системы, а это означает, что каждой блокировкой необходимо управлять. Предоставляя информацию на уровне приложений, можно значительно повысить производительность, идентифицируя только те блокировки, которые необходимы для порядка последовательности доступа к реплицируемым объектам (см. [Taiani et al., 2005]).

Примечание 7.7 (дополнительно: реплицированные вызовы)

Другая проблема, которая должна быть решена, – это реплицированные вызовы. Рассмотрим объект A, вызывающий другой объект B, как показано на рис. 7.25. Предполагается, что объект B вызывает независимо еще один объект C. Если B реплицируется, каждая реплика B, в принципе, будет вызывать C. Проблема в том, что C теперь вызывается несколько раз, а не только один раз. Если вызванный метод на C приводит к переводу 100 000 долларов, то, очевидно, рано или поздно кто-то будет жаловаться.

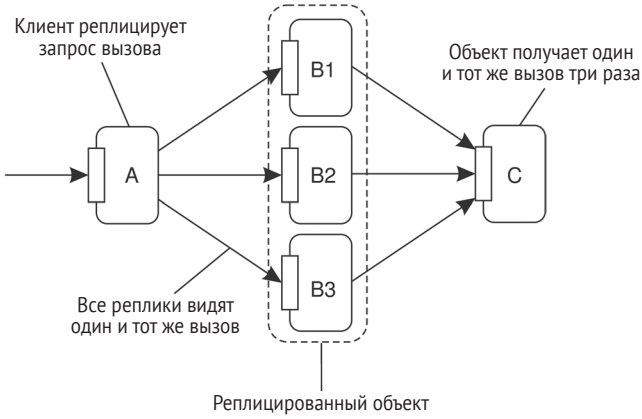


Рис. 7.25 ❖ Проблема реплицированных вызовов методов

Существует не так много универсальных решений проблемы повторных вызовов. Одно из решений – просто запретить это [Maassen et al., 2001], что имеет смысл, когда речь идет о производительности. Однако при тиражировании в целях отказоустойчивости может быть применено следующее решение, предложенное в [Mazouni et al., 1995]. Их решение не зависит от политики репликации, то есть точных деталей того, как реплики поддерживаются согласованными. Проблема в том, чтобы обеспечить уровень связи с поддержкой репликации, поверх которого выполняются (реплицируемые) объекты. Когда реплицированный объект B вызывает другой реплицируемый объект C, запросу на вызов сначала присваивается один и тот же уникальный идентификатор каждой реплике B. В этот момент координатор реплик B передает свой запрос всем репликам объекта C, в то время как другие реплики B хранят свою копию запроса вызова, как показано на рис. 7.26. В результате к каждой реплике C направляется только один запрос.

Тот же механизм используется, чтобы гарантировать, что только одно ответное сообщение вернулось к репликам B. Эта ситуация показана на рис. 7.26. Координатор реплик C замечает, что имеет дело с реплицированным ответным сообщением, которое было сгенерировано каждой репликой C. Однако только координатор перенаправляет этот ответ на реплики объекта B, в то время как другие реплики C удерживают свою копию ответного сообщения.

Когда реплика B получает ответное сообщение для запроса вызова, которое она либо перенаправила на C, либо удержала, поскольку не являлась координатором, ответ затем передается фактическому объекту.

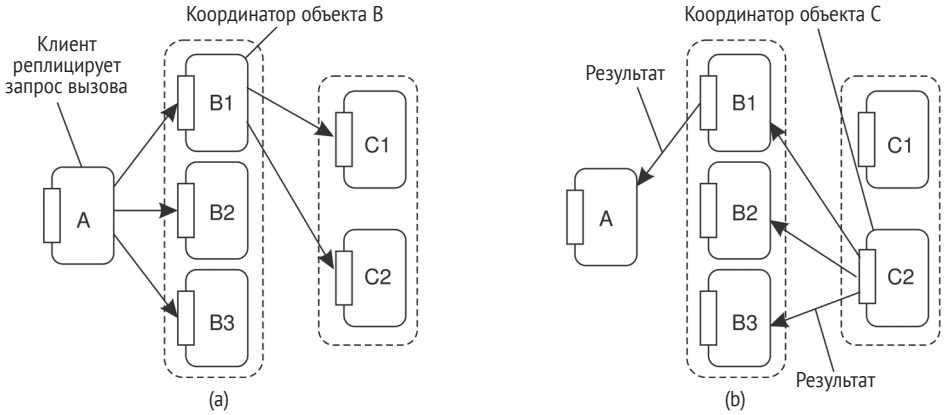


Рис. 7.26 ❖ a) Пересылка запроса на вызов
от реплицируемого объекта другому реплицируемому объекту;
b) возврат ответа от одного реплицированного объекта другому

По существу, только что описанная схема основана на использовании многоадресной связи, но предотвращает многоадресную передачу одного и того же сообщения разными репликами, то есть схема, основанная на отправителе. Альтернативное решение состоит в том, чтобы позволить получающей реплике обнаруживать несколько копий входящих сообщений, принадлежащих одному и тому же вызову, и передавать только одну копию соответствующему объекту. Детали этой схемы оставляем читателю в качестве упражнения.

7.5. Согласованность протоколов

Теперь мы сосредоточимся на фактической реализации моделей согласованности, взглянув на несколько протоколов согласованности. Протокол согласованности описывает реализацию конкретной модели согласованности. Мы следим за организацией нашего обсуждения моделей согласованности, сначала рассмотрев модели, ориентированные на данные, а затем протоколы для моделей, ориентированных на клиента.

Непрерывная последовательность

В работе [Yu and Vahdat, 2000] разработан ряд протоколов для решения трех форм согласованности. Далее мы кратко рассмотрим ряд решений, для ясности опуская некоторые детали.

Ограничивающее числовое отклонение

Сначала сосредоточимся на одном решении для сохранения пределов численного отклонения. Опять же, наша цель – не вдаваться во все детали каж-

дого протокола, а дать общее представление. Подробности ограничивающего числового отклонения можно найти в [Yu and Vahdat, 2000].

Мы концентрируемся на записи в один элемент данных x. Каждая запись W(x) имеет **ассоциированное значение**, которое представляет собой число, на которое обновляется x, и обозначается как *val*(W(x)), или просто *val*(W). Для простоты мы предполагаем, что *val*(W) > 0. Каждая запись W первоначально отправляется одному из *N* доступных серверов реплики, и в этом случае данный сервер становится источником записи, обозначаемым как *origin*(W). Если мы рассмотрим систему в определенный момент времени, то увидим несколько отправленных записей, которые все еще необходимо распространить на все серверы. С этой целью каждый сервер S_i будет отслеживать записи в журнале **log** L_i о том, что он выполнил свою собственную локальную копию x.

Пусть $TW[i, j]$ будет эффектом выполнения записей сервером S_i, которые исходят от сервера S_j:

$$TW[i,j] = \sum \{val(W) | origin(W) = S_j \text{ и } W \in L_i\}.$$

Обратите внимание, что $TW[i,i]$ представляет агрегированные записи, имеющиеся в S_i. Наша цель – в любой момент времени t позволить текущему значению $v(x)$ на сервере S_i отклоняться в пределах границ от фактического значения v(x). Это фактическое значение полностью определяется всеми представленными записями. То есть если v_0 является начальным значением x, то

$$v = v_0 + \sum_{k=1}^{N} TW[k,k]$$

и

$$v_i = v_0 + \sum_{k=1}^{N} TW[i,k].$$

Обратите внимание, что $v_i \leq v$. Давайте сосредоточимся только на абсолютных отклонениях. В частности, для каждого сервера S_i мы связываем верхнюю границу δ_i так, чтобы обеспечить

$$v - v_i \leq \delta_i.$$

Записи, отправленные на сервер S_i, необходимо будет распространить на все остальные серверы. Это можно сделать разными способами, но, как правило, эпидемический протокол позволяет быстро распространять обновления. В любом случае, когда сервер S_i распространяет запись, исходящую из S_j в S_k, последний сможет узнать о значении $TW[i, j]$ во время отправки записи. Другими словами, S_k может поддерживать вид $TW_k[i, j]$ того, что, по его мнению, S_i будет иметь в качестве значения для $TW[i, j]$.

Очевидно, что

$$0 \leq TW_k[i,j] \leq TW[i,j] \leq TW[j,j].$$

Идея заключается в том, что когда сервер S_k замечает, что S_i не находится в правильном темпе с обновлениями, которые были отправлены в S_k, он пересылает записи из своего журнала в S_i. Эта пересылка эффективно *продвигает* представление $TW_k[i, k]$, что S_k имеет $TW[i, k]$, делая отклонение $TW[i, k] - TW_k[i, k]$ меньшим. В частности, S_k обращает внимание на $TW[i, k]$, когда приложение представляет новую запись, которая будет увеличивать $TW[k, k] - TW_k[i, k]$ за пределы $\delta_i(N - 1)$. Мы оставляем читателю в качестве упражнения показать, что продвижение всегда гарантирует, что $v - v_i < \delta_i$.

Граничные отклонения устаревания

Есть много способов сохранить устаревание реплик в указанных пределах. Один простой подход – позволить серверу S_k сохранять векторные часы реального времени RVC_k, где $RVC_k[i] = t_i$ означает, что S_k видел все записи, которые были отправлены в S_i до времени t_i. В этом случае мы предполагаем, что каждая отправленная запись помечается временем своего исходного сервера и что t_i обозначает время, *локальное* для S_i.

Если часы между серверами реплики слабо синхронизированы, то приемлемый протокол для ограничения устаревания будет следующим. Всякий раз, когда сервер S_k отмечает, что $t_k - RVC_k[i]$ собирается превысить указанный предел, он просто начинает извлекать записи, исходящие из S_i, с отметкой времени позже $RVC_k[i]$.

Обратите внимание, что в этом случае сервер реплики отвечает за поддержание своей копии x в актуальном состоянии в отношении записей, которые были выпущены в другом месте. Напротив, при поддержании числовых границ мы следовали принудительному подходу, позволяя исходному серверу поддерживать реплики в актуальном состоянии путем пересылки записей. Проблема с выталкивающими записями в случае устаревания заключается в том, что не может быть никаких гарантий согласованности, когда заранее неизвестно, каким будет максимальное время распространения. Эта ситуация несколько улучшается за счет загрузки обновлений, поскольку несколько серверов могут помочь сохранить свежую (то есть обновленную) копию сервера x.

Ограничение отклонений порядка

Напомним, что отклонения порядка при непрерывной согласованности вызваны тем фактом, что сервер реплики предварительно применяет обновления, которые были ему отправлены. В результате каждый сервер будет иметь локальную очередь предварительных записей, для которой все еще необходимо определить фактический порядок, в котором они должны применяться к локальной копии x. Отклонение порядка ограничено указанием максимальной длины очереди предварительных записей.

Как следствие определение необходимости упорядочения последовательности очень просто: оно возникает, когда длина этой локальной очереди превышает указанную максимальную длину. В этот момент сервер больше не будет принимать новые отправленные записи, но вместо этого попыта-

ется зафиксировать предварительные записи, вступив в контакт с другими серверами, в порядке, в котором должны выполняться его записи. Иными словами, необходимо обеспечить глобально согласованный порядок предварительных записей.

Первичные протоколы

На практике мы видим, что распределенные приложения обычно следуют моделям согласованности, которые относительно легко понять. К этим моделям относятся те, которые ограничивают отклонения от устаревания, и в меньшей степени также те, которые ограничивают числовые отклонения. Что касается моделей, которые обрабатывают последовательное упорядочение операций, последовательную согласованность операций, особенно популярными являются те, в которых операции могут быть сгруппированы с помощью блокировки или транзакций.

Как только модели согласованности становятся более сложными для понимания разработчиками приложений, они игнорируются ими, даже если производительность при этом может быть улучшена. Это происходит потому, что когда разработчику интуитивно не понятна семантика модели согласованности, ему сложно создавать правильные приложения. Простота ценится (и, возможно, оправданно).

В случае последовательной согласованности оказывается, что на практике преобладают **первичные протоколы** (primary-based protocols). В этих протоколах каждый элемент данных x в хранилище данных имеет связанный с ним первичный элемент, который отвечает за координацию операций записи в x. Может быть сделано разделение: или первичный элемент фиксируется на удаленном сервере, или операции записи выполняются локально после перемещения первичного элемента в процесс, который инициирует операцию записи.

Протоколы удаленной записи

Простейший первичный протокол, который поддерживает репликацию, – это протокол, в котором все операции записи необходимо перенаправлять на фиксированный одиночный сервер, и операции чтения могут выполняться локально. Такие схемы также известны как **протоколы первичного резервного копирования** (primary-backup protocols) [Budhijara et al., 1993]. Протокол первичного резервного копирования работает, как показано на рис. 7.27. Процесс, желающий выполнить операцию записи для элемента данных x, перенаправляет эту операцию на основной (первичный) сервер для x. Основной выполняет обновление своей локальной копии x, а затем передает обновление на серверы резервного копирования. Каждый сервер резервного копирования также выполняет обновление и отправляет подтверждение первичному серверу. Когда все резервные копии обновили свою локальную копию, основной сервер отправляет подтверждение начальному процессу, который, в свою очередь, информирует клиента.

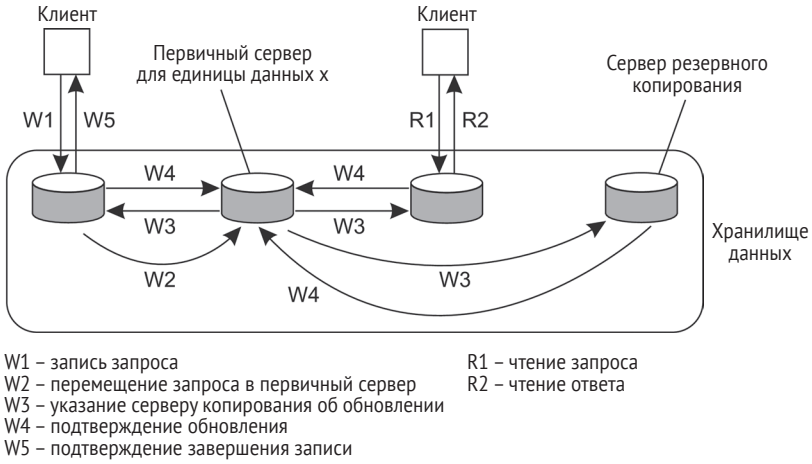


Рис. 7.27 ❖ Принцип протокола первичного резервного копирования

Потенциальная проблема производительности в этой схеме состоит в том, что может пройти относительно много времени, прежде чем процессу, инициировавшему обновление, будет разрешено продолжить. По сути, обновление реализовано как операция блокировки. Альтернативой является использование неблокирующего подхода. Как только первичный сервер обновил свою локальную копию x, он возвращает подтверждение. После этого он сообщает серверам резервного копирования также выполнить обновление. Неблокирующие протоколы первичного резервного копирования обсуждаются в [Budhiraja and Marzullo, 1992].

Основная проблема с неблокирующими протоколами первичного резервного копирования связана с отказоустойчивостью. В схеме блокировки клиентский процесс точно знает, что операция обновления поддерживается несколькими другими серверами. При неблокирующем решении это не так. Преимущество этой схемы, конечно, в том, что операции записи могут значительно ускориться.

Протоколы первичного резервного копирования обеспечивают простую реализацию последовательной согласованности, поскольку первичный сервер может упорядочить все входящие записи в глобально уникальном временном порядке. Очевидно, что все процессы видят все операции записи в одном и том же порядке, независимо от того, какой сервер резервного копирования они используют для выполнения операций чтения. Кроме того, при использовании протоколов блокировки процессы всегда будут видеть результаты своей последней операции записи (обратите внимание, что это не может быть гарантировано неблокирующим протоколом без принятия специальных мер).

Протоколы локальной записи

Вариант протоколов первичного резервного копирования – это протокол, в котором первичная копия мигрирует между процессами, которые хотят

выполнить операцию записи. Как и прежде, всякий раз, когда процесс хочет обновить элемент данных x, он находит основную копию x и впоследствии перемещает ее в свое собственное местоположение, как показано на рис. 7.28. Основным преимуществом этого подхода является то, что несколько последовательных операций записи могут выполняться локально, тогда как процессы чтения могут по-прежнему получать доступ к своей локальной копии. Однако такое улучшение может быть достигнуто только в том случае, если используется неблокирующий протокол, по которому обновления распространяются на реплики, после того как основной сервер завершил локальное выполнение обновлений.

Этот протокол локальной записи первичной резервной копии также может использоваться применительно к мобильным компьютерам, которые могут работать в автономном режиме. Перед отключением мобильный компьютер становится основным сервером для каждого элемента данных, который он ожидает обновить. При отключении все операции обновления выполняются локально, в то время как другие процессы могут выполнять операции чтения (но не обновления). Позднее, при повторном подключении, обновления распространяются с первичного на резервные копии, что снова приводит хранилище данных в согласованное состояние.

В качестве последнего варианта этой схемы неблокирующие первичные протоколы с локальной записью также используются для распределенных файловых систем в целом. В этом случае может существовать фиксированный центральный сервер, через который обычно выполняются все операции записи, как в случае первичной резервной копии с удаленной записью. Однако сервер временно позволяет одной из реплик выполнять серию локальных обновлений, поскольку это может значительно повысить производительность. Когда сервер реплики завершен, обновления распространяются на центральный сервер, откуда они затем распространяются на другие серверы реплики.

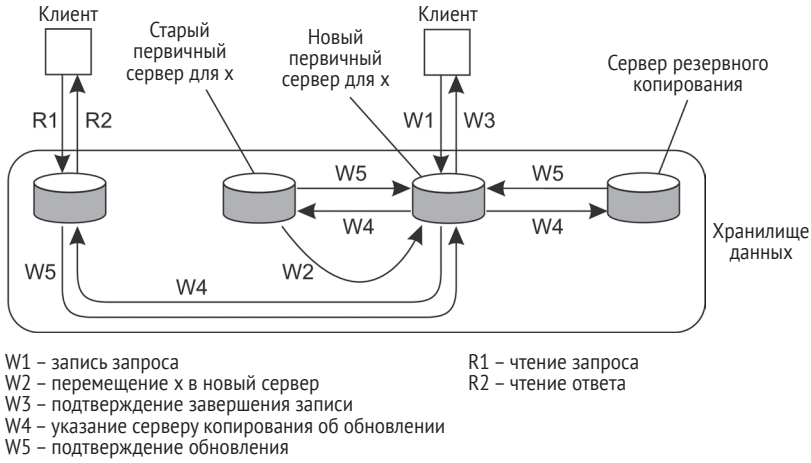


Рис. 7.28 ❖ Протокол первичного резервного копирования, в котором первичный сервер мигрирует в процесс, желающий выполнить обновление

Протоколы реплицируемой записи

В протоколах реплицированной записи операции записи могут выполняться в нескольких репликах, а не только в одной, как в случае первичных реплик. Можно провести различие между активной репликацией, в которой операция передается всем репликам, и протоколами согласованности, основанными на голосовании большинства.

Активная репликация

В активной репликации каждая реплика имеет связанный процесс, который выполняет операции обновления. В отличие от других протоколов, обновления обычно распространяются посредством операции записи, которая вызывает обновление. Другими словами, эта операция посылается каждой реплике. Вместе с тем можно также отправлять и обновление.

Проблема с активной репликацией состоит в том, что операции должны выполняться везде в одном и том же порядке. Следовательно, необходим полностью упорядоченный многоадресный механизм. Практический подход к выполнению полного упорядочения осуществляется с помощью центрального координатора, также называемого **секвенсором** (sequencer). Один из подходов заключается в том, чтобы сначала переслать каждую операцию в секвенсор, который присваивает ей уникальный порядковый номер, и затем перенаправить эту операцию всем репликам. Операции выполняются в соответствии с их порядковым номером.

Примечание 7.8 (дополнительно: достижение масштабируемости)

Обратите внимание, что использование секвенсора может легко создать проблемы с масштабируемостью. Фактически если требуется упорядоченная групповая адресация, комбинация симметричной групповой адресации, возможно, понадобятся временны́е метки Лампорта [Lamport, 1978] и секвенсоры. Такое решение описано в [Rodrigues et al., 1996]. Суть этого решения заключается в том, чтобы иметь операции многоадресного обновления нескольких секвенсоров друг друга и упорядочивать обновления, используя механизм полного упорядочения Lamport, как описано в разделе 6.2. Непоследовательные процессы сгруппированы так, что в каждой группе используется один секвенсор. Любой неупорядоченный процесс отправляет запросы на обновление своему секвенсору и ждет, пока не получит подтверждения того, что его запрос был обработан (то есть многоадресную рассылку другим секвенсорам путем упорядочения). Очевидно, что существует компромисс между количеством процессов, которые действуют как секвенсор, и теми, которые этого не делают, а также выбором процессов, которые будут действовать как секвенсор. Как выясняется, этот компромисс во многом зависит от приложения и, в частности, от относительной частоты обновления в каждом процессе.

Протоколы на основе кворума

Другой подход к поддержке реплицированных записей заключается в использовании голосования, как первоначально было предложено Томасом

[Thomas, 1979] и обобщено Гиффордом [Gifford, 1979]. Основная идея состоит в том, чтобы требовать от клиентов запрашивать и получать разрешение нескольких серверов перед чтением или записью реплицированного элемента данных.

В качестве простого примера того, как работает алгоритм, рассмотрим распределенную файловую систему и предположим, что файл реплицируется на N серверах. Мы могли бы создать правило, гласящее, что для обновления файла клиент должен сначала связаться хотя бы с половиной серверов плюс один (большинством) и заставить их согласиться выполнить обновление. Как только они согласились, файл изменяется, и новый номер версии связывается с новым файлом. Номер версии используется для идентификации версии файла и является одинаковым для всех недавно обновленных файлов.

Чтобы прочитать реплицированный файл, клиент должен также связаться по крайней мере с половиной серверов плюс один и попросить их отправить номера версий, связанных с файлом. Если все номера версий совпадают, это должна быть самая последняя версия, поскольку попытка обновить только оставшиеся серверы не удастся, так как их недостаточно.

Например, если есть пять серверов и клиент определяет, что у трех из них есть версия 8, невозможно, чтобы у двух других была версия 9. В конце концов, любое успешное обновление с версии 8 до версии 9 требует, чтобы три сервера согласились, а не только два.

После того как изначально была введена репликация на основе кворума, была предложена несколько более общая схема. В ней для чтения файла, в котором существует N реплик, клиент должен собрать **кворум чтения** (read quorum), произвольную коллекцию любых серверов N_R или более. Точно так же, чтобы изменить файл, требуется **кворум записи** (write quorum) как минимум N_W-серверов. Значения N_R и N_W подчиняются следующим двум ограничениям:

1) $N_R + N_W > N$;
2) $N_W > N/2$.

Первое ограничение используется для предотвращения конфликтов чтения-записи, тогда как второе предотвращает конфликты записи-записи. Только после того, как соответствующее количество серверов согласится участвовать, файл может быть прочитан или записан.

Чтобы увидеть, как работает этот алгоритм, рассмотрим рис. 7.29а с $N_R = 3$ и $N_W = 10$. Представьте, что самый последний кворум записи состоял из 10 серверов от C до L. Все они получают новую версию и новое число версии. Любой последующий кворум чтения из трех серверов должен содержать хотя бы одного члена этого набора. Когда клиент смотрит на номера версий, он узнает, какая из них самая свежая, и выберет ее.

На рис. 7.29 мы видим еще два примера. На рис. 7.29b может возникнуть конфликт запись-запись, потому что $N_W < N/2$.

В частности, если один клиент выберет {A, B, C, E, F, G} в качестве своего набора записи, а другой клиент выберет {D, H, I, J, K, L} в качестве своего набора записи, то очевидно, что мы столкнемся с тем, что оба обновления будут приняты без обнаружения фактического конфликта.

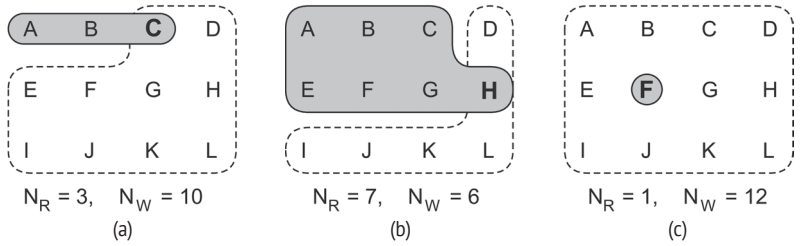


Рис. 7.29 ❖ Три примера алгоритма голосования.
Серые области обозначают прочитанный кворум; белые – кворум для записи.
Серверы на пересечении обозначены жирным шрифтом: а) правильный выбор набора для чтения и записи; б) выбор, который может привести к конфликтам записи-записи; с) правильный выбор, известный как ROWA (Read One, Write All; читай один, пиши все)

Ситуация, показанная на рис. 7.29с, особенно интересна, поскольку она устанавливает N_R равным единице, что позволяет прочитать реплицированный файл, найдя любую копию и используя ее. Однако цена, которую платят за эту хорошую производительность чтения, состоит в том, что для записи обновлений необходимо приобрести все копии. Эта схема обычно называется **ROWA** (Read One, Write All; **читай один, пиши все**). Существует несколько вариантов протоколов репликации на основе кворума. Хороший обзор в [Jalote, 1994].

Протоколы кеширования

Кеши образуют особый случай репликации в том смысле, что они обычно контролируются клиентами, а не серверами. Однако протоколы когерентности кеша, которые обеспечивают соответствие кеша репликам, инициируемым сервером, в принципе, не сильно отличаются от протоколов согласованности, которые обсуждались до сих пор.

Было проведено много исследований в области разработки и реализации кешей, особенно в контексте многопроцессорных систем с разделяемой памятью. Многие решения основаны на поддержке базового оборудования, например при условии, что может быть выполнено отслеживание или эффективное вещание. В контексте распределенных систем на основе промежуточного программного обеспечения, которые построены на основе операционных систем общего назначения, программные решения для кеширования наиболее интересны. В этом случае часто поддерживается два отдельных критерия для классификации протоколов кеширования (см. также [Min and Baer, 1992], [Lilja, 1993] или [Tartalja and Milutinovic, 1997]).

Во-первых, решения для кеширования могут отличаться по своей **стратегии обнаружения когерентности** (coherence detection strategy), то есть когда фактически обнаруживаются несоответствия. В статических решениях предполагается, что компилятор выполняет необходимый анализ перед выполнением и определяет, какие данные могут фактически привести к несоответствиям, поскольку они могут быть кешированы. Компилятор просто вставляет инструкции, которые позволяют избежать несоответствий. В рас-

пределенных системах, изучаемых в этой книге, обычно применяются динамические решения. В этих решениях несоответствия обнаруживаются во время выполнения. Например, сервер проверяет, были ли изменены кешированные данные с момента их кеширования.

В случае распределенных баз данных протоколы, основанные на динамическом обнаружении, могут быть дополнительно классифицированы с учетом того, когда именно во время транзакции выполняется обнаружение. В работе [Franklin et al., 1997] различают следующие три случая. Первый, когда во время транзакции осуществляется доступ к кешированному элементу данных, клиент должен проверить, соответствует ли этот элемент данных версии, хранящейся на (возможно, реплицированном) сервере. Транзакция не может продолжать использовать кешированную версию, пока ее согласованность не будет окончательно подтверждена.

Второй, оптимистичный подход – позволить транзакции продолжаться во время проверки. В этом случае предполагается, что кешированные данные были обновлены на момент начала транзакции. Если это предположение позже окажется ложным, транзакцию придется прервать.

Третий подход заключается в проверке актуальности кешированных данных лишь при фиксации транзакции. По сути, когда транзакция только начинает работать с кешированными данными и надеется на лучшее. После того как вся работа выполнена, полученные данные проверяются на согласованность. Если использовались устаревшие данные, транзакция прерывается.

Другой проблемой при разработке протоколов когерентности кеша является **стратегия усиления когерентности** (coherence enforcement strategy), которая определяет, как кеши поддерживаются в соответствии с копиями, хранящимися на серверах. Самое простое решение – вообще запретить совместное использование данных. Вместо этого общие данные хранятся только на серверах, которые поддерживают согласованность с использованием одного из первичных протоколов или протоколов репликации и записи, описанных выше. Клиенты могут кешировать только личные данные. Очевидно, что это решение может предложить лишь ограниченные улучшения производительности.

Когда общие данные могут быть кешированы, существует два подхода для обеспечения согласованности кеша. Первый – разрешить серверу отправлять недействительные данные на все кеши всякий раз, когда элемент данных изменяется. Второй – просто распространять обновление. Большинство систем кеширования используют одну из этих двух схем. Динамический выбор между отправкой недействительных данных или обновленнных данных иногда поддерживается в базах данных клиент-сервер.

Наконец, необходимо также рассмотреть, что происходит, когда процесс изменяет кешированные данные. При использовании кешей только для чтения операции обновления могут выполняться лишь серверами, которые впоследствии следуют некоторому протоколу распространения, чтобы гарантировать, что обновления распространяются на кеши. Во многих случаях применяется подход, основанный на извлечении (pull-based approach). В этом случае клиент обнаруживает, что его кеш устарел, и запрашивает обновление на сервере.

Альтернативный подход – позволить клиентам напрямую изменять кешированные данные и пересылать обновления на серверы. Этот подход используется в **кешах сквозной записи** (write-through caches), которые часто используются в распределенных файловых системах. По существу, сквозное кеширование аналогично первичному протоколу локальной записи, в котором клиентский кеш стал временно основным. Чтобы гарантировать (последовательную) согласованность, необходимо, чтобы клиенту были предоставлены эксклюзивные разрешения на запись, иначе могут возникнуть конфликты запись-запись.

Кеши сквозной записи потенциально предлагают улучшенную производительность по сравнению с другими схемами, поскольку все операции могут выполняться локально. Дальнейшие улучшения могут быть сделаны, если мы задержим распространение обновлений, разрешив многократные записи, прежде чем информировать серверы. Это приводит к так называемому **кешу с обратной записью** (write-back cache), который, опять же, в основном применяется в распределенных файловых системах.

Примечание 7.9 (пример: кеширование на стороне клиента в NFS)

В качестве практического примера рассмотрим общую модель кеширования в NFS, показанную на рис. 7.30. Каждый клиент может иметь кеш-память, которая содержит данные, ранее считанные с сервера. Кроме того, может также существовать дисковый кеш, который добавляется в качестве расширения кеша памяти с использованием тех же параметров согласованности.

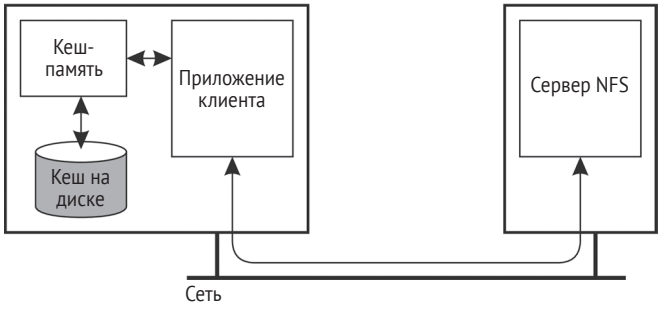


Рис. 7.30 ❖ Кеширование на стороне клиента в NFS

Как правило, клиенты кешируют данные файла, атрибуты, дескрипторы файла и каталоги. Существуют различные стратегии для обеспечения согласованности кешированных данных, кешированных атрибутов и т. д. Давайте сначала посмотрим на данные кеширования файла.

Программа NFSv4 поддерживает два разных подхода для кеширования файловых данных. Самый простой подход – это когда клиент открывает файл и кеширует данные, которые он получает от сервера в результате различных операций чтения. Кроме того, операции записи могут выполняться и в кеше. Когда клиент закрывает файл, NFS требует, чтобы в случае внесения изменений кешированные данные отправлялись обратно на сервер. Этот подход соответствует реализации семантики сеанса, как обсуждалось ранее.

Как только (часть) файл был кеширован, клиент может сохранить свои данные в кеше даже после закрытия файла. Кроме того, несколько клиентов на одном компьютере могут совместно использовать один кеш. NFS требует, чтобы всякий раз, когда клиент открывал ранее закрытый файл, который был (частично) кеширован, клиент должен немедленно повторно проверить кешированные данные. Повторная проверка выполняется путем проверки того, когда файл был последний раз изменен, и аннулирования кеша в случае, если он содержит устаревшие данные.

В NFSv4 сервер может делегировать некоторые свои права клиенту при открытии файла. **Открытое делегирование** (open delegation) происходит, когда клиентскому компьютеру разрешено локально обрабатывать операции открытия и закрытия от других клиентов на том же компьютере. Обычно сервер отвечает за проверку того, должно ли открытие файла быть успешным или нет, например потому, что необходимо учитывать резервирование общих ресурсов. При открытом делегировании клиентскому компьютеру иногда разрешается принимать такие решения, избегая необходимости связываться с сервером.

Например, если сервер делегировал открытие файла клиенту, который запросил разрешения на запись, запросы блокировки файлов от других клиентов на том же компьютере также могут обрабатываться локально. Сервер по-прежнему будет обрабатывать запросы блокировки от клиентов на других компьютерах, просто отказывая этим клиентам в доступе к файлу. Обратите внимание, что эта схема не работает в случае делегирования файла клиенту, который запросил только разрешения на чтение. В этом случае, когда другой локальный клиент хочет иметь права на запись, он должен связаться с сервером; невозможно обработать запрос локально.

Важным следствием делегирования файла клиенту является то, что сервер должен иметь возможность отозвать делегирование, например когда другому клиенту на другом компьютере необходимо получить права доступа к файлу. Для отзыва делегирования требуется, чтобы сервер мог выполнить обратный вызов клиенту, как показано на рис. 7.31.

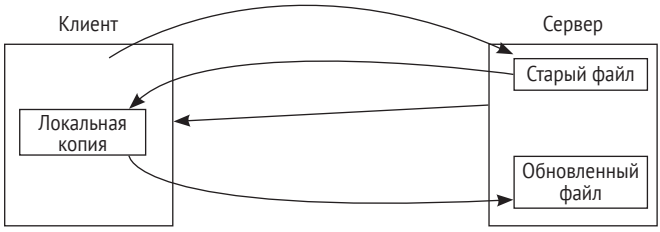


Рис. 7.31 ❖ Использование механизма обратного вызова NFSv4 для вызова делегирования файла

Обратный вызов реализован в NFS с использованием базовых механизмов RPC (remote distance control, дистанционное управление). Однако обратите внимание, что обратные вызовы требуют, чтобы сервер отслеживал клиентов, которым он делегировал файл. Здесь мы видим другой пример, где сервер NFS больше не может быть реализован без сохранения состояния. Обратите внимание, однако, что комбинация серверов делегирования и состояний может привести к различным проблемам при сбоях клиента и сервера. Например, что должен делать сервер, когда он делегировал файл клиенту, который теперь не отвечает?

Клиенты также могут кешировать значения атрибутов, но в значительной степени остаются самостоятельными, когда речь идет о сохранении согласованных ке-

шированных значений. В частности, значения атрибутов одного и того же файла, кешированные двумя разными клиентами, могут отличаться, если только клиенты не поддерживают эти атрибуты взаимно согласованными. Модификации значения атрибута должны быть немедленно отправлены на сервер, таким образом следуя политике когерентности сквозного кеша.

Аналогичный подход применяется для дескрипторов файлов кеширования (или, скорее, для сопоставления дескрипторов имен в файлах) и каталогов. Чтобы смягчить последствия несоответствий, NFS использует аренду кешированных атрибутов, файловых дескрипторов и каталогов. По истечении некоторого времени записи в кеше, таким образом, автоматически становятся недействительными, и требуется повторная проверка перед повторным использованием.

Реализация согласованности, ориентированной на клиента

Что касается нашей последней темы о протоколах согласованности, давайте обратим внимание на реализацию согласованности, ориентированной на клиента. Реализация согласованности, ориентированной на клиента, относительно проста, если игнорировать проблемы с производительностью.

В наивной реализации клиент-ориентированной согласованности каждой операции записи W присваивается глобально уникальный идентификатор. Такой идентификатор присваивается сервером, на который была отправлена запись. Мы называем этот сервер источником W. Затем для каждого клиента мы отслеживаем два набора записей. Набор чтения для клиента состоит из записей, относящихся к операциям чтения, выполняемым клиентом. Аналогично набор записи состоит из (идентификаторов) записей, выполняемых клиентом.

Монотонно-читаемая согласованность реализуется следующим образом. Когда клиент выполняет операцию чтения на сервере, этому серверу передается клиентский набор для чтения, чтобы проверить, все ли идентифицированные записи выполнялись локально. Если нет, он связывается с другими серверами, чтобы убедиться, что он обновлен перед выполнением операции чтения. В качестве альтернативы операция чтения пересылается на сервер, где уже выполнялись операции записи. После выполнения операции чтения операции записи, которые выполнялись на выбранном сервере и которые имеют отношение к операции чтения, добавляются в клиентский набор чтения.

Обратите внимание, что должна быть возможность точно определить, где выполнялись операции записи, идентифицированные в наборе чтения. Например, идентификатор записи может включать в себя идентификатор сервера, на который была передана операция. Этот сервер необходим, например, для записи операции записи, чтобы ее можно было воспроизвести на другом сервере. Кроме того, операции записи должны выполняться в порядке их отправки. Порядок может быть достигнут, позволяя клиенту генерировать глобально уникальный порядковый номер, который включен в идентификаторе записи. Если каждый элемент данных может быть изменен только его владельцем, последний может предоставить порядковый номер.

Согласованность монотонной записи реализована аналогично монотонному чтению. Всякий раз, когда клиент инициирует новую операцию записи на сервере, серверу передается клиентский набор записи. (Опять же, размер набора может быть чрезмерно большим с учетом требований к производительности. Альтернативное решение обсуждается ниже.) Затем оно гарантирует, что идентифицированные операции записи выполняются в первую очередь и в правильном порядке. После выполнения новой операции идентификатор записи этой операции добавляется в набор записи. Обратите внимание, что приведение текущего сервера в соответствие с набором записи клиента может привести к значительному увеличению времени ответа клиента, поскольку клиент затем ожидает завершения операции.

Аналогично согласованность чтения-записи-записи требует, чтобы сервер, на котором выполняется операция чтения, видел все операции записи в наборе записи клиента. Записи могут быть просто извлечены с других серверов перед выполнением операции чтения, хотя это может привести к снижению времени ответа. Альтернативно клиентское программное обеспечение может искать сервер, на котором идентифицированные операции записи в наборе записи клиента уже выполнены.

Наконец, согласованность записи-следования-чтения может быть реализована, сначала приводя выбранный сервер в соответствие с операциями записи в наборе чтения клиента, а затем добавив идентификатор операции записи в набор записи вместе с идентификаторами в наборе чтения (которые теперь стали релевантными для только что выполненной операции записи).

Примечание 7.10 (дополнительно: повышение эффективности)

Легко видеть, что набор чтения и набор записи, связанные с каждым клиентом, могут стать очень большими. Чтобы эти наборы были управляемыми, операции чтения и записи клиента группируются в сеансы. Сеанс обычно связан с приложением: он открывается при запуске приложения и закрывается при его выходе. Однако сеансы также могут быть связаны с приложениями, которые временно закрыты, такими как пользовательские агенты для электронной почты. Всякий раз, когда клиент закрывает сеанс, наборы просто очищаются. Конечно, если клиент открывает сеанс, который он никогда не закрывает, связанные наборы для чтения и записи могут все еще стать очень большими.

Основная проблема с наивной реализацией заключается в представлении наборов для чтения и записи. Каждый набор состоит из нескольких идентификаторов для операций записи. Всякий раз, когда клиент пересылает запрос на чтение или запись на сервер, серверу также передается набор идентификаторов, чтобы посмотреть, были ли все операции записи, относящиеся к запросу, выполнены этим сервером.

Эта информация может быть более эффективно представлена с помощью векторных временных меток следующим образом. Во-первых, всякий раз, когда сервер принимает новую операцию записи W, он назначает этой операции глобально уникальный идентификатор вместе с отметкой времени *ts*(W). Последующей операции записи, отправленной на этот сервер, присваивается более высокая временная метка. Каждый сервер S_i поддерживает векторную метку времени WVC_i, где $WVC_i[j]$ равна метке времени самой последней операции записи, исходящей из S_j, которая была обработана S_i.

Для ясности предположим, что для каждого сервера записи из S_j обрабатываются в том порядке, в котором они были отправлены. Всякий раз, когда клиент отправляет запрос на выполнение операции чтения или записи O на конкретном сервере, этот сервер возвращает свою текущую временную метку вместе с результатами O. Впоследствии наборы чтения и записи представляются векторными временными метками. Более конкретно, для каждого сеанса A мы строим векторную временную метку SVC_A с установленным $SVC_A[i]$, равным максимальной временной отметке всех операций записи в A, которые исходят от сервера S_i:

$$SVC_A[j] = \max\{ts(W) | W \in A \text{ и } origin(W) = S_j\}.$$

Другими словами, метка времени сеанса всегда представляет самые последние операции записи, которые были замечены приложениями и которые выполняются как часть этого сеанса. Компактность достигается путем представления всех наблюдаемых операций записи, исходящих с одного и того же сервера, через одну временную метку.

В качестве примера предположим, что клиент в рамках сеанса A входит в систему на сервере S_i. Для этого он передает SVC_A S_i. Предположим, что $SVC_A[j] > WVC_i[j]$. Это означает, что S_i еще не видел все записи, происходящие из S_j, которые видел клиент. В зависимости от требуемой согласованности серверу S_i теперь, возможно, придется извлекать эти записи, прежде чем они смогут последовательно отчитаться перед клиентом. Как только операция будет выполнена, сервер S_i вернет текущую временную метку WVC_i. На этом этапе SVC_A настраивается на

$$SVC_A[j] \leftarrow \max\{SVC_A[j], WVC_i[j]\}.$$

Снова мы видим, как векторные временные метки могут обеспечить элегантный и компактный способ представления истории в распределенной системе.

7.6. Пример: кеширование и репликация в сети

Веб-сеть – возможно, самая большая распределенная система из когда-либо созданных. Происходя из сравнительно простой клиент-серверной архитектуры, теперь это сложная система, состоящая из множества методов, обеспечивающих строгие требования к производительности и доступности. Эти требования привели к многочисленным предложениям для кеширования и репликации веб-контента. Там, где исходные схемы (которые все еще в значительной степени развернуты) были нацелены на поддержку статичного контента, также были приложены большие усилия для поддержки динамичного контента, то есть поддержки документов, которые генерируются на месте в результате запроса, а также тех, которые содержат сценарии и т. п. Обзор традиционного веб-кеширования и репликации предоставлен в [Rabinovich and Spastscheck, 2002].

Кеширование на стороне клиента в сети обычно происходит в двух местах. Во-первых, большинство браузеров оснащены относительно простым средством кеширования. Всякий раз, когда документ извлекается, он сохраняется в кеше браузера, откуда загружается в следующий раз. Во-вторых, на сайте клиента часто работает веб-прокси. Веб-прокси принимает запросы от локальных клиентов и передает их веб-серверам. Когда приходит ответ,

результат передается клиенту. Преимущество этого подхода заключается в том, что прокси-сервер может кешировать результат и при необходимости возвращать этот результат другому клиенту. Иными словами, веб-прокси может реализовать общий кеш. С таким большим количеством документов, генерируемых на лету, сервер обычно предоставляет документ по частям, инструктируя клиента кешировать только те части, которые вряд ли изменятся при следующем запросе документа.

В дополнение к кешированию в браузерах и прокси интернет-провайдеры обычно также размещают кеши в своих сетях. Такие схемы в основном используются для уменьшения сетевого трафика (что хорошо для провайдера) и для повышения производительности (что хорошо для конечных пользователей). Однако при наличии нескольких кешей на пути запроса от клиента к серверу существует риск увеличения задержек, когда кеши не содержат запрошенную информацию.

Примечание 7.11 (дополнительно: совместное кеширование)

В качестве альтернативы построению иерархических кешей можно также организовать кеши для совместного развертывания, как показано на рис. 7.32. При совместном или распределенном кешировании всякий раз, когда происходит потеря кеша на веб-прокси, прокси-сервер сначала проверяет количество соседних прокси-серверов, чтобы определить, содержит ли один из них запрошенный документ. Если такая проверка не пройдена, прокси-сервер перенаправляет запрос на веб-сервер, ответственный за документ. В более традиционных настройках эта схема в основном используется с веб-кешами, принадлежащими к той же организации или учреждению.

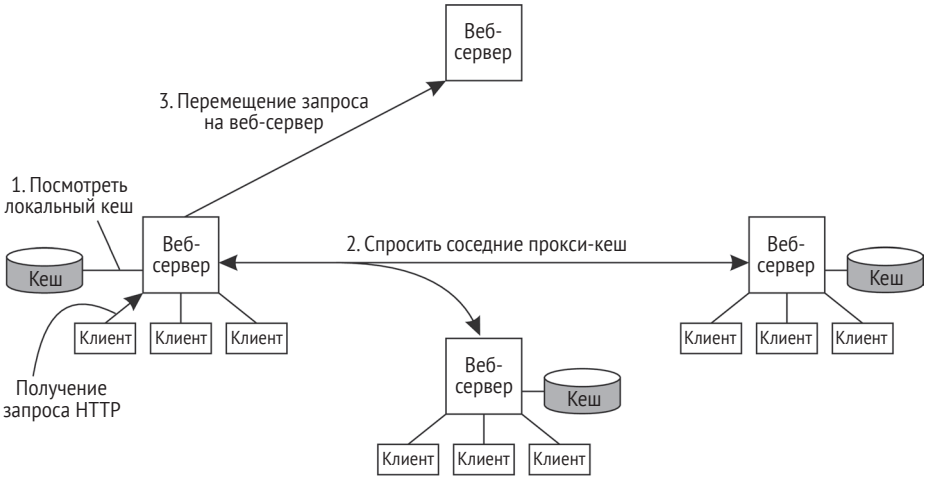


Рис. 7.32 ❖ Принцип кооперативного кеширования

В работе [Wolman et al., 1999] показано, что совместное кеширование может быть эффективным только для относительно небольших групп клиентов (порядка десятков тысяч пользователей). Однако такие группы также могут обслуживаться с по-

мощью одного прокси-кеша, что намного дешевле с точки зрения связи и использования ресурсов.

Тем не менее в исследовании десятилетия спустя в [Wendell and Freedman, 2011] показано, что в высокодецентрализованной системе совместное кеширование действительно оказывается очень эффективным. Эти исследования не обязательно противоречат друг другу: в обоих случаях делается вывод, что эффект совместного кеширования сильно зависит от требований клиентов.

Сравнение иерархического и кооперативного кеширований, выполненное в [Rodriguez et al., 2001], дает понять, что существуют различные компромиссы. Например, поскольку кооперативные кеши обычно связаны через высокоскоростные каналы, время передачи, необходимое для извлечения документа, намного меньше, чем для иерархического кеша. Кроме того, как и следовало ожидать, требования к хранилищу являются менее строгими для кооперативных кешей, чем иерархические.

В веб-сети были развернуты различные протоколы согласованности кеша. Чтобы гарантировать, что документ, возвращаемый из кеша, является непротиворечивым, некоторые веб-прокси сначала отправляют на сервер условный HTTP-запрос с дополнительным заголовком запроса If-Modied-Since (изменялся ли до этого), определяя время последнего изменения, связанное с кешированным документом. Только если документ был изменен с того времени, сервер вернет весь документ. В противном случае веб-прокси может просто вернуть свою кешированную версию запрашивающему локальному клиенту, что соответствует протоколу на основе извлечения.

К сожалению, эта стратегия требует, чтобы прокси связывался с сервером для каждого запроса. Для повышения производительности за счет более слабой согласованности широко используемый веб-прокси Squid [Wessels, 2004] назначает срок действия T_{expire}, который зависит от того, как давно документ был изменен в последний раз при его кешировании. В частности, если $T_{last\_modified}$ – это время последнего изменения документа (записанное его владельцем), а T_{cached} – это время его кеширования, то

$$T_{expire} = \alpha(T_{cached} - T_{last\_modified}) + T_{cached},$$

где $\alpha = 0{,}2$ (это значение было получено из практического опыта). До T_{expire} документ считается действительным, и прокси не будет связываться с сервером. По истечении времени прокси-сервер запрашивает у сервера новую копию, если она не была изменена. Отметим, что Squid также позволяет ограничивать срок истечения минимальным и максимальным временем.

Альтернативой протоколу на основе извлечения является уведомление сервером прокси-серверов о том, что документ был изменен, отправляя недействительную информацию. Проблема этого подхода для веб-прокси заключается в том, что серверу может потребоваться отслеживать большое количество прокси, что неизбежно приводит к проблеме масштабируемости. Тем не менее благодаря объединению аренды и аннулирования состояние, которое должно поддерживаться на сервере, может поддерживаться в допустимых пределах. Обратите внимание, что это состояние в значительной степени определяется временем истечения срока, установленным для аренды: чем оно меньше, тем меньше кешей сервер должен отслеживать. Вместе с тем

протоколы аннулирования для кешей веб-прокси практически не применяются. Сравнение политик согласованности веб-кеширования можно найти в [Cao and Ozsu, 2002]. Их вывод заключается в том, что разрешение серверу отправлять недействительные результаты может превзойти любой другой метод с точки зрения пропускной способности и предполагаемой задержки клиента, при этом сохраняя кешированные документы в соответствии с документами на исходном сервере.

Наконец, мы должны также упомянуть, что было проведено много исследований, чтобы выяснить, каковы лучшие стратегии замены кеша. Существует множество предложений, но в целом простые стратегии замены, такие как вытеснение наименее использованного объекта, работают достаточно хорошо. Углубленный обзор стратегий замены представлен в [Podling and Boszormenyi, 2003]; [Ali et al., 2011] предоставляет более свежий обзор, который также включает в себя методы предварительной выборки через интернет.

По мере того как важность интернета продолжает расти как средства, с помощью которого организации могут представлять себя и напрямую взаимодействовать с конечными пользователями, мы наблюдаем сдвиг между поддержанием содержания веб-сайта и обеспечением легкого и постоянного доступа к сайту. Это различие проложило путь для **сетей доставки контента** (content delivery networks, CDN). Основная идея, лежащая в основе CDN, заключается в том, что они действуют как служба веб-хостинга, обеспечивая инфраструктуру для распространения и репликации веб-документов нескольких сайтов через интернет. Размер инфраструктуры может быть впечатляющим. Например, по данным на 2016 год, программа Akamai имеет более 200 000 серверов, распределенных по 120 странам.

Размер CDN требует, чтобы размещенные документы автоматически распространялись и реплицировались. В большинстве случаев крупномасштабная CDN организована по линиям контура управления с обратной связью, как показано на рис. 7.33 и подробно описано в [Sivasubramanian et al., 2004b].

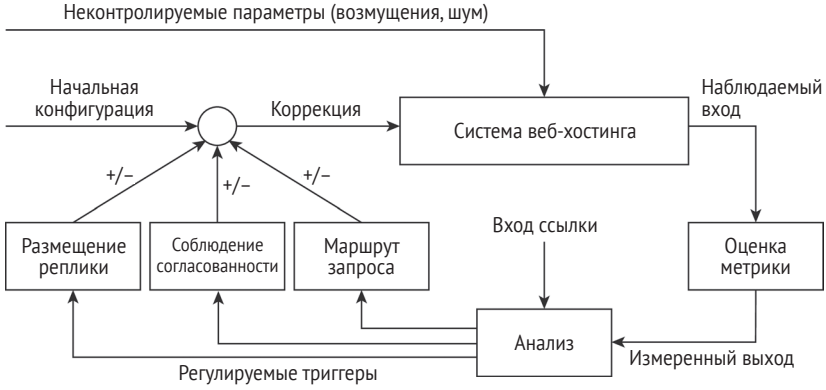


Рис. 7.33 ❖ Общая организация CDN как системы управления с обратной связью

Существуют три различных вида аспектов, связанных с репликацией в системах веб-хостинга: оценка метрик, запуск адаптации и принятие соответствующих мер. Последние можно подразделить на решения о размещении реплик, соблюдении согласованности и маршрутизации клиентских запросов. Далее мы кратко остановимся на каждом из них.

Интересным аспектом CDN является то, что им необходимо найти компромисс между многими аспектами, когда речь идет о размещении реплицированного контента. Например, время доступа к документу может быть оптимальным, если документ массово реплицируется, но в то же время это влечет за собой финансовые затраты, а также затраты с точки зрения использования полосы пропускания для распространения обновлений. По большому счету, есть много предложений для оценки того, насколько хорошо работает CDN. Эти предложения могут быть сгруппированы в несколько классов.

Во-первых, существуют *метрики задержки*, по которым измеряется время для выполнения действия, например выборки документа. Как ни банально это может показаться, оценка задержек становится сложной, когда, например, процессу, принимающему решение о размещении реплик, необходимо знать задержку между клиентом и некоторым удаленным сервером. Как правило, необходимо будет развернуть алгоритм глобального позиционирования узлов, как описано в главе 6.

Вместо оценки задержки и может быть более важно измерить доступную полосу пропускания между двумя узлами. Эта информация особенно важна, когда необходимо передать большие документы, так как в подобном случае отклик системы в значительной степени определяется временем передачи документа. Существуют различные инструменты для измерения доступной полосы пропускания, но во всех случаях оказывается, что точных измерений трудно достичь (см. также [Strauss et al., 2003], [Shriram and Kaur, 2007], [Chaudhari and Biradar, 2015] и [Atxutegi et al., 2016]).

Другой класс состоит из *пространственных метрик*, которые в основном измеряют расстояния между узлами с точки зрения количества скачков маршрутизации на уровне сети или скачков между автономными системами. Опять же, определение количества скачков между двумя произвольными узлами может быть очень трудным, а может даже не коррелировать с задержкой [Huffaker et al., 2002]. Более того, простой просмотр таблиц маршрутизации не будет работать, когда развернуты такие низкоуровневые методы, как **многопротокольная коммутация по меткам** (multi-protocol label switching MPLS). MPLS нарушает маршрутизацию на уровне сети, используя методы виртуальных каналов для немедленной и эффективной пересылки пакетов к месту назначения (см. также [Guichard et al., 2005]). Пакеты могут следовать по совершенно другим маршрутам, чем те, которые указаны в таблицах маршрутизаторов сетевого уровня.

Третий класс состоит из показателей использования сети, которые чаще всего влекут за собой требуемую пропускную способность. Вычисление потребляемой полосы пропускания с точки зрения количества байтов для передачи, как правило, не вызывает затруднений. Однако, чтобы сделать это правильно, необходимо принять во внимание, как часто документ читается, как часто онобновляется и как часто тиражируется.

Метрики согласованности говорят нам, в какой степени реплика отклоняется от своей основной копии. Мы уже широко обсуждали, как можно измерить согласованность в контексте непрерывной согласованности [Yu and Vahdat, 2002].

Наконец, *финансовые показатели* образуют еще один класс для оценки эффективности работы CDN. Хотя это и не технический вопрос, учитывая, что большинство CDN работают на коммерческой основе, ясно, что во многих случаях финансовые показатели будут решающими. Кроме того, финансовые показатели тесно связаны с реальной инфраструктурой интернета. Например, большинство коммерческих CDN размещают серверы на границе интернета, а это означает, что они нанимают ресурсы у интернет-провайдеров, непосредственно обслуживающих конечных пользователей. На этом этапе бизнес-модели переплетаются с технологическими проблемами, и эта область не совсем прозрачна. Существует лишь немного материалов о связи между финансовыми показателями и технологическими проблемами [Janiga et al., 2001].

Из этих примеров становится ясно, что простое измерение производительности CDN или даже оценка его производительности сама по себе может быть чрезвычайно сложной задачей. На практике для коммерческих CDN проблема, которая действительно имеет значение, заключается в том, могут ли они соответствовать соглашениям об уровне обслуживания, заключенным с клиентами. Эти соглашения часто формулируются просто с точки зрения того, как быстро должны обслуживаться клиенты. Затем CDN должен убедиться, что эти соглашения выполнены.

Другой вопрос, который необходимо решить, – это когда и как должны инициироваться адаптации. Простая модель состоит в том, чтобы периодически оценивать метрики и впоследствии по мере необходимости принимать меры. Такой подход часто встречается на практике. Специальные процессы, расположенные на серверах, собирают информацию и периодически проверяют изменения.

Примечание 7.12 (дополнительно: резкое увеличение обращений)

Основным недостатком периодической оценки является то, что могут быть пропущены внезапные изменения. Одним из видов внезапного изменения, которому уделяется значительное внимание, является **резкое увеличение обращений** (flash crowd). Увеличение обращений – это законный внезапный всплеск запросов на конкретный веб-документ. Во многих случаях этот тип пакетов может привести к остановке всего сервиса, что, в свою очередь, приведет к каскаду сбоев в обслуживании.

Обращение с этим явлением может быть затруднено. Одним из решений является массовая репликация веб-сайта, и как только скорость запросов начинает быстро увеличиваться, запросы следует перенаправлять на реплики для загрузки мастер-копий. Этот тип сверхпоставки – очевидно, не тот путь, по которому следует идти. Вместо этого необходимо динамически создавать реплики, когда спрос возрастает, и начинать перенаправлять запросы, когда ситуация становится сложной. Теперь, когда должным образом работают облачные вычисления, в сочетании с тем фактом, что клонирование виртуальных машин является относительно простым, успешное реагирование на внезапные изменения требований запросов практиче-

ски осуществимо. Кроме того, как мы обсудим ниже, хорошо себя зарекомендовало использование методов доставки контента по сетям.

Вместе с тем было бы даже лучше, чем реагировать, предсказывать всплески, однако на самом деле это очень сложно, если не практически невозможно. На рис. 7.34 показаны следы доступа для трех разных веб-сайтов, которые пострадали от внезапного всплеска запросов.

В качестве ориентира на рис. 7.34a показаны обычные трассировки доступа, охватывающие два дня. Есть также несколько очень сильных пиков, но в остальном ничего шокирующего не происходит. В отличие от этого, на рис. 7.34b показан двухдневный след с четырьмя внезапными скоплениями всплесков. Там присутствует еще некоторая закономерность, которая может быть обнаружена через некоторое время, чтобы можно было принять меры. Тем не менее ущерб может быть нанесен до достижения этой точки.

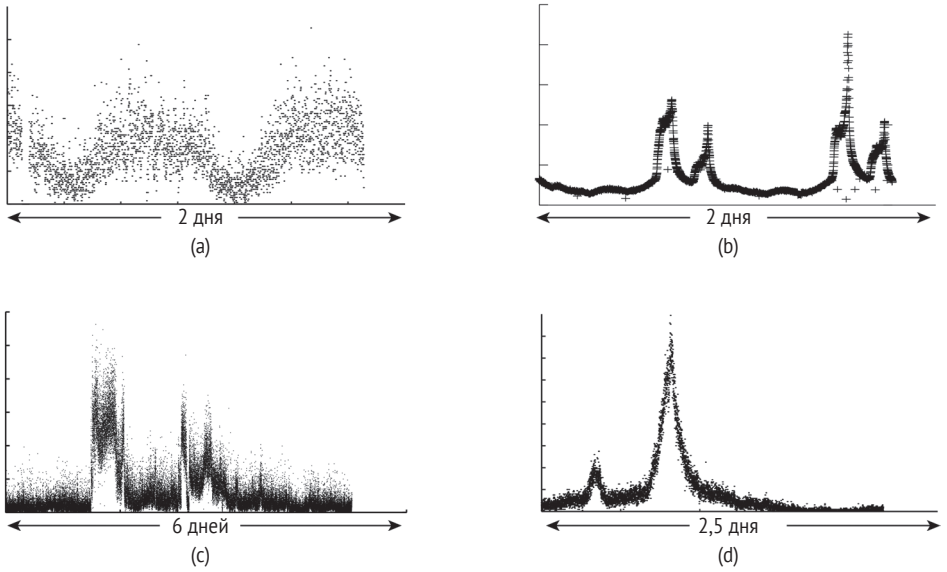


Рис. 7.34 ❖ Один нормальный и три различных образца доступа, отражающих поведение всплесков (адаптировано из [Baryshnikov et al., 2005])

Рисунок 7.34c показывает след, охватывающий шесть дней, по крайней мере с двумя всплесками. В этом случае у любого предиктора будет серьезная проблема, поскольку оказывается, что оба увеличения частоты запросов происходят практически мгновенно. Наконец, на рис. 7.34d показана ситуация, в которой первый пик, вероятно, не должен вызывать адаптаций, но второй, очевидно, должен. С таким типом поведения можно достаточно разобраться с помощью анализа времени выполнения.

Как уже упоминалось, существует, по сути, только три (связанные) возможности, которые можно предпринять для изменения поведения службы вебхостинга: изменение размещения реплик, изменение обеспечения согласованности и принятие решения о том, как и когда перенаправлять клиентские запросы. Мы уже подробно обсудили первые две меры. Перенаправление

клиентских запросов заслуживает еще большего внимания. Прежде чем мы обсудим некоторые из компромиссов, давайте сначала рассмотрим, как согласованность и репликация решаются на практике, рассматривая ситуацию программы Akamai [Dilley et al., 2002; Nygren et al., 2010].

Основная идея заключается в том, что каждый веб-документ состоит из основной страницы HTML (или XML), в которую были встроены несколько других документов, таких как изображения, видео и аудио. Для отображения всего документа необходимо, чтобы внедренные документы выбирались также браузером пользователя. Предполагается, что эти встроенные документы редко изменяются, поэтому имеет смысл их кешировать или копировать.

На каждый встроенный документ обычно ссылаются через URL. Однако в CDN Akamai такой URL-адрес изменяется так, что он ссылается на виртуального призрака, который является ссылкой на реальный сервер в CDN. URL также содержит имя хоста исходного сервера по причинам, которые мы объясним далее. Модифицированный URL-адрес разрешается так, как показано на рис. 7.35.

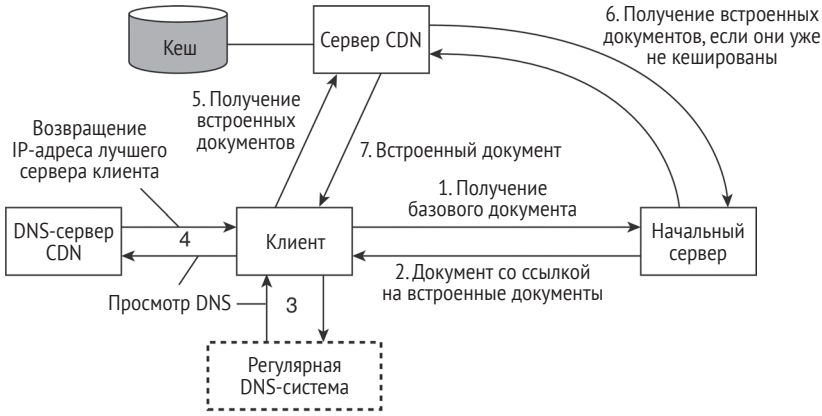


Рис. 7.35 ❖ Принцип работы CDN Akamai

Имя виртуального призрака включает DNS-имя, такое как ghosting.com, которое разрешается обычной системой имен DNS на DNS-сервере CDN (результат шага 3). Каждый такой DNS-сервер отслеживает серверы, расположенные близко к клиенту. Для этого может быть использована любая из метрик близости, которые мы обсуждали ранее. По сути, DNS-серверы CDN перенаправляют клиента на сервер реплики, который лучше всего подходит для этого клиента (шаг 4), что может означать ближайший, наименее загруженный или комбинацию нескольких таких метрик (фактическая политика перенаправления является внутренней политикой пользователя).

Наконец, клиент пересылает запрос на внедренный документ на выбранный сервер CDN. Если на этом сервере еще нет документа, он извлекает его с исходного веб-сервера (показанного на шаге 6), кеширует его локально и затем передает клиенту. Если документ уже был в кеше сервера CDN, он может быть немедленно возвращен. Обратите внимание, что для извлечения

встроенного документа сервер реплики должен иметь возможность отправлять запрос исходному серверу, поэтому его имя хоста также содержится в URL-адресе встроенного документа.

Интересным аспектом этой схемы является простота, с помощью которой можно обеспечить согласованность документов. Ясно, что всякий раз, когда основной документ изменяется, клиент всегда сможет получить его с исходного сервера. В случае встроенных документов необходимо придерживаться другого подхода, поскольку эти документы, в принципе, извлекаются с ближайшего сервера реплики. С этой целью URL для встроенного документа не только относится к специальному имени хоста, которое в конечном итоге приводит к DNS-серверу CDN, но также содержит уникальный идентификатор, который изменяется каждый раз, когда изменяется встроенный документ. По сути, этот идентификатор изменяет имя внедренного документа. Как следствие, когда клиент перенаправляется на определенный сервер CDN, этот сервер не найдет поименованный документ в своем кеше и, таким образом, получит его с исходного сервера. Старый документ в конечном итоге будет удален из кеша сервера, поскольку на него больше нет ссылок.

Этот пример уже показывает важность перенаправления клиентских запросов. В принципе, путем правильного перенаправления клиентов CDN может сохранять контроль над производительностью, воспринимаемой клиентом, а также с учетом глобальной производительности системы, например избегая отправки запросов на сильно загруженные серверы. Эти так называемые **политики адаптивного перенаправления** (adaptive redirection policies) могут применяться, когда информация о текущем поведении системы предоставляется процессам, которые принимают решения о перенаправлении. Это частично возвращает нас к методам оценки метрик, которые обсуждались ранее.

Помимо различных политик, важный вопрос – является ли переадресация запроса прозрачной для клиента или нет. По сути, существует только три метода перенаправления: передача обслуживания TCP, перенаправление DNS и перенаправление HTTP.

Мы уже обсуждали передачу обслуживания TCP. Этот метод применим только для кластеров серверов и не распространяется на глобальные сети.

Перенаправление DNS – это прозрачный механизм, благодаря которому клиент может полностью не знать, где находятся документы. Двухуровневое перенаправление Akamai является одним из примеров этой техники. Мы также можем напрямую развернуть DNS, чтобы вернуть один из нескольких адресов, как обсуждали ранее. Обратите, однако, внимание, что перенаправление DNS может применяться только ко всему сайту: имена отдельных документов не вписываются в пространство имен DNS.

Наконец, перенаправление HTTP является непрозрачным механизмом. Когда клиент запрашивает конкретный документ, ему может быть предоставлен альтернативный URL-адрес как часть ответного сообщения HTTP, на которое он затем перенаправляется. Важным наблюдением является то, что этот URL-адрес виден для браузера клиента. Фактически пользователь может решить добавить в закладки URL-адрес направляемого, что может сделать политику перенаправления бесполезной.

До этого момента мы в основном сосредоточивались на кешировании и репликации статичного веб-контента. На практике мы видим, что интернет все чаще предлагает более динамически генерируемый контент, а также расширяется и предлагает сервисы, которые могут вызываться удаленными приложениями. В этих ситуациях мы также видим, что кеширование и репликация могут значительно помочь в улучшении общей производительности, хотя методы достижения таких улучшений более тонкие, чем те, которые мы обсуждали до сих пор (см. также [Conti et al., 2005]).

При рассмотрении вопроса о повышении производительности веб-приложений с помощью кеширования и репликации все усложняется тем, что можно развернуть несколько решений, но ни одно из них не будет лучшим. Давайте рассмотрим ситуацию с крайним сервером, как показано на рис. 7.36 (см. также [Sivasub-ramanian et al., 2007]). В этом случае мы предполагаем, что в CDN каждый размещенный сайт имеет исходный сервер, который действует как уполномоченный сайт для всех операций чтения и обновления. Крайний (пограничный) сервер используется для обработки клиентских запросов и имеет возможность хранить (частичную) информацию так же, как и на исходном сервере.

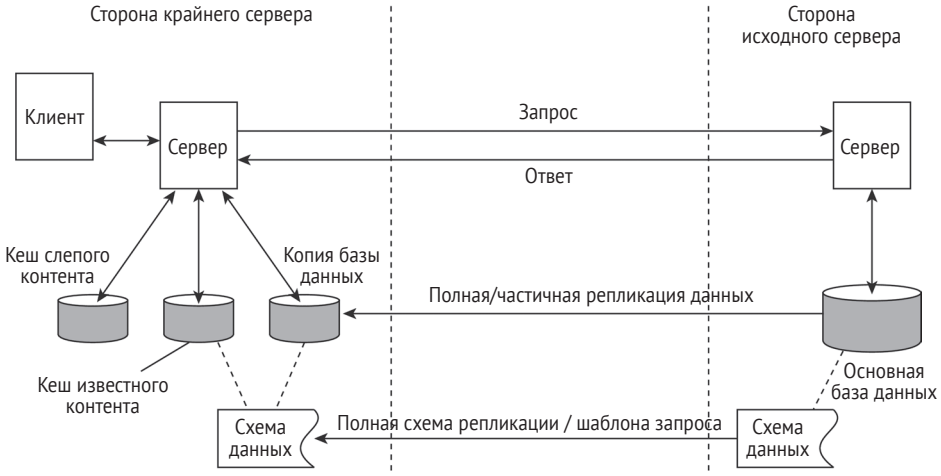


Рис. 7.36 ❖ Альтернативы для кеширования и репликации с веб-приложениями

Напомним, что архитектура крайнего сервера предусматривает запрос данных веб-клиентами через крайний сервер, в свою очередь получающий информацию от исходного сервера, связанного с конкретным веб-сайтом, на который ссылается клиент. Как показано на рис. 7.36, мы предполагаем, что исходный сервер состоит из базы данных, которая динамически создает ответы. Хотя мы показали только один веб-сервер, обычно каждый сервер организован в соответствии с многоуровневой архитектурой, как мы обсуждали ранее. Крайний сервер теперь может быть организован примерно по следующим направлениям.

Во-первых, для повышения производительности мы можем решить применить полную репликацию данных, хранящихся на исходном сервере. Эта схема хорошо работает, когда коэффициент обновления низкий и когда запросы требуют расширенного поиска в базе данных. Как упомянуто выше, мы предполагаем, что все обновления выполняются на исходном сервере, который берет на себя ответственность за поддержание согласованного состояния реплик и крайних серверов. Таким образом, операции чтения могут выполняться на крайних серверах. Мы видим, что здесь повысить производительность за счет репликации не удастся, если коэффициент обновления высок, поскольку каждое обновление будет вызывать связь по глобальной сети, чтобы привести реплики в согласованное состояние. Как показано в [Sivasubramanian et al., 2004a], отношение чтение/обновление является определяющим фактором того, до какой степени база данных источника в глобальной области должна быть реплицирована.

Другой случай полной репликации – это когда запросы в основном сложные. В случае реляционной базы данных это означает, что для запроса требуется поиск и обработка нескольких таблиц, как это обычно происходит в случае операции соединения. В отличие от сложных запросов, простые запросы для ответа требуют обычно доступа только к одной таблице. В последнем случае может быть достаточно **частичной репликации** (partial replication), при которой на крайнем сервере сохраняется лишь подмножество данных.

Альтернативой частичной репликации является использование **кешей с учетом содержимого** (content-aware caches). Основная идея в этом случае заключается в том, что крайний сервер поддерживает локальную базу данных, которая теперь адаптирована к типу запросов, которые могут обрабатываться на исходном сервере. Для объяснения в полнофункциональной системе баз данных запрос будет работать с базой данных, в которой данные были организованы в таблицы, так что, например, избыточность минимизирована. Такие базы данных также называются **нормализованными** (normalized). В подобных базах данных любой запрос, который соответствует схеме данных, в принципе может быть обработан, хотя, возможно, и при значительных затратах. Благодаря кешам с поддержкой содержимого крайний сервер поддерживает базу данных, которая организована в соответствии со структурой запросов. Это означает предположение, что запросы придерживаются ограниченного числа шаблонов и что различные типы запросов, которые могут быть обработаны, также ограничены. В этих случаях при получении запроса крайний сервер сопоставляет запрос с доступными шаблонами и впоследствии просматривает свою локальную базу данных, чтобы, если это возможно, составить ответ. Если запрошенные данные недоступны, запрос перенаправляется на исходный сервер, после чего ответ кешируется перед возвратом его клиенту.

По сути, пограничный сервер проверяет, можно ли ответить на запрос данными, хранящимися локально. Это также называется **проверкой содержимого запроса** (query containment check). Обратите внимание, что такие данные хранились локально как ответы на ранее выполненные запросы. Этот подход работает лучше всего, когда запросы имеют тенденцию повторяться.

Часть сложности кеширования с учетом содержимого обусловлена тем фактом, что данные на крайнем сервере должны быть согласованными. Для этого исходному серверу необходимо знать, какие записи связаны с какими шаблонами, чтобы можно было надлежащим образом адресовать любое обновление записи или любое обновление таблицы, например отправив сообщение о недействительности на соответствующий крайний сервер. Еще один источник сложности связан с тем, что запросы по-прежнему необходимо обрабатывать на крайних серверах. Другими словами, для обработки запросов требуется значительная вычислительная мощность. Учитывая, что базы данных часто образуют узкое место в производительности на веб-серверах, могут потребоваться альтернативные решения. Наконец, кеширование результатов запросов, которые охватывают несколько таблиц (то есть когда запросы сложные), так что проверка выполнения запроса может быть выполнена эффективно, не тривиально. Причина в том, что организация результатов может сильно отличаться от организации таблиц, с которыми работал запрос.

Эти наблюдения приводят нас к третьему решению, а именно к **контент-слепому кешированию** (content-blind caching). Идея слепого кеширования контента чрезвычайно проста: когда клиент отправляет запрос на пограничный сервер, сервер сначала вычисляет уникальное значение хеш-функции для этого запроса. Используя это хеш-значение, он впоследствии просматривает в своем кеше, обрабатывал ли он этот запрос раньше. Если нет, запрос передается источнику, и результат кешируется перед возвратом клиенту. Если запрос был обработан раньше, ранее кешированный результат возвращается клиенту.

Основным преимуществом этой схемы является уменьшение вычислительных усилий, которое требуется от крайнего сервера по сравнению с подходами к базе данных, описанными выше. Однако слепое кеширование содержимого может быть расточительным с точки зрения хранения, поскольку кеши могут содержать гораздо больше избыточных данных по сравнению с кешированием с учетом содержимого или репликацией базы данных. Обратите внимание, что такая избыточность также усложняет процесс обновления кеша, поскольку исходному серверу может потребоваться точный учет того, какие обновления могут потенциально повлиять на результаты кешированных запросов. Эти проблемы могут быть смягчены, если предположить, что запросы могут соответствовать только ограниченному набору предопределенных шаблонов, как мы обсуждали выше.

7.7. Резюме

Существует две главные причины для репликации данных: повышение надежности распределенной системы и повышение производительности. Репликация создает проблему согласованности: всякий раз, когда реплика обновляется, эта реплика становится отличной от других. Чтобы обеспечить согласованность реплик, нам необходимо распространять обновления таким образом, чтобы не было отмечено временных несоответствий. К сожалению,

это может серьезно снизить производительность, особенно в крупных распределенных системах.

Единственное решение данной проблемы – немного ослабить согласованность. Существуют различные модели согласованности. Для непрерывной согласованности цель состоит в том, чтобы установить границы для числового отклонения между репликами, отклонения устаревания и отклонения в порядке операций.

Числовое отклонение – это величина, на которое реплики могут отличаться. Этот тип отклонения сильно зависит от приложения, но может, например, использоваться при тиражировании резервов. Отклонение устаревания относится ко времени, в течение которого реплика все еще считается согласованной, несмотря на то что обновления могли иметь место некоторое время назад. Отклонение устаревания часто используется для веб-кешей. Наконец, отклонение порядка относится к максимальному количеству предварительных записей, которые могут быть ожидающими на любом сервере без синхронизации с другими серверами реплики.

Упорядочение последовательности операций давно составляет основу для многих моделей согласованности. Существует множество вариантов, но среди разработчиков приложений преобладают лишь некоторые. Последовательная согласованность по существу обеспечивает семантику, которую программисты ожидают при параллельном программировании: все операции записи рассматриваются всеми в одном и том же порядке. Менее используемой, но все же актуальной является причинно-следственная согласованность, которая отражает то, что операции, которые потенциально зависят друг от друга, выполняются в порядке этой зависимости.

Более слабые модели согласованности учитывают последовательность операций чтения и записи. В частности, они предполагают, что каждая серия надлежащим образом заключена в квадратные скобки посредством сопутствующих операций над переменными синхронизации, такими как блокировки. Хотя это требует явных усилий со стороны программистов, данные модели, как правило, легче реализовать более эффективным способом, чем, например, чисто последовательную согласованность.

В отличие от этих моделей, ориентированных на данные, исследователи в области распределенных баз данных для мобильных пользователей определили ряд моделей согласованности, ориентированных на клиента. Такие модели не учитывают тот факт, что данные могут совместно использоваться несколькими пользователями, но вместо этого концентрируются на последовательности, которую должен предлагать отдельный клиент. Основное предположение состоит в том, что клиент со временем подключается к различным репликам, но такие различия должны быть прозрачными. По сути, клиент-ориентированные модели согласованности гарантируют, что всякий раз, когда клиент подключается к новой реплике, эта реплика обновляется с использованием данных, которыми этот клиент манипулировал ранее и которые могут находиться на других сайтах реплики.

Для распространения обновлений могут применяться разные методы. Необходимо различать, что именно распространяется, куда распространяются обновления и кем инициируется распространение. Мы можем принять

решение о распространении уведомлений, операций или состояния. Также не каждая реплика всегда должна обновляться немедленно. Какая реплика обновляется и в какое время, зависит от протокола распространения. Наконец, можно сделать выбор: отправлять ли обновления в другие реплики или реплика извлекает обновления из другой реплики.

Протоколы согласованности описывают конкретные реализации моделей согласованности. Что касается последовательной согласованности и ее вариантов, можно провести различие между первичными протоколами и протоколами реплицированной записи. В первичных протоколах все операции обновления пересылаются в первичную копию, которая впоследствии обеспечивает правильное упорядочение и пересылку обновления. В протоколах репликации и записи обновление направляется нескольким репликам одновременно. В этом случае правильное упорядочение операций часто становится более сложным.

Мы уделяем отдельное внимание кешированию и репликации в интернете и связанных с ним сетях доставки контента. Как выясняется, используя существующие серверы и службы, многие из методов, обсужденных ранее, могут быть легко реализованы с использованием соответствующих методов перенаправления. Особенно сложным является кеширование контента, когда задействованы базы данных, так как в этих случаях большая часть того, что возвращает веб-сервер, генерируется динамически. Однако даже тогда путем тщательного администрирования того, что уже было кешировано на самом краю, можно изобрести действенные и эффективные схемы кеширования.

Глава 8

Отказоустойчивость

Характерной особенностью распределенных систем, которая отличает их от систем с одним компьютером, является понятие частичного отказа: часть системы выходит из строя, а оставшаяся часть продолжает работать, и, возможно, правильно. Важной целью при проектировании распределенных систем является создание системы таким образом, чтобы она могла автоматически восстанавливаться после частичных сбоев, не оказывая серьезного влияния на общую производительность. В частности, всякий раз, когда происходит сбой, система должна продолжать работать приемлемым образом во время ремонта. Другими словами, распределенная система должна быть отказоустойчивой.

В этой главе мы более подробно рассмотрим методы обеспечения отказоустойчивости. После предоставления некоторой общей информации мы рассмотрим устойчивость процессов посредством группирования процессов. В этом случае несколько идентичных процессов взаимодействуют, создавая видимость одного логического процесса, чтобы гарантировать, что один или несколько из них могут выйти из строя без уведомления клиента. Особенно трудным моментом в группах процессов является достижение консенсуса между членами группы, в отношении которого должна выполняться запрашиваемая клиентом операция. К настоящему времени семейство протоколов Paxos является общепринятым, но относительно сложным алгоритмом, который мы объясняем, создавая его с нуля. Кроме того, мы тщательно изучаем случаи, в которых можно достичь консенсуса и при каких обстоятельствах.

Достижение отказоустойчивости и надежная связь тесно связаны. Наряду с надежной клиент-серверной связью мы обращаем внимание на надежную групповую связь и, в частности, на атомарную многоадресную рассылку. В последнем случае сообщение доставляется или всем неработающим процессам в группе, или вообще ни одному. Атомарная многоадресная передача значительно упрощает разработку отказоустойчивых решений.

Атомарность – это свойство, которое важно во многих приложениях. В этой главе мы обращаем внимание на так называемые протоколы распределенной фиксации, по которым группа процессов выполняет либо совместную фиксацию своей локальной работы, либо коллективно прерывает работу и возвращает систему к предыдущему состоянию.

Наконец, мы рассмотрим, как восстанавливается система после сбоя. В частности, разбираем, когда и как следует сохранять состояние распреде-

ленной системы, чтобы впоследствии разрешить восстановление до этого состояния.

8.1. Введение в отказоустойчивость

Отказоустойчивость была предметом многочисленных исследований в области компьютерных наук. В этом разделе мы начнем с представления основных понятий, связанных со сбоями обработки, а затем обсудим модели сбоя. Ключевым методом обработки сбоев является избыточность, которая также обсуждается. Для получения более общей информации об отказоустойчивости в распределенных системах смотрите, например, [Jalote, 1994; Shooman, 2002] или [Koren and Krishna, 2007].

Базовые концепции

Чтобы понять роль отказоустойчивости в распределенных системах, нам прежде всего необходимо более внимательно посмотреть, что на самом деле означает отказ распределенной системы. Отказоустойчивость системы тесно связана с так называемой надежностью системы. Надежность – это термин, который охватывает ряд полезных требований для распределенных систем, включая следующие [Kopetz and Verissimo, 1993]:

- доступность;
- надежность;
- безопасность;
- ремонтопригодность.

Доступность (availability) определяется как свойство готовности системы к немедленному использованию. В общем, это относится к вероятности того, что система работает правильно в любой момент и доступна для выполнения своих функций от имени своих пользователей. Другими словами, система высокой доступности – это система, которая, скорее всего, будет работать в данный момент времени.

Надежность (reliability) относится к свойству, которое обеспечивает продолжительную работу системы без сбоев. В отличие от доступности, надежность определяется с точки зрения временного интервала, а не момента времени. Высоконадежная система – это система, которая, скорее всего, будет продолжать работать без перерыва в течение относительно длительного периода времени. Это тонкое, но важное отличие по сравнению с доступностью. Если система выходит из строя в среднем на одну, казалось бы, случайную миллисекунду каждый час, она имеет доступность более 99,9999 %, но все еще ненадежна. Аналогичным образом система, которая никогда не выходит из строя, но закрывается на две недели в августе каждого года, обладает высокой надежностью, но ее доступность составляет всего 96 %. Это два разных свойства.

Безопасность (safety) относится к ситуации, когда система временно не работает должным образом, не происходит никаких катастрофических со-

бытий. Например, многие системы управления технологическим процессом, например используемые для управления атомными электростанциями или отправки людей в космос, обязаны обеспечить высокую степень безопасности. Если такие системы управления временно выходят из строя даже на очень короткий промежуток времени, последствия могут быть катастрофическими. Многие примеры из прошлого (и, вероятно, еще многие в будущем) показывают, как сложно создавать безопасные системы.

Наконец, **ремонтопригодность** (maintainability) определяет, насколько легко может быть восстановлена отказавшая система. Высокая ремонтопригодность системы показывает также высокий уровень доступности, особенно если отказ может быть обнаружен и устранен автоматически. Однако, как мы увидим далее в этой главе, автоматическое устранение неисправности – задача не из простых.

Примечание 8.1 (дополнительная информация: традиционные показатели)

Мы можем быть немного более точными, когда дело доходит до описания доступности и надежности. Формально доступность $A(t)$ компонента во временном интервале $[0, t)$ определяется как среднее времени, в течение которого компонент функционировал правильно в продолжение этого интервала. **Долгосрочная доступность** (long-term availability) компонента A определяется как $A(\infty)$.

Аналогично надежность $R(t)$ компонента во временном интервале $[0, t)$ формально определяется как условная вероятность того, что он функционировал правильно в течение этого интервала, учитывая, что он функционировал правильно в момент времени $T = 0$. Следуя [Pradhan, 1996], для определения $R(t)$ рассмотрим систему из N одинаковых компонентов. Пусть $N_0(t)$ обозначает количество правильно работающих компонентов в момент времени t, а $N_1(t)$ – количество неисправных компонентов. Тогда ясно, что

$$R(t) = \frac{N_0(t)}{N} = 1 - \frac{N_1(t)}{N} = \frac{N_0(t)}{N_0(t) + N_1(t)}.$$

Скорость отказа компонентов может быть выражена как производная $dN_1(t)/dt$. Разделив эту производную на количество правильно работающих компонентов за время t, получаем **функцию интенсивности отказов** (failure rate function) $z(t)$:

$$z(t) = \frac{1}{N_0(t)} \frac{dN_1(t)}{dt}.$$

Из

$$\frac{dR(t)}{dt} = -\frac{1}{N} \frac{dN_1(t)}{dt}$$

следует, что

$$z(t) = \frac{1}{N_0(t)} \frac{dN_1(t)}{dt} = -\frac{N}{N_0(t)} \frac{dR(t)}{dt} = -\frac{1}{R(t)} \frac{dR(t)}{dt}.$$

Если мы сделаем упрощенное предположение, что компонент не стареет (и, следовательно, по существу не имеет фазы износа), его частота отказов будет постоянной, то есть $z(t) = z$, подразумевая, что

$$\frac{dR(t)}{dt} = -zR(t).$$

Поскольку $R(0) = 1$, мы получаем

$R(t) = e^{-zt}$.

Другими словами, если мы игнорируем старение компонента, то видим, что постоянная частота отказов приводит к экспоненциальному распределению надежности, имеющему форму, показанную на рис. 8.1.

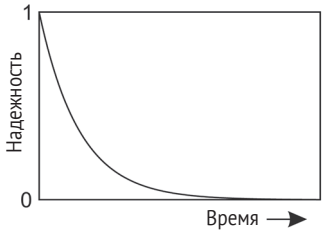


Рис. 8.1 ❖ Надежность компонента с постоянной частотой отказов

Традиционно отказоустойчивость связана со следующими тремя показателями:

- **среднее время до отказа** (Mean Time To Failure, MTTF): среднее время до отказа компонента;
- **среднее время восстановления** (Mean Time To Repair, MTTR): среднее время, необходимое для восстановления компонента;
- **среднее время между отказами** (Mean Time Between Failures, MTBF): просто MTTF + MTTR.

Обратите внимание, что

$$A = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF + MTTR}.$$

Кроме того, эти метрики имеют смысл, только если у нас есть точное представление о том, что на самом деле является отказом. Как мы увидим позже, выявление возникновения отказа может быть не столь очевидным.

Зачастую надежные системы также необходимы для обеспечения высокой степени безопасности, особенно когда речь идет о таких вопросах, как **целостность** (integrity). Мы обсудим безопасность в следующей главе.

Говорят, что система **выходит из строя** (fail), когда она не может выполнить свои обещания. В частности, если распределенная система предназначена для предоставления своим пользователям ряда услуг, система перестала работать, когда одна или несколько из этих услуг не могут быть ею (полностью) предоставлены. **Ошибка** (error) – это часть состояния системы, которая может привести к сбою. Например, при передаче пакетов по сети следует ожидать, что некоторые пакеты были повреждены при поступлении в приемник. Повреждение в этом контексте означает, что получатель может неправильно определить значение бита (например, считывание 1 вместо 0) или даже быть неспособным обнаружить, что что-то пришло.

Причина ошибки называется **неисправностью** (fault). Очевидно, важно выяснить, что вызвало ошибку. Например, неправильная или плохая среда

передачи может легко привести к повреждению пакетов. В этом случае относительно легко устранить неисправность. Однако ошибки передачи также могут быть вызваны плохими погодными условиями, например в беспроводных сетях. Изменение погоды для уменьшения или предотвращения ошибок несколько сложнее.

В качестве другого примера сбойная программа явно является ошибкой, которая могла произойти из-за того, что программа ввела ветку кода, содержащую программный дефект (то есть программную ошибку). Причиной такой ошибки является, как правило, программист. Другими словами, программист виноват в ошибке (программная ошибка), что, в свою очередь, приводит к сбою программы.

Создание надежных систем тесно связано с контролем неисправностей. Как объяснено в [Avizienis et al., 2004], можно провести различие между предупреждением, устойчивостью, устранением и прогнозированием неисправностей. Для наших целей наиболее важной проблемой является отказоустойчивость, означающая, что система может предоставлять свои услуги даже при наличии неисправностей. Например, применяя коды с исправлением ошибок для передачи пакетов, можно в определенной степени допустить относительно плохие линии передачи и снизить вероятность того, что ошибка (поврежденный пакет) может привести к сбою.

Неисправности обычно классифицируются как вре́менные, периодические или постоянные. **Вре́менная неисправность** (transient faults) происходит один раз, а затем исчезает. Неисправность исчезает, если операция повторяется. Птица, летящая через луч микроволнового передатчика, может привести к потере битов в некоторой сети. Если время передачи истекло и делается очередная попытка, вероятно, система будет снова работать.

Периодическая неисправность (intermittent fault) возникает, затем она исчезает сама по себе, потом вновь появляется и т. д. Ненадежный контакт на разъеме часто вызывает периодическую неисправность. Периодические неисправности вызывают значительные затруднения, поскольку их трудно диагностировать. Как правило, при хорошей диагностике система работает нормально.

Постоянная неисправность (permanent fault) – это та неисправность, которая продолжает существовать, пока неисправный компонент не будет заменен. Сгоревшие микросхемы, программные ошибки и сбои на головке диска являются примерами постоянных неисправностей.

Модели отказов

Система, которая выходит из строя, не обеспечивает адекватного обслуживания, для которого она была разработана. Если мы рассмотрим распределенную систему как совокупность серверов, которые взаимодействуют друг с другом и со своими клиентами, то неадекватное предоставление услуг означает, что серверы, каналы связи или, возможно, и то, и другое не выполняют то, что должны делать. Однако неисправный сервер не всегда может быть неисправностью, которую мы ищем. Если такой сервер зависит

от других серверов для адекватного предоставления своих услуг, возможно, необходимо искать причину неисправности в другом месте.

Подобные отношения зависимости в распределенных системах появляются в изобилии. Выходящий из строя диск может осложнить жизнь файловому серверу, который разработан для обеспечения высокодоступной файловой системы. Если такой файловый сервер является частью распределенной базы данных, может быть поставлена на карту надлежащая работа всей базы данных, поскольку только часть ее данных может быть доступна.

Чтобы лучше понять, насколько серьезен на самом деле отказ, было разработано несколько классификационных схем. Одна из таких схем показана на рис. 8.2 и основана на схемах, описанных в [Cristian, 1991] и [Hadzilacos and Toueg, 1993].

Тип отказа	Описание поведения сервера
Аварийный отказ	Останавливается, но работает правильно до остановки
Пропуск при отказе *Пропуск приема* *Пропуск отправки*	Не удается ответить на входящие запросы Не удается получить входящие сообщения Не удается отправить сообщение
Сбой синхронизации	Ответ находится за пределами указанного интервала времени
Ошибка ответа *Ошибка значения* *Сбой перехода состояния*	Ответ неверен Ошибка значения ответа Отклоняется от правильного потока управления
Произвольный отказ	Может привести к произвольным ответам в произвольные моменты времени

Рис. 8.2 ❖ Различные типы отказов

Аварийный отказ (crash failure) происходит, когда сервер преждевременно останавливается, но работал правильно, пока не остановился. Важным аспектом аварийных отказов является то, что после остановки сервера от него больше нет никаких ответов. Типичным примером такого отказа является операционная система, которая останавливается, и для его устранения есть только одно решение: перезагрузка операционной системы. Многие системы персональных компьютеров страдают от таких сбоев настолько часто, что люди начинают считать это нормальным, а перемещение кнопки сброса от задней части компьютера к передней части имело свои причины. Возможно, однажды ее снова можно будет переместить назад или вообще убрать.

Пропуск при отказе (omission failure) возникает, когда сервер не отвечает на запрос. Что-то при этом может пойти не так. В случае **пропуска приема** (`receive`-omission faillure) сервер, возможно, никогда не получал запрос. Обратите внимание, что вполне может быть так, что соединение между клиентом и сервером было правильно установлено, но поток не прослушивал входящие запросы. Кроме того, ошибка пропуска приема, как правило, не влияет на текущее состояние сервера, поскольку сервер не знает ни о каком отправленном ему сообщении.

Аналогично **отказ отправки пропуска** (send-omission failure) происходит, когда сервер выполнил свою работу, но каким-то образом не удается отправить ответ. Такой отказ может произойти, например, когда буфер отправки переполнен, а сервер не подготовлен к такой ситуации. Обратите внимание, что, в отличие от ошибки пропуска приема, сервер теперь может находиться в состоянии, в котором он только что завершил обслуживание для клиента. Как следствие, если отправка его ответа не удалась, сервер должен быть готов к тому, чтобы клиент повторил свой предыдущий запрос.

Другие типы пропусков, не связанных со связью, могут быть вызваны ошибками программного обеспечения, такими как бесконечные циклы или неправильное управление памятью, из-за которых сервер «зависает».

Другой класс отказов связан с синхронизацией. **Отказ временно́й синхронизации** (state-transition failure) возникает, когда ответ находится за пределами указанного интервала в реальном времени. Например, в случае потокового видео предоставление данных слишком рано может легко вызвать проблемы у получателя, если не хватает места в буфере для хранения всех входящих данных. Более распространенным, однако, является то, что сервер отвечает слишком поздно, и в этом случае происходит сбой производительности.

Серьезным типом сбоя является **ошибка ответа** (response failure), из-за которого ответ сервера просто неверен. Могут возникнуть два типа ошибок ответа. В случае ошибки значения сервер просто предоставляет неправильный ответ на запрос. Например, поисковая система, которая систематически возвращает веб-страницы, не относящиеся ни к одному из используемых поисковых терминов, не работает.

Другой тип ошибки ответа известен как **ошибка перехода состояния** (state-transition failure). Этот вид отказа происходит, когда сервер неожиданно реагирует на входящий запрос. Например, если сервер получает сообщение, которое он не может распознать, происходит сбой перехода состояния, если не было принято никаких мер для обработки таких сообщений. В частности, неисправный сервер может неправильно выполнять действия по умолчанию, которые он никогда не должен был инициировать.

Наиболее серьезными являются **произвольные отказы** (**arbitrary failures**), известные также как **византийские отказы** (Byzantine failures). Фактически, когда происходят **произвольные отказы**, клиенты должны быть готовы к худшему. В частности, может случиться так, что сервер производит вывод, который он никогда не должен был производить, но который не может быть обнаружен как неправильный.

Византийские отказы были впервые проанализированы в [Pease et al., 1980] и [Lamport et al., 1982]. Мы вернемся к таким отказам ниже.

Примечание 8.2 (дополнительная информация: ошибки при пропуске и комиссии)

Стало привычным ассоциировать возникновение византийских отказов со злонамеренно работающими процессами. Термин «византийский» относится к Византийской империи, времени (330–1453 гг.) и территориям (Балканы и современная Турция), на которых в правящих кругах были распространены бесконечные заговоры, интриги и несправедливость.

Однако, возможно, и не удастся обнаружить, был ли отказ на самом деле не слишком вредным или вредоносным. Работает ли сетевой компьютер с плохо спроектированной операционной системой, которая негативно влияет на производительность других компьютеров, нарушающих работу? В этом смысле лучше провести следующее различие, которое фактически исключает оценку:

- **ошибка пропуска** (omission failure) возникает, когда компонент не может выполнить действие, которое он должен был выполнить;
- **ошибка действия** (commission failure) возникает, когда компонент выполняет действие, которое он не должен был выполнять.

Это различие, введенное в [Mohan et. al., 1983], также иллюстрирует, что иногда бывает трудно разделить надежность и безопасность.

Многие из вышеупомянутых случаев имеют дело с ситуацией, когда процесс P больше не воспринимает какие-либо действия другого процесса Q. Однако может ли P сделать вывод, что Q действительно остановился? Чтобы ответить на этот вопрос, нам нужно провести различие между двумя типами распределенных систем:

1) в **асинхронной системе** (asynchronous system) не делается никаких предположений о скорости выполнения процесса или времени доставки сообщений. Следствием этого является то, что когда процесс P больше не воспринимает какие-либо действия от Q, он не может заключить, что Q потерпел крах. Вместо этого он может быть медленным, или его сообщения могут быть потеряны;
2) в **синхронной системе** (synchronous system) скорость выполнения процесса и время доставки сообщений ограничены. Это также означает, что когда Q больше не проявляет активности, как это ожидается, процесс P может по праву сделать вывод, что Q потерпел крах.

К сожалению, чисто синхронные системы существуют только в теории. С другой стороны, простое утверждение о том, что каждая распределенная система является синхронной, как мы видим на практике, было бы чрезмерно пессимистичным при проектировании распределенных систем и предположении, что они обязательно являются асинхронными. Вместо этого более реалистично предположить, что распределенная система является **частично синхронной** (partially synchronous): большую часть времени она ведет себя как синхронная система, но когда нет никаких ограничений по времени, она ведет себя как асинхронная система.

Другими словами, асинхронное поведение является исключением, что означает, что обычно мы можем использовать время простоя, чтобы сделать вывод о том, что процесс действительно завершился сбоем, однако иногда такое заключение будет неверным. На практике это означает, что нам необходимо разрабатывать отказоустойчивые решения, которые могут противостоять неправильному решнию об остановке процесса.

В этом контексте остановка из-за отказов может быть классифицирована следующим образом, от наименее до наиболее серьезных (см. также [Cachin et al., 2011]). Мы позволяем процессу P попытаться обнаружить, что процесс Q не выполняется.

- **Остановка при отказе** (Fail-stop failures) относится к аварийным сбоям, которые могут быть надежно обнаружены. Это может происходить

при предположении об исправности линий связи и когда процесс P обнаружения неисправностей может отнести к отказу чрезмерную задержку ответа от Q.

- **Шумоподобные отказы** (Fail-noisy failures) похожи на сбои остановки при отказе, за исключением того, что P только *в конечном итоге* придет к правильному выводу, что Q отказал. Это означает, что может быть какое-то априори неизвестное время, когда обнаружение P поведения Q ненадежно.
- При работе со **скрытыми отказами** (fail-silent failures) мы предполагаем, что каналы связи не являются неисправными, но этот процесс P не может отличить аварийные отказы от отказов неправильного действия.
- **Отказоустойчивые сбои** (Fail-safe failures) покрывают случай обработки произвольных сбоев процессом Q, но эти сбои являются доброкачественными: они не могут причинить никакого вреда.
- И наконец, при работе с **отказами при произвольных сбоях** (fail-arbitrary failures) Q может дать сбой любым возможным способом; сбои могут быть ненаблюдаемыми, а также вредными для правильного поведения других процессов.

Очевидно, что иметь дело с произвольными сбоями – это худшее, что может случиться. Как мы вскоре обсудим, мы можем спроектировать распределенные системы таким образом, чтобы они могли выдерживать даже такие сбои.

Маскировка отказов посредством избыточности

Если система должна быть отказоустойчивой, лучшее, что она может сделать, – это попытаться скрыть возникновение сбоев от других процессов. Ключевой метод маскировки неисправностей заключается в использовании избыточности. Возможны три вида: информационная избыточность и физическая избыточность (см. также [Джонсон, 1995]). При **информацинной избыточности** (information redundancy) добавляются дополнительные биты, чтобы обеспечить восстановление искаженных битов. Например, код Хемминга может быть добавлен к передаваемым данным для восстановления после шума в линии передачи.

При **временно́й избыточности** (time redundancy) выполняется действие, а затем, если необходимо, его выполнение повторяется. Этот подход используют транзакции. Если транзакция прерывается, она может быть передана без вреда. Другой известный пример – повторная передача запроса на сервер при отсутствии ожидаемого ответа. Временна́я избыточность особенно полезна, когда неисправности являются временными или прерывистыми.

При **физической избыточности** (резервировании) (physical redundancy) добавляется дополнительное оборудование или процессы, чтобы система в целом могла допускать потерю или неисправность некоторых компонентов. Таким образом, физическое резервирование может быть выполнено либо в аппаратном, либо в программном обеспечении. Например, в систему можно добавить дополнительные процессы, чтобы в случае сбоя небольшого

их числа система все еще могла функционировать правильно. Другими словами, путем репликации процессов может быть достигнута высокая степень отказоустойчивости. Мы вернемся к данному типу избыточности программного обеспечения позже в этой главе.

Примечание 8.3 (дополнительная информация: тройное модульное резервирование)

Показательна работа избыточности применительно к конструкции электронных устройств. Рассмотрим, например, схему на рис. 8.3a. Здесь сигналы проходят последовательно через устройства A, B и C. Если одно из них неисправно, окончательный результат, вероятно, будет неверным.

На рис. 8.3b каждое устройство реплицируется три раза. После каждого этапа в цепь включен трехкратный избиратель. Каждый избиратель представляет собой схему, которая имеет три входа и один выход. Если два или три входа совпадают, выход равен этому входу. Если все три входа различны, выход не определен. Этот вид конструкции известен как **тройное модульное резервирование** (Triple Modular Redundancy, TMR).

Предположим, что элемент A2 не работает. Каждый из избирателей V1, V2 и V3 получает два хороших (идентичных) входа и один мошеннический вход, и каждый из них выводит правильное значение на второй этап. По сути, эффект отказа A2 полностью маскируется, так что входы в B1, B2 и B3 точно такие же, как если бы не было ошибки.

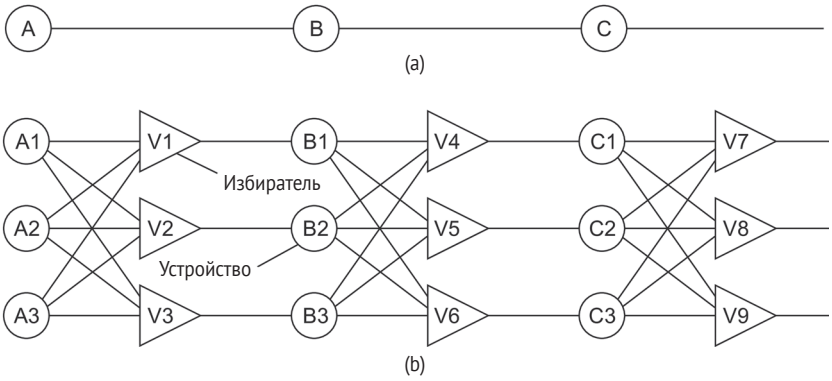


Рис. 8.3 ❖ Тройное модульное резервирование

Теперь рассмотрим, что произойдет, если B3 и C1 также неисправны, в дополнение к A2.

Эти эффекты тоже замаскированы, поэтому три конечных выхода все еще верны.

Во-первых, может быть неочевидно, почему на каждом этапе нужны три избирателя. В конце концов, один избиратель также может обнаружить и пропустить хотя бы мнение большинства. Однако избиратель тоже является компонентом и тоже может быть неисправен. Предположим, например, неисправность избирателя V1. В этом случае ввод в B1 будет неправильным, но пока все остальное работает, B2 и B3 будут выдавать одинаковый выход, а V4, V5 и V6 будут давать правильный результат на третьем этапе. Ошибка в V1 фактически ничем не отличается от ошибки в B1. В обоих случаях B1 дает неправильный вывод, но в обоих случаях он отклоняется позже, и окончательный результат все еще корректен.

Хотя не все отказоустойчивые распределенные системы используют TMR, методика очень общая и должна дать четкое представление о том, что такое отказоустойчивая система, в отличие от системы, отдельные компоненты которой являются высоконадежными, но чья организация не выдерживает отказов (т. е. работает правильно даже при наличии неисправных компонентов). Конечно, TMR можно применять рекурсивно, например чтобы сделать микросхему высоконадежной, использовать TMR внутри нее, не извещая разработчиков, которые будут использовать микросхему в своей собственной схеме, возможно, содержащей несколько копий микросхем с избирателями.

8.2. Устойчивость процесса

Теперь, когда основные вопросы отказоустойчивости были обсуждены, давайте сосредоточимся на том, как отказоустойчивость действительно может быть достигнута в распределенных системах. Первая тема, которую мы обсудим, – это защита от сбоев процессов, которая достигается путем репликации процессов в группы. На следующих страницах мы рассмотрим общие проблемы проектирования групп процессов и обсудим, что же такое отказоустойчивая группа. Кроме того, разберем, как достичь консенсуса внутри группы процессов, когда нельзя доверять одному или нескольким ее членам в предоставлении правильных ответов.

Устойчивость групповых процессов

Ключевой подход к сохранению работоспособности при наличии ошибочного процесса состоит в том, чтобы объединить несколько идентичных процессов в группу. Ключевое свойство, которым обладают все группы, заключается в том, что когда сообщение отправляется самой группе, его получают все члены группы. Таким образом, если один из процессов в группе завершится неудачей, мы надеемся, что другой процесс заменит его [Guerraoui and Schiper, 1997].

Группы процессов могут быть динамическими. Новые группы могут быть созданы, а старые группы уничтожены. Процесс может присоединиться к группе или покинуть группу во время работы системы. Процесс может быть членом нескольких групп одновременно. Следовательно, необходимы механизмы для управления группами и членства в группах.

Цель введения групп – позволить процессу иметь дело с коллекциями других процессов как единой абстракции. Таким образом, процесс P может отправить сообщение группе $\mathbf{Q} = \{Q_1, \ldots, Q_N\}$ серверов без необходимости знать, кто они, сколько их, где они расположены, и которые могут меняться от одного вызова к другому. Для P группа **Q** представляется одним логическим процессом.

Организация групп

Важное различие между группами связано с их внутренней структурой. В некоторых группах все процессы одинаковы. Здесь нет отличительного лиде-

ра, и все решения принимаются коллективно. В других группах существует какая-то иерархия. Например, один процесс является координатором, а все остальные – работниками. В этой модели, когда запрос на работу генерируется внешним клиентом или одним из работников, он отправляется координатору. Координатор затем решает, какой работник лучше всего подходит для выполнения запроса, и направляет его туда. Конечно, возможны и более сложные иерархии. Эти шаблоны связи показаны на рис. 8.4.

Каждая из этих организаций имеет свои преимущества и недостатки. Плоская группа симметрична и не имеет единой точки отказа. Если происходит сбой одного из процессов, группа просто становится меньше, но в противном случае может продолжать существовать. Недостатком является то, что принятие решений является более сложным. Например, чтобы что-то решить, часто приходится проводить голосование, что приводит к некоторым затратам.

Иерархическая группа обладает противоположными свойствами. Потеря координатора останавливает всю группу, но пока она работает, она может принимать решения, не затрагивая всех остальных. На практике, когда координатор в иерархической группе отказывает, его роль должна быть передана, и один из рабочих избирается новым координатором. Мы обсудили алгоритмы выбора лидера в главе 6.

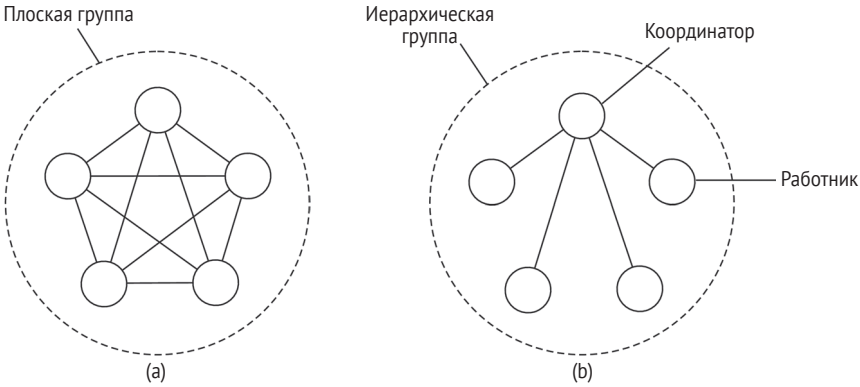


Рис. 8.4 ❖ Связь в группе: a) плоской; b) иерархической

Управление членством

Когда присутствует групповое взаимодействие, необходим некоторый метод для создания и удаления групп, а также для разрешения процессам присоединяться к группам и выходить из них. Одним из возможных подходов является создание группового сервера, на который можно отправлять все эти запросы. Затем сервер групп может поддерживать полную базу данных всех групп и их точного членства. Этот метод эффективен и довольно прост в реализации. К сожалению, он имеет недостаток, общий для всех централизованных методов: единую точку отказа. Если сервер группы дает сбой, управление группой перестает существовать. Вероятно, большинство или все

группы должны быть восстановлены с нуля, возможно, придется прекратить любую работу.

Противоположный подход заключается в распределении членства в группах. Например, если (надежная) многоадресная рассылка доступна, не входящий в группу может отправить сообщение всем членам группы, сообщив о своем желании присоединиться к группе.

В идеале, чтобы покинуть группу, участник просто посылает всем прощальное сообщение. В контексте отказоустойчивости допущение семантики сбоя аварийной остановки обычно не подходит. Проблема в том, что нет объявления о том, что процесс завершается сбоем, как это происходит, когда процесс уходит добровольно. Другие участники должны обнаружить это экспериментально, заметив, что отказавший участник больше ни на что не реагирует. Как только определено, что отказавший член действительно не работает (и не просто работает медленно), его можно удалить из группы.

Другая запутанная ситуация состоит в том, что уход и соединение должны быть синхронными с отправкой сообщений данных. Иными словами, начиная с момента, когда процесс присоединился к группе, он должен получать все сообщения, отправленные этой группе. Точно так же, как только процесс покинул группу, он больше не должен получать сообщений от группы, и другие участники не должны больше получать сообщений от него. Один из способов убедиться, что объединение или выход интегрированы в поток сообщений в нужном месте, – преобразовать эту операцию в последовательность сообщений, отправляемых всей группе.

Последний вопрос, касающийся членства в группе: что делать, если происходит остановка стольких процессов, что группа больше вообще не может функционировать? Некоторый протокол необходим для перестройки группы. Неизменно, некоторый процесс должен будет взять на себя инициативу, чтобы вновь начать работу, но что произойдет, если два или три попытаются это сделать одновременно? Протокол должен быть в состоянии противостоять этому. Опять же, может потребоваться координация, например с помощью алгоритма выбора лидера.

Маскировка и репликация отказа

Частью решения для построения отказоустойчивых систем являются группы процессов.

В частности, наличие группы идентичных процессов позволяет нам маскировать один или несколько неисправных процессов в этой группе. Другими словами, мы можем реплицировать процессы и организовывать их в группу, чтобы заменить один (уязвимый) процесс группой (отказоустойчивой). Как обсуждалось в предыдущей главе, существует два способа приблизиться к такой репликации: с помощью первичных протоколов или протоколов репликации и записи.

Первичная репликация в случае отказоустойчивости обычно отображается в форме протокола первичного резервного копирования. В этом случае группа процессов организована иерархическим образом, в котором пер-

вичный координирует все операции записи. На практике основной объект фиксируется, хотя его роль может быть передана одной из резервных копий, если это необходимо. Фактически, когда происходит сбой основного сервера, резервные копии выполняют некоторый алгоритм выбора, чтобы выбрать новый основной сервер.

Протоколы реплицированной записи используются в форме активной репликации, а также с помощью протоколов на основе кворума. Эти решения соответствуют организации набора идентичных процессов в плоскую группу. Основным преимуществом является то, что такие группы не имеют единой точки отказа за счет распределенной координации.

Важной проблемой при использовании групп процессов для допуска ошибок является то, сколько требуется репликаций. Чтобы упростить наше обсуждение, давайте рассмотрим только системы с реплицированной записью. Говорят, что система является **устойчивой к *k* неисправностям** (*k*-fault tolerant), если она может преодолеть неисправности в *k* компонентах и при этом соответствовать спецификациям. Если компоненты, скажем процессы, молча терпят неудачу, то наличие $k + 1$ таких компонентов достаточно для обеспечения устойчивости к ошибкам. Если *k* из них просто останавливаются, то можно использовать ответ от другого.

С другой стороны, если процессы демонстрируют произвольные отказы, продолжая работать при отказе и отправляя ошибочные или случайные ответы, необходимо минимум $2k + 1$ процессов для достижения устойчивости к отказам. В худшем случае *k* неудачных процессов могут случайно (или даже намеренно) генерировать один и тот же ответ. Тем не менее оставшиеся $k + 1$ также дадут тот же ответ, так что клиент или избиратель может просто поверить большинству.

Теперь предположим, что в группе с отказоустойчивостью *k* один процесс завершается неудачно. Группа в целом все еще выполняет требования, поскольку она может терпеть отказ до *k* своих членов (из которых один только что потерпел неудачу). Но что произойдет, если более чем *k* участников потерпят неудачу? В этом случае все ставки на безотказность отменены, и что бы ни делала группа, ее результатам, если таковые имеются, нельзя доверять. Еще один способ взглянуть на это состоит в том, чтобы посчитать, что вся группа процессов, скорее всего, имитирует поведение одного надежного процесса.

Консенсус в неисправных системах со сбоями

Как уже упоминалось, мы приняли модель относительно клиентов и серверов, в которой потенциально очень большое количество клиентов теперь отправляют команды *группе процессов*, которые совместно ведут себя *как единый, очень надежный процесс*. Чтобы выполнять эту работу, нам необходимо сделать важное предположение:

> *В отказоустойчивой группе процессов каждый исправный процесс выполняет те же команды и в том же порядке, что и любой другой исправный процесс.*

Формально это означает, что члены группы должны прийти к **единому мнению**, какую команду выполнять. Если отказы не могут произойти, достичь **консенсуса** легко. Например, мы можем использовать полностью упорядоченную многоадресную рассылку Лампорта (Lamport), как описано в разделе 6.2. Или, для простоты, используя централизованный секвенсор, который, давая порядковый номер каждой команде, которая должна быть выполнена, также выполнит эту работу. К сожалению, жизнь не обходится без неудач, и достижение консенсуса в группе процессов при более реалистичных предположениях оказывается непростым делом.

Для иллюстрации рассматриваемой проблемы предположим, что у нас есть группа процессов $\mathbf{P} = \{P_1, \ldots, P_n\}$, которая работает в семантике прекращения работы из-за отказа. Другими словами, мы предполагаем, что аварийные отказы членов группы могут быть надежно обнаружены. Обычно клиент связывается с членом группы, запрашивая его выполнить команду. Каждый член группы ведет список предлагаемых команд, некоторые из которых он получил непосредственно от клиентов, другие – от иных членов группы. Мы можем достичь консенсуса с помощью адаптированного подхода, используемого в [Cachin et al., 2011] и относящегося к **консенсусу лавинной маршрутизации** (flooding consensus).

Концептуально алгоритм работает в раундах. В каждом раунде процесс P_i отправляет свой список предложенных команд, которые он видел до сих пор, каждому другому процессу в **P**. В конце раунда каждый процесс объединяет все полученные предложенные команды в новый список, из которого он затем будет детерминированно определять выбор команды для выполнения, если это возможно. Важно понимать, что алгоритм выбора одинаков для всех процессов. Другими словами, имеются абсолютно одинаковые списки. Все они будут выбирать одни и те же команды для выполнения (и удалять команды из этого списка).

Нетрудно увидеть, что этот подход работает до тех пор, пока процессы не выходят из строя. Проблемы начинаются, когда во время раунда *r* процесс P_i обнаруживает, что, скажем, процесс P_k вышел из строя. Чтобы сделать это конкретным, предположим, что у нас есть группа из четырех процессов $\{P_1, \ldots, P_4\}$ и что P_1 выходит из строя во время раунда *r*. Также предположим, что P_2 получает список предложенных команд от P_1 до его сбоя, а P_3 и P_4 – нет (другими словами, P_1 выходит из строя до того, как получит возможность отправить свой список P_3 и P_4). Эта ситуация показана на рис. 8.5.

Предполагая, что все процессы знали, кто был членом группы в начале раунда *r*, P_2 готов принять решение о том, какую команду выполнять, когда он получает соответствующие списки других членов: у него есть все команды, предложенные до сих пор. Не так относительно P_3 и P_4. Например, P_3 может обнаружить, что P_1 вышел из строя, но он не знает, получил ли P_2 или P_4 список P_1. С точки зрения P_3, если есть другой процесс, который получил предложенные P_1 команды, тогда этот процесс может принять иное решение, чем он сам. Как следствие лучшее, что может сделать P_3, – это отложить свое решение до следующего раунда. То же самое в этом примере относится и к P_4. Процесс решит перейти к следующему раунду, когда он получит сообщение от каждого нефункционального процесса. Это предполагает, что каждый про-

цесс может надежно обнаружить сбой другого процесса, так как в противном случае он не сможет решить, какие процессы исправны.

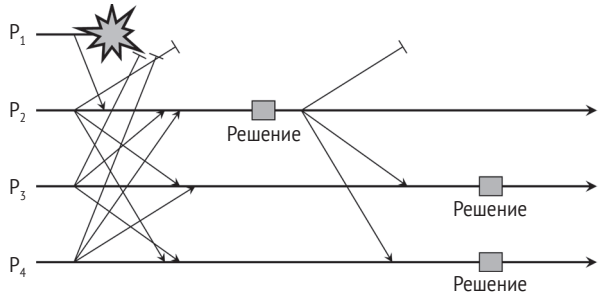


Рис. 8.5 ❖ Достижение консенсуса за счет лавинной маршрутизации при наличии сбоев [Cachin et al., 2011]

Поскольку процесс P_2 получил все команды, он действительно может принять решение и впоследствии передать это решение другим. Затем во время следующего раунда $r + 1$ процессы P_3 и P_4 также смогут принять решение: они решат выполнить ту же команду, выбранную P_2.

Чтобы понять, почему этот алгоритм корректен, важно понимать, что процесс перейдет к следующему раунду без принятия решения, только когда он обнаружит, что другой процесс не удался. В конце концов, это означает, что в худшем случае остается не более одного исправного процесса, и этот процесс может просто решить, какую предлагаемую команду выполнить. Еще раз отметим, что мы предполагаем надежное обнаружение сбоев.

Но что происходит, когда решение процесса P_2, отправленное в P_3, было потеряно? В этом случае P_3 все еще не может принять решение. Хуже того, нам нужно убедиться, что он принимает то же решение, что и P_2 и P_4. Если P_2 не вышел из строя, можно предположить, что повторная передача своего решения исправит положение. Если P_2 действительно вышел из строя, это также будет обнаружено P_4, который затем повторно передаст свое решение. Тем временем P_3 перешел в следующий раунд и после получения решения от P_4 прекратит выполнение алгоритма.

Пример: Paxos

Алгоритм консенсуса, основанный на лавинной маршрутизации, не очень реалистичен хотя бы потому, что опирается на модель сбоя аварийного останова. Более реалистичным является предположение об отказоустойчивой модели сбоев, в которой процесс в конечном итоге надежно обнаружит сбой другого процесса. Далее мы опишем упрощенную версию широко принятого консенсусного алгоритма, известного как **Paxos**. Первоначально он был опубликован в 1989 году в качестве технического доклада Лесли Лэмпорта (Leslie Lamport), но прошло около десяти лет, прежде чем было решено, что, может быть, не такая плохая идея – разъяснить и распространять его

[Lamport, 1998]. Первоначальную публикацию нелегко понять, примером могут служить другие публикации, нацеленные на ее объяснение [Lampson, 1996; Prisco et.al., 1997; Lamport, 2001; van Renesse and Altinbuken, 2015].

Основная идея Paxos

Предположения, при которых работает Paxos, довольно слабые:

- распределенная система является частично синхронной (фактически она может быть даже асинхронной);
- связь между процессами может быть ненадежной, что означает, что сообщения могут быть потеряны, продублированы или переупорядочены;
- сообщения, которые повреждены, могут быть обнаружены как таковые (и, следовательно, впоследствии проигнорированы);
- все операции являются детерминированными: после запуска выполнения точно известно, что оно будет делать;
- процессы могут демонстрировать аварийные сбои, но не произвольные сбои, и процессы не договариваются.

В целом это реалистичные предположения для многих практических распределенных систем.

Мы в основном следуем объяснениям, данным в [Lamport, 2001] и [Kirsch and Amir, 2008]. Алгоритм работает как сеть *логических* процессов, которые бывают разных типов. Во-первых, есть **клиенты**, которые запрашивают выполнение конкретной операции. На стороне сервера каждый клиент представлен **заявителем** (proposer), тем, кто **выдвигает предположение**, и который пытается принять запрос клиента. Как правило, один выдвигающий предположение назначается **лидером** (lider) и направляет протокол для достижения консенсуса.

Нам нужно установить, что предложенная операция принята **акцептором** (принимающим) (acceptor). Если большинство акцепторов принимают одно и то же предложение, оно считается *выбранным*. Однако то, что выбрано, все еще должно быть *изучено*. Для этого у нас будет несколько **процессов обучения** (learner), каждый из которых выполнит выбранное предложение, как только оно будет получено от большинства акцепторов.

Важно отметить, что один заявитель, акцептор и учащийся образуют один *физический* процесс, работающий на одной машине, с которой взаимодействует клиент, как показано на рис. 8.6. Таким образом, мы предполагаем, что если, например, произойдет сбой заявителя, то физический процесс, частью которого он является, потерпит крах. Реплицируя этот сервер, мы стремимся обеспечить отказоустойчивость при наличии сбоев.

Основа модели заключается в том, что ведущий разработчик получает запросы от клиентов, по одному за один раз. Неведущий заявитель направляет любой клиентский запрос лидеру. Ведущий заявитель направляет свое предложение всем получателям (акцепторам), предлагая каждому принять запрошенную операцию. Каждый получатель впоследствии будет транслировать обучающее сообщение. Если учащийся получает одно и то же обучающее сообщение от большинства акцепторов, он знает, что достигнут консенсус относительно того, какую операцию выполнить, и выполнит ее.

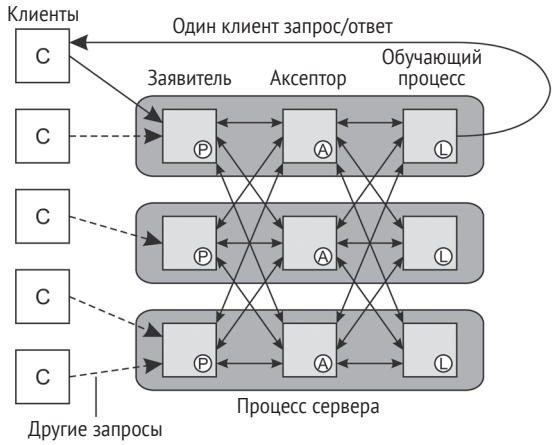


Рис. 8.6 ❖ Разные логические процессы организации в Paxos

Есть как минимум две конкретные проблемы, требующие дальнейшего внимания. Во-первых, серверам нужно не только достичь консенсуса относительно того, какую операцию выполнять, но также убедиться, что каждый из них действительно ее выполняет. Другими словами, как мы можем точно знать, что большинство исправных серверов будут выполнять операцию? По сути, есть только один выход: повторная передача обучающих сообщений. Тем не менее, чтобы сделать эту работу, акцептор должен будет регистрировать свои решения (что, в свою очередь, требует механизма для очистки журналов). Поскольку мы предполагаем глобально упорядоченные временные метки предложений (как будет объяснено чуть далее), пропущенные сообщения могут быть легко обнаружены, а также принятые операции всегда будут выполняться в одном и том же порядке всеми обучаемыми.

Как правило, сервер, на котором размещен ведущий заявитель, также будет информировать клиента о выполнении запрошенной операции. Если другой процесс взял на себя инициативу, он также будет обрабатывать ответ клиенту.

Это приводит нас ко второму важному вопросу: провалу лидера. Жизнь была бы легкой, если бы был надежно обнаружен отказ лидера, после чего был бы избран новый лидер, а позже выздоравливающий лидер немедленно заметил бы, что мир вокруг него изменился. К сожалению, жизнь не так проста. Paxos был разработан, чтобы мириться с теми, кто предлагает, кто все еще верит, что они лидируют. В результате предложения могут быть разосланы одновременно разными авторами (каждый из которых считает себя лидером). Поэтому нам необходимо убедиться, что эти предложения можно отличить друг от друга, чтобы акцепторы обрабатывали только предложения от нынешнего лидера.

Обратите внимание, что опора на ведущего заявителя подразумевает, что любая практическая реализация Paxos должна сопровождаться алгоритмом выбора лидера.

В принципе, этот алгоритм может работать независимо от Paxos, но обычно является его частью.

Чтобы отличить параллельные предложения от разных заявителей, каждое предложение p имеет уникально связанную (логическую) временную метку *ts*(p). То, как достигается уникальность, остается за реализацией, но в ближайшее время мы опишем некоторые детали. Пусть *oper*(p) обозначает операцию, связанную с предложением p.

Хитрость заключается в том, чтобы несколько предложений были приняты, но чтобы каждое из этих принятых предложений имело одну и ту же связанную операцию. Этого можно достичь, гарантируя, что если выбрано предложение p, то любое предложение с более высокой отметкой времени также будет иметь такую же связанную операцию. Другими словами, мы требуем, чтобы

p выбран ⇒ для всех p′ *с* *ts*(p′) > *ts*(p): *oper*(p′) = *oper*(p).

Конечно, чтобы выбрать p, его нужно принять. Это означает, что мы можем гарантировать выполнение нашего требования, гарантируя, что если выбрано p, то любое предложение с более высокой отметкой времени, принятое любым акцептором, будет иметь ту же связанную операцию, что и p. Однако этого недостаточно, так как предположим, что в определенный момент заявитель просто отправляет новое предложение p′ с самой высокой отметкой времени на тот момент времени получателю A, который ранее не получал никакого предложения. Обратите внимание, что это действительно может произойти в соответствии с нашими предположениями относительно потери сообщений и множеством заявителей, каждый из которых считает себя лидером. В отсутствие каких-либо других предложений A просто примет p′. Чтобы не допустить возникновения такой ситуации, мы должны гарантировать:

> *если выбрано предложение* p*, то любое предложение с более высокой отметкой времени, выданное заявителем, будет иметь ту же связанную операцию, что и* p.

При объяснении алгоритма Paxos ниже мы действительно увидим, что заявителю может потребоваться принять операцию, исходящую от акцепторов, в пользу своей собственной. Это произойдет после того, как ведущий заявитель потерпел неудачу, но предложенная операция уже была принята большинством его участников.

Процессы в совокупности формально обеспечивают безопасность в том смысле, что будут изучены только предложенные операции и что одновременно будет изучаться не более одной операции. В общем, безопасность не пострадает. Кроме того, Paxos обеспечивает *условную живучесть* (conditional liveness) системы в том смысле, что если достаточное количество процессов останется в рабочем состоянии, то предложенная операция в конечном итоге будет изучена (и, следовательно, выполнена). **Живучесть**, которая говорит нам, что в конечном итоге все будет в порядке, в Paxos не гарантируется, если не будут сделаны некоторые адаптации. Мы возвращаемся к понятию живучести в примечании 8.4.

Теперь обратимся к двум этапам, каждый из которых состоит из двух подэтапов. На первом этапе ведущий заявитель взаимодействует с акцепторами, чтобы получить запрошенную операцию, принятую для выполнения. Первый этап необходим, чтобы исключить любые проблемы, вызванные различными

заявителями, каждый из которых верит, что является лидером. Лучшее, что может случиться, – это то, что отдельный получатель обещает рассмотреть действия заявителя и игнорировать другие запросы. Хуже, если заявитель опоздал, и вместо этого ему будет предложено принять запрос другого заявителя. По-видимому, произошла смена руководства, и могут быть прежние запросы, которые необходимо обработать в первую очередь.

На втором этапе акцепторы проинформируют участников предложения о своих обещаниях. Ведущий заявитель, по сути, выполняет несколько иную роль, предпочитая ту операцию, которая должна быть выполнена, и затем сообщает об этом акцепторам.

- **Этап 1a (подготовка).** Цель этого этапа заключается в том, чтобы заявитель P, который считает, что он является лидером, и предлагает операцию o, пытается **привязать** (anchor) временную метку своего предложения, в том смысле, что любая более низкая временная метка была неудачной или что o также была ранее предложена (то есть с некоторой более низкой временной отметкой предложения). С этой целью P передает свое предложение акцепторам. Для операции o заявитель выбирает номер предложения m больше, чем любой из его ранее выбранных номеров. Это приводит к метке **времени предложения** (proposal timestamp) $t = (m, i)$, где i – это (числовой) идентификатор процесса P. Обратите внимание, что

  $$(m, i) < (n, j) \Leftrightarrow (m < n) \text{ или } (m = n \text{ и } i < j).$$

  Эта временная метка для предложения p является реализацией ранее упомянутой временной метки ts(p). Заявитель P отправляет PREPARE(t) всем аксепторам (но учтите, что сообщения могут быть потеряны). При этом он (1) просит акцепторов пообещать не принимать никаких предложений с более низкой отметкой времени предложения и (2) сообщить ему о принятом предложении, если таковое имеется, с самой высокой отметкой времени меньше t. Если такое предложение существует, заявитель примет его.
- **Этап 1b (обещание):** акцептор A может получить несколько предложений. Предположим, что он получает PREPARE(t) от P. Есть три случая, которые следует рассмотреть:
  1) t является самой высокой отметкой времени предложения, полученной до сих пор от любого заявителя. В этом случае A вернет P PREPARE(t), заявив, что A будет игнорировать любые будущие предложения с более низкой отметкой времени;
  2) если t является самой высокой отметкой времени, но другое предложение (t', o′) уже было принято, акцептор A также возвращает (t', o′) в P. Это позволит P принять решение о конечной операции, которая должна быть принята;
  3) во всех остальных случаях ничего не делать: очевидно, что есть еще одно предложение с более высокой отметкой времени, которое обрабатывается.

  Как только первый этап завершен, ведущий заявитель P знает, что обещали акцепторы. По сути, ведущий заявитель знает, что все акцепторы

согласились на одну и ту же операцию. Это даст P возможность сообщить акцепторам, что они могут действовать дальше. Это необходимо, потому что хотя ведущий заявитель знает, по какому согласованию операций был достигнут этот консенсус, он неизвестен другим. Опять же, мы предполагаем, что P получил ответ от большинства акцепторов (чьи соответствующие ответы могут отличаться).

- **Этап 2a (прием):** необходимо рассмотреть два случая:
  1) если P не получает ни одной принятой операции от какого-либо из получателей, он направит свое собственное предложение для принятия, отправив ACCEPT(t, o) всем акцепторам;
  2) в противном случае ему сообщили о другой операции o′, которую он примет и направит на прием. Это делается путем отправки ACCEPT(t, o′), где t – это временна́я метка предложения P, а o′ – самая высокая операция с временно́й меткой предложения среди всех принятых операций, которые были возвращены получателями на этапе 1b.
- **Этап 2b (изучение):** наконец, если акцептор получает ACCEPT(t, o′), но поскольку ранее не отправлял обещание с более высокой отметкой времени предложения, он примет операцию o′ и сообщит всем учащимся выполнить o′, отправив LEARN(o′).

  В этот момент акцептор может забыть об o′. Обучающийся L, получающий обучение (o′) от большинства акцепторов, выполнит операцию o′. Теперь мы также знаем, что большинство учащихся придерживаются той же идеи, какую операцию выполнять.

Важно понимать, что это описание Paxos действительно отражает только его суть: использование ведущего заявителя для инициирования акцепторов к выполнению той же операции. Когда дело доходит до практических реализаций, нужно сделать гораздо больше (и больше, чем мы хотим описать здесь).

Прекрасное описание того, что значит реализовать Paxos, было изложено в [Kirsch and Amir, 2008]. Еще одно описание его практического применения можно найти в [Chandra et al., 2007].

Понимание Paxos

Чтобы правильно понять Paxos, а также многие другие алгоритмы согласованности, полезно посмотреть, как его дизайн мог бы развиваться. Мы говорим «мог бы», так как эволюция алгоритма никогда не документировалась. Нижеследующее описание работы Paxos в значительной степени основано на ее описании в [Meling and Jehl, 2013][1].

В качестве отправной точки для понимания Paxos мы рассмотрим сервер, который хотим сделать более надежным. К настоящему времени мы знаем, что этого можно достичь путем репликации и обеспечения того, чтобы все команды, отправленные клиентами, выполнялись всеми серверами в одном и том же порядке. Самая простая ситуация – добавить один сервер, соз-

[1] Выражаем особую благодарность Мелину (Meling) и Джелу (Jehl) за помощь в понимании Paxos.

дав тем самым группу из двух процессов, скажем S_1 и S_2. Кроме того, чтобы убедиться, что все команды выполняются в одном и том же порядке, мы назначаем один процесс секвенсором, который увеличивает и ассоциирует уникальную метку времени с каждой отправленной командой. Серверы обязаны выполнять команды в соответствии со своей отметкой времени. В Paxos такой сервер называется **лидером** (leader). Мы также можем считать его **основным сервером** (primary server), а другой выступает в качестве **резервного сервера** (buckup server).

Мы предполагаем, что клиент передает свою команду запроса всем серверам. Если сервер замечает, что пропускает команду, он может рассчитывать на переадресацию при необходимости другим сервером. Мы не будем описывать, как это происходит, но молча предположим, что все команды хранятся на серверах и что нам просто нужно убедиться, что серверы согласны с тем, какую команду выполнять дальше. Как следствие вся оставшаяся связь между серверами состоит из управляющих сообщений. Чтобы прояснить этот момент, рассмотрим ситуацию, показанную на рис. 8.7. (В дальнейшем мы будем использовать индексы для обозначения процессов и индексы для обозначения операций и состояний.)

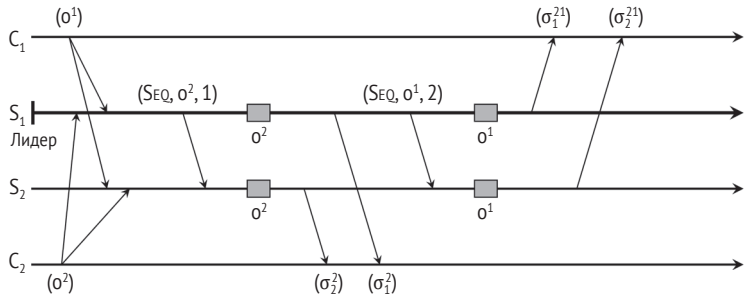


Рис. 8.7 ❖ Два клиента, взаимодействующих с группой процессов из двух серверов

В этом примере сервер S_1 является лидером и, как таковой, будет раздавать метки времени отправленным запросам. Клиент C_1 отправил команду o^1, а C_2 отправил o^2. S_1 дает команду S_2 выполнить операцию o^2 с отметкой времени 1, а затем операцию o^1 с отметкой времени 2. После обработки команды сервер вернет результат соответствующему клиенту. Мы обозначаем это с помощью (σ_i^j), где i – индекс сервера отчетов, а j – состояние, в котором он находился, выраженное как последовательность операций, которые он выполнил. В нашем примере клиент C_1 увидит результат (σ_k^{21}), означающий, что каждый сервер выполнил o^1 после выполнения o^2.

В Paxos, когда лидер связывает временную метку с операцией, он делает это, отправляя сообщение о принятии на другой сервер (серверы). Поскольку мы предполагаем, что сообщения могут быть потеряны, сервер, принимающий операцию, делает это, сообщая лидеру, что он изучил операцию, возвращая сообщение LEARN(o).

Когда руководитель не замечает, что операция o изучена, он просто повторно передает сообщение ACCEPT(o, *t*), где *t* является исходной отметкой времени. Обратите внимание, что в нашем описании мы пропускаем этап согласования операции, которая должна быть выполнена: мы предполагаем, что лидер принял решение и теперь должен достичь консенсуса по выполнении этой операции.

Компенсация за потерянное сообщение относительно проста, но что происходит, когда также происходит сбой сервера? Сначала давайте предположим, что отказ может быть надежно обнаружен. Рассмотрим ситуацию, показанную на рис. 8.8a. Проблема, конечно, в том, что сервер S_2 никогда не узнает (об) операции o^1. Эту ситуацию можно предотвратить, потребовав, чтобы сервер мог выполнить операцию, только если он знает, что другой сервер также изучил эту операцию, как показано на рис. 8.8b.

Нетрудно увидеть, что с большей группой процессов мы можем попасть в ту же ситуацию, что и на рис. 8.8a. Просто рассмотрим группу из трех серверов $\{S_1, S_2, S_3\}$, где S_1 является лидером. Если его сообщение ACCEPT(o^1, *t*) для S_3 потеряно, но все же знает, что S_2 изучил o^1, то он все равно не должен выполнять o^1, пока не получит также сообщение LEARN(o^1) от S_3. Эта ситуация показана на рис. 8.9.

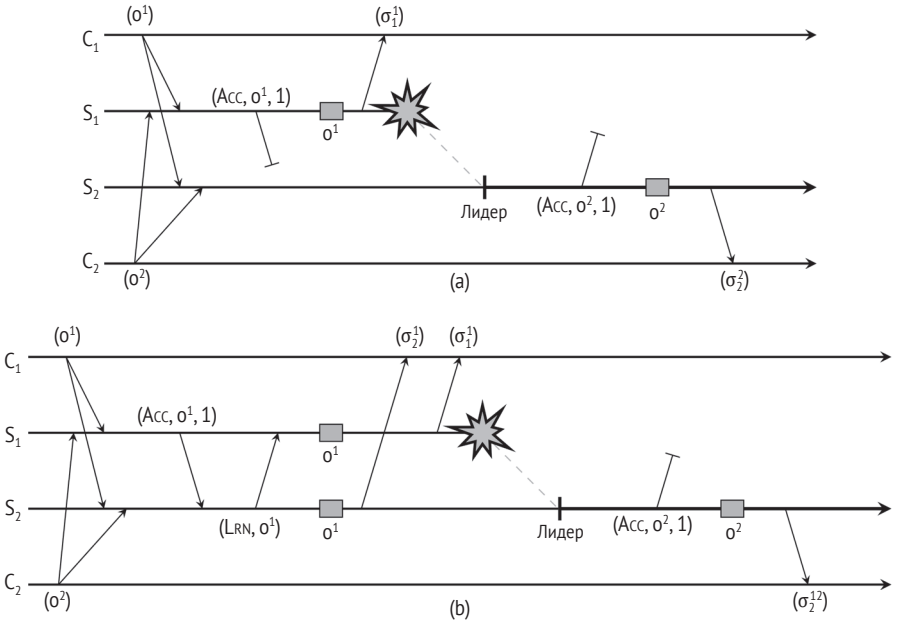


Рис. 8.8 ❖ a) Что может произойти, когда лидер терпит крах в сочетании с потерянным приемом; b) решение, требующее, чтобы другой сервер также изучил операцию перед ее выполнением

А теперь представьте, что произойдет, если LEARN(o^1), возвращенный S_2, не доберется до S_1. Конечно, S_1 не будет выполнять o^1, но в противном случае у нас все еще будут проблемы: S_2 выполнит o^1, в то время как S_3 возглавит

и выполнит o^2, даже не зная, что o^1 уже обработан. Другими словами, также S_2 должен ждать с выполнением o^1, пока не узнает, что S_3 изучил эту операцию. Это приводит нас к следующему:

В Paxos сервер S *не может выполнить операцию* o *до тех пор, пока не получит* LEARN(o) *от всех других исправных серверов.*

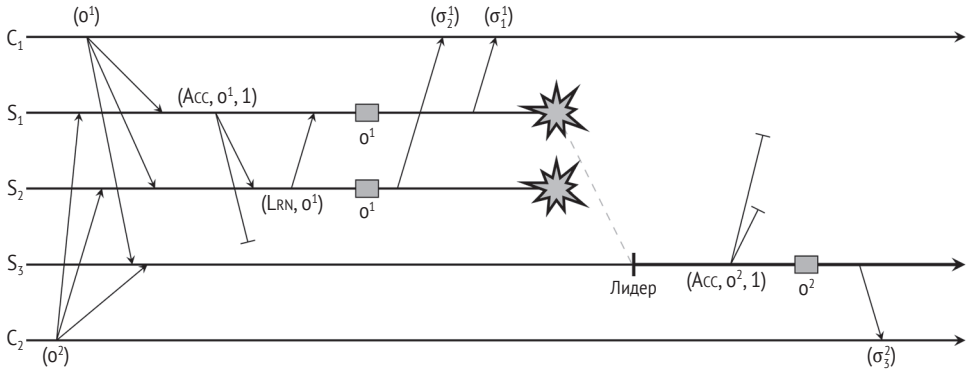


Рис. 8.9 ❖ Ситуация при работе с тремя серверами, два из которых выходят из строя

До этого момента мы предполагали, что процесс может надежно определить, что другой процесс потерпел крах. На практике это не так.

Как мы вскоре обсудим более подробно, стандартным подходом к обнаружению сбоев является установка тайм-аута ожидаемых сообщений. Например, каждому серверу необходимо отправить сообщение о том, что оно еще живо, и в то же время другие серверы устанавливают тайм-ауты для ожидаемого получения таких сообщений. Если время ожидания истекло, подозревается, что отправитель потерпел неудачу. В частично синхронной или полностью асинхронной системе другого решения не существует. Однако следствием этого является то, что сбой может быть ложно обнаружен, поскольку доставка таких сообщений «Я жив» (alive) могла быть просто отложена или потеряна.

Предположим, что Paxos реализовал механизм обнаружения сбоев, но два сервера ошибочно пришли к выводу, что другой отказал, как показано на рис. 8.10. Проблема ясна: каждый может теперь самостоятельно решить выполнить свою операцию выбора, приводящую к расходящемуся поведению. Именно в этот момент нам нужно ввести дополнительный сервер и потребовать, чтобы сервер мог выполнить операцию, только если он уверен, что большинство выполнит эту операцию. Обратите внимание, что в случае трех серверов выполнение операции o сервером S может иметь место, как только S получит по меньшей мере одно (другое) сообщение LEARN(o). Вместе с отправителем этого сообщения S составит большинство.

Теперь мы подошли к тому моменту, когда должно быть ясно, что для правильной работы Paxos требуются как минимум три реплицированных

сервера $\{S_1, S_2, S_3\}$. Давайте сосредоточимся на ситуации, когда один из этих серверов выходит из строя. Мы делаем следующие предположения:

- первоначально S_1 является лидером;
- сервер может надежно определить, что пропустил сообщение. Последнее может быть реализовано, например, с помощью временных меток в виде строго возрастающих порядковых номеров. Всякий раз, когда сервер замечает, что пропустил сообщение, он может просто запросить повторную передачу и наверстать упущенное, прежде чем продолжить;
- когда нужно выбрать нового лидера, остальные серверы следуют строго детерминированному алгоритму. Например, мы можем смело предположить, что в случае сбоя S_1 лидером станет S_2. Аналогично, если S_2 отказывает, S_3 станет основным, и т. д.;
- клиенты могут получать дубликаты ответов, но без необходимости распознавать дубликаты они больше не составляют часть протокола Paxos. Другими словами, клиент не может помочь серверу разрешить ситуацию.

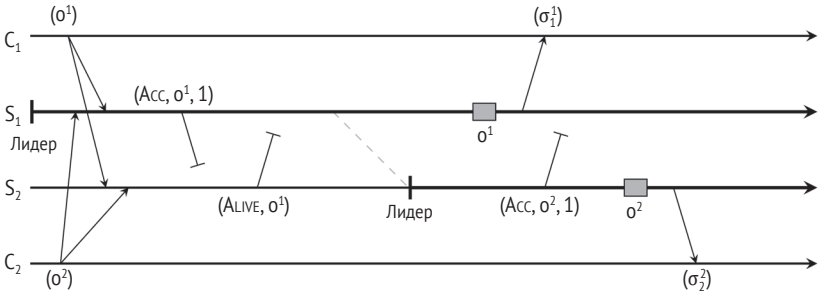


Рис. 8.10 ❖ Ситуация в случае ложных обнаружений сбоев

В этих условиях довольно легко увидеть, что независимо от того, что один из S_2 или S_3 терпит крах, Paxos будет вести себя правильно. Конечно, мы все еще требуем, чтобы выполнение операции могло иметь место, только если сервер знает, что большинство выполнит эту операцию.

Предположим теперь, что S_1 терпит крах после выполнения операции o^1. Худшее, что может случиться в этом случае, – это то, что S_3 полностью не знает о ситуации, пока новый лидер S_2 не скажет ему принять операцию o^2. Обратите внимание, что это объявляется в сообщении ACCEPT(o^2, 2), так что отметка времени $t = 2$ оповестит S_3, что оно пропустило предыдущее сообщение ACCEPT. S_3 сообщит об этом S_2, который затем может повторить передачу ACCEPT(o^1, 1), позволяя S_3 наверстать упущенное.

Аналогично, если S_2 пропустил ACCEPT(o^1, 1), но обнаружил, что S_1 потерпел крах, он в конечном итоге либо отправит ACCEPT(o^1, 1), либо ACCEPT(o^2, 1) (т. е. в обоих случаях, используя временную метку $t = 1$, которая ранее использовалась S_1). Опять же, у S_3 достаточно информации, чтобы снова поставить S_2 на правильный путь. Если S_2 послал ACCEPT(o^1, 1), S_3 может просто сказать S_2, что он уже узнал o^1. В другом случае, когда S_2 отправляет команду ACCEPT(o^2, 1), S_3

сообщит S_2, что он явно пропустил операцию o^1. Мы заключаем, что когда S_1 терпит крах после выполнения операции, Paxos ведет себя правильно.

Так что же может произойти, если S_1 терпит крах сразу после отправки ACCEPT(o^1, 1) на два других сервера? Предположим еще раз, что S_3 полностью игнорирует ситуацию, потому что сообщения теряются, пока S_2 не возглавит и не объявит, что o^2 должно быть принято. Как и раньше, S_3 может сказать S_2, что он (то есть S_3) пропустил операцию o^1, так что S_2 может помочь S_3 наверстать упущенное. Если S_2 пропускает сообщения, но обнаруживает, что S_1 потерпел крах, тогда, как только он возьмет на себя руководство и предложит операцию, он будет использовать устаревшую временную метку. Это заставит S_3 сообщить S_2, что он пропустил операцию o^1, что спасает положение снова, Paxos ведет себя правильно.

Проблемы могут возникнуть при ложном обнаружении сбоев. Рассмотрим ситуацию, показанную на рис. 8.11. Мы видим, что сообщения приема от S_1 значительно задерживаются и что S_2 ложно обнаруживает сбой S_1. S_2 берет на себя лидерство и отправляет ACCEPT(o^2, 1), то есть с отметкой времени $t = 1$. Однако когда приходит окончательное подтверждение ACCEPT(o^1, 1), S_3 ничего не может сделать: это не ожидаемое сообщение. К счастью, если он знает, кто является текущим лидером, в сочетании с детерминированными выборами лидера он может спокойно отказаться от ACCEPT(o^1, 1), зная, что к настоящему времени S_2 стал основным. Мы пришли к выводу, что лидер должен указать свой идентификатор в сообщении о принятии.

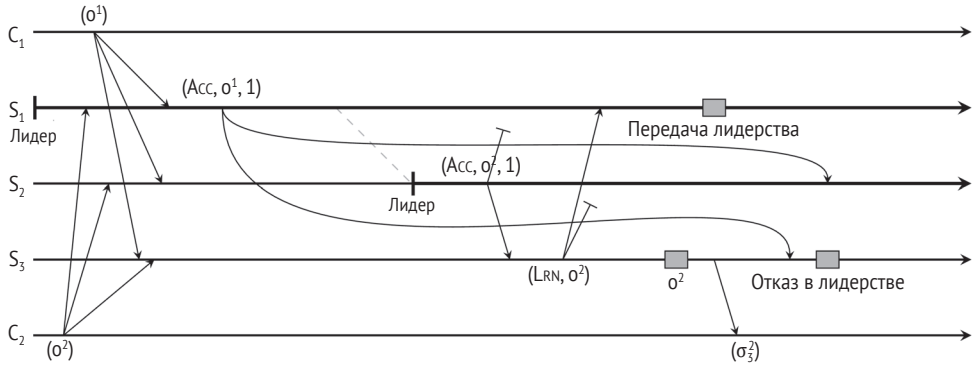


Рис. 8.11 ❖ Зачем нужно вводить идентификатор текущего лидера

Мы рассмотрели почти все случаи и показали, что Paxos ведет себя правильно. К сожалению, несмотря на правильность, алгоритм все еще может остановиться. Рассмотрим ситуацию, показанную на рис. 8.12. Здесь мы видим, что поскольку сообщения обучения, возвращаемые S_3, теряются, ни S_1, ни S_2 никогда не смогут узнать, что на самом деле выполнял S_3: он обучается (и выполняет) ACCEPT(o^1, 1) до или после обучения ACCEPT(o^2, 1), или, возможно, он не выучил ни ту, ни другую операцию? Решение этой проблемы обсуждается в примечании 8.4.

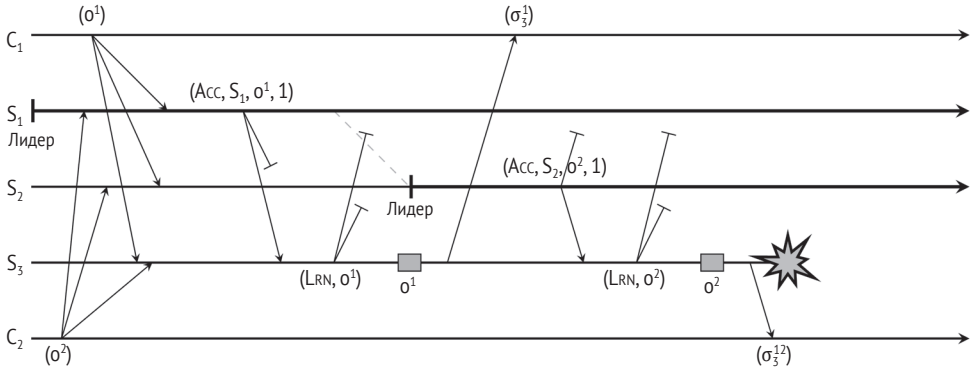


Рис. 8.12 ❖ Когда Paxos не может двигаться дальше

Примечание 8.4 (дополнительно: прогресс в Paxos)

До этого момента мы обсуждали разработку Paxos с точки зрения обеспечения **безопасности**. По сути, безопасность означает, что ничего плохого не произойдет, или, иначе говоря, поведение алгоритма будет правильным. Чтобы гарантировать, что со временем произойдет нечто хорошее, обычно называемое **живучестью** (liveness) алгоритма, нам нужно сделать несколько больше. В частности, нам нужно выйти из ситуации, изображенной на рис. 8.12.

Проблема в этой ситуации заключается в том, что серверы не имеют единого мнения о том, кто является лидером. Как только S_2 решает, что он должен взять на себя руководство, он должен быть уверен, что все невыполненные операции, инициированные S_1, были должным образом выполнены. Другими словами, он должен быть уверен, что его собственному руководству не будут препятствовать операции, которые еще не были завершены всеми неотказавшими процессами. Если руководство переходит слишком быстро и предлагается новая операция, предыдущая операция, которая была выполнена по крайней мере одним сервером, может не получить шанс быть выполненной всеми серверами.

С этой целью Paxos навязывает явное лидерство, и именно отсюда вытекает роль заявителей. Когда сервер выходит из строя, следующий в очереди должен будет стать основным (напомним, что Paxos предполагает детерминированный алгоритм выбора лидера), а также убедиться, что все невыполненные операции выполняются. Это явное поглощение реализуется путем передачи сообщения о предложении: PROPOSE(S_i), где S_i – следующий сервер, который будет лидером. Когда сервер S_j получает это сообщение, он отвечает сообщением PROMISE(o^j, t_j), содержащим самую последнюю выполненную операцию o^j и соответствующую ей метку времени t_j. Обратите внимание, что S_i особенно заинтересован в самой последней операции o^*, выполняемой большинством серверов. «Приняв» эту операцию от (очевидно, отказавшего) сервера S^*, который изначально предлагал ее принятие, S_i может эффективно завершить то, что S не смог завершить из-за своего сбоя.

Есть две очевидные оптимизации этой процедуры. Первая: серверы возвращают не самую последнюю выполненную операцию, а последнюю *обученную* операцию, которая все еще ожидает выполнения, если таковая имеется. Кроме того, так как может случиться, что у коллекции серверов больше нет ожидающих операций, S_i также может предложить следующую операцию o', когда первоначально предлагает перехватить лидерство, что приводит к сообщению предложения PROPOSE(S_i, o^i). По сути, это ситуация, которую мы описали ранее, но теперь должно быть ясно, откуда на самом деле возникает идея предложений.

Когда S_i получает большинство обещаний для операции o′, а самая высокая возвращаемая временная метка равна t^*, он передает широковещательные сообщения ACCEPT(S_i, o*, t^*), что, по сути, является ретрансляцией последней операции, предложенной до того, как S_i захватил лидерство. Если такого o не существует, S_i предложит принять собственную операцию o^i.

Консенсус в неисправных системах с произвольными отказами

До сих пор мы предполагали, что при аварийном отказе затрагивались только реплики, и в этом случае группа процессов должна состоять из $2k + 1$ серверов, чтобы выдержать k аварийных участников. Важным предположением в этих случаях является то, что процесс не вступает в договоренность с другим процессом или, более конкретно, является последовательным в своих сообщениях другим. Ситуации, показанные на рис. 8.13, возникать не должны. В первом случае мы видим, что процесс P_2 пересылает значение или операцию, отличную от той, что предполагается. Согласно Paxos, это может означать, что основной процесс сообщает резервным копиям, что не была принята операция o, а вместо этого распространяет другую операцию o′. Во втором случае P_1 сообщает разное разным процессам, например наличие лидера, отправляющего операцию o в некоторые резервные копии и в то же время операцию o′ другим. Еще раз отметим, что это должны быть не злонамеренные действия, а просто упущение или выход из строя.

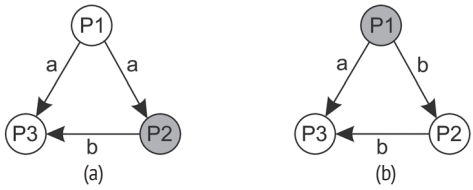


Рис. 8.13 ❖ Процесс в реплицируемой группе, действующий непоследовательно: а) неправильная пересылка; б) разные процессы

В этом разделе мы рассмотрим достижение консенсуса в отказоустойчивой группе процессов, в которой k участников могут отказать в случае произвольных отказов. В частности, мы покажем, что нам нужно, по крайней мере, $3k + 1$ членов, чтобы достичь консенсуса в соответствии с этими предположениями об отказах.

Рассмотрим группу процессов, состоящую из n членов, один из которых обозначен как *основной* (*первичный*), P, а остальные $n - 1$ – как *резервные копии* $B_1, \ldots, B_{n-1}$. Мы делаем следующие предположения:

- клиент отправляет значение v ∈ {T, F} первичному процессу, где v обозначает либо истина (true), либо ложь (false);

- сообщения могут быть потеряны, но это можно обнаружить;
- сообщения не могут быть повреждены без их обнаружения (и, следовательно, впоследствии игнорируются);
- получатель сообщения может надежно обнаружить своего отправителя.

Чтобы достичь так называемого **византийского соглашения** (Byzantine agreement, BA), нам необходимо выполнить следующие два требования:

1) **BA1**: каждый исправный процесс резервного копирования хранит одно и то же значение;
2) **BA2**: если первичный процесс не является неисправным, то каждый исправный процесс резервного копирования сохраняет именно то, что отправил основной.

Обратите внимание, что если первичный процесс неисправен, BA1 сообщает нам, что резервные копии могут хранить то же самое, но отличающееся (и, следовательно, неправильное) значение от значения, первоначально отправленного клиентом. Кроме того, должно быть ясно, что если первичный процесс не является неисправным, выполение требования BA2 подразумевает, что BA1 также выполняется.

Почему 3k процессов недостаточно

Чтобы понять, почему наличия только 3*k* процессов недостаточно для достижения консенсуса, давайте рассмотрим ситуацию, в которой мы хотим допустить сбой одного процесса, то есть $k = 1$. Рассмотрим рис. 8.14, который по сути является продолжением рис. 8.13.

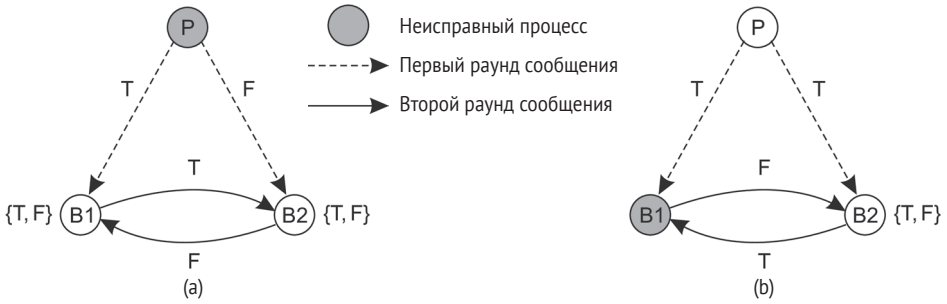


Рис. 8.14 ❖ Невозможность достичь консенсуса с тремя процессами и попытка допустить один произвольно неудачный процесс

На рис. 8.14a мы видим, что неисправный первичный P отправляет два разных значения в резервные копии B_1 и B_2 соответственно. Для достижения консенсуса оба процесса резервного копирования пересылают полученное значение другому, что приводит ко второму раунду обмена сообщениями. В этот момент каждый из B_1 и B_2 получил набор значений {T, F}, из которых невозможно сделать заключение.

Точно так же мы не можем достичь консенсуса, когда пересылаются неправильные значения. На рис. 8.14b основной процесс P и резервная копия B_2 работают правильно, а B_1 – нет. Вместо того чтобы пересылать значение T

в процесс B_2, он отправляет неверное значение F. В результате B_2 теперь увидит набор значений {T, F}, из которого он не может сделать никаких выводов. Другими словами, P и B_2 не могут достичь консенсуса. В частности, B_2 не может решить, что хранить, поэтому мы не можем удовлетворить требование BA2.

Примечание 8.5 (дополнительно: случай, когда $k > 1$ и $n \leq 3k$)

Обобщение этой ситуации для других значений k не так уж сложно. Как объяснено в [Kshemkalyani and Singhal, 2008], мы можем использовать простую схему редукции. Предположим, что есть решение для случая, когда $k \geq 1$ и $n \leq 3k$. Разбиение n процессов Q_1, ..., Q_n на три непересекающихся множества $\mathbf{S_1}$, $\mathbf{S_2}$ и $\mathbf{S_3}$, вместе содержащих все процессы. Более того, пусть каждый набор $\mathbf{S_k}$ имеет меньше или равное $n/3$ количество членов.

Формально это означает, что

- $\mathbf{S_1} \cap \mathbf{S_2} = \mathbf{S_1} \cap \mathbf{S_3} = \mathbf{S_2} \cap \mathbf{S_3} = \emptyset$;
- $\mathbf{S_1} \cup \mathbf{S_2} \cup \mathbf{S_3} = \{Q_1,\ldots,Q_n\}$;
- для каждого $\mathbf{S_i}$, $|\mathbf{S_i}| \leq n/3$.

Теперь рассмотрим ситуацию, в которой три процесса Q_1^*, Q_2^* и Q_3^* *моделируют* действия, которые происходят внутри и между процессами $\mathbf{S_1}$, $\mathbf{S_2}$ и $\mathbf{S_3}$ соответственно. Другими словами, если процесс на $\mathbf{S_1}$ отправляет сообщение другому процессу на $\mathbf{S_2}$, то Q_1^* отправит такое же сообщение Q_2. То же самое относится к процессу коммуникации в группе. Предположим, что Q_1 неисправен, а Q_2 и Q_3 нет. Предполагается, что все процессы, моделируемые Q_1, являются неисправными и, таким образом, приведут к неправильным сообщениям, отправляемым в Q_2 и Q_3 соответственно. Для Q_2 (и Q_3) это не так: все сообщения, поступающие от процессов в $\mathbf{S_2}$ (и $\mathbf{S_3}$ соответственно), считаются правильными. Поскольку $n \leq 3k$ и для каждого множества $\mathbf{S_i}$ имеем $|\mathbf{S_i}| \leq n/3$, то самое большее $n/3$ моделируемых процессов Q_1, ..., Q_n неисправны. Другими словами, мы удовлетворяем условие, для которого предполагали, что будет найдено общее решение.

Теперь мы можем встретиться с противоречием: если бы существовало решение для общего случая, то процессы Q_1, Q_2 и Q_3 могли бы смоделировать это решение, которое тогда также было бы решением для частного случая, когда $n = 3$ и $k = 1$. Но мы только что доказали, что этого не может быть, то есть пришли к противоречию. Мы заключаем, что наше предположение о том, что существует общее решение для $k \geq 1$ и $n \leq 3k$, неверно.

Почему достаточно 3k + 1 процессов

Теперь давайте сосредоточимся на случае, когда у нас есть группа из $3k + 1$ процессов.

Наша цель – показать, что мы можем найти решение, в котором k членов группы могут пострадать от случайных сбоев, но оставшиеся работающие процессы все равно достигнут консенсуса. Опять, сначала сосредоточимся на случае $n = 4$, $k = 1$. Рассмотрим рис. 8.15, где показана ситуация с одним основным P и тремя процессами резервного копирования B_1, B_2 и B_3.

На рис. 8.15а мы показали ситуацию, в которой первичный P неисправен и предоставляет несогласованную информацию для своих резервных копий. В нашем решении процессы передадут то, что они получают, другим. Во время первого раунда P отправляет T на B_1, F на B_2 и T на B_3 соответственно.

Каждая из резервных копий затем отправляет то, что у них есть, другим. Если произошел только первичный сбой, это означает, что после двух раундов каждая из резервных копий получит набор значений {T, T, F}, что означает, что они могут достичь консенсуса по значению T.

Когда мы рассматриваем случай, когда одна из резервных копий не работает, мы получаем ситуацию, показанную на рис. 8.15b. Предположим, что (исправный) первичный сервер отправляет T во все резервные копии, но B_2 неисправен. Если B_1 и B_3 отправят T другим резервным копиям во втором раунде, худшее, что может сделать B_2, – это отправить F, как показано на рисунке. Несмотря на эту ошибку, B_1 и B_3 придут к одному и тому же выводу, а именно что P разослал T, тем самым выполнив наше требование BA2, как показано выше.

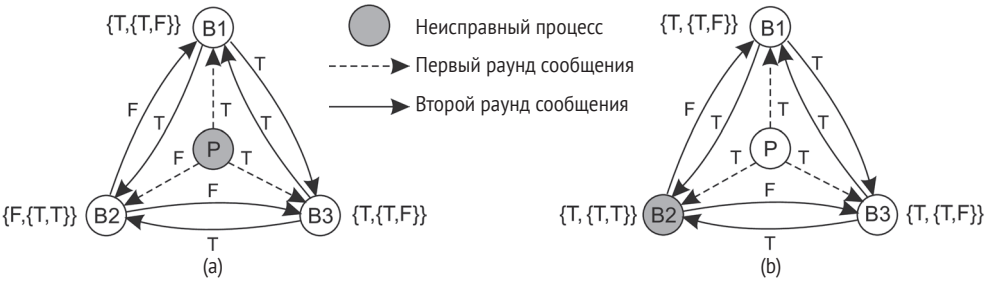


Рис. 8.15 ❖ Достижение консенсуса с четырьмя процессами, один из которых произвольно неисправен

Примечание 8.6 (дополнительно: случай, когда $k > 1$ и $n = 3k + 1$)

В качестве наброска к общему решению рассмотрим более сложный случай, когда $n = 7$ и $k = 2$. Пусть первичный P отправит значение v_0. Используя обозначения, аналогичные примененным в [Kshemkalyani and Singhal, 2008], мы действуем следующим образом.

Обратите внимание, что мы эффективно используем индекс 0 для обозначения основного P (представьте, что P равен специальному процессу резервного копирования B_0).

1. Мы разрешим P отправить v_0 шести резервным копиям. Резервная копия B_i сохраняет полученное значение как $v_{i,0}( )$. Эта запись указывает, что значение было получено процессом B_i, что оно было отправлено с помощью $P = B_0$ и что значение v_0 было отправлено непосредственно в B_i, а не через другой процесс (с использованием записи ( )). Для конкретизации рассмотрим B_4, который будет хранить v_0 в $v_{4,0}( )$.
2. Каждая резервная копия B_i, в свою очередь, будет отправлять $v_{i,0}( )$ в каждую из остальных пяти резервных копий, которая хранится в B_j как $v_{i,j}(0)$. Это обозначение указывает, что значение, сохраненное в B_j, было отправлено B_i, но оно возникло из $P = B_0$ (через обозначение (0). Концентрируясь на B_4, он в конечном итоге сохраняет $v_{4,5}(0)$, то есть значение $v_{5,0}( )$, которое отправлено из B_5 на B_4.
3. Предположим, что B_i теперь имеет значение $v_{i,j}(0)$. Он опять отправит это значение всем процессам, кроме $P = B_0$, B_j и B_i (то есть самому себе). Если B_k получает $v_{i,j}(0)$ от B_i, он сохраняет полученное значение в $v_{k,j}(j, 0)$. Действительно, к тому времени v_0 пройдет путь $P \rightarrow B_j \rightarrow B_i \rightarrow B_k$. Например, B_2 может отправить

$v_{2,1}(0)$ в B_4, который сохранит его как $v_{4,2}(1,0)$. Обратите внимание, что B_4 может отправлять *это* значение только процессам B_3, B_5 и B_6. Нет смысла отправлять его другим процессам.

4. Продолжая эту мысль, предположим, что B_i имеет значение $v_{i,j}(k, 0)$, которое оно отправляет трем не равным оставшимся процессам, $P = B_0$, B_k, B_j и B_i (самому себе). Вернемся к B_4. B_4 также получит аналогичное сообщение от, скажем, B_2, приводящее, возможно, к значению $v_{4,2}(3, 1, 0)$. Это значение может быть отправлено только на B_5 и B_6, после чего остается лишь один раунд.

После отправки всех этих сообщений каждая резервная копия может снова выйти из рекурсии. Давайте опять посмотрим на процесс B_4. В последнем раунде будет храниться всего пять сообщений:

- $v_{4,1}(6, 5, 3, 2, 0)$;
- $v_{4,2}(6, 5, 3, 1, 0)$;
- $v_{4,3}(6, 5, 2, 1, 0)$;
- $v_{4,5}(6, 3, 2, 1, 0)$;
- $v_{4,6}(5, 3, 2, 1, 0)$.

С этими пятью значениями он может начать вычислять *оценки* v_0, то есть значение, которое, по его мнению, должно быть v_0. Для этого мы предполагаем, что каждый (исправный) процесс выполняет одну и ту же процедуру *majority*(), которая выбирает уникальное значение из заданного набора входных данных. На практике это будет большинство среди входного набора. Если большинства нет, выбирается значение по умолчанию. Приведем несколько примеров:

$$w_{4,1}(5, 3, 2, 0) \leftarrow majority(v_{4,1}(5, 3, 2, 0), v_{4,1}(6, 5, 3, 2, 0))$$
$$w_{4,1}(6, 3, 2, 0) \leftarrow majority(v_{4,1}(6, 3, 2, 0), v_{4,2}(6, 5, 3, 1, 0))$$
$$w_{4,5}(3, 2, 1, 0) \leftarrow majority(v_{4,5}(3, 2, 1, 0), v_{4,5}(6, 3, 2, 1, 0))$$
$$w_{4,5}(6, 3, 2, 0) \leftarrow majority(v_{4,5}(6, 3, 2, 0), v_{4,5}(6, 3, 2, 1, 0))$$
$$w_{4,6}(3, 2, 1, 0) \leftarrow majority(v_{4,6}(3, 2, 1, 0), v_{4,6}(5, 3, 2, 1, 0))$$
$$w_{4,6}(5, 3, 2, 0) \leftarrow majority(v_{4,6}(5, 3, 2, 0), v_{4,1}(6, 5, 3, 2, 0))$$

В свою очередь, это позволяет вычислять оценки, такие как:

$$w_{4,1}(3, 2, 0) \leftarrow majority(v_{4,1}(3, 2, 0), w_{4,5}(3, 2, 1, 0), w_{4,6}(3, 2, 1, 0))$$
$$w_{4,5}(3, 2, 0) \leftarrow majority(v_{4,5}(3, 2, 0), w_{4,1}(5, 3, 2, 0), w_{4,6}(5, 3, 2, 0))$$
$$w_{4,6}(3, 2, 0) \leftarrow majority(v_{4,6}(3, 2, 0), w_{4,1}(6, 3, 2, 0), w_{4,5}(6, 3, 2, 0))$$

И отсюда, например:

$$w_{4,3}(2, 0) \leftarrow majority(v_{4,3}(2, 0), w_{4,1}(3, 2, 0), w_{4,5}(3, 2, 0), w_{4,6}(3, 2, 0))$$

Этот процесс продолжается до тех пор, пока, в конце концов, B_4 не сможет выполнить

$$w_{4,0}(\,) \leftarrow majority(v_{4,0}(\,), w_{4,1}(0), w_{4,2}(0), w_{4,3}(0), w_{4,5}(0), w_{4,6}(0))$$

и достичь финального результата.

Теперь давайте посмотрим, почему эта схема действительно работает. Обозначим через **BAP(n, k)** приведенный выше набросок протокола для достижения консенсуса. **BAP(n, k)** начинает с того, что основной сервер отправляет свое значение v_0 в $n - 1$ резервных копий. В случае если основной сервер работает правильно, каждая из резервных копий действительно получит v_0. Если основной файл работает корректно, некоторые резервные копии получают v_0, в то время как другие получают $\overline{v0}$ (т. е. противоположность v_0).

Поскольку мы предполагаем, что резервная копия B_i не может знать, правильно ли работает первичный сервер, он должен будет проверить другие резервные копии. Поэтому мы позволяем B_i снова запустить протокол, но в этом случае со значе-

нием $v_{i,0}(\,)$ и с меньшей группой процессов, а именно $\{B_1, \ldots, B_{i-1}, B_{i+1}, \ldots, B_n\}$. Другими словами, B_i выполняет **BAP($n-1, k-1$)** с общим количеством $n-2$ других процессов. Обратите внимание, что на данный момент существует $n-1$ экземпляров **BAP($n-1, k-1$)**, выполняемых параллельно.

В итоге мы видим, что эти выполнения приводят к тому, что каждая резервная копия B_i принимает большинство из значений $n-1$:

- одно значение получается из первичного: $v_{i,0}()$;
- $n-2$ значений поступают из других резервных копий, в частности поступающее из B_i имеет дело со значениями $v_{i,1}(\,), \ldots, v_{i,i-1}(\,), v_{i,i+1}(\,), \ldots, v_{i,n-1}(\,)$.

Однако, поскольку B_i не может доверять полученному значению $v_{i,j}(\,)$, он должен будет проверить это значение с другими $n-2$ резервными копиями: $\{B_1, \ldots, B_{i-1}, B_{i+1}, \ldots, B_{j-1}, B_{j+1}, \ldots, B_{n-1}\}$. Это приводит к выполнению **BAP($n-2, k-2$)**, из которых в общей сложности $n-2$ экземпляров будут работать параллельно. Эта история продолжается, и в конечном итоге процесс резервного копирования должен будет запустить **BAP($n-k, 0$)**, которое просто возвращает значение, отправленное первичным сервером, после чего мы можем переместить рекурсию, как описано выше.

Примечание 8.7 (дополнительно: правильность протокола византийского соглашения)

Из общей схемы, приведенной в примечании 8.6, нетрудно понять, почему протокол правильный. Следуя [Koren and Krishna, 2007], мы используем индукцию по k, чтобы доказать, что **BAP(n, k)** соответствует требованиям BA1 и BA2 для $n \geq 3k + 1$ и для всех $k \geq 0$.

Сначала рассмотрим случай $k = 0$. Другими словами, мы предполагаем, что нет дефектных процессов. В этом случае, что бы первичный сервер ни отправлял в резервные копии, это значение будет последовательно распространяться по всей системе, и никакое другое значение никогда не появится. Другими словами, для любого n **BAP($n, 0$)** является правильным. Теперь рассмотрим случай $k > 0$.

Сначала разберем случай, когда основной процесс работает правильно. Без потери общности мы можем предположить, что основной сервер отправляет T. Все резервные копии получают одно и то же значение, а именно T. Каждая резервная копия будет тогда выполнять **BAP($n-1, k-1$)**. По индукции мы знаем, что каждый из этих экземпляров будет выполнен правильно. Это означает, что для любой исправной резервной копии B все другие исправные резервные копии будут хранить значение, отправленное B, а именно T. Каждая неисправная резервная копия получает в общей сложности $n-1$ значений, из которых $n-2$ поступают из других резервных копий. Из этих $n-2$ не более k значений могут быть неправильными (то есть F). С $k \leq (n-1)/3$ это означает, что каждая исправная резервная копия получает по крайней мере $1 + (n-2) - (n-1)/3 = (2n-2)/3$ значения T. Поскольку $(2n-2)/3 > n/3$ для всех $n > 2$, это означает, что каждая исправная резервная копия может получить правильное большинство голосов от общего числа полученных значений, удовлетворяя, таким образом, требованию BA2.

Давайте теперь рассмотрим случай, когда первичная копия неисправна, а это означает, что не более $k-1$ резервных копий также могут работать неправильно. Предполагается, что основной сервер отправляет любое значение, которое ему нравится. Всего существует $n-1$ резервных копий, из которых не более $k-1$ являются неисправными. Каждое резервное копирование выполняет **BAP($n-1, k-1$)**, и по индукции каждый из этих экземпляров выполняется правильно. В частности, для каждой исправной резервной копии B все другие исправные резервные копии будут голосовать за значение, отправленное B. Это означает, что все исправные резерв-

ные копии будут иметь один и тот же вектор из $n - 2$ результатов своих резервных копий. Любая разница между двумя исправными резервными копиями может быть вызвана только тем фактом, что первичный сервер отправил что-то еще каждому из них. В результате при применении *majority*() к этим полным векторам результат для каждой резервной копии будет одинаковым, так что требование BA1 будет выполнено.

Пример: практическая византийская отказоустойчивость

Византийская отказоустойчивость долгое время была относительно экзотической темой отчасти потому, что оказалось, что объединить безопасность, жизнеспособность и практическую работу было трудно. Примерно в 2000 году Барбаре Лисков (Barbara Liskov) и в то время ее ученице Мигеле Кастро (Miguel Castro) удалось придумать практическую реализацию протокола для репликации серверов, способных обрабатывать произвольные сбои. Давайте кратко рассмотрим их решение, которое было названо **практическая византийская отказоустойчивость** (Practical Byzantine Fault Tolerance, PBFT) [Castro and Liskov, 2002]).

Как и Paxos, PBFT делает только несколько предположений о своей среде. Он не делает никаких предположений о поведении серверов реплики: предполагается, что неисправный сервер демонстрирует произвольное поведение. Кроме того, сообщения могут быть потеряны, задержаны и получены не по порядку. Однако предполагается, что отправитель сообщения может быть идентифицирован (что достигается подписью сообщений, как мы обсудим в главе 9). Исходя из этих предположений и до тех пор, пока не выходит из строя не более k серверов, можно доказать, что PBFT безопасен, а это означает, что клиент всегда получит правильный ответ. Если мы можем дополнительно предположить синхронность, что означает, что задержки сообщений и время ответа ограничены, это также обеспечивает жизнеспособность. На практике это означает, что PBFT предполагает частично синхронную модель, в которой неограниченные задержки являются исключением, например, вызванным атакой.

Чтобы понять алгоритм, давайте сделаем шаг назад и частично рассмотрим то, что мы уже обсуждали при создании группы процессов, устойчивых к отказам. Существенной проблемой является то, что такая группа ведет себя как единый центральный сервер. Как следствие в предположении наличия только аварийных отказов, когда клиент отправляет запрос, он должен ожидать $k + 1$ идентичных ответов. В случае сбоя сервера будет получено меньше ответов, но они будут одинаковыми.

Первая проблема, которую нам нужно решить, состоит в том, что все параллельные запросы обрабатываются в одном и том же порядке. С этой целью PBFT использует модель первичного резервного копирования с общим количеством серверов реплики $3k + 1$. Для простоты давайте пока предположим, что первичный сервер исправен. В этом случае клиент C отправляет запрос на выполнение операции o первичному серверу (обозначен как P на рис. 8.16). Первичный сервер имеет представление о текущей коллекции исправных серверов реплик, выраженной в терминах **представления** v, ко-

торое является простым числом. Первичный сервер назначает временную метку t для o, которая затем увеличивается, чтобы использовать ее для последующего запроса. Затем первичный сервер отправляет (подписанное) сообщение о предварительной подготовке (t, v, o) в резервные копии.

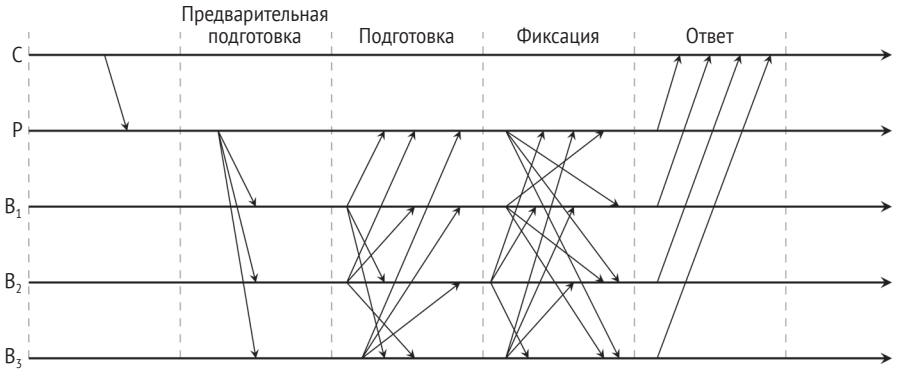


Рис. 8.16 ❖ Различные фазы в PBFT. C является клиентом, P – основным сервером, а B_1, B_2, B_3 являются резервными копиями. Мы предполагаем, что B_2 неисправен

Резервная копия (исправная) примет *предварительную подготовку* (*preprepare*), если она находится в v и никогда ранее не принимала операцию с отметкой времени t в v. Каждая резервная копия, которая принимает *предварительную подготовку*, отправляет (снова подписанное) сообщение подготовки PREPARE(t, v, o) другим, включая основную. Ключевой аспект состоит в том, что когда исправный сервер реплик S получил $2k$ подготовленных сообщений PREPARE(t, v, o), все они соответствуют самому сообщению *предварительной подготовки* S, полученному первичным сервером (то есть все имеют одинаковое значение для t, v и o соответственно), среди исправных серверов существует консенсус в отношении того, какая операция выполняется в первую очередь. Чтобы понять, почему, пусть **сертификат подготовки** (prepare certificate) PC(t,v,o) обозначает сертификат, основанный на таком наборе $2k + 1$ сообщений. Пусть теперь PC(t, v, o′) будет другим подготовительным сертификатом с теми же значениями для t и v соответственно, но с другой операцией o′. Поскольку каждый подготовительный сертификат основан на значениях $2k + 1$ от общего количества серверов реплики $3k + 1$, пересечение двух сертификатов обязательно будет основано на сообщениях из подмножества не менее $k + 1$ серверов. Из этого подмножества мы знаем, что есть по крайней мере один исправный сервер, который отправит такое же сообщение о подготовке. Следовательно, o = o′.

После того как сервер реплики получил сертификат подготовки, он фиксирует операцию, передавая COMMIT(t, v, o) другим участникам в v. Каждый сервер S, в свою очередь, собирает $2k$ таких сообщений фиксации с других серверов, что приводит к **сертификату фиксации** (commit certificate) выполнения операции o. В этот момент он выполняет o и отправляет ответ

клиенту. Опять, при наличии своего собственного сообщения и $2k$ других сообщений S знает, что среди неисправных серверов существует консенсус относительно того, какую операцию фактически следует сейчас выполнить. Клиент собирает все результаты и принимает в качестве ответа ответ, который возвращается по крайней мере $k + 1$ репликами, из которых он знает, что есть по крайней мере один исправный сервер реплики, способствующий этому ответу.

Итак, все хорошо. Однако мы также должны иметь дело с ситуацией, когда отказывает основной сервер. Если резервная копия обнаруживает отказ основного (первичного) сервера, она передает сообщение об изменении представления на представление $v + 1$. Мы хотим установить, что запрос, который еще обрабатывался во время отказа первичного сервера, в конечном итоге будет выполнен *один раз и только один раз* всеми исправными серверами. Для этого нам необходимо убедиться, что не существует двух сертификатов фиксации с одинаковой отметкой времени, которые имеют разные связанные операции, независимо от представления, с которым связан каждый из них. Эту ситуацию можно предотвратить, если иметь кворум $2k + 1$ сертификатов фиксации, как и прежде, но на этот раз на основе сертификатов подготовки. Другими словами, мы хотим восстановить сертификаты фиксации, но теперь для нового представления, и только для того, чтобы убедиться, что на исправном сервере не пропущены никакие операции. В этом отношении обратите внимание, что мы можем генерировать сертификат для операции, которую сервер S уже выполнил (что можно понять, посмотрев на временные метки), но этот сертификат будет игнорироваться S, пока он хранит учетную запись своей истории выполнения.

Резервный сервер будет передавать (подписанное) сообщение об изменении вида VIEW-CHANGE($v + 1$, **P**), где **P** – это набор его подготовительных сертификатов. (Обратите внимание, что мы игнорируем проблемы сбора мусора.) PBFT включает детерминированную первичную функцию (w), известную всем резервным копиям, которая возвращает информацию о том, кому из следующих первичных данных будет предоставлено представление w. Эта резервная копия будет ждать до тех пор, пока не получит в общей сложности $2k + 1$ сообщений об изменении представления сообщений, что приведет к **сертификату изменения представления X** (view-change certificate X) подготовительных сертификатов. Затем новый первичный сервер передает новое представление NEW-VIEW($v + 1$, **X**, **O**), где **O** состоит из сообщений предварительной подготовки, так что:

- предварительная подготовка PRE-PREPARE(t, $v + 1$, o) ∈ **O**, если подготовительный сертификат $PC(t, v', o) \in X$ такой, что не существует подготовительного сертификата $PC(t, v'', o')$ с $v'' > v'$,
  или
- предварительная подготовка PRE-PREPARE(t, $v + 1$, *none*) ∈ **O**, если нет сертификата подготовки $PC(t, v', o') \in$ **X**.

Что происходит, так это то, что любая незавершенная, предварительно подготовленная операция из предыдущего представления перемещается в новое представление, но с учетом только самого последнего представления, которое привело к установке текущего нового представления. Немного

упрощая, можно сказать, что каждая резервная копия будет проверять **O** и **X**, чтобы удостовериться, что все операции действительно являются подлинными, и транслировать сообщения подготовки для всех сообщений предварительной подготовки в **O**.

Мы пропустили много элементов PBFT, которые касаются его правильности и прежде всего его эффективности. Например, мы не затрагивали журналы сбора мусора или эффективные способы аутентификации сообщений. Такие подробности можно найти в [Castro and Liskov, 2002]. Описание оболочки, которая позволит объединить византийскую отказоустойчивость с устаревшими приложениями, присутствует в [Castro et al., 2003]. В частности, эффективность византийской отказоустойчивости была предметом многочисленных исследований, что привело ко многим новым протоколам (см., например, [Zyzzyva Kotla et al., 2009] и Abstract [Guerraoui et al., 2010]), но даже эти новые предложения часто полагаются на оригинальную реализацию PBFT. То, что есть еще возможности для улучшения, когда фактически используется PBFT для разработки надежных приложений, обсуждается в [Chondros et al., 2012]. Например, PBFT предполагает статическое членство (то есть клиенты и серверы заранее известны друг другу), но также предполагает, что память сервера реплики действует как стабильное, постоянное хранилище. Эти и другие недостатки наряду с присущей византийской отказоустойчивостью и сложностью стали препятствием для его широкого использования.

Некоторые ограничения по реализации отказоустойчивости

Организация реплицированных процессов в группу помогает повысить отказоустойчивость. Однако к настоящему времени должно быть понятно, что за это приходится платить потенциальной потерей производительности. В решениях, которые обсуждались до сих пор, процессам в отказоустойчивой группе может потребоваться обмен многочисленными сообщениями, прежде чем принимать решение. Протокол византийского соглашения – превосходная иллюстрация того, насколько тесно могут быть связаны процессы. Возникает вопрос, всегда ли возможна реализация конкретных форм отказоустойчивости, таких как способность противостоять произвольным отказам.

Относительно достижения консенсуса

Как мы уже упоминали, если клиент может основывать свои решения с помощью механизма голосования, мы можем допустить, что k из $2k + 1$ процессов лгут о своем результате. Однако мы предполагаем, что процессы не объединяются для получения неправильного результата. В общем, все становится более сложным, если мы требуем, чтобы группа процессов достигла консенсуса, что необходимо во многих случаях. Существует три требования для достижения консенсуса [Fischer et al., 1985]:

1) процессы выдают одно и то же выходное значение;
2) каждое выходное значение должно быть действительным;
3) каждый процесс должен в конечном итоге обеспечивать вывод.

Некоторые примеры, когда необходимо достичь консенсуса, включают выбор координатора, принятие решения о совершении транзакции или разделение задач между работниками. Когда все коммуникации и процессы совершенны, достижение консенсуса часто бывает простым, но когда это не так, возникают проблемы.

Общая цель алгоритмов распределенного консенсуса состоит в том, чтобы все неработающие процессы достигли консенсуса по какой-либо проблеме и установили этот консенсус в течение конечного числа шагов. Проблема усложняется тем фактом, что различные предположения о базовой системе требуют разных решений, даже при условии, что решения существуют. В работе [Turek and Shasha, 1992] выделяются следующие случаи:

- синхронные и асинхронные системы. Несколько перефразируя наше прежнее определение, система является **синхронной** (synchronous) тогда и только тогда, когда известно, что процессы работают в режиме ступенчатой блокировки. Формально это означает, что должна быть некоторая постоянная $c \geq 1$, такая, что если какой-либо процесс сделал $c + 1$ шагов, любой другой процесс сделал хотя бы 1 шаг;
- задержка связи ограничена или нет. Задержка ограничена тогда и только тогда, когда мы знаем, что каждое сообщение доставляется с глобальным и заранее определенным максимальным временем;
- доставка сообщений заказана (в режиме реального времени) или нет. Другими словами, мы отличаем ситуацию, когда сообщения от разных отправителей доставляются в том порядке, в котором они были отправлены в реальном глобальном времени, от ситуации, в которой у нас нет таких гарантий;
- передача сообщений осуществляется посредством одноадресной или многоадресной рассылки.

Как оказалось, достижение консенсуса возможно только для ситуаций, показанных на рис. 8.17. Можно показать, что во всех других случаях решения не существует. Обратите внимание, что на практике большинство распределенных систем предполагают, что процессы ведут себя асинхронно, передача сообщений является одноадресной, а задержки связи не ограничены. Как следствие нам необходимо использовать заказанную (надежную) доставку сообщений, например предоставляемую TCP. Опять-таки, в практических ситуациях мы принимаем синхронное поведение по умолчанию, но учитываем, что могут быть и неограниченные задержки. Рисунок 8.17 иллюстрирует нетривиальную природу распределенного консенсуса, когда процессы могут потерпеть неудачу.

Достижение консенсуса может быть невозможным. В работе [Fischer et al., 1985] доказано, что если нельзя гарантировать, что сообщения будут доставлены в течение известного, конечного времени, невозможно достичь консенсуса, если хотя бы один процесс неисправен (хотя этот процесс молча завершается неудачей). Проблема таких систем заключается в том, что произвольно медленные процессы неотличимы от отказавших (то есть вы не

можете отличить мертвых от живых). Эти и другие теоретические результаты приведены в обзоре [Barborak et al., 1993] и работе [Turek and Shasha, 1992].

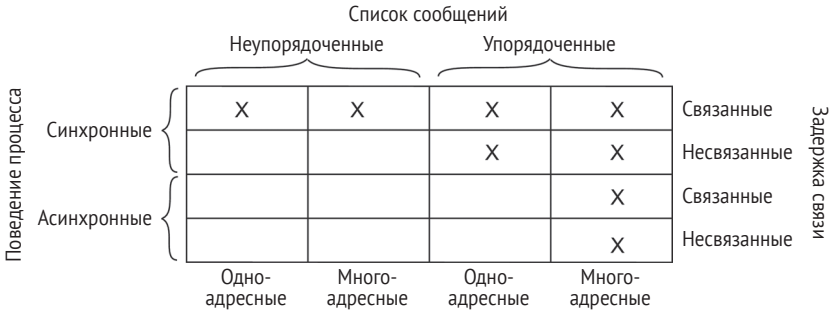


Рис. 8.17 ❖ Обстоятельства, при которых может быть достигнут распределенный консенсус

Согласованность, доступность и разделение

Достижение согласованности сильно зависит от условий, при которых консенсус может (не) быть достигнут. Согласованность в этом случае означает, что когда у нас есть группа процессов, в которую клиент отправляет запросы, то ответы, возвращенные этому клиенту, являются правильными. Мы имеем дело со **свойством безопасности** (safety property): свойством, которое утверждает, что ничего плохого не произойдет. Для наших целей операции, которые мы рассматриваем, выполняются в четко определенном порядке одним централизованным сервером. К настоящему времени мы знаем больше: эти операции выполняются группой процессов, чтобы противостоять сбоям *k* членов группы.

Мы ввели группы процессов для повышения отказоустойчивости и, более конкретно, для повышения доступности. Доступность – это, как правило, свойство **живучести** (liveness): в конце концов, все будет хорошо. Что касается наших групп процессов, мы стремимся в конечном итоге получить (правильный) ответ на каждый запрос, выданный клиентом. Быть последовательным в ответах и в то же время высокодоступным не является необоснованным требованием для служб, которые являются частью распределенной системы. К сожалению, мы, может быть, просим слишком много.

В практических ситуациях наше предположение о том, что процессы в группе действительно могут общаться друг с другом, может быть ложным. Сообщения могут быть потеряны; группа может быть разделена из-за неисправной сети. В 2000 году Эрик Брюер (Eric Brewer) сформулировал важную теорему, которая позже была подтверждена в [Gilbert and Lynch 2002]:

Теорема CAP. *Любая сетевая система, предоставляющая общие данные, может предоставлять только два из следующих трех свойств:*

- ***C*:** *согласованность, при которой общий и реплицируемый элемент данных отображается в виде единой актуальной копии;*
- ***A*:** *доступность, с помощью которой всегда будут выполняться обновления;*
- ***P*:** *допустимость разделения группы процессов (например, из-за сбоя сети).*

Другими словами, в сети, подверженной сбоям связи, невозможно реализовать атомарную разделяемую память для чтения/записи, которая гарантирует ответ на каждый запрос [Gilbert and Lynch, 2012].

Эта теорема известна как **теорема CAP**. Она впервые опубликована в [Fox and Brewer, 1999]. Как объяснил Брюер [Brewer, 2012], один из способов понять теорему – это подумать о двух процессах, которые не могут обмениваться данными из-за неисправной сети. Разрешение одному процессу принимать обновления приводит к несогласованности, поэтому мы можем иметь только свойства {*C*, *P*}. Если должна быть обеспечена иллюзия согласованности, в то время как два процесса не могут обмениваться данными, то один из двух процессов должен будет притвориться недоступным, подразумевая наличие лишь {*A*, *P*}. Только если два процесса могут обмениваться данными, можно поддерживать как согласованность, так и высокую доступность, а это означает, что у нас есть лишь {*C*, *A*}, но больше нет свойства *P*.

Обратите внимание также на связь с достижением консенсуса; фактически, когда консенсус требует доказать, что процессы производят одинаковый результат, обеспечение согласованности слабее. Это также означает, что если достижение CAP невозможно, то и консенсус тоже.

Теорема CAP полностью сводится к достижению компромисса между безопасностью и живучестью, основанного на наблюдении, что достижение обоих в изначально ненадежной системе не может быть достигнуто. Практические распределенные системы ненадежны по своей природе. Брюер и его коллеги заметили, что в практических распределенных системах нужно просто сделать выбор, несмотря на то что другой процесс не может быть достигнут. Иными словами, нам нужно что-то делать, когда разделение проявляет себя большой задержкой.

Суть в том, что если кажется, что разделение имеет место, следует продолжить (допуская разделение в пользу согласованности или доступности), одновременно запуская процедуру восстановления, которая может смягчить последствия потенциальных несогласованностей. Точное решение о том, как поступить, зависит от приложения: во многих случаях наличие дублирующих ключей в базе данных может быть легко установлено (это подразумевает, что мы должны допускать несогласованность), в то время как дублирующие переводы больших сумм денег могут и не быть (это означает, что мы должны решиться терпеть меньшую доступность). Можно утверждать, что теорема CAP существенно перемещает проектировщиков распределенных систем от теоретических решений к инженерным решениям. Чтобы узнать, как можно сделать такой шаг, заинтересованный читатель может обратиться к [Brewer, 2012].

Обнаружение отказов

Из наших обсуждений, возможно, стало ясно, что для правильной маскировки отказов нам, как правило, необходимо также их обнаруживать. Обнаружение отказов является одним из краеугольных камней отказоустойчивости в распределенных системах. Для группы процессов это означает, что исправ-

ные участники группы должны иметь возможность решать, кто еще остается участником, а кто нет. Другими словами, мы должны быть в состоянии знать, что участник потерпел неудачу.

Что касается обнаружения отказов процесса, то, по сути, есть только два механизма. Либо процессы активно отправляют сообщения «ты жив?» друг другу (от которых они, очевидно, ожидают ответа), либо пассивно ждут, пока сообщения не поступят от различных процессов. Последний подход имеет смысл только тогда, когда можно гарантировать, что общения для этого достаточно.

Существует огромное количество теоретических работ по детекторам отказов. Все они сводятся к тому, что для проверки наличия отказа используется механизм тайм-аута. Говорят, что если процесс P проверяет другой процесс Q, чтобы выяснить, не произошел ли отказ, то P начинает подозревать, что Q потерпел крах, если Q не ответил в течение некоторого времени.

Примечание 8.8 (более подробная информация: об идеальных детекторах отказов)

Должно быть ясно, что в синхронной распределенной системе предполагаемый сбой соответствует известному сбою. На практике, однако, часто приходится иметь дело с частично синхронными системами. В этом случае имеет смысл иметь дело с совершенными детекторами отказов. Тогда, если другой процесс Q не ответил на зонд P после истечения t временных единиц, процесс P будет подозревать, что Q завершился сбоем. Однако если Q позже отправит сообщение, которое (также) получено и P, процесс P (1) прекратит подозревать Q, и (2) увеличит значение времени ожидания t. Обратите внимание, что если Q не работает (и не восстанавливается), P будет продолжать подозревать Q.

В реальных настройках возникают проблемы с использованием детекторов и тайм-аутов. Например, из-за ненадежных сетей простое утверждение о том, что процесс не выполнен, поскольку он не возвращает ответ на тестовое сообщение, может быть неправильным. Другими словами, довольно легко генерируются ложные срабатывания. Если ложный положительный результат приводит к тому, что совершенно здоровый процесс удаляется из списка участников, то, очевидно, мы делаем что-то не так. Другая серьезная проблема заключается в том, что тайм-ауты весьма приблизительны. Как отмечено в [Birman, 2012], вряд ли найдется работа по созданию хороших подсистем обнаружения сбоев, в которых учитывается больше, чем просто отсутствие ответа на одиночное сообщение. Это утверждение становится еще более очевидным при рассмотрении эксплуатируемых в отрасли распределенных систем.

Существуют различные проблемы, которые необходимо учитывать при разработке подсистемы обнаружения отказов (см. также [Zhuang et al., 2005]). Например, обнаружение отказов может происходить посредством сплетен, в которых каждый узел регулярно сообщает своим соседям, что он все еще работает. Как мы уже упоминали, альтернатива состоит в том, чтобы позволить узлам активно проверять друг друга.

Обнаружение отказов также может быть выполнено как побочный эффект регулярного обмена информацией с соседями, как в случае распространения информации на основе сплетен (о чем мы говорили в главе 4). Этот подход по существу принят также в системе Obduro [Vogels, 2003]: процессы периодически сплетничают о доступности своих услуг. Подобная информация постепенно распространяется через сеть. В конце концов, каждый процесс будет знать о каждом другом процессе, но, что более важно, будет иметь достаточно информации, доступной локально, чтобы решить, произошел сбой процесса или нет. Участник, для которого информация о доступности устарела, предположительно потерпел неудачу.

Еще одна важная проблема заключается в том, что в идеале подсистема обнаружения отказов должна отличать отказы сети от отказов узлов. Один из способов решения этой проблемы – не позволить одному узлу решить, произошел ли отказ одного из его соседей. Вместо этого, замечая тайм-аут в тестовом сообщении, узел запрашивает других соседей, чтобы выяснить, могут ли они достичь предполагаемого неисправного узла. Конечно, положительная информация также может быть передана: если узел еще жив, эта информация может быть передана другим заинтересованным сторонам (которые могут обнаружить отказ соединения с подозреваемым узлом).

Это подводит нас к иной ключевой проблеме: как следует информировать другие работающие процессы, когда обнаруживается отказ участника? Один простой и несколько радикальный подход заключается в следующем. Как предлагается в FUSE [Dunagan et al., 2004], процессы могут быть объединены в группу, охватывающую глобальную сеть. Члены группы создают связующее дерево, которое используется для мониторинга отказов членов. Участники отправляют сообщения *ping* своим соседям. Когда сосед не отвечает, проверяющий узел немедленно переключается в состояние, в котором он также больше не будет отвечать на эхо-запросы от других узлов. Посредством рекурсии отказ одного узла быстро преобразуется в уведомление об отказе группы.

8.3. Надежная связь клиент-сервер

Во многих случаях отказоустойчивость в распределенных системах концентрируется на неисправных процессах. Однако необходимо также учитывать сбои связи. Большинство рассмотренных ранее моделей отказов одинаково хорошо применимы и к каналам связи. В частности, канал связи может демонстрировать сбои, пропуски, временны́е задержки и произвольные сбои. На практике при создании надежных каналов связи основное внимание уделяется маскировке аварий и пропусков. Произвольные сбои могут возникать в форме повторяющихся сообщений, что связано с тем, что в компьютерной сети сообщения могут буферизироваться в течение относительно длительного времени и повторно вводиться в сеть после того, как исходный отправитель уже отправил повторную передачу (см., например, [Tanenbaum и Wetherall, 2010]).

Двухточечная связь

Во многих распределенных системах надежная двухточечная связь устанавливается с использованием надежного транспортного протокола, такого как TCP. Протокол TCP маскирует пропуски сбоев, которые возникают в виде потерянных сообщений с использованием подтверждений и повторных передач. Такие сбои полностью скрыты от TCP-клиента.

Однако сбои соединений не маскируются. Аварийный отказ может произойти, когда (по какой-либо причине) TCP-соединение будет внезапно разорвано и сообщения больше не передаются по каналу. В большинстве случаев клиент информируется об отказе канала, вызвав исключение. Единственный способ маскировать такие отказы – позволить распределенной системе автоматически установить новое соединение, просто повторно отправив запрос на подключение. Основное предположение заключается в том, что другая сторона все еще или снова начинает реагировать на такие запросы.

Семантика RPC при наличии отказов

Давайте теперь подробнее рассмотрим взаимодействие клиент-сервер при использовании средств связи высокого уровня, таких как **удаленные вызовы процедур** (Remote Procedure Calls, RPC).

Цель RPC – скрыть связь, делая удаленные вызовы процедур похожими на локальные. За некоторыми исключениями, мы подошли к этой цели довольно близко. Действительно, до тех пор, пока и клиент, и сервер работают отлично, RPC хорошо выполняет свою работу. Проблема возникает, когда возникают ошибки. Именно тогда различия между локальными и удаленными вызовами не всегда легко замаскировать.

Чтобы структурировать наше обсуждение, давайте различать пять различных классов сбоев, которые могут возникнуть в системах RPC, следующим образом:

1) клиент не может найти сервер;
2) сообщение запроса от клиента к серверу потеряно;
3) сервер отказывает после получения запроса;
4) ответное сообщение с сервера клиенту теряется;
5) клиент пропадает после отправки запроса.

Каждая из этих категорий ставит разные проблемы и требует разных решений.

Клиент не может найти сервер

Начнем с того, что клиент не может найти подходящий сервер. Например, все серверы могут быть недоступны. В качестве альтернативы предположим, что клиент скомпилирован с использованием определенной версии клиентской заглушки, а двоичный файл не используется в течение значительного периода времени. Тем временем сервер развивается, устанавливается новая версия интерфейса, генерируются и вводятся в эксплуатацию новые заглуш-

ки. Когда клиент в конце концов запустится, связующее звено не сможет сопоставить его с сервером и сообщит о сбое. Хотя этот механизм используется для защиты клиента от случайной попытки связаться с сервером, который может отказать из-за несоответствия требуемых параметров или того, что он должен делать, проблема того, как следует устранять этот сбой, остается.

Одно из возможных решений – иметь ошибку, вызывающую **исключение** (exeption). В некоторых языках (например, Java) программисты могут писать специальные процедуры, которые вызываются при определенных ошибках, таких как деление на ноль. В языке C для этой цели могут использоваться обработчики сигналов. Другими словами, мы могли бы определить новый тип сигнала SIGNOSERVER и позволить обрабатывать его так же, как и другие сигналы.

У данного подхода тоже есть недостатки. Начнем с того, что не у каждого языка есть исключения или сигналы. Другой аспект заключается в том, что необходимость написать обработчик исключения или сигнала разрушает прозрачность, которую мы стремились достичь.

Предположим, что вы программист, и ваш босс говорит вам написать процедуру добавления (`append`). Вы улыбаетесь и говорите, что это будет написано, проверено и задокументировано в течение пяти минут. Затем босс говорит, что вы также должны написать обработчик исключений, просто на случай, если процедура не понадобится в определенное время. В таком случае довольно сложно поддерживать иллюзию, что удаленные процедуры ничем не отличаются от локальных, так как написание обработчика исключений для «Не удается найти сервер» было бы довольно необычным запросом в нераспределенной системе. Это уже слишком для прозрачности.

Потерянные сообщения запроса

Второй пункт в списке имеет дело с потерянными сообщениями запроса. С этим разобраться проще всего: достаточно, чтобы операционная система или заглушка клиента запускали таймер при отправке запроса. Если время таймера истекает до того, как ответ или подтверждение возвращаются, сообщение отправляется снова. Если сообщение было действительно потеряно, сервер не сможет определить разницу между повторной передачей и оригиналом, и все будет работать. Если, конечно, так много сообщений с запросами потеряно, что клиент сдается и ложно приходит к выводу, что сервер не работает, в этом случае мы возвращаемся к «Не удается найти сервер». Если запрос не был потерян, единственное, что нам нужно сделать, – это позволить серверу определить, что он имеет дело с повторной передачей. К сожалению, сделать это не так просто, как мы об этом сказали.

Сбои сервера

Следующая неисправность в нашем списке – сбой сервера. Нормальная последовательность событий на сервере показана на рис. 8.18a. Запрос приходит, выполняется, и ответ отправляется. Теперь рассмотрим рис. 8.18b. Запрос поступает и выполняется, как и раньше, но сервер отказывает, прежде

чем он может отправить ответ. Наконец, посмотрите на рис. 8.18c. Снова приходит запрос, но на этот раз сервер отказывает до того, как запрос может быть выполнен. И конечно же, ответ не возвращается.

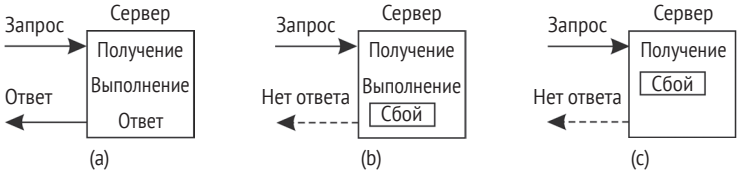


Рис. 8.18 ❖ Сервер в коммуникации клиент-сервер:
a) нормальный случай; б) сбой после выполнения; c) сбой до выполнения

Раздражающая часть рис. 8.18 состоит в том, что правильное поведение отличается для b и c. В b система должна сообщить клиенту о сбое (например, вызвать исключение), тогда как в c она может просто повторно передать запрос. Проблема в том, что операционная система клиента не может определить, что есть что. Все, что она знает, – то, что время на ее таймере истекло.

Существуют три философских подхода к тому, что здесь делать [Spector, 1982]. Одна философия заключается в том, чтобы дождаться перезагрузки сервера (или позволить промежуточному программному обеспечению клиента прозрачно выполнить повторную привязку к новому серверу) и повторить операцию. Идея: продолжать попытки до тех пор, пока не будет получен ответ, а затем передать его клиенту. Этот метод называется **семантикой «хотя бы один раз»** (at-least-once semantics) и гарантирует, что RPC был выполнен хотя бы один раз, но, возможно, больше.

Вторая философия сразу же сдается и сообщает о неудаче. Этот подход называется семантикой **«не более одного раза»** (at-most-once semantics) и гарантирует, что RPC был выполнен не более одного раза, но, возможно, вообще не выполнен.

Третья философия – ничего не гарантирует. При сбое сервера клиент не получает никакой помощи и никаких сообщений о том, что произошло. RPC может выполняться от нуля до большого количества раз. Основное достоинство этой схемы в том, что ее легко реализовать.

Ни один из них не очень привлекателен. То, что хотелось бы, – это **семантика «точно один раз»** (exactly-once semantics), но в общем случае это никак не создать. Представьте, что удаленная операция состоит из обработки документа, такого как создание ряда файлов PDF из LATEX и других источников. Сервер отправляет сообщение о завершении клиенту, когда документ полностью обработан. Также предположим, что когда клиент выдает запрос, он получает подтверждение того, что запрос был доставлен на сервер. Сервер может следовать двум стратегиям. Он может либо отправить сообщение о завершении непосредственно перед тем, как он фактически

скажет обработчику документов выполнить свою работу, либо после обработки документа.

Предположим, что сервер отказывает и впоследствии восстанавливается. Он сообщает всем клиентам о том, что он только что вышел из строя, но теперь снова работает. Проблема в том, что клиент не знает, был ли фактически выполнен его запрос на обработку документа.

Клиент может следовать четырем стратегиям. Во-первых, клиент может решить никогда не повторять запрос, рискуя тем, что документ не будет обработан. Во-вторых, он может принять решение всегда повторять запрос, но это может привести к тому, что документ будет обработан дважды (что может легко повлечь за собой значительный объем работы при работе со сложными документами). В-третьих, он может принять решение о повторном запросе, только если он еще не получил подтверждение того, что его запрос был доставлен на сервер. В этом случае клиент рассчитывает на то, что сервер потерпел крах до того, как запрос мог быть доставлен. Четвертая, и последняя, стратегия – повторный запрос, только если он получил подтверждение для запроса. Имея две стратегии для сервера и четыре для клиента, можно рассмотреть восемь комбинаций. К сожалению, как выясняется, ни одна комбинация не является удовлетворительной: можно показать, что для любой комбинации либо запрос теряется навсегда, либо выполняется дважды.

Примечание 8.9 (дополнительно: почему полностью прозрачное восстановление сервера невозможно)

Чтобы объяснить ситуацию восстановления сервера, обратите внимание, что на сервере могут происходить три события: отправка сообщения о завершении (M), завершение обработки документа (P) и сбой (C). Обратите внимание, что сбой *во время* обработки документа считается таким же, как сбой до его завершения. Эти события могут происходить в шести разных порядках:

1) $M \rightarrow P \rightarrow C$: происходит сбой после отправки сообщения о завершении и обработки документа;
2) $M \rightarrow C(\rightarrow P)$: сбой происходит после отправки сообщения о завершении, но до того, как документ может быть (полностью) обработан;
3) $P \rightarrow M \rightarrow C$: происходит сбой после отправки сообщения о завершении и обработки документа;
4) $P \rightarrow C(\rightarrow M)$: документ обработан, после чего происходит сбой до отправки сообщения о завершении;
5) $C(\rightarrow P \rightarrow M)$: сбой происходит до того, как сервер может завершить обработку документа;
6) $C(\rightarrow M \rightarrow P)$: сбой происходит до того, как сервер может что-либо сделать.

Скобки указывают на событие, которое больше не может произойти, потому что сервер уже вышел из строя. На рис. 8.19 показаны все возможные комбинации. Как можно легко проверить, не существует комбинации стратегии клиента и стратегии сервера, которая бы работала правильно при всех возможных последовательностях событий. Суть в том, что клиент никогда не узнает, произошел ли сбой сервера непосредственно перед или после печати текста.

	Стратегии M ← P			Стратегии P ← M		
Изменение стратегии	**MPC**	**MC(P)**	**C(MP)**	**PMC**	**PC(M)**	**C(PM)**
Всегда	DUP	OK	OK	DUP	DUP	OK
Никогда	OK	ZERO	ZERO	OK	OK	ZERO
Только когда есть запрос	DUP	OK	ZERO	DUP	OK	ZERO
Только когда нет запроса	OK	ZERO	OK	OK	DUP	OK
Клиент	**Сервер**			**Сервер**		

OK – документ обработан один раз
DUP – документ обработан дважды
ZERO – документ не обработан вообще

Рис. 8.19 ❖ Различные комбинации стратегий клиента и сервера при наличии сбоев сервера. События в скобках никогда не происходят из-за предшествующего сбоя

Одним словом, возможность сбоев серверов радикально меняет характер RPC и четко отличает однопроцессорные системы от распределенных. В первом случае сбой сервера также подразумевает сбой клиента, поэтому восстановление невозможно и не нужно. В последнем случае мы можем и должны принять меры.

Потерянные ответные сообщения

С потерянными ответами также могут возникнуть проблемы. Очевидное решение – просто снова полагаться на таймер, установленный операционной системой клиента. Если в течение разумного периода времени ответа не ожидается, просто отправьте запрос еще раз. Проблема с этим решением в том, что клиент не совсем уверен, почему ответа не было. Был ли запрос, или ответ потерян, или просто сервер медленный? Это может иметь значение.

В частности, некоторые операции можно безопасно повторять так часто, как это необходимо, без нанесения ущерба. Запрос, такой как запрос первых 1024 байтов файла, не имеет побочных эффектов и может быть выполнен так часто, как это необходимо, без какого-либо вреда. Запрос, который имеет это свойство, называется **идемпотентным** (idempotent).

Рассмотрим запрос к банковскому серверу с просьбой перевести деньги с одного счета на другой. Если запрос прибывает и выполняется, но ответ потерян, клиент не узнает об этом и повторно отправит сообщение. Сервер банка будет интерпретировать этот запрос как новый и также выполнит его. Сумма будет переведена дважды. Перевод денег не идемпотентен.

Одним из способов решения этой проблемы является попытка структурировать все запросы идемпотентным способом. На практике, однако, многие запросы (например, перевод денег) по своей природе не идемпотентны, поэтому необходимо что-то еще. Другой метод заключается в том, чтобы клиент назначал каждому запросу порядковый номер. Имея сервер, отслеживающий последний полученный порядковый номер от каждого клиента, который его использует, сервер может определить разницу между исходным запросом и повторной передачей и может отказать в выполнении любого

запроса во второй раз. Однако серверу все равно придется отправить ответ клиенту. Обратите внимание, что этот подход требует, чтобы сервер поддерживал администрирование каждого клиента. Кроме того, не ясно, как долго поддерживать это администрирование. Дополнительной гарантией является наличие в заголовке сообщения бита, который используется для различения начальных запросов от повторных передач (идея заключается в том, что всегда можно безопасно отправить исходный запрос; повторные передачи привлекут большее внимание).

Клиент неисправен

Последний элемент в списке сбоев – это сбой клиента. Что произойдет, если клиент отправит запрос на сервер для выполнения некоторой работы и произойдет сбой до того, как сервер ответит? В этот момент идут активные вычисления, но никто не ожидает результата. Такое нежелательное бесхозное вычисление называется **сиротой** (orphan).

Бесхозные вычисления могут вызвать множество проблем, которые могут помешать нормальной работе системы. Как минимум, это затраты на обработку. Они также могут блокировать файлы или иным образом связывать ценные ресурсы.

Наконец, если клиент перезагружается и снова выполняет RPC, но ответ сироты возвращается сразу после этого и может создать путаницу.

Что можно сделать с сиротами? Было предложено четыре решения [Nelson, 1981]. Во-первых, перед тем как клиентская заглушка отправляет сообщение RPC, она делает запись в журнале, сообщающую, что собирается сделать. Журнал хранится на диске или на другом носителе, который выживает после сбоев. После перезагрузки журнал проверяется, и сирота уничтожается. Это решение называется **уничтожением сирот** (orphan extermination).

Недостаток данной схемы – очень большие затраты на запись на диск для каждого RPC. Кроме того, это может даже не работать, поскольку сами сироты могут создавать RPC, создавая таким образом **потомков сирот** (grandorphans), которых трудно или невозможно найти. Наконец, сеть может быть разделена, например, из-за неисправного шлюза, что делает невозможным их уничтожение, даже если они могут быть обнаружены. В общем, это не перспективный подход.

Со вторым решением, называемым **реинкарнацией** (reincarnation), или **перевоплощением**, все эти проблемы могут быть решены без необходимости записи на диск. При этом время делится на последовательно пронумерованные эпохи. Когда клиент перезагружается, он передает сообщение всем машинам, объявляя о начале новой эпохи. Когда приходит такое сообщение, все удаленные вычисления уничтожаются. Конечно, если сеть разделена, некоторые сироты могут выжить. К счастью, однако, когда они что-либо сообщают, их ответы будут содержать устаревший номер эпохи, что облегчает их обнаружение.

Третье решение – вариант этой идеи, но несколько менее драконовский.

Это называется **мягким перевоплощением** (gentle reincarnation). Когда приходит широковещательная рассылка, каждая машина проверяет наличие

локальных удаленных вычислений и, если они есть, старается найти своих владельцев. Только если владельцы нигде не могут быть найдены, вычисление уничтожается.

В четвертом решении, называемом **истечением срока** (expiration), каждому RPC дается стандартное время *T* для выполнения работы. Если он не может закончить его, он должен явно запросить другую часть. Конечно, это довольно неприятно. С другой стороны, если после сбоя клиент ждет время *T* перед перезагрузкой, все сироты обязательно исчезнут. Проблема, которая должна быть решена здесь, заключается в выборе разумного значения *T* для RPC с сильно различающимися требованиями.

На практике все эти методы являются грубыми и нежелательными. Хуже того, убийство сироты может иметь непредвиденные последствия. Например, предположим, что сирота получила блокировки одного или нескольких файлов либо записей базы данных. Если сирота внезапно погибнет, эти блокировки могут остаться навсегда. Кроме того, сирота, возможно, уже сделала записи в различных удаленных очередях для запуска других процессов в будущем, поэтому даже убийство сироты может не удалить все ее следы. Возможно, все может начаться сначала, с непредвиденными последствиями. Ликвидация сирот обсуждается более подробно в [Panzieri and Shrivastava, 1988].

8.4. Надежное групповое общение

Учитывая, насколько важна устойчивость процессов при репликации, неудивительно, что надежные многоадресные службы также важны. Такие сервисы гарантируют, что сообщения доставляются всем участникам группы процессов. К сожалению, надежная многоадресная передача оказывается на удивление коварной. В этом разделе мы подробнее рассмотрим вопросы, связанные с надежной доставкой сообщения группе процессов. Давайте сначала определим, что такое надежная групповая коммуникация на самом деле. Интуитивно это означает, что сообщение, которое отправляется группе процессов, должно быть доставлено каждому члену этой группы. Если мы отделим логику обработки сообщений от основной функциональности члена группы, то сможем удобно провести различие между *получением* сообщений и *доставкой* сообщений, как показано на рис. 8.20. Сообщение принимается компонентом обработки сообщений, который, в свою очередь, доставляет сообщение компоненту, содержащему основные функциональные возможности члена группы. Неформально сообщение, полученное процессом P, также будет доставлено P.

Например, чтобы доставка сообщений от одного и того же отправителя происходила в том же порядке, в котором они были отправлены, обычно добавляется компонент обработки сообщений. Аналогично обеспечение надежной передачи сообщений – это функция, которая может и должна быть отделена от основной функциональности члена группы и обычно реализуется компонентом обработки сообщений (если не базовой операционной системой).

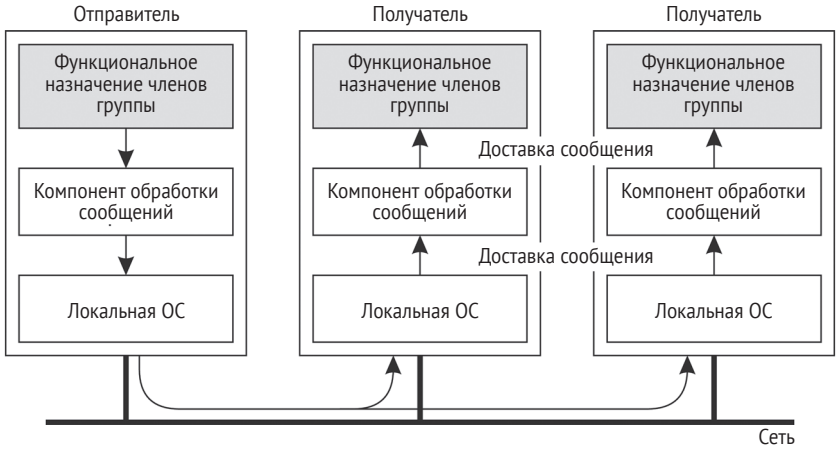


Рис. 8.20 ❖ Различие между получением и доставкой сообщений

С этим разделением между получением и доставкой сообщений мы можем более точно определить, что означает надежная групповая связь. Давайте сделаем различие между надежной связью при наличии неисправных процессов и надежной связью, когда предполагается, что процессы работают правильно. В первом случае групповое общение считается надежным, если можно гарантировать, что сообщение получено и впоследствии доставлено всеми исправными членами группы.

Хитрость здесь в том, что до того, как сообщение может быть доставлено, должно быть достигнуто соглашение о том, как на самом деле выглядит группа. Если отправитель намеревался получить сообщение от каждого члена группы **G**, но по какой-то причине во время доставки у нас фактически есть другая группа $\mathbf{G}' \neq \mathbf{G}$, мы должны спросить себя, может сообщение быть доставлено или нет.

Ситуация становится проще, если мы можем игнорировать консенсус в отношении членства в группах. В частности, давайте сначала предположим, что процесс отправки имеет список предполагаемых получателей. В этом случае он может просто развернуть надежные протоколы транспортного уровня, такие как TCP, и один за другим отправлять свое сообщение каждому получателю. Если процесс получения завершается неудачно, сообщение может быть повторно отправлено позже, когда процесс восстанавливается, или он игнорируется вообще (например, потому что отправитель покинул группу). В случае если от члена группы ожидается отправка ответа, даже если это просто подтверждение, обмен данными можно ускорить, отделяя отправку запроса от получения ответа, как показано диаграммами последовательности сообщений на рис. 8.21.

Большинство транспортных уровней предлагают надежные двухточечные каналы; они редко предлагают надежную связь группе процессов. Лучшее, что они предлагают, – это позволить процессу установить соединение «точка-точка» друг с другом, с которым он хочет установить связь. Когда группы процессов относительно малы, такой подход к установлению надежности

является простым и практичным решением. С другой стороны, часто можно ожидать, что базовая система связи предлагает ненадежную многоадресную передачу, что означает, что многоадресное сообщение может быть частично потеряно и доставлено некоторым, но не всем предполагаемым получателям.

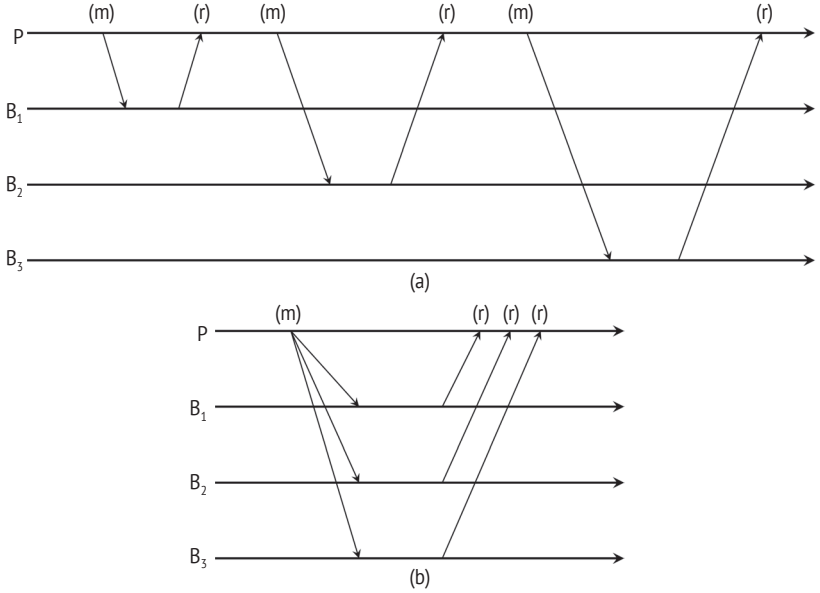


Рис. 8.21 ❖ a) Отправитель посылает запросы, но ждет ответа перед отправкой следующего; b) запросы отправляются параллельно, после чего отправитель ожидает входящих ответов

Простое решение для обеспечения надежного группового общения показано на рис. 8.22. Процесс отправки присваивает порядковый номер каждому сообщению, которое он передает, и сохраняет его локально в буфере истории. Предполагая, что получатели известны отправителю, отправитель просто сохраняет сообщение в своем буфере истории, пока каждый получатель не вернет подтверждение.

Получатель может заподозрить, что ему не хватает сообщения m с порядковым номером *s*, когда он получил сообщения с порядковыми номерами, превышающими *s*. В этом случае он возвращает отрицательное подтверждение отправителю, запрашивая повторную передачу m.

Существуют различные компромиссы дизайна. Например, чтобы уменьшить количество сообщений, возвращаемых отправителю, подтверждения могут быть дополнены другими сообщениями. Кроме того, повторная передача сообщения может быть выполнена с использованием двухточечной связи с каждым запрашивающим процессом или с использованием одного многоадресного сообщения, отправленного всем процессам. Общие вопросы надежной многоадресной рассылки обсуждаются в [Popescu et al., 2007]. Обзор по надежной многоадресной рассылке в контексте систем публикации/подписки, которые также актуальны здесь, приводится в [Esposito et al., 2013].

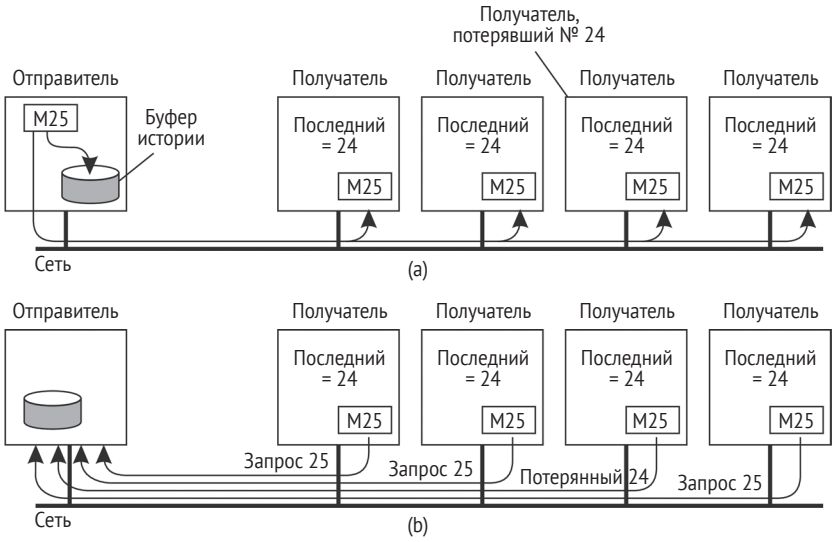


Рис. 8.22 ❖ Решение для надежной многоадресной рассылки: a) передача сообщений; b) сообщение обратной связи

Примечание 8.10 (дополнительно: масштабируемость в надежной многоадресной передаче)

Основная проблема только что описанной надежной многоадресной схемы заключается в том, что она не может поддерживать большое количество получателей. Если есть *N* получателей, отправитель должен быть готов принять как минимум *N* подтверждений. Со многими получателями отправитель может быть завален такими сообщениями обратной связи, что также называется **взрывом обратной связи** (feedback implosion). При репликации процессов для обеспечения отказоустойчивости такая ситуация маловероятна, поскольку группы процессов относительно малы. При репликации для повышения производительности мы имеем другой случай. Кроме того, нам также может понадобиться учитывать, что получатели распределены по глобальной сети.

Одно из решений проблемы взрыва обратной связи состоит в том, что получатели не подтверждают получение сообщения. Вместо этого получатель возвращает сообщение обратной связи только для того, чтобы сообщить отправителю, что он пропустил сообщение. Можно показать, что возвращение только таких отрицательных подтверждений в целом лучше масштабируется [Towsley et al., 1997], но не может быть никаких твердых гарантий того, что взрыва обратной связи никогда не произойдет.

Другая проблема, связанная с возвратом только отрицательных подтверждений, заключается в том, что теоретически отправитель будет вынужден хранить сообщение в своем буфере истории навсегда. Поскольку отправитель никогда не может знать, правильно ли было доставлено сообщение всем получателям, его всегда следует подготовить для получателя, запрашивающего повторную передачу старого сообщения. На практике отправитель удаляет сообщение из своего буфера истории по истечении некоторого времени, чтобы предотвратить переполнение буфера. Однако удаление сообщения выполняется с риском того, что запрос на повторную передачу не будет выполнен.

Существует несколько предложений о масштабируемой надежной многоадресной рассылке. Сравнение между различными схемами можно найти в [Levine and Garcia-Luna-Aceves, 1998]. Теперь мы кратко обсудим два совершенно разных подхода, которые представляют многие существующие решения.

Неиерархический контроль обратной связи. Ключевой проблемой масштабируемых решений для надежной многоадресной рассылки является уменьшение количества сообщений обратной связи, возвращаемых отправителю. Популярной моделью, которая была применена к нескольким глобальным приложениям, является подавление обратной связи. Эта схема лежит в основе протокола **«Масштабируемая надежная многоадресная связь»** (Scalable Reliable Multicasting. SRM), разработанного в [Floyd et al., 1997] и работающего следующим образом.

Во-первых, в SRM получатели никогда не подтверждают успешную доставку многоадресного сообщения, а вместо этого сообщают только об отсутствии сообщения. Как обнаружена потеря сообщения, остается за приложением. Только отрицательные подтверждения возвращаются в качестве обратной связи. Всякий раз, когда получатель замечает, что пропустил сообщение, он направляет свой отзыв остальной группе.

Многоадресная обратная связь позволяет другому члену группы подавлять свою обратную связь. Предположим, несколько получателей пропустили сообщение m. Каждому из них потребуется вернуть отрицательное подтверждение отправителю S, чтобы можно было повторно передать m. Однако если мы предположим, что повторные передачи всегда являются многоадресными для всей группы, то достаточно, чтобы только один запрос на повторную передачу достиг S.

По этой причине получатель R, который не получил сообщение m, планирует сообщение обратной связи с некоторой случайной задержкой. То есть запрос на повторную передачу не отправляется до тех пор, пока не истечет некоторое случайное время. Если в то же самое время другой запрос на повторную передачу для m достигает R, R будет подавлять свою собственную обратную связь, зная, что m будет передан в ближайшее время. Таким образом, в идеале только одно сообщение обратной связи будет достигать S, которое, в свою очередь, впоследствии повторно. Эта схема показана на рис. 8.23.

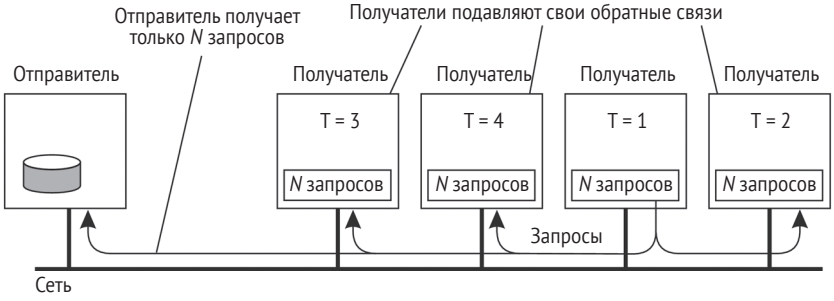


Рис. 8.23 ❖ Несколько получателей запланировали запрос на повторную передачу, но первый запрос на повторную передачу приводит к подавлению других

Подавление обратной связи продемонстрировало достаточно хорошее масштабирование и использовалось в качестве основного механизма в ряде интернет-приложений для совместной работы, таких как общая доска. Однако такой подход также создает ряд серьезных проблем. Во-первых, для обеспечения того, чтобы отправителю был возвращен только один запрос на повторную передачу, требуется

достаточно точное планирование сообщений обратной связи на каждом получателе. В противном случае многие получатели будут по-прежнему возвращать свои отзывы одновременно. Соответственно, установить таймеры в группе процессов, которые распределены по глобальной сети, не так просто.

Другая проблема заключается в том, что многоадресная обратная связь также прерывает те процессы, на которые сообщение было успешно доставлено. Иными словами, другие получатели вынуждены получать и обрабатывать бесполезные для них сообщения. Единственное решение этой проблемы – позволить получателям, которые не получили сообщение m, присоединиться к отдельной группе многоадресной рассылки для m, как объяснено в [Kasera et al., 1997].

К сожалению, это решение требует, чтобы группы могли управляться очень эффективным образом, что трудно осуществить в глобальной системе. Поэтому лучший подход состоит в том, чтобы позволить получателям, которые, как правило, пропускают одни и те же сообщения, объединяться и совместно использовать один и тот же многоадресный канал для сообщений обратной связи и повторных передач. Подробности этого подхода можно найти в [Liu et al., 1998].

Для повышения масштабируемости SRM полезно позволить получателям помочь в локальном восстановлении. В частности, если получатель, которому сообщение m было успешно доставлено, получает запрос на повторную передачу, он может принять решение о многоадресной передаче m даже до того, как запрос на повторную передачу достигнет исходного отправителя. Дополнительные подробности можно найти в [Floyd et al., 1997] и [Liu et al., 1998].

Иерархическое управление с обратной связью. Подавление обратной связи, как только что описано, является в основном неиерархическим решением. Однако достижение масштабируемости для очень больших групп получателей требует применения иерархических подходов. Решение показано на рис. 8.24. Группа получателей разделена на несколько подгрупп, которые впоследствии организованы в дерево. В каждой подгруппе может использоваться любая надежная схема многоадресной рассылки, которая работает для небольших групп. Каждая подгруппа назначает местного координатора, который представляет эту группу в многоадресном дереве. Связь в дереве между двумя узлами соответствует надежной связи между координаторами соответствующих подгрупп.

Когда процесс S в группе **G** хочет отправить сообщение, он просто использует надежную схему многоадресной рассылки для **G**, чтобы охватить всех ее членов, включая координатора группы, скажем C. Координатор C, в свою очередь, направит сообщение своим соседним координаторам. Как правило, координатор пересылает входящее сообщение m всем своим соседним координаторам, кроме того, от которого он получил m. Координатор будет надежно многоадресно передавать входящее сообщение всем членам подгруппы, которую он представляет, и, в частности, обрабатывать повторные передачи для этой группы.

В схеме на основе запросов (ACK), если координатор C группы **G** отправляет сообщение m координатору C′ другой, соседней группы **G**′, он будет хранить m в своем буфере истории по крайней мере до тех пор, пока C′ не отправит подтверждение. В схеме на основе *N* запросов (NACK), только если **G**′ обнаружит, что он пропустил m (и, следовательно, также все члены **G**′ и все координаторы, которым **G**′ должен был бы переслать m), он отправит сообщение NACK в **C**. Таким образом, видно, что одно сообщение подтверждения или отсутствия от координатора объединяет множество управляющих сообщений обратной связи от других процессов, что приводит к гораздо более масштабируемой надежной схеме многоадресной рассылки. Масштабируемость дополнительно улучшается, позволяя координатору обрабатывать повторные передачи соседним координаторам, которым он направил сообщение.

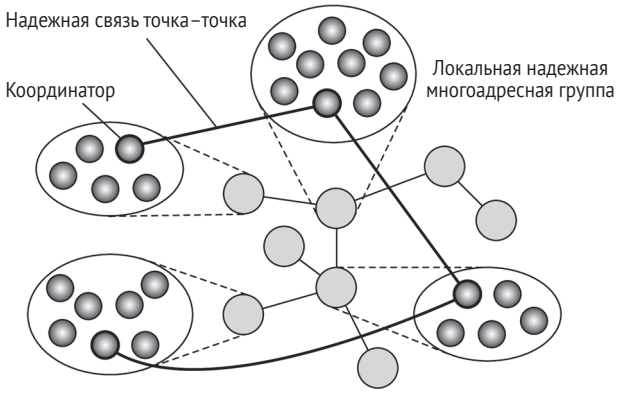


Рис. 8.24 ❖ Сущность иерархической надежной многоадресной рассылки. Каждый локальный координатор пересылает сообщение своим соседним координаторам в дереве и позже обрабатывает запросы на повторную передачу

Обратите внимание, что неиерархический контроль обратной связи, который мы обсуждали ранее, может быть использован для улучшения масштабируемости одной группы многоадресной рассылки. Вместе с иерархическим контролем обратной связи мы бы объединили относительно большие подгруппы надежного многоадресного вещания в потенциально большие деревья, что позволило бы поддерживать надежное многоадресное вещание для очень больших групп процессов.

Основная проблема с иерархическими решениями – это построение и управление деревом: как формируются подгруппы, какие процессы назначаются координаторами и как подгруппы организуются в дереве. Во многих случаях дерево должно быть построено динамически. К сожалению, традиционные решения сетевого уровня почти не предоставляют адекватных услуг для управления деревьями. По этой причине приобрели популярность многоадресные решения на уровне приложений, которые мы обсуждали в разделе 4.4.

Масштабируемая надежная многоадресная рассылка на основе сплетен. Наконец, давайте кратко рассмотрим многоадресные схемы, основанные на сплетнях, в частности следующую двухъярусную антиэнтропийную схему, которая подробно обсуждалась в разделе 4.4.

В этой схеме узел P выбирает другой узел Q случайным образом и впоследствии обменивается обновлениями с Q. Другими словами, P отправляет обновления, которые Q ранее не видел, в Q и извлекает любые обновления, которые есть у Q, но которые были пропущены P. После обмена оба процесса имеют одинаковые данные. Ясно, что эта схема уже изначально устойчива, поскольку, если по какой-либо причине связь между P и Q прерывается, P просто выберет какой-то другой узел для обмена обновлениями. Конечным эффектом является то, что скорость, с которой обновление распространяется по системе, замедляется, но на надежность влияет только в крайних случаях. Тем не менее это замедление считается важным для некоторых приложений. В этом свете может представлять интерес сравнение между традиционной многоадресной рассылкой на основе дерева и многоадресной рассылкой на основе сплетен с целью агрегирования, как обсуждалось в [Nyers and Jelasity, 2015].

Создание надежных многоадресных схем, которые могут масштабироваться до большого числа приемников, распределенных по большим сетям, является сложной проблемой. Единого лучшего решения не существует, и каждое решение порождает новые проблемы.

Атомарная многоадресная рассылка

Давайте теперь вернемся к ситуации, в которой нам нужно добиться надежной многоадресной передачи при наличии сбоев процесса. В частности, в распределенной системе часто требуется гарантия доставки сообщения всем членам группы или вообще никому. Эта проблема также известна как **атомарная (неделимая) многоадресная проблема** (atomic multicast problem).

Чтобы понять, почему атомарность так важна, рассмотрим реплицированную базу данных, созданную как приложение поверх распределенной системы. Распределенная система предлагает надежные средства многоадресной рассылки. В частности, она позволяет создавать группы процессов, в которые можно надежно отправлять сообщения. Поэтому реплицированная база данных строится как группа процессов, по одному процессу для каждой реплики. Операции обновления всегда являются многоадресными для всех реплик и впоследствии выполняются локально. Таким образом, мы предполагаем, что используется протокол активной репликации.

Для простоты предположим, что клиент связывается с репликой P и просит ее выполнить обновление. Реплика делает это путем многоадресной рассылки обновления другим членам группы. К сожалению, до завершения многоадресной рассылки P дает сбой, оставляя остальную часть группы в трудном положении: некоторые члены группы получат запрос на обновление; другие нет. Если участники, получившие запрос, доставят его в базу данных, то, очевидно, у нас будет несовместимая реплицируемая база данных. Некоторые реплики будут обрабатывать обновление, другие вообще не будут. Этой ситуации нужно избегать, и мы должны либо сообщить, что обновление доставлено всем исправным членам, либо вообще никому. В первом случае отражается сбой P после завершения многоадресной рассылки, а во втором – сбой P до того, как он даже получил возможность запросить обновление.

Обе эти ситуации приемлемы и соответствуют случаю, когда клиент связывается с одним сервером, которому разрешен сбой. Если ряд членов группы выполнят обновление, а другие – нет, на карту поставлена прозрачность распределения, но, что еще хуже, клиент не будет знать, что делать с ситуацией.

Виртуальная синхроннность

Надежная многоадресная передача при наличии сбоев процессов может быть точно определена с точки зрения групп процессов и изменений в членстве в группах. Как и раньше, мы делаем различие между получением и доставкой сообщения. В частности, мы снова принимаем модель, в которой распределенная система состоит из компонентов обработки сообщений, как показано на рис. 8.20. Полученное сообщение локально буферизуется в этом компоненте до тех пор, пока оно не может быть доставлено приложению, которое логически размещается в качестве члена группы на более высоком уровне.

Вся идея атомарной многоадресной рассылки заключается в том, что многоадресное сообщение m однозначно связано со списком процессов, которые

должны его доставить. Этот список доставки соответствует **представлению группы** (group view), а именно представлению набора процессов, содержащихся в группе, которое отправитель имел в то время, когда сообщение m было многоадресным. Важным наблюдением является то, что каждый процесс в этом списке имеет одинаковое представление. Другими словами, все они должны согласиться с тем, что m должен доставляться каждым из них, а не каким-либо другим процессом.

Теперь предположим, что сообщение m является многоадресным, когда его отправитель, скажем P, имеет представление группы **G**. Кроме того, предположим, что во время многоадресной передачи другой процесс Q присоединяется к группе или покидает ее. Это изменение в членстве в группе, естественно, объявляется всем процессам в **G**. С другой стороны, **изменение представления** (view change) происходит путем многоадресной рассылки сообщения vc, сообщающего о присоединении или выходе из Q. Теперь у нас есть два находящихся одновременно в пути многоадресных сообщения: m и vc. Нам нужно гарантировать, что m либо доставляется всеми процессами в **G** до того, как кто-либо выполнит изменение представления, как указано в vc, либо m не доставляется вообще. Обратите внимание, что это требование сравнимо с тотально упорядоченной многоадресной рассылкой, о которой мы говорили в главе 6.

Сразу возникает вопрос: если m не доставляется каким-либо процессом, как мы можем говорить о надежном протоколе многоадресной рассылки? В принципе, существует только один случай, когда доставка m может быть неудачной: изменение членства в группе является результатом сбоя отправителя m P. В этом случае либо все оставшиеся (исправные) члены **G** должны доставить m, прежде чем согласиться, что P больше не является членом группы, либо никто не должен доставить m.

Как упоминалось ранее, последнее соответствует ситуации, когда P считается потерпевшим аварию до того, как у него была возможность отправить m.

Эта более сильная форма надежной многоадресной рассылки гарантирует, что многоадресная рассылка сообщений группе представления **G** доставляется каждым исправным процессом в **G**. Если отправитель сообщения завершает работу во время многоадресной рассылки, сообщение либо доставляется всем оставшимся процессам, либо игнорируется каждым из них. Говорят, что такая надежная многоадресная передача является **виртально синхронной** (virtually synchronous) [Birman and Joseph, 1987].

Чтобы проиллюстрировать это, рассмотрим четыре процесса, показанных на рис. 8.25. В определенный момент времени у нас есть группа, состоящая из S_1, S_2, S_3 и S_4. После многоадресной рассылки S_3 вылетает. Однако перед сбоем удалось выполнить многоадресную передачу сообщения процессам S_2 и S_4, но не S_1. Однако виртуальная синхронность в этом случае гарантирует, что сообщение не будет доставлено вообще, эффективно устанавливая ситуацию, при которой сообщение никогда не отправлялось до сбоя S_3.

После того как S_3 был удален из группы, связь продолжается между остальными членами группы. Позже, когда S_3 восстанавливается, он может снова присоединиться к группе, после того как его состояние было обновлено.

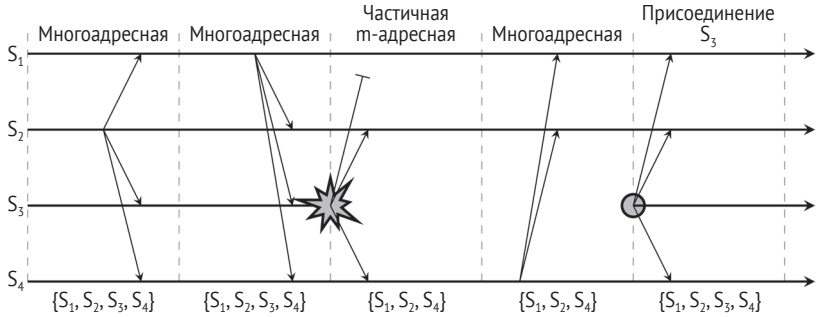


Рис. 8.25 ❖ Принцип виртуальной синхронной многоадресной рассылки

Принцип виртуальной синхронности исходит из того факта, что все многоадресные передачи происходят между изменениями представления. Иными словами, изменение представления действует как барьер, через который не может проходить многоадресная передача. В некотором смысле это сопоставимо с использованием переменной синхронизации в распределенных хранилищах данных, как обсуждалось в предыдущей главе. Все многоадресные передачи, которые находятся в процессе передачи, пока происходит изменение представления, завершаются до того, как изменение представления вступает в силу. Реализация виртуальной синхронности не является тривиальной, как мы увидим далее.

Порядок сообщений

Виртуальная синхронизация позволяет разработчику приложения думать о многоадресной рассылке как о происходящих рассылках в эпохи, разделенные изменениями членства в группах. Однако еще ничего не было сказано о порядке многоадресных рассылок. В общем, различают четыре разных порядка рассылок:

1) неупорядоченные многоадресные рассылки;
2) FIFO многоадресные упорядоченные рассылки (первым пришел – первым вышел);
3) причинно упорядоченные многоадресные передачи;
4) полностью упорядоченные многоадресные рассылки.

Надежная неупорядоченная многоадресная рассылка (reliable, unordered multicast) – это практически синхронная многоадресная рассылка, для которой нет никаких гарантий относительно порядка, в котором полученные сообщения доставляются различными процессами. Для объяснения предположим, что надежная многоадресная передача поддерживается библиотекой, предоставляющей примитив отправки и получения. Операция получения блокирует абонента до тех пор, пока сообщение не будет доставлено.

Теперь предположим, что отправитель P_1 осуществляет многоадресную рассылку двух сообщений группе, в то время как два других процесса в этой группе ожидают прибытия сообщений, как показано на рис. 8.26. Предполагая, что процессы не завершаются сбоем или не покидают группу во время этих многоадресных рассылок, возможно, что компонент обработки сообще-

ний в P_2 сначала получит сообщение m_1, а затем m_2. Поскольку нет никаких ограничений на порядок сообщений, сообщения могут доставляться в порядке их получения. Напротив, компонент обработки сообщений в P_3 может сначала получить сообщение m_2, за которым следует m_1, и доставить эти два в том же порядке в приложение более высокого уровня P_3.

Порядок событий	Процесс P_1	Процесс P_2	Процесс P_3
1	отправляет m_1	получает m_1	получает m_2
2	отправляет m_2	получает m_2	получает m_1

Рис. 8.26 ❖ Три процесса обмена данными в одной группе. Порядок событий процесса показан вдоль вертикальной оси

В случае **надежной FIFO-упорядоченной многоадресной передачи** (reliable FIFO-ordered multicasts) уровень компонента обработки сообщений вынужден доставлять входящие сообщения из того же процесса в том же порядке, в котором они были отправлены. Рассмотрим взаимодействие в группе из четырех процессов, как показано на рис. 8.27. При порядке FIFO единственное, что имеет значение, – это то, что сообщение m_1 всегда доставляется перед m_2, и, аналогично, это сообщение m_3 всегда доставляется перед m_4. Данное правило должно соблюдаться всеми процессами в группе. Другими словами, когда уровень связи в P_3 получает первым m_2, он будет ожидать доставки с P_3, пока не получит и не доставит m_1.

Порядок событий	Процесс P_1	Процесс P_2	Процесс P_3	Процесс P_4
1	отправляет m_1	получает m_1	получает m_3	отправляет m_3
2	отправляет m_2	получает m_3	получает m_1	отправляет m_4
3		получает m_2	получает m_2	
4		получает m_4	получает m_4	

Рис. 8.27 ❖ Четыре процесса в одной группе с двумя разными отправителями и возможный порядок доставки сообщений в соответствии с FIFO – заказанная групповая передача

Вместе с тем нет никаких ограничений в отношении доставки сообщений, отправленных различными процессами. Другими словами, если процесс P_2 получает m_1 до m_3, он может доставить оба сообщения в этом порядке. Между тем процесс P_3 мог получить m_3 до получения m_1. Порядок FIFO гласит, что P_3 может доставить m_3 перед m_1, хотя этот порядок доставки отличается от порядка P_2.

Наконец, **надежная причинно упорядоченная многоадресная** (reliable causally ordered multicast) рассылка доставляет сообщения так, чтобы потен-

циальная причинно-следственная связь между различными сообщениями сохранялась. Другими словами, если сообщение m_1 причинно предшествует другому сообщению m_2, независимо от того, были ли они многоадресной рассылкой одного и того же отправителя, то транспортный уровень на каждом получателе будет всегда доставлять m_2, после того как он получил и доставил m_1. Обратите внимание, что причинно упорядоченные многоадресные рассылки могут быть реализованы с использованием векторных временных меток, как обсуждалось в главе 6.

Помимо этих трех порядков, может быть дополнительное ограничение, что доставка сообщений также должна быть полностью упрядочена. Полная упорядоченная доставка означает, что независимо от того, является ли неупорядоченной доставка сообщений, FIFO-упорядоченной или причинно упорядоченной, дополнительно требуется, чтобы при доставке сообщений они доставлялись всем членам группы в одном и том же порядке.

Например, при комбинации FIFO и полностью упорядоченной многоадресной передаче процессы P_2 и P_3 на рис. 8.27 могут как сначала доставить сообщение m_3, так и затем сообщение m_1. Однако если P_2 доставит m_1 до m_3, а P_3 доставит m_3 до доставки m_1, они нарушат ограничение общего порядка. Обратите внимание, что порядок FIFO все еще должен соблюдаться. Другими словами, m_2 должен быть доставлен после m_1, и, соответственно, m_4 должен быть доставлен после m_3.

Виртуальная синхронная надежная многоадресная рассылка, предлагающая полную упорядоченную доставку сообщений, называется **атомарной многоадресной рассылкой** (atomic multicasting). Три различных ограничения порядка сообщений, рассмотренных выше, приводят к шести формам надежной многоадресной передачи, как показано на рис. 8.28 [Hadzilacos and Toueg, 1993].

Многоадресная рассылка	Порядок доставки базового сообщения	ТО доставка?
Надежная многоадресная рассылка	Нет	Нет
FIFO многоадресная рассылка	FIFO-упорядоченная доставка	Нет
Причинно упорядоченная многоадресная рассылка	Причинно упорядоченная многоадресная доставка	Нет
Атомарная многоадресная рассылка	Нет	Да
FIFO атомарная многоадресная рассылка	FIFO-упорядоченная атомарная многоадресная доставка	Да
Причинно упорядоченная многоадресная рассылка	Причинно упорядоченная доставка	Да

Рис. 8.28 ❖ Шесть различных версий вируальной синхронной надежной многоадресной рассылки с учетом полностью упорядоченной доставки

Примечание 8.11 (дополнительно: реализация виртуальной синхронизации)

Давайте теперь рассмотрим возможную реализацию виртуальной синхронной надежной многоадресной рассылки. Примером такой реализации является система Isis, отказоустойчивая распределенная система, которая уже несколько лет используется в промышленности. Мы сосредоточимся на некоторых вопросах реализации этой методики, как описано в [Birman et al., 1991].

Надежная многоадресная передача в Isis использует доступные надежные средства связи точка-точка базовой сети, в частности TCP. Многоадресная передача сообщения m группе процессов осуществляется путем надежной отправки m каждому члену группы. Как следствие, хотя каждая передача гарантированно будет успешной, нет никаких гарантий, что все члены группы получат m. В частности, отправитель может потерпеть неудачу, прежде чем передать m каждому участнику.

Помимо надежной двухточечной связи, Isis также предполагает, что сообщения из одного и того же источника принимаются транспортным уровнем в том порядке, в котором они были отправлены этим источником. На практике данное требование решается путем использования TCP-соединений для связи точка-точка.

Основная проблема, которую необходимо решить, состоит в том, чтобы гарантировать, что все сообщения, отправленные для просмотра **G**, будут доставлены всем исправным процессам в **G** до того, как произойдет следующее изменение членства в группе. Первая проблема, о которой необходимо позаботиться, – убедиться, что каждый процесс в **G** получил все сообщения, которые были отправлены в **G**. Обратите внимание, что, поскольку отправитель сообщения m в **G**, возможно, потерпел крах до завершения многоадресной рассылки, в **G** могут быть процессы, которые никогда не получат m. Поскольку отправитель потерпел крах, эти процессы должны получить m откуда-то еще.

Решение этой проблемы состоит в том, чтобы позволить каждому процессу в **G** сохранять m, пока он не будет точно знать, что все члены в **G** получили его. Если m было получено всеми членами группы **G**, то m называется стабильным. Только стабильные сообщения могут быть доставлены. Для обеспечения стабильности достаточно выбрать произвольный исправный процесс в **G** и запросить его отправить m всем другим процессам в **G**.

Чтобы быть более конкретным, предположим, что текущим представлением является $\mathbf{G}_i$, но необходимо установить следующее представление $\mathbf{G}_{i+1}$. Без ограничения общности мы предполагаем, что $\mathbf{G}_i$ и $\mathbf{G}_{i+1}$ отличаются не более чем одним процессом. Процесс P замечает изменение представления, когда получает сообщение об изменении представления. Такое сообщение может исходить от процесса, желающего присоединиться или покинуть группу, или от процесса, который обнаружил сбой процесса в $\mathbf{G}_i$, который теперь должен быть удален, как показано на рис. 8.29a.

Когда процесс P получает сообщение об изменении представления для $\mathbf{G}_{i+1}$, он сначала пересылает копию любого нестабильного сообщения от $\mathbf{G}_i$, которое он все еще имеет, каждому процессу в $\mathbf{G}_{i+1}$, а затем помечает его как стабильный. Напомним, что Isis предполагает, что двухточечная связь является надежной, поэтому перенаправленные сообщения никогда не теряются. Такая пересылка гарантирует, что все сообщения в $\mathbf{G}_i$, которые были получены хотя бы одним процессом, получены всеми исправными процессами в $\mathbf{G}_i$. Обратите внимание, что было бы также достаточно выбрать одного координатора для пересылки нестабильных сообщений.

Чтобы указать, что P больше не имеет нестабильных сообщений и что он готов установить $\mathbf{G}_{i+1}$, как только другие процессы смогут это сделать, он отправляет многоадресное **сообщение о сбросе** (flush message) $\mathbf{G}_{i+1}$, как показано на рис. 8.29b. После того как P получил сообщение о сбросе для $\mathbf{G}_{i+1}$ от каждого другого процесса, он может безопасно установить новое представление, как показано на рис. 8.29c.

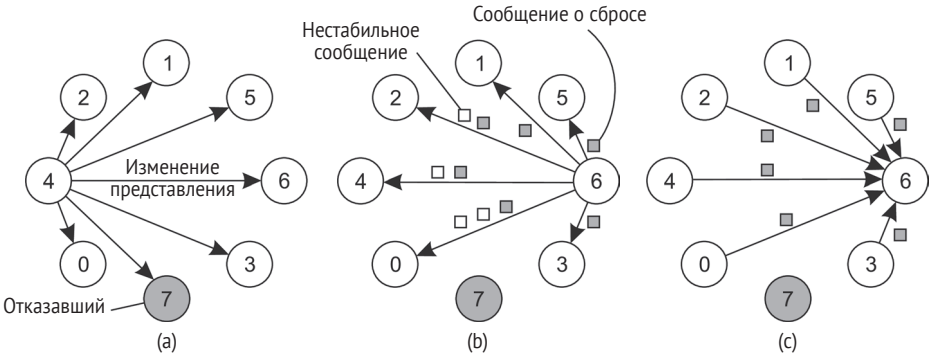


Рис. 8.29 ❖ a) Процесс 4 замечает, что процесс 7 завершился сбоем, и отправляет изменение представления; b) процесс 6 рассылает все свои нестабильные сообщения, после чего следует сообщение о сбросе; c) процесс 6 устанавливает новое представление, когда оно получает сообщение о сбросе от всех остальных

Когда процесс Q получает сообщение m, в то время как Q все еще полагает, что текущим представлением является $\mathbf{G_i}$, он доставляет m, принимая во внимание любые дополнительные ограничения порядка сообщений. Если он уже получил m, то считает сообщение дубликатом и сбрасывает его.

Поскольку процесс Q в конечном итоге получит сообщение об изменении представления для $\mathbf{G_{i+1}}$, он также сначала перенаправит любое из своих нестабильных сообщений и впоследствии завершит работу, отправив сообщение сброса для $\mathbf{G_{i+1}}$. Следует отметить, что из-за предполагаемого упорядочения FIFO-сообщений, обеспечиваемого базовым транспортным уровнем, сообщение о сбросе от какого-либо процесса всегда принимается после получения нестабильного сообщения от этого же процесса.

Основная проблема в протоколе, описанном до сих пор, состоит в том, что он не может справиться с ошибками процесса, пока объявляется новое изменение представления. В частности, предполагается, что до тех пор, пока новый элемент $\mathbf{G_{i+1}}$ не будет установлен каждым участником в $\mathbf{G_{i+1}}$, ни один процесс в $\mathbf{G_{i+1}}$ не будет завершен (что приведет к следующему представлению $\mathbf{G_{i+2}}$). Эта проблема решается путем объявления изменений представления для любого представления $\mathbf{G_{i+k}}$, даже если предыдущие изменения еще не были установлены всеми процессами. Детали довольно сложные, но принцип должен быть ясен.

8.5. Распределенная фиксация

Проблема атомарной многоадресной рассылки, рассмотренная в предыдущем разделе, является примером более общей проблемы, известной как **распределенная фиксация** (distributed commit). Проблема распределенной фиксации заключается в том, что каждый член группы процессов завершает выполнение операции или вообще не выполняет ни одной операции. В случае надежной многоадресной передачи операция представляет собой доставку сообщения (транзакцию). С распределенными транзакциями опера-

ция может быть фиксацией транзакции на одном сайте, который принимает участие в транзакции. Другие примеры распределенного коммита и способы его решения обсуждаются в [Tanisch, 2000].

Распределенная фиксация часто устанавливается с помощью координатора. В простой схеме этот координатор сообщает всем другим вовлеченным процессам, называемым участниками, выполнять или нет (локально) данную операцию. Эта схема называется **протоколом однофазной фиксации** (one-phase commit protocol). У нее есть очевидный недостаток: если один из участников не может выполнить операцию, невозможно сообщить об этом координатору. Например, в случае распределенных транзакций локальная фиксация может быть невозможна, поскольку это нарушит ограничения управления параллелизмом.

На практике требуются более сложные схемы, наиболее распространенной из которых является **протокол двухфазной фиксации** (two-phase commit protocol, 2PC), который мы подробно обсудим ниже. Основным недостатком этого протокола является то, что он обычно не может эффективно обработать сбой координатора. С этой целью был разработан трехфазный протокол, который мы обсудим отдельно в примечании 8.13.

Первоначальный протокол двухфазной фиксации принадлежит Грею (Gray) [1978]. Не ограничивая общности, рассмотрим распределенную транзакцию, включающую участие нескольких процессов, каждый из которых выполняется на другой машине. Предполагая, что сбоев не происходит, протокол состоит из следующих двух этапов, каждый из которых состоит также из двух подэтапов (см. также [Bernstein and Newcomer, 2009]).

1-й этап: координатор отправляет сообщение с запросом голосования всем участникам. Когда участник получает сообщение с запросом на голосование, он возвращает либо сообщение с подтверждением голосования координатору, сообщая координатору, что он готов локально зафиксировать свою часть транзакции, либо иным образом сообщая об отмене голосования.

2-й этап: координатор собирает все голоса участников. Если все участники проголосовали за совершение сделки, то и координатор тоже. В этом случае он отправляет сообщение глобальной фиксации всем участникам. Однако если один участник проголосовал за прерывание транзакции, координатор также примет решение отменить транзакцию и отправит сообщение глобальной отмены многоадресной рассылки.

Каждый участник, проголосовавший за фиксацию, ожидает окончательной реакции координатора. Если участник получает сообщение глобальной фиксации, он локально фиксирует транзакцию. В противном случае при получении сообщения о глобальном прерывании транзакция также прерывается локально.

Первый этап – это этап голосования, состоящий из подэтапов 1 и 2. Второй этап – этап принятия решения, состоящий из подэтапов 3 и 4. Эти четыре этапа показаны в виде конечных диаграмм состояний на рис. 8.30.

Некоторые проблемы возникают, когда базовый протокол 2PC используется в системе, где происходят сбои. Во-первых, обратите внимание, что у координатора и участников есть состояния, в которых они блокируют ожидание входящих сообщений. Следовательно, протокол может легко потерпеть не-

удачу, когда при его сбое из-за неопределенно долгого ожидания сообщения от этого процесса могут быть не выполнены другие процессы. По этой причине используются механизмы тайм-аута. Эти механизмы объяснены далее.

Рис. 8.30 ❖ a) Конечный автомат для координатора в 2PC; b) конечный автомат для участника

Если взглянуть на конечные автоматы на рис. 8.30, то можно увидеть, что существует всего три состояния, в которых либо координатор, либо участник заблокированы в ожидании входящего сообщения. Во-первых, участник может ожидать в своем начальном состоянии (INIT) сообщения запроса голосования от координатора. Если это сообщение не получено через некоторое время, участник просто решит локально прервать транзакцию и, таким образом, отправит сообщение об отмене голосования координатору.

Точно так же координатор может быть заблокирован в состоянии ожидания (WAIT), ожидая голосов каждого участника. Если не все голоса были собраны по истечении определенного периода времени, координатор должен также проголосовать за прерывание, а затем отправить глобальное прерывание всем участникам.

Наконец, участник может быть заблокирован в состоянии готов (READY), ожидая глобального голосования, отправленного координатором. Если это сообщение не получено в течение определенного времени, участник не может просто решить прервать транзакцию. Вместо этого он должен выяснить, какое сообщение на самом деле отправил координатор. Самое простое решение этой проблемы – позволить каждому участнику заблокироваться, пока координатор не восстановится снова.

Лучшее решение – позволить участнику P связаться с другим участником Q, чтобы узнать, сможет ли он из текущего состояния Q решить, что ему де-

лать. Например, предположим, что Q достиг состояния фиксации (COMMIT). Это возможно, только если координатор отправил сообщение глобальной фиксации в Q непосредственно перед сбоем. По-видимому, это сообщение еще не было отправлено участнику P. Следовательно, участник P теперь может также принять решение о локальной фиксации. Аналогично, если Q находится в состоянии ABORT, P также может безопасно прервать.

Теперь предположим, что Q все еще находится в начальном состоянии INIT. Такая ситуация может возникнуть, когда координатор отправил запрос на голосование всем участникам, но это сообщение достигло P (который впоследствии ответил сообщением о голосовании – фиксации), но не достигло Q.

Состояние Q	Действие P
COMMIT	Сделать переход в COMMIT
ABORT	Сделать переход в ABORT
INIT	Сделать переход в ABORT
READY	Связаться с другим участником

Рис. 8.31 ❖ Действия, предпринимаемые участником P, когда он находится в состоянии READY и связался с другим участником Q

Другими словами, координатор потерпел крах во время многоадресной передачи запроса на голосование. В этом случае безопасно прервать транзакцию: и P, и Q могут перейти в состояние ABORT.

Наиболее сложная ситуация возникает, когда Q также находится в состоянии READY, ожидая ответа от координатора. В частности, если окажется, что все участники находятся в состоянии READY, решение не может быть принято. Проблема заключается в том, что хотя все участники готовы взять на себя обязательства, им все еще нужен голос координатора, чтобы принять окончательное решение. Следовательно, протокол блокируется до восстановления координатора.

Различные варианты приведены на рис. 8.31.

Чтобы процесс действительно мог восстановиться, необходимо, чтобы он сохранил свое состояние в постоянном хранилище. Например, если участник находился в состоянии INIT, он может безопасно решить локально прервать транзакцию при ее восстановлении, а затем сообщить об этом координатору. Аналогичным образом, когда он уже принял решение, например когда произошел сбой, находясь в состоянии COMMIT или ABORT, он снова должен вернуться в это состояние и повторно передать свое решение координатору.

Проблемы возникают, когда участник отказывает, находясь в состоянии READY.

В этом случае при восстановлении он не может самостоятельно решить, что ему делать дальше, то есть зафиксировать или прервать транзакцию. Следовательно, он вынужден связываться с другими участниками, чтобы найти то, что он должен делать, аналогично ситуации, когда время ожидания истекает, в то время когда он находится в состоянии READY, как описано выше.

У координатора есть только два критических состояния, которые необходимо отслеживать. Когда он запускает протокол 2PC, он должен записать, что

входит в состояние WAIT, чтобы он мог повторно передать сообщение запроса голосования всем участникам после восстановления. Аналогичным образом, если решение пришло на втором этапе, достаточно, чтобы это решение было записано, и его можно повторно передать при восстановлении.

Таким образом, возможно, что участнику потребуется заблокироваться, пока координатор не восстановится. Эта ситуация возникает, когда все участники получили и обработали сообщение с запросом голосования от координатора, в то время как координатор потерпел крах. В этом случае участники не могут совместно принять решение о последнем действии. По этой причине 2PC относится к **протоколам блокировки фиксации** (blocking commit protocol).

Есть несколько решений, чтобы избежать блокировки. Одним из решений является использование многоадресного примитива, с помощью которого получатель немедленно передает полученное сообщение всем другим процессам [Babaoglu and Toueg, 1993]. Можно показать, что этот подход позволяет участнику принять окончательное решение, даже если координатор еще не восстановился. Другое решение состоит в использовании трех вместо двух фаз, как мы обсуждаем в примечании 8.13.

Примечание 8.12 (дополнительно: 2PC в Python)

Схема действий, выполняемых координатором, приведена на рис. 8.32. Координатор начинает с многоадресной передачи запроса на голосование всем участникам, чтобы собрать их голоса. Впоследствии он записывает, что входит в состояние **WAIT**, после чего ожидает входящих голосов от участников.

```
class Coordinanor:

  def run(self):
    yetToReceive = list(participants)
    self.log.info('WAIT')
    self.chan.sendTo(participants, VOTE_REQUEST)
    while len(stillToReceive)> 0:
      msg = self.chan.recvFrom(participants, TIMEOUT)
      if (not msg) or (msg[1] == VOTE_ABORT):
        self.log.info('ABORT')
        self.chan.sendTo(participants, GLOBAL_ABORT)
        return
      else: # msg[1] == VOTE_COMMIT
        yetToReceive.remove(msg[0])
    self.log.info('COMMIT')
    self.chan.sendTo(participants, GLOBAL_COMMIT)
```

Рис. 8.32 ❖ Шаги, предпринятые координатором в протоколе 2PC

Если не все голоса были собраны, но больше голосов не было получено в течение заданного заранее интервала времени, координатор предполагает, что один или несколько участников потерпели неудачу. Следовательно, он должен прервать транзакцию и выполнить многоадресную передачу глобального прерывания (оставшимся) участникам. Аналогично, если только один участник решит прервать транзакцию,

координатор должен будет отозвать транзакцию. Если все участники проголосуют за принятие, глобальная фиксация сначала регистрируется и впоследствии отправляется всем процессам. В противном случае координатор выполняет многоадресную передачу глобального прерывания (после записи в локальный журнал).

После получения запроса на голосование участник выполняет свою работу. Все ставки отменяются, если его работа не удалась, но в противном случае он проголосует за совершение транзакции. Он записывает свое решение в локальный журнал и информирует координатора, отправляя сообщение с подтверждением голосования. Затем участник должен дождаться глобального решения. Предполагая, что это решение (которое снова должно прийти от координатора) приходит вовремя, оно просто записывается в локальный журнал, после чего его можно выполнить (последнее в коде не показано).

На рис. 8.33 показаны шаги, предпринятые участником. Сначала процесс ожидает запроса на голосование от координатора. Если сообщение не приходит, транзакция просто прерывается. Видимо, координатор потерпел неудачу.

```
class Participant:
  def run(self):
    self.log.info('INIT')
    msg = self.chan.recvFrom(coordinator, TIMEOUT)
    if (not msg): # Сбой координатора - отказаться полностью
      decision = LOCAL_ABORT
    else: # координатор отправит VOTE_REQUEST
      decision = self.do_work()
      if decision == LOCAL_ABORT:
        self.chan.sendTo(coordinator, VOTE_ABORT)
      else: # Готово к принятию, войдите в состояние ГОТОВ
        self.log.info('READY')
        self.chan.sendTo(coordinator, VOTE_COMMIT)
        msg = self.chan.recvFrom(coordinator, TIMEOUT)
        if (not msg): # Сбой координатора - проверить остальные
          self.chan.sendTo(all_participants, NEED_DECISION)
          while True:
            msg = self.chan.recvFromAny()
            if msg[1] in [GLOBAL_COMMIT, GLOBAL_ABORT, LOCAL_ABORT]:
              decision = msg[1]
              break
        else: # Координатор пришел к решению
          decision = msg[1]
    if desicion == GLOBAL_COMMIT:
      self.log.info('COMMIT')
    else: # Решение в [GLOBAL_ABORT, LOCAL_ABORT]:
      self.log.info('ABORT')
    while True: # Помочь любому другому участнику в случае сбоя координатора
      msg = self.chan.recvFrom(all_participants)
      if msg[1] == NEED_DECISION:
        self.chan.sendTo([msg[0]], decision)
```

Рис. 8.33 ❖ Шаги, предпринятые участником процесса в 2PC

Однако если участник ожидает тайм-аута в ожидании решения координатора, он выполняет протокол завершения, сначала отправляя сообщение запроса решения

на многоадресную рассылку DECISION-REQUEST, после чего впоследствии блокирует его, ожидая ответа. Когда приходит решающий ответ (возможно, от координатора, который, как предполагается, в конечном итоге восстанавливается), участник записывает решение в свой локальный журнал и обрабатывает его соответствующим образом. Любой запрос от другого участника об окончательном решении остается без ответа, пока это решение неизвестно.

Мы продолжаем поддерживать каждого участника, после того как он решил либо зафиксировать, либо прервать. Затем он может помочь другим участникам, нуждающимся в решении, обнаружив, что координатор потерпел крах. С этой целью участник блокирует входящие сообщения и возвращает свое собственное решение по запросу. Обратите внимание, что мы фактически предоставляем реализацию, которая поддерживает частично синхронное поведение: мы предполагаем, что тайм-ауты могут быть применены как механизм для обнаружения отказов, но, принимая во внимание, что мы можем принять ошибочное заключение о крахе сервера.

Примечание 8.13 (дополнительно: трехфазная фиксация)

Проблема с протоколом двухфазной фиксации заключается в том, что после сбоя координатора участники могут быть не в состоянии принять окончательное решение. Следовательно, участникам, возможно, придется оставаться заблокированными до восстановления координатора. В [Скин, 1981] разработан вариант 2PC, называемый **протоколом трехфазной фиксации** (three-phase commit protocol, 3PC), который позволяет избежать блокирования процессов при наличии аварийных остановов. Хотя 3PC широко упоминается в литературе, на практике это применяется нечасто, поскольку условия, при которых 2PC блокируются, встречаются редко. Мы обсуждаем этот протокол, так как он дает дополнительное представление о решении проблем отказоустойчивости в распределенных системах.

Как и 2PC, 3PC также формулируется в терминах координатора и количества участников. Соответствующие им конечные автоматы показаны на рис. 8.34. Суть протокола заключается в том, что состояния координатора и каждого участника удовлетворяют следующим двум условиям:

1) нет единого состояния, из которого можно было бы перейти непосредственно в состояние COMMIT или ABORT;
2) не существует состояния, в котором невозможно принять окончательное решение и из которого может быть выполнен переход в состояние COMMIT.

Можно показать, что эти два условия необходимы и достаточны для отсутствия блокирования протокола фиксации [Skeen and Stonebraker, 1983].

Координатор в 3PC начинает с отправки сообщения с запросом на голосование всем участникам, после чего ожидает входящих ответов. Если какой-либо участник проголосует за отмену транзакции, окончательным решением будет также прерывание транзакции, поэтому координатор отправляет глобальное прерывание. Однако когда транзакция может быть зафиксирована, отправляется сообщение о подготовке к фиксации. Только после того, как каждый участник подтвердил, что теперь он готов к фиксации, координатор отправит окончательное сообщение глобальной фиксации, с помощью которого транзакция фактически фиксируется.

Есть только несколько ситуаций, в которых процесс блокируется во время ожидания входящих сообщений. Во-первых, если участник ожидает запроса на голосование от координатора, находясь в начальном состоянии (UNIT), он в конечном итоге сделает переход в состояние прерывания (ABORT), тем самым предполагая, что координатор потерпел крах. Эта ситуация идентична ситуации в 2PC. Аналогично координатор может быть в состоянии ожидания (WAIT) голосов от участни-

ков. По истечении времени ожидания координатор сделает вывод о сбое участника и, таким образом, прервет транзакцию путем многоадресной рассылки сообщения глобального прерывания.

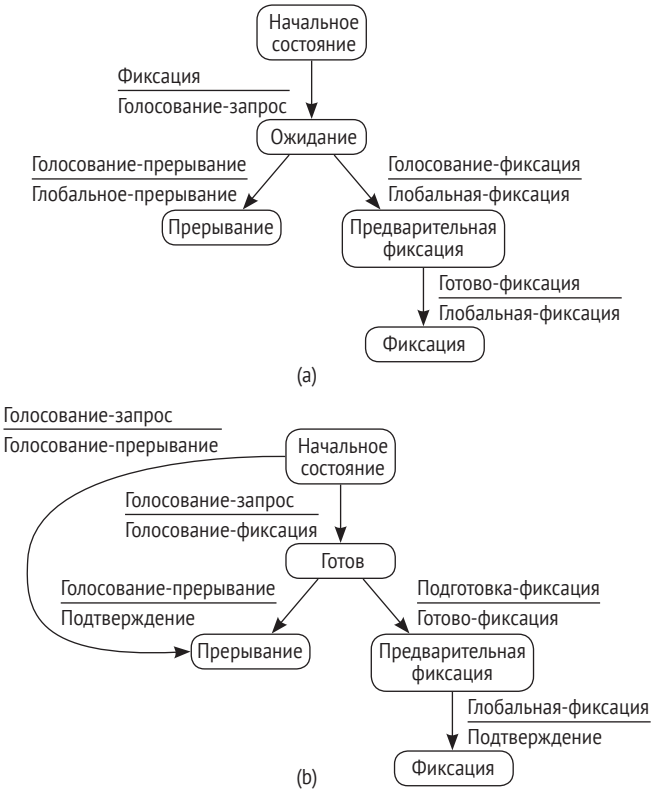


Рис. 8.34 ❖ a) Конечный автомат для координатора в 3PC; b) конечный автомат для участника

Теперь предположим, что координатор заблокирован в состоянии предварительной фиксации PRECOMMIT. По тайм-ауту он решит, что один из участников потерпел крах, но известно, что этот участник проголосовал за совершение транзакции. Следовательно, координатор может проинструктировать участников операции о фиксации путем многоадресной передачи сообщения глобальной фиксации. Кроме того, он полагается на протокол восстановления для потерпевшего аварию участника, который фиксирует свою часть транзакции при повторном запуске.

Участник P может заблокировать в состоянии готово (READY) или в состоянии PRECOMMIT. По тайм-ауту P может сделать вывод только о том, что координатор потерпел неудачу, так что теперь ему нужно выяснить, что делать дальше. Как и в 2PC, если P связывается с любым другим участником, находящимся в состоянии COMMIT (или ABORT), P также должен перейти в это состояние. Кроме того, если все участники находятся в состоянии PRECOMMIT, транзакция может быть совершена.

Аналогично 2PC, если другой участник Q все еще находится в состоянии INIT, транзакция может быть безопасно отменена. Важно отметить, что Q может находиться в состоянии INIT, только если ни один другой участник не находится

в состоянии PRECOMMIT. Участник может достичь PRECOMMIT только в том случае, если координатор достиг состояния PRECOMMIT до сбоя и, таким образом, получил голос для принятия от каждого участника. Другими словами, ни один участник не может находиться в состоянии INIT, пока другой участник находится в состоянии PRECOMMIT.

Если каждый из участников, с которым P может связаться, находится в состоянии READY (и они вместе составляют большинство), транзакция должна быть прервана. Следует отметить, что другой участник, возможно, потерпел крах и позже восстановится. Однако ни P, ни кто-либо из участников операции не знают, каким будет состояние потерпевшего крах участника, когда он восстановится. Если процесс восстанавливается до состояния INIT, то решение об отмене транзакции является единственно правильным решением. В худшем случае процесс может восстановиться до состояния PRECOMMIT, но в этом случае он не может причинить никакого вреда, продолжая прерывать транзакцию.

Эта ситуация является основным отличием от 2PC, где потерпевший крах участник мог вернуться в состояние COMMIT, в то время как все остальные были все еще в состоянии READY. В этом случае оставшиеся операционные процессы не могут принять окончательное решение и должны будут ждать восстановления аварийного процесса. С 3PC, если какой-либо рабочий процесс находится в состоянии READY, ни один сбойный процесс не восстановится до состояния, отличного от INIT, ABORT или PRECOMMIT. По этой причине выживающие процессы всегда могут прийти к окончательному решению.

Наконец, если процессы, которых может достичь P, находятся в состоянии PRECOMMIT (и они составляют большинство), тогда безопасно совершать транзакцию. Опять же, можно показать, что в этом случае все остальные процессы будут либо в состоянии READY, либо, по крайней мере, восстановятся в состояние READY, PRECOMMIT или COMMIT после сбоя. Более подробную информацию о 3PC можно найти в [Bernstein et al., 1987] и [Özsu and Valduriez, 2011].

8.6. Восстановление

До сих пор мы в основном концентрировались на алгоритмах, которые позволяют нам терпеть ошибки. Однако, как только произошел сбой, важно, чтобы процесс, в котором произошел сбой, мог вернуться в правильное состояние. Далее мы сконцентрируемся на том, что на самом деле означает восстановление до правильного состояния, а затем и том, когда и как можно записать и восстановить состояние распределенной системы с помощью контрольных точек и регистрации сообщений.

Введение

Основой отказоустойчивости является восстановление после ошибки. Напомним, что ошибка – это та часть системы, которая может привести к отказу. Вся идея восстановления после ошибок состоит в том, чтобы заменить ошибочное состояние безошибочным состоянием. Существуют две формы восстановления после ошибок.

При **обратном восстановлении** (backward recovery) основной проблемой является приведение системы из ее нынешнего ошибочного состояния обратно в ранее правильное состояние. Для этого необходимо время от времени регистрировать состояние системы и, когда что-то идет не так, восстанавливать такое записанное состояние. Каждый раз (когда) записывается текущее состояние системы, говорят, что создается **контрольная точка** (checkpoint).

Другая форма восстановления после ошибок – **прямое восстановление** (forward recovery). В этом случае, когда система вошла в ошибочное состояние, вместо того чтобы возвращаться к предыдущему состоянию контрольной точки, делается попытка привести систему в правильное новое состояние, из которого она может продолжить работу. Основная проблема с механизмами прямого восстановления после ошибок заключается в том, что необходимо заранее знать, какие ошибки могут возникнуть. Только в этом случае можно исправить эти ошибки и перейти в новое состояние.

Различие между обратным и прямым восстановлением ошибок легко объясняется при рассмотрении вопроса о реализации надежной связи. Общий подход к восстановлению из потерянного пакета состоит в том, чтобы позволить отправителю повторно передать этот пакет. По сути, повторная передача пакета устанавливает, что мы пытаемся вернуться к предыдущему правильному состоянию, а именно к тому, в котором отправляется потерянный пакет. Таким образом, надежная связь посредством повторной передачи пакетов является примером применения методов обратного восстановления после ошибок.

Альтернативный подход заключается в использовании метода, известного как **коррекция стирания** (erasure correction). При таком подходе отсутствующий пакет создается из других успешно доставленных пакетов. Например, в (n, k)-блочном коде стирания набор из k *исходных пакетов* кодируется в набор из n кодированных пакетов, так что любого набора из k кодированных пакетов достаточно для восстановления k исходных пакетов. Типичными значениями являются $k = 16$ или $k = 32$ и $k < n \leq 2k$ (см., например, [Rizzo, 1997]). Если пакетов было отправлено недостаточно, отправителю придется продолжать передачу пакетов, пока не будет создан ранее потерянный пакет. Коррекция стирания является типичным примером метода прямого восстановления после ошибок.

В целом методы обратного восстановления после ошибок широко применяются в качестве общего механизма восстановления в распределенных системах после сбоев. Основным преимуществом обратного восстановления после ошибок является то, что он является общепринятым методом, независимым от какой-либо конкретной системы или процесса. Другими словами, он может быть интегрирован в (уровень промежуточного программного обеспечения) распределенную систему как сервис общего назначения.

Однако обратное исправление ошибок также создает некоторые проблемы [Singhal and Shivaratri, 1994]. Во-первых, восстановление системы или процесса до предыдущего состояния обычно является относительно дорогостоящей операцией с точки зрения производительности. Как будет обсуждаться в последующих разделах, обычно требуется много работы для восстановления, например после сбоя процесса или сбоя сайта. Потенциальным вы-

ходом из этой проблемы является разработка очень дешевых механизмов, с помощью которых компоненты просто перезагружаются.

Во-вторых, поскольку механизмы обратного восстановления после ошибок не зависят от распределенного приложения, для которого они фактически используются, нельзя дать никаких гарантий, что после восстановления такой же или аналогичный сбой больше не произойдет. Если такие гарантии необходимы, обработка ошибок часто требует, чтобы приложение попадало в цикл восстановления. Другими словами, полная прозрачность отказов, как правило, не может быть обеспечена механизмами обратного восстановления после ошибок.

Наконец, хотя для восстановления после ошибок требуется контрольная точка, некоторые состояния просто нельзя откатить. Например, как только (возможно, злонамеренно) человек взял 1000 долларов, внезапно появившихся из неправильно функционирующего банкомата, существует лишь небольшая вероятность того, что деньги будут вложены обратно в банкомат. Аналогично восстановление в предыдущее состояние в большинстве систем Unix, после того как с энтузиазмом набрано

```
/bin/rm -fr *
```

но из неправильного рабочего каталога, несколько человек могут побледнеть. Некоторые вещи просто необратимы.

Контрольная точка позволяет восстановить предыдущее правильное состояние. Однако взятие контрольного пункта часто является дорогостоящей операцией и может привести к серьезным потерям производительности. Как следствие многие отказоустойчивые распределенные системы сочетают контрольные точки с **протоколированием сообщений** (message logging). В этом случае, после того как контрольная точка была пройдена, процесс регистрирует свои сообщения перед отправкой (так называемая регистрация на основе отправителя). Альтернативное решение состоит в том, чтобы позволить принимающему процессу сначала зарегистрировать входящее сообщение, прежде чем доставлять его в приложение, которое он выполняет. Эта схема также упоминается как **регистрация на основе получателя** (receiver-based logging). Когда происходит сбой принимающего процесса, необходимо восстановить последнее состояние контрольной точки и оттуда воспроизвести отправленные сообщения. Следовательно, объединение контрольных точек с ведением журнала сообщений позволяет восстановить состояние, которое находится за пределами самой последней контрольной точки, без затрат на контрольные точки.

В системе, где используется только контрольная точка, процессы будут восстановлены в состояние контрольной точки. С этого момента их поведение может отличаться от того, что было до сбоя. Например, поскольку время связи не является детерминированным, сообщения теперь могут доставляться в другом порядке, что, в свою очередь, приводит к разным реакциям получателей. Однако если осуществляется ведение журнала сообщений, происходит фактическое воспроизведение событий, которые произошли с момента последней контрольной точки. Такое повторение облегчает взаимодействие с внешним миром.

Рассмотрим, например, случай, когда произошел сбой, поскольку пользователь предоставил ошибочный ввод. Если используется только контрольная точка, система должна будет принять контрольную точку, прежде чем принимать ввод пользователя, чтобы восстановить точно такое же состояние. При ведении журнала сообщений можно использовать более старую контрольную точку, после которой может происходить воспроизведение событий до того момента, когда пользователь должен будет предоставить ввод. На практике сочетание меньшего количества контрольных точек и регистрации сообщений является более эффективным, чем использование множества контрольных точек.

В работе [Elnozahy et al., 2002] предоставлен обзор контрольных точек и регистрации в распределенных системах. Различные алгоритмические подробности можно найти в [Chow and Johnson, 1997].

Контрольная точка

В отказоустойчивой распределенной системе для обратного восстановления после ошибок требуется, чтобы система регулярно сохраняла свое состояние[1].

В частности, нам необходимо записать согласованное глобальное состояние, называемое **распределенным снимком** (distributed snapshot). В распределенном снимке если процесс P записал получение сообщения, то также должен быть процесс Q, который записал отправку этого сообщения. В конце концов, это должно откуда-то прийти.

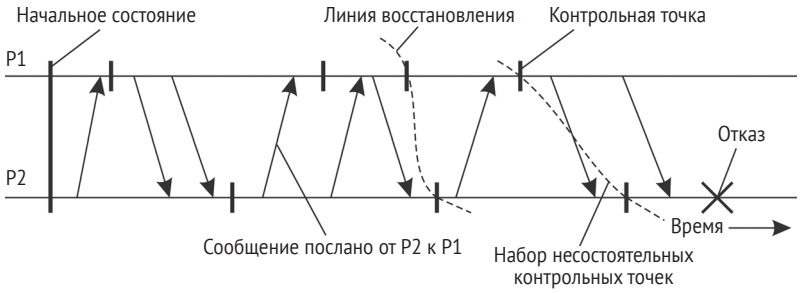


Рис. 8.35 ❖ Линия восстановления

Для восстановления после сбоя процесса или системы требуется, чтобы мы создавали согласованное глобальное состояние из локальных состояний, сохраняемых каждым процессом. В частности, лучше всего восстановить самый последний распределенный снимок, также называемый **линией восстановления** (recovery line). Другими словами, линия восстановления соответствует последней согласованной коллекции контрольных точек, как показано на рис. 8.35.

---

[1] Мы предполагаем, что каждый процесс имеет доступ к надежной локальной памяти.

Скоординированная контрольная точка

В согласованной контрольной точке все процессы синхронизируются для совместной записи своего состояния в локальное хранилище. Основным преимуществом скоординированной контрольной точки является то, что сохраненное состояние автоматически согласовано на глобальном уровне. Простым решением является использование протокола двухфазной блокировки. Координатор сначала передает сообщение-запрос (checkpoint-request) контрольной точки всем процессам. Когда процесс получает такое сообщение, он берет локальную контрольную точку, ставит в очередь любое последующее сообщение, переданное ему приложением, которое он выполняет, и подтверждает координатору, что принял контрольную точку. Когда координатор получил подтверждение от всех процессов, он отправляет сообщение о завершении контрольной точки, чтобы позволить (заблокированным) процессам продолжить работу.

Легко видеть, что этот подход также приведет к глобально согласованному состоянию, потому что никакое входящее сообщение никогда не будет зарегистрировано как часть контрольной точки. Причина в том, что любое сообщение, которое следует за запросом на получение контрольной точки, не считается частью локальной контрольной точки. В то же время исходящие сообщения (передаваемые процессу контрольной точки приложением, которое оно запускает) помещаются в очередь локально, пока не получено сообщение о завершении контрольной точки.

Усовершенствование этого алгоритма состоит в отправке запроса контрольной точки только тем процессам, которые зависят от восстановления координатора и игнорируют другие процессы. Процесс зависит от координатора, если он получил сообщение, прямо или косвенно причинно связанное с сообщением, отправленным координатором со времени последней контрольной точки. Это приводит к понятию **инкрементный снимок** (incremental snapshot).

Чтобы сделать инкрементный снимок, координатор отправляет запрос контрольной точки только тем процессам, которым он отправил сообщение с момента последней проверки контрольной точки. Когда процесс P получает такой запрос, он пересылает запрос всем тем процессам, которым сам P отправил сообщение с момента последней контрольной точки, и т. д. Процесс перенаправляет запрос только один раз. Когда все процессы были идентифицированы, вторая многоадресная передача используется для фактического запуска контрольных точек и позволяет процессам продолжаться там, где они остановились.

Независимая контрольная точка

Теперь рассмотрим случай, когда каждый процесс время от времени просто записывает свое локальное состояние некоординированным образом. Чтобы обнаружить линию восстановления, необходимо, чтобы каждый процесс был отправлен до своего последнего сохраненного состояния. Если эти локальные состояния совместно не образуют распределенный снимок, необходим

дальнейший откат. Этот процесс отката может привести к так называемому **эффекту домино** (domino effect) и показан на рис. 8.36.

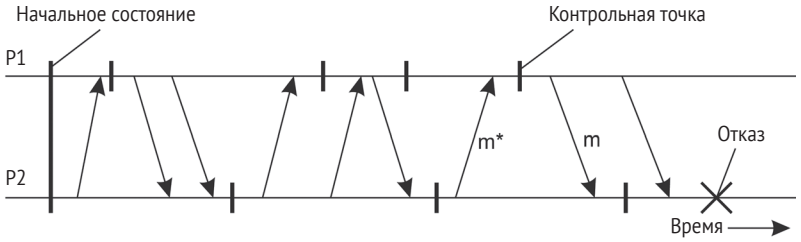


Рис. 8.36 ❖ Эффект домино

Когда происходит сбой процесса P_2, нам нужно восстановить его состояние до последней сохраненной контрольной точки. Как следствие процесс P_1 также необходимо будет откатить. К сожалению, два последних сохраненных локальных состояния не образуют согласованного глобального состояния: состояние, сохраненное P_2, указывает на получение сообщения m, но никакой другой процесс не может быть идентифицирован как его отправитель. Следовательно, P_2 необходимо откатить к более раннему состоянию.

Однако следующее состояние, до которого выполняется откат P_2, также нельзя использовать как часть распределенного снимка. В этом случае P_1 будет записывать получение сообщения m, но не записано событие отправки этого сообщения. Поэтому необходимо также откатить P_1 к предыдущему состоянию. В этом примере оказывается, что линия восстановления фактически является начальным состоянием системы.

Поскольку процессы используют локальные контрольные точки, независимые друг от друга, этот метод также называется **независимой контрольной точкой** (independent checkpointing). Его реализация требует, чтобы зависимости регистрировались таким образом, чтобы процессы могли совместно откатиться до согласованного глобального состояния. Для этого пусть $CP_i(m)$ обозначает m-ю контрольную точку, взятую процессом P_i. Также пусть $INT_i(m)$ обозначает интервал между контрольными точками $CP_i(m-1)$ и $CP_i(m)$.

Когда процесс P_i отправляет сообщение с интервалом $INT_i(m)$, он связывает пару (i, m) с процессом получения. Когда процесс P_j получает сообщение в интервальном $INT_j(n)$ вместе с парой индексов (i, m), он записывает зависимость $INT_i(m) \rightarrow INT_j(n)$. Всякий раз, когда P_j принимает контрольную точку $CP_j(n)$, он дополнительно сохраняет эту зависимость в своем локальном хранилище вместе с остальной информацией восстановления, которая является частью $CP_j(n)$.

Теперь предположим, что в определенный момент процесс P_1 должен откатиться до контрольной точки $CP_j(m-1)$. Чтобы обеспечить глобальную согласованность, мы должны обеспечить, чтобы все процессы, которые получили сообщения от P_j и были отправлены в интервале $INT_i(m)$, откатились до состояния контрольной точки, предшествующей получению таких сообщений. В частности, процесс P_j в нашем примере необходимо будет откатить хотя

бы до контрольной точки $CP_j(n-1)$. Если $CP_j(n-1)$ не приводит к глобально согласованному состоянию, может потребоваться дальнейший откат.

Расчет линии восстановления требует анализа интервальных зависимостей, записанных каждым процессом, когда была взята контрольная точка. Не вдаваясь в подробности, скажем, что такие расчеты довольно сложны. Кроме того, как выясняется, доминирующим фактором производительности часто является отсутствие координации между процессами, а затраты возникают в результате необходимости сохранения состояния в локальном стабильном хранилище. Следовательно, скоординированные контрольные точки, которые намного проще, чем независимые контрольные точки, часто более популярны и, вероятно, останутся такими даже при увеличении размеров систем [Elnozahy and Plank, 2004].

Регистрация сообщений

С учетом того, что контрольные точки могут быть дорогостоящей операцией, были предприняты попытки уменьшить количество контрольных точек, но при этом обеспечить повторное отображение. Одним из таких методов является регистрация сообщений. Основная идея, лежащая в основе регистрации сообщений, заключается в том, что если передача сообщений может быть воспроизведена, мы все равно можем достичь глобально согласованного состояния, но без необходимости восстанавливать это состояние из локальных хранилищ. Вместо этого в качестве отправной точки принимается состояние контрольной точки, и все сообщения, которые были отправлены с тех пор, просто повторно передаются и обрабатываются соответствующим образом.

Этот подход прекрасно работает в предположении, что называется **кусочно-детерминированной моделью исполнения** (piece-wise deterministic execution model). В такой модели предполагается, что выполнение каждого процесса происходит в виде последовательности интервалов, в которых происходят события. Эти события аналогичны тем, которые обсуждались в контексте отношения между событиями Лампорта в главе 6. Например, событие может быть выполнением инструкции, отправкой сообщения и т. д. Предполагается, что каждый интервал в кусочно-детерминированной модели начинается с недетерминированного события, такого как получение сообщения. Однако с этого момента выполнение процесса является полностью детерминированным. Интервал заканчивается последним событием до того, как происходит недетерминированное событие.

По сути, интервал может быть воспроизведен с известным результатом, то есть полностью детерминированным образом, при условии что он воспроизводится, начиная с того же недетерминированного события, что и раньше. Следовательно, если мы регистрируем все недетерминированные события в такой модели, становится возможным полностью воспроизвести все выполнение процесса детерминистическим способом.

Учитывая, что журналы сообщений необходимы для восстановления после сбоя процесса, чтобы восстановить глобально согласованное состояние, становится важно точно знать, когда сообщения должны регистрироваться.

Следуя подходу, описанному в [Alvisi and Marzullo, 1998], выясняется, что многие существующие схемы регистрации сообщений можно легко охарактеризовать, если мы сконцентрируемся на том, как они справляются с потерянными процессами.

Потерянный процесс (orphan process) – это процесс, который пережил сбой другого процесса, но состояние которого не соответствует сбойному процессу после его восстановления. В качестве примера рассмотрим ситуацию, показанную на рис. 8.37. Процесс Q получает сообщения m_1 и m_2 от процессов P и R соответственно и впоследствии отправляет сообщение m_3 в R. Однако, в отличие от всех других сообщений, сообщение m_2 не регистрируется. Если происходит сбой процесса Q и последующее восстановление, воспроизводятся только зарегистрированные сообщения, необходимые для восстановления Q, в нашем примере m_1. Поскольку m_2 не было зарегистрировано, его передача не будет воспроизведена, что означает, что передача m_3 также может быть не воспроизведена.

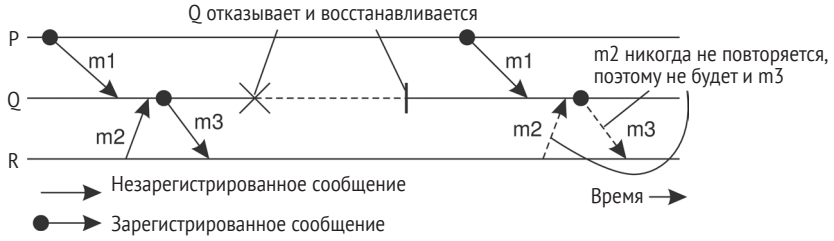


Рис. 8.37 ❖ Неправильное воспроизведение сообщений после восстановления, приводящее к потерянному процессу R

Однако ситуация после восстановления Q несовместима с ситуацией до его восстановления. В частности, R содержит сообщение (m_3), которое было отправлено до сбоя, но чей прием и доставка не происходят при воспроизведении того, что произошло до сбоя. Подобных несоответствий, очевидно, следует избегать.

Примечание 8.14 (дополнительно: характеристика схем регистрации сообщений)

Чтобы охарактеризовать различные схемы регистрации сообщений, мы следуем подходу, описанному [Alvisi and Marzullo, 1998]. Считается, что каждое сообщение m имеет заголовок, который содержит всю информацию, необходимую для повторной передачи m и правильной обработки. Например, каждый заголовок будет идентифицировать отправителя и получателя, а также порядковый номер, чтобы распознать его как дубликат. Кроме того, номер доставки может быть добавлен, чтобы решить, когда именно он должен быть передан получающему приложению.

Сообщение считается **стабильным** (stable), если оно больше не может быть потеряно, например потому, что оно было записано в надежное локальное хранилище. Таким образом, стабильные сообщения могут быть использованы для восстановления путем воспроизведения их передачи.

Каждое сообщение m приводит к набору **DEP**(m) процессов, которые зависят от доставки m. В частности, **DEP**(m) состоит из тех процессов, в которые было доставлено m. Кроме того, если другое сообщение m причинно зависит от доставки m и m* доставлено процессу Q, тогда Q также будет содержаться в **DEP**(m). Обратите внимание, что m* причинно зависит от доставки m, если оно было отправлено тем же процессом, который ранее доставил m или который доставил другое сообщение, причинно зависевшее от доставки m.

Набор **COPY**(m) состоит из тех процессов, которые имеют копию m, но еще не (надежно) ее сохранили. Когда процесс Q доставляет сообщение m, он также становится членом **COPY**(m). Обратите внимание, что **COPY**(m) состоит из тех процессов, которые могли передать копию m, которая может быть использована для воспроизведения передачи m. Если все эти процессы дают сбой, воспроизведение передачи m явно неосуществимо.

Используя эти обозначения, теперь легко определить, что такое потерянный процесс (процесс-сирота). Пусть **FAIL** обозначает набор сбойных процессов, и пусть Q – один из выживших. Q является потерянным процессом, если есть сообщение m, такое, что Q содержится в **DEP**(m), в то же время каждый процесс в **COPY**(m) завершился сбоем. Более формально:

Q (сирота) $\Leftrightarrow \exists m : Q \in$ **DEP**(m) и **COPY**(m) $\subseteq$ **FAIL**.

Другими словами, сиротский процесс появляется, когда он зависит от m, но нет способа воспроизвести передачу m.

Чтобы избежать потерянных процессов, нам необходимо убедиться, что если каждый процесс в **COPY**(m) потерпит крах, то в **DEP**(m) не останется ни одного выжившего процесса. Другими словами, все процессы в **DEP**(m) также должны были аварийно завершаться. Это условие может быть выполнено, если мы можем гарантировать, что всякий раз, когда процесс становится членом **DEP**(m), он также становится членом **COPY**(m). Другими словами, всякий раз, когда процесс становится зависимым от доставки m, он сохраняет копию m.

По существу, теперь есть два подхода, которым можно следовать. Первый подход представлен так называемыми **протоколами пессимистической регистрации** (pessimistic logging protocols). Эти протоколы учитывают, что для каждого нестабильного сообщения m существует не более одного процесса, зависящего от m. Другими словами, **протоколы пессимистической регистрации** гарантируют, что каждое нестабильное сообщение m доставляется не более чем одному процессу. Обратите внимание, что как только m доставляется, скажем, к процессу P, P становится членом **COPY**(m).

Худшее, что может случиться, – это сбой процесса P, когда m не было зарегистрировано. При пессимистической регистрации P не разрешается отправлять какие-либо сообщения после доставки m, не убедившись, что m записано в надежное хранилище. Следовательно, никакие другие процессы никогда не станут зависимыми от доставки m к P, без возможности воспроизведения передачи m. Таким образом, бесхозные процессы всегда избегаются.

Напротив, в **протоколе оптимистической регистрации** (optimistic logging protocol) фактическая работа выполняется после сбоя. В частности, предположим, что для некоторого сообщения m произошел сбой каждого процесса в **COPY**(m). При оптимистическом подходе любой потерянный процесс в **DEP**(m) откатывается в состояние, в котором он больше не принадлежит **DEP**(m). Очевидно, что оптимистичные протоколы протоколирования должны отслеживать зависимости, что усложняет их реализацию.

Как отмечено в [Elnozahy et al., 2002], пессимистическое ведение журнала намного проще, чем оптимистический подход, и это предпочтительный способ ведения журнала сообщений при практическом проектировании распределенных систем.

Вычисления, ориентированные на восстановление

Этот способ обработки восстановления – начать обработку заново. Основное достоинство данного метода, лежащее в его основе маскирования отказов, – оптимизация, которая может оказаться гораздо дешевле для восстановления, чем те системы, которые свободны от сбоев в течение длительного времени. Этот подход упоминается в [Candea et al., 2004a] как **ориентированные на восстановление вычисления** (recovery-oriented computing).

Существуют различные виды ориентированных на восстановление вычислений. Один из способов – просто перезагрузить компьютер (часть системы) и изучить возможность перезагрузки интернет-серверов [Candea et al., 2004b; 2006]. Чтобы иметь возможность перезагрузить только часть системы, важно, чтобы ошибка была правильно локализована. На этом этапе перезагрузка означает просто удаление всех экземпляров идентифицированных компонентов, а также потоков, работающих с ними, и (часто) просто перезапуск связанных запросов. Обратите внимание, что локализация ошибок сама по себе может быть нетривиальным занятием [Steinder and Sethi, 2004].

Чтобы включить перезагрузку в качестве практического метода восстановления, необходимо, чтобы компоненты были в значительной степени отделены в том смысле, что между различными компонентами практически отсутствуют зависимости. Если существуют сильные зависимости, то для локализации и анализа неисправностей может потребоваться перезапуск всего сервера, после чего применение традиционных методов восстановления может оказаться более эффективным, чем те, которые мы только что обсуждали.

Другим преимуществом вычислений, ориентированных на восстановление, является применение контрольных точек и методов восстановления, но продолжение выполнения в измененной среде. Многих сбоев можно просто избежать, если программам дается больше буферного пространства, память обнуляется перед выделением, изменяется порядок доставки сообщений (если это не влияет на семантику) и т. д. [Qin et al., 2005]. Основная идея заключается в устранении сбоев программного обеспечения (в то время как многие из обсуждаемых методик нацелены на аппаратные сбои или основаны на них). Поскольку выполнение программного обеспечения является в высокой степени детерминированным, изменение среды выполнения может сэкономить время без каких-либо исправлений.

8.7. Резюме

Отказоустойчивость является важным аспектом при проектировании распределенных систем. Отказоустойчивость определяется как характеристика, с помощью которой система может маскировать возникновение и восстановление после сбоев. Другими словами, система является отказоустойчивой, если она может продолжать работать при наличии отказов.

Существует несколько типов сбоев. Аварийный сбой – когда процесс просто останавливается. Ошибка пропуска возникает, когда процесс не отвечает на входящие запросы. Когда процесс отвечает слишком рано или слишком поздно на запрос, говорят, что он имеет ошибку синхронизации. Ответ на входящий запрос, но неправильным образом, является примером ошибки ответа. Наиболее сложные ошибки, с которыми приходится сталкиваться, – это те, в которых процесс проявляет какие-либо ошибки, называемые произвольными, или византийскими, отказами.

Резервирование является ключевым методом, необходимым для достижения отказоустойчивости. Применительно к процессам становится важным понятие групп процессов. Группа процессов состоит из ряда процессов, которые тесно сотрудничают в предоставлении услуг. В отказоустойчивых группах процессов один или несколько процессов могут выйти из строя, не влияя на доступность службы, реализуемой группой. Часто для обеспечения отказоустойчивости необходимо, чтобы связь внутри группы была высоконадежной и придерживалась строгих свойств упорядоченности и атомарности.

Основная проблема заключается в том, что членам группы процессов необходимо достичь консенсуса при наличии различных сбоев. К настоящему времени алгоритм Paxos является хорошо отлаженным и очень надежным алгоритмом консенсуса. Используя $2k + 1$ серверов, он может обеспечить k-отказоустойчивость. Однако если необходимо устранение произвольных сбоев. нам нужно $3k + 1$ серверов.

Надежное групповое общение, также называемое надежной многоадресной передачей, имеет разные формы. Оказывается, что реализация высокой надежности возможна, когда группы относительно малы. Однако как только необходимо поддерживать очень большие группы, масштабируемость надежной многоадресной рассылки становится проблематичной.

Ключевой проблемой в достижении масштабируемости является уменьшение количества сообщений обратной связи, с помощью которых получатели сообщают об успешном или неуспешном получении многоадресного сообщения.

Положение становится хуже, когда должна быть обеспечена атомарность. В атомарных протоколах многоадресной рассылки важно, чтобы у каждого члена группы было одинаковое представление о том, каким членам доставлено многоадресное сообщение. Атомарная групповая адресация может быть точно сформулирована в терминах виртуальной синхронной модели. По сути, эта модель вводит границы, в пределах которых членство в группе не изменяется и сообщения членов надежно передаются. Сообщение никогда не может нарушить эту границу.

Изменения членства в группах требуют согласования каждым процессом одного и того же списка участников. Такое соглашение может быть достигнуто с помощью протокола фиксации, из которых наиболее широко применяется протокол двухфазной фиксации. В протоколе двухфазной фиксации координатор сначала проверяет, согласны ли все процессы выполнить одну и ту же операцию (т. е. согласны ли они все на фиксацию), и многоадресная передача результатов этого опроса происходит во втором раунде. Протокол

трехфазной фиксации используется для обработки сбоя координатора без необходимости блокировать все процессы для достижения соглашения, пока координатор не восстановится.

Восстановление в отказоустойчивых системах достигается путем неизменно регулярного определения состояния системы. Контрольная точка полностью распределена. К сожалению, получение контрольной точки – дорогостоящая операция. Для повышения производительности многие распределенные системы сочетают контрольные точки с ведением журнала сообщений. Регистрация связи между процессами обеспечивает возможность воспроизвести работу системы после сбоя.

Глава 9

Безопасность

Последний принцип распределенных систем, который мы обсуждаем, – это безопасность. Безопасность ни в коем случае не немаловажный принцип. Однако можно утверждать, что это один из самых сложных принципов, поскольку безопасность должна распространяться на всю систему. Единственный дефект разработки в отношении безопасности может сделать все меры безопасности бесполезными. В этой главе мы сосредоточимся на различных механизмах, которые обычно включены в распределенные системы для поддержки безопасности.

Начнем с представления основных вопросов безопасности. Встраивание всех видов механизмов безопасности в систему на самом деле не имеет смысла, если неизвестно, как эти механизмы использовать и против чего. Это требует, чтобы нам была известна политика безопасности, которая должна выполняться. Понятие политики безопасности наряду с некоторыми общими проблемами проектирования механизмов, которые помогают применять такие политики, обсуждается в первую очередь. Мы также кратко коснемся необходимой криптографии.

Безопасность в распределенных системах можно условно разделить на две части. Одна часть касается связи между пользователями или процессами, возможно, на разных машинах. Основным механизмом обеспечения безопасной связи является механизм безопасного канала. Защищенные каналы, а точнее аутентификация, целостность сообщений и конфиденциальность, обсуждаются в отдельном разделе.

Другая часть – авторизация, которая состоит в обеспечении получения процессом только тех прав доступа к ресурсам в распределенной системе, на которую он имеет право. Авторизация рассматривается в отдельном разделе, посвященном контролю доступа. В дополнение к традиционным механизмам контроля доступа мы также фокусируемся на контроле доступа, когда нам приходится иметь дело с мобильным кодом, таким как агенты.

Мы также вернемся к наименованию и обратим внимание на довольно неприятную проблему, заключающуюся в том, что имя, используемое для получения объекта, принадлежит этому объекту, а также в том, как объединить безопасное наименование с понятными для человека именами.

Безопасные каналы и контроль доступа требуют механизмов для распространения криптографических ключей, а также механизмов добавления и удаления пользователей из системы. Эти темы охватывают аспекты управления безопасностью. В отдельном разделе мы обсуждаем вопросы, касаю-

щиеся управления криптографическими ключами, безопасного управления группами и выдачи сертификатов, подтверждающих право владельца на доступ к указанным ресурсам.

9.1. Введение

Мы начнем наше описание безопасности в распределенных системах с рассмотрения некоторых общих проблем безопасности. Во-первых, необходимо определить, что такое защищенная система. Мы отличаем политики безопасности от механизмов безопасности. Наша вторая задача – рассмотреть некоторые общие вопросы проектирования защищенных систем. Наконец, мы кратко обсудим некоторые криптографические алгоритмы, которые играют ключевую роль в разработке протоколов безопасности.

Угрозы безопасности, политики и механизмы

Безопасность в компьютерной системе тесно связана с понятием надежности. Неформально надежная компьютерная система – это система, которой мы по праву доверяем в предоставлении услуг [Laprie, 1995]. Надежность включает доступность, безопасность и ремонтопригодность. Однако если мы хотим доверять компьютерной системе, следует также учитывать конфиденциальность и целостность. **Конфиденциальность** (confidentiality) относится к свойству компьютерной системы раскрывать информацию только уполномоченным сторонам. **Целостность** (integrity) – это характеристика тех изменений в ресурсах системы, которые могут быть сделаны только авторизованным способом. Другими словами, неправильные изменения в защищенной компьютерной системе должны быть обнаружимы и исправимы. Основными активами любой компьютерной системы являются ее оборудование, программное обеспечение и данные.

Еще один подход к безопасности в компьютерных системах заключается в защите предлагаемых им сервисов и данных от угроз безопасности. Следует учитывать четыре типа существующих **угроз безопасности** (security threats) [Peeger, 2003]:

1) перехват (Interception);
2) прерывание (Interuption);
3) модификация (Modification);
4) фабрикация (Fabrication).

Концепция перехвата относится к ситуации, когда неавторизованная сторона получила доступ к услуге или данным. Типичным примером перехвата является случай, когда связь между двумя сторонами была подслушана кем-то другим. Перехват также происходит, когда данные незаконно копируются, например после взлома личного каталога человека в файловой системе.

Примером прерывания является случай, когда файл поврежден или потерян. В более общем смысле прерывание относится к ситуации, в которой услуги или данные становятся недоступными, непригодными, уничтожен-

ными и т. д. В этом смысле атаки типа «отказ в обслуживании», с помощью которых кто-либо злонамеренно пытается сделать услугу недоступной для других сторон, представляют собой угрозу безопасности, которая классифицируется как прерывание.

Модификации включают несанкционированное изменение данных или подделку службы, чтобы она больше не придерживалась своих первоначальных спецификаций. Примеры модификаций включают перехват и последующее изменение переданных данных, подделку записей базы данных и изменение программы, чтобы она тайно регистрировала действия своего пользователя.

Под фабрикацией понимается ситуация, в которой генерируются дополнительные данные или действия, которые обычно не существуют. Например, злоумышленник может попытаться добавить запись в файл пароля или базу данных. Кроме того, иногда можно взломать систему путем воспроизведения ранее отправленных сообщений. Мы встретимся с такими примерами позже в этой главе.

Обратите внимание, что прерывание, модификация и фабрикация могут рассматриваться как форма фальсификации данных.

Просто заявить, что система должна быть способна защитить себя от всех возможных угроз безопасности, – это неприемлемый способ создать защищенную систему. Сначала необходимо дать описание требований безопасности, то есть политики безопасности. **Политика безопасности** (security policy) точно описывает, какие действия разрешено выполнять объектам в системе, а какие запрещено. Объекты включают пользователей, сервисы, данные, машины и т. д. После того как политика безопасности была установлена, становится возможным сосредоточиться на механизмах безопасности, с помощью которых можно применять политику. Важными механизмами безопасности являются:

1) шифрование (Encryption);
2) аутентификация (Autentication);
3) авторизация (Authorization);
4) аудит (Auditing).

Шифрование имеет основополагающее значение для компьютерной безопасности. Оно превращает данные в то, что злоумышленник не может понять. Другими словами, шифрование обеспечивает средства для обеспечения конфиденциальности данных. Кроме того, шифрование позволяет нам проверять, были ли данные изменены. Таким образом, оно также обеспечивает поддержку проверок целостности.

Аутентификация используется для проверки заявленной личности пользователя, клиента, сервера, хоста или другого объекта. В случае клиентов основная предпосылка заключается в том, что перед тем, как служба начинает выполнять какую-либо работу от имени клиента, служба должна узнать личность клиента (если служба не доступна для всех). Как правило, пользователи проходят проверку подлинности с помощью паролей, но есть много других способов проверки подлинности клиентов.

После того как клиент был аутентифицирован, необходимо проверить, авторизован ли этот клиент для выполнения запрошенного действия; ти-

пичным примером является доступ к записям в медицинской базе данных. В зависимости от того, кто имеет доступ к базе данных, может быть предоставлено разрешение на чтение записей, изменение определенных полей в записи, а также на добавление или удаление записи.

Инструменты аудита используются для отслеживания того, какие клиенты получили доступ к чему и каким образом. Хотя аудит на самом деле не обеспечивает никакой защиты от угроз безопасности, журналы аудита могут быть чрезвычайно полезны для анализа взлома системы безопасности и последующего принятия мер против злоумышленников. По этой причине злоумышленники, как правило, стремятся не оставлять никаких следов, которые могли бы в конечном итоге привести к раскрытию их личности. В этом смысле регистрация доступа иногда делает атаку рискованной.

Проблемы дизайна

Распределенная система или любая компьютерная система в этом отношении должна предоставлять услуги безопасности, с помощью которых может быть реализован широкий спектр политик безопасности. Существует ряд важных проблем проектирования, которые необходимо учитывать при внедрении служб безопасности общего назначения. На следующих страницах мы обсуждаем три таких вопроса: сфокусированность управления, расслоение механизмов безопасности и простоту (см. также [Gollmann, 2006]).

Контроль

При рассмотрении защиты (возможно, распределенного) приложения можно, по существу, следовать трем различным подходам, как показано на рис. 9.1. Первый подход заключается в том, чтобы сосредоточиться непосредственно на защите данных, связанных с приложением. Здесь подразумевается, что независимо от различных операций, которые могут быть выполнены с элементом данных, первостепенной задачей является обеспечение целостности данных. Как правило, этот тип защиты встречается в системах баз данных, в которых могут быть сформулированы различные ограничения целостности, которые автоматически проверяются каждый раз при изменении элемента данных (см., например, [Doorn and Rivero, 2002]).

Второй подход заключается в том, чтобы сконцентрироваться на защите, указав, какие именно операции могут быть вызваны и кем, когда должны быть доступны определенные данные или ресурсы. В этом случае фокус контроля тесно связан с механизмами контроля доступа, которые мы подробно обсудим позже в этой главе. Например, в объектно-ориентированной системе может быть решено указать для каждого метода, который предоставляется клиентам, какие клиенты могут вызывать этот метод. Альтернативно методы контроля доступа могут применяться ко всему интерфейсу, предлагаемому объектом, или ко всему самому объекту. Таким образом, этот подход допускает различные детализации контроля доступа.

Третий подход заключается в том, чтобы сосредоточиться непосредственно на пользователях путем принятия мер, с помощью которых только опре-

деленные люди имеют доступ к приложению, независимо от операций, которые они хотят выполнять. Например, база данных в банке может быть защищена путем отказа в доступе кому-либо, кроме высшего руководства банка и специально уполномоченных лиц.

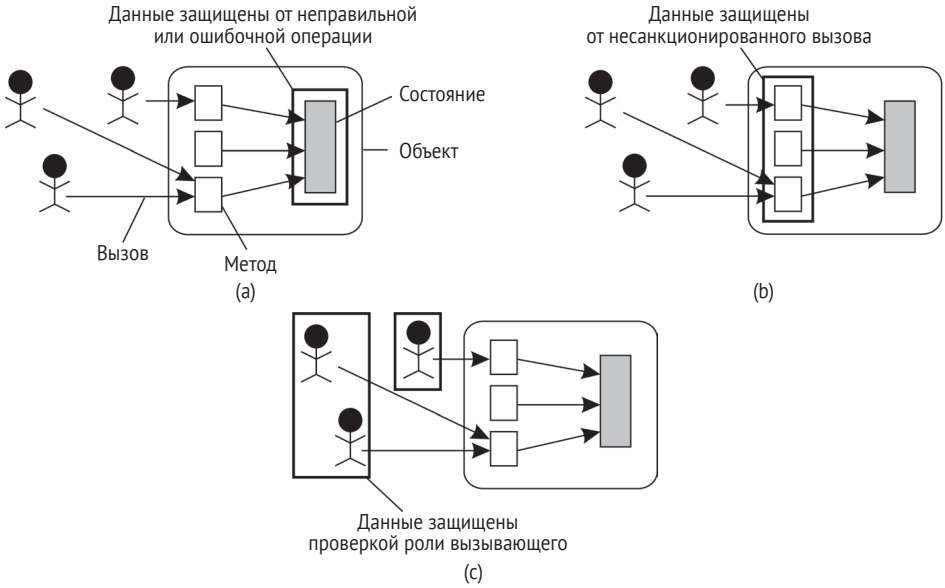


Рис. 9.1 ❖ Три подхода к защите от угроз безопасности:
a) защита от недействительных операций; b) защита от несанкционированного доступа;
c) защита от неавторизованных пользователей

В качестве другого примера во многих университетах определенные данные и приложения разрешается использовать только преподавателям и сотрудникам, в то время как доступ студентам запрещен. По сути, контроль направлен на определение ролей, которые имеют пользователи, и после проверки роли пользователя доступ к ресурсу предоставляется или запрещается. Таким образом, в рамках разработки защищенной системы необходимо определить роли, которые могут иметь пользователи, и предоставить механизмы для поддержки контроля доступа на основе ролей. Мы вернемся к ролям позже в этой главе.

Уровни механизмов безопасности

Важной проблемой при разработке безопасных систем является решение, на каком уровне следует размещать механизмы безопасности. Уровень в этом контексте связан с логической организацией системы. Например, компьютерные сети часто организованы как уровни, следуя некоторой эталонной модели, как мы обсуждали в главе 4. В главе 1 мы представили организацию распределенных систем, состоящих из отдельных уровней для приложений, промежуточного программного обеспечения, служб операционной системы

и ядра операционной системы. Объединение двух организаций приводит примерно к тому, что показано на рис. 9.2.

По сути, на рис. 9.2 услуги общего назначения отделяются от услуг связи.

Рис. 9.2 ❖ Логическая организация распределенной системы в несколько уровней

Это разделение важно для понимания уровня безопасности в распределенных системах и, в частности, понятия доверия. Разница между доверием и безопасностью важна. Система либо безопасна, либо нет (принимая во внимание различные вероятностные меры), но вопрос о том, считает ли клиент систему безопасной, является вопросом доверия. Безопасность – технический аспект; доверие – это эмоциональное восприятие [Bishop, 2003]. На каком уровне находятся механизмы безопасности, зависит от доверия клиента к тому, насколько безопасны услуги на определенном уровне. В качестве примера рассмотрим организацию, расположенную в разных узлах, которые подключены через низкоуровневую магистраль, соединяющую различные локальные сети в географически разнесенных узлах, как показано на рис. 9.3. Подобные соединения могут быть настроены с использованием таких методов, как **многосторонний протокол коммутации по меткам** (Multiprotocol Label Switching, MPLS) или методы для **виртуальных частных сетей** (Virtual Private Networks, VPN).

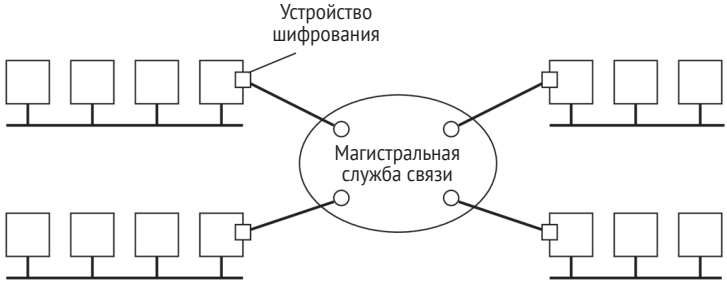


Рис. 9.3 ❖ Несколько сайтов, подключенных через глобальную магистральную службу

Безопасность может быть обеспечена путем размещения устройств шифрования на каждом магистральном коммутаторе, как показано на рис. 9.3. Эти устройства автоматически шифруют и дешифруют пакеты, отправляемые между сайтами, но в противном случае не обеспечивают безопасную связь между хостами на одном сайте. Если Алиса на сайте A отправляет сообщение Бобу на сайт B и беспокоится о том, что ее сообщение будет перехвачено, она должна, по крайней мере, доверять шифрованию межсайтового трафика для правильной работы. Это означает, например, что она должна доверять системному администратору и что на обоих сайтах приняты надлежащие меры против подделки устройств.

Теперь предположим, что Алиса не доверяет безопасности межсайтового трафика. Она может решить принять свои собственные меры, применяя, например, **службу безопасности транспортного уровня** (Transport Layer Security, TLS), которую можно использовать для безопасной отправки сообщений по TCP-соединению. Здесь важно отметить, что TLS позволяет Алисе установить безопасное соединение с Бобом. Все сообщения транспортного уровня будут зашифрованы – и в нашем примере также на уровне канала, но это не касается Алисы. В этом случае Алисе придется довериться TLS. Другими словами, она считает, что TLS является безопасным.

В распределенных системах механизмы безопасности часто размещаются на уровне промежуточного программного обеспечения. Если Алиса не доверяет TLS, она может использовать локальную безопасную службу вызова удаленных процедур (Remote Procedure Call, RPC). Опять же, ей придется доверять этой службе RPC, чтобы делать то, что она обещает, например не передавать информацию или должным образом проверять подлинность клиентов и серверов.

Службам безопасности, размещенным на уровне промежуточного программного обеспечения распределенной системы, можно доверять, только если службы, на которые они полагаются, являются действительно безопасными. Например, если защищенная служба RPC частично реализована посредством TLS, то доверие к службе RPC зависит от того, насколько доверяют TLS. Если TLS не является доверенной службой, то не может быть доверия и к безопасности службы RPC.

Распределение механизмов безопасности

Зависимости между службами в отношении доверия приводят к понятию **достоверная вычислительная база** (Trusted Computing Base, TCB). TCB – это набор всех механизмов безопасности в (распределенной) компьютерной системе, которые необходимы для реализации политики безопасности и которым, таким образом, нужно доверять. Чем меньше TCB, тем лучше. Если распределенная система построена как промежуточное программное обеспечение в существующей сетевой операционной системе, ее безопасность может зависеть от безопасности базовых локальных операционных систем. Другими словами, TCB в распределенной системе может включать в себя локальные операционные системы на различных хостах.

Рассмотрим файловый сервер в распределенной файловой системе. Такой сервер может нуждаться в различных механизмах защиты, предлагаемых его локальной операционной системой. Подобные механизмы включают в себя не только механизмы защиты файлов от доступа к процессам, отличным от файлового сервера, но и механизмы для защиты файлового сервера от злонамеренного сбоя.

Таким образом, распределенные системы на базе промежуточного программного обеспечения требуют доверия к существующим локальным операционным системам, от которых они зависят. Если такого доверия нет, то часть функциональности локальных операционных систем может потребоваться включить в саму распределенную систему. Рассмотрим микроядерную операционную систему, в которой большинство служб операционной системы работают как обычные пользовательские процессы. В этом случае, например, файловая система может быть полностью заменена другой с конкретными потребностями распределенной системы, включая различные меры безопасности.

В соответствии с этим подходом необходимо отделить службы безопасности от других типов служб путем распределения служб на разных компьютерах в зависимости от требуемого уровня безопасности. Например, для защищенной распределенной файловой системы может быть возможно изолировать файловый сервер от клиентов, разместив сервер на машине с доверенной операционной системой, возможно, с выделенной защищенной файловой системой. Клиенты и их приложения могут быть размещены на ненадежных машинах.

Такое разделение эффективно сокращает TCB до относительно небольшого количества машин и программных компонентов. За счет последующей защиты этих машин от атак безопасности извне можно повысить общее доверие к безопасности распределенной системы.

Простота

Другая важная проблема проектирования, связанная с принятием решения о том, на каком уровне размещать механизмы безопасности, состоит в простоте. Проектирование защищенной компьютерной системы обычно считается сложной задачей. Следовательно, если разработчик системы сможет использовать несколько простых механизмов, которые легко понять и которым доверяют, тем лучше.

К сожалению, простых механизмов не всегда достаточно для реализации политик безопасности. Рассмотрим еще раз ситуацию, в которой Алиса хочет отправить сообщение Бобу, как обсуждалось выше. Шифрование на канальном уровне – это простой и понятный механизм защиты от перехвата межсайтового трафика сообщений. Однако нужно гораздо больше, если Алиса хочет быть уверенной, что только Боб получит ее сообщения. В этом случае требуются сервисы аутентификации на уровне пользователя, и Алисе, возможно, нужно знать, как работают эти сервисы, чтобы доверять им. Поэтому для аутентификации на уровне пользователя может потребоваться хотя бы представление о криптографических ключах и знание таких механизмов, как сертификаты, несмотря на тот факт, что многие службы безопасности высокоавтоматизированы и скрыты от пользователей.

В других случаях само приложение является по своей сути сложным, а внедрение безопасности только усугубляет ситуацию. Примером области применения, включающей сложные протоколы безопасности, является область цифровых платежных систем. Сложность протоколов цифровых платежей часто обусловлена тем фактом, что для осуществления платежа необходимо общаться нескольким сторонам. В этих случаях важно, чтобы базовые механизмы, используемые для реализации протоколов, были относительно просты и понятны. Простота будет способствовать доверию, которое конечные пользователи окажут приложению, и, что более важно, убедит разработчиков в том, что в системе нет дыр в безопасности.

Примечание 9.1 (дополнительно: включение защиты в методологию разработки)

Поскольку (распределенная) система безопасности является сложной задачей, может быть целесообразно не просто утверждать, что безопасность должна быть включена на начальных этапах разработки, но фактически поддерживать безопасное проектирование методологическим способом. Существует несколько подходов, чтобы сделать именно это, как описано в [Uzunov et al., 2012]. Авторы различают методологию разработки программного обеспечения и жизненного цикла на основе кода и модели.

В первом случае основная идея заключается в обеспечении различных функций безопасности без учета общего дизайна или архитектуры системы. Простым примером всегда является обеспечение безопасного сетевого взаимодействия, независимо от того, всегда ли такая безопасность необходима. Аналогичным образом может потребоваться убедиться, что импортированный код всегда выполняется на (защищенной) виртуальной машине или что буферы никогда не будут переполняться. Однако методологии, основанные на коде, также включают такие аспекты, как моделирование угроз и тестирование на проникновение.

Хотя методологии, основанные на коде, как представляется, применяются более широко, методологии, основанные на моделях, считаются более эффективными. Дело в том, что основанные на модели подходы учитывают весь дизайн или архитектуру распределенной системы. Как правило, они интегрированы в процесс проектирования путем улучшения языков моделирования, таких как UML. В целом идея заключается в том, что различные функции безопасности делаются явными на ранних этапах разработки требований и проектирования. К сожалению, это имеет смысл, только если можно дать некоторую гарантию того, что моделируемое также и (правильно) реализовано. На практике это оказывается препятствием, если нет автоматизированных средств перехода от модели к реализации.

В итоге в [Uzunov et al., 2012] сделан вывод, что для успешной разработки системы безопасности в распределенных системах еще предстоит пройти долгий путь. Их заключение иллюстрирует трудности в систематическом обеспечении безопасности систем. Такова жизнь.

Криптография

Основой безопасности в распределенных системах является использование криптографических методов. Основная идея применения этих методов проста. Рассмотрим отправителя S, который хочет передать сообщение m

получателю R. Чтобы защитить сообщение от угроз безопасности, отправитель сначала **зашифровывает** (encrypts) его в неразборчивое сообщение m′, а затем отправляет m′ в R. R, в свою очередь, должен расшифровать (decrypt) полученное сообщение в первоначальном виде m.

Шифрование и дешифрование осуществляются с помощью криптографических методов, параметризованных ключами, как показано на рис. 9.4. Исходная форма отправляемого сообщения называется **открытым текстом** (plaintext), обозначенным буквой P на рис. 9.4. Зашифрованная форма называется **зашифрованным текстом** (ciphertext), обозначенным буквой C.

Для описания различных протоколов безопасности, которые используются при создании служб безопасности для распределенных систем, полезно иметь нотацию для связи простого текста, зашифрованного текста и ключей. Следуя общепринятым условным обозначениям, мы будем использовать $C = E_K(P)$ для обозначения того, что зашифрованный текст C получается путем шифрования открытого текста P с использованием ключа K. Аналогично $P = D_K(C)$ используется для выражения дешифрования зашифрованного текста C с использованием ключа K, в результате чего появляется открытый текст P.

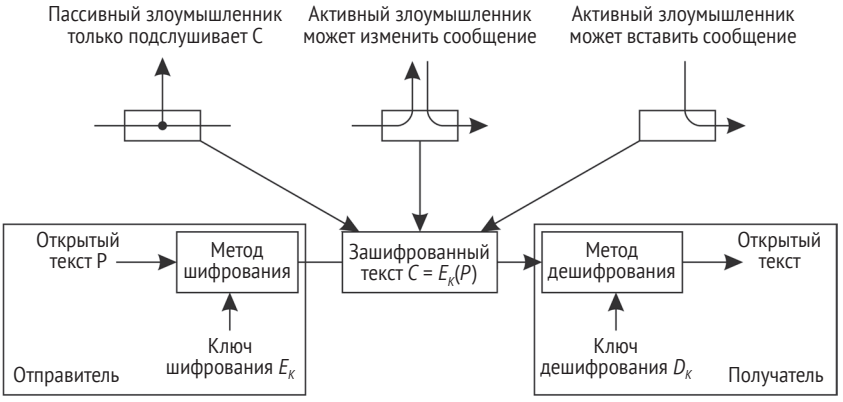


Рис. 9.4 ❖ Злоумышленники и перехватчики сообщения

Возвращаясь к нашему примеру, показанному на рис. 9.4, при передаче сообщения в виде зашифрованного текста C есть три различные атаки, от которых мы должны защищаться и для которых помогает шифрование. Во-первых, **злоумышленник** (intruder) может перехватить сообщение, и отправитель или получатель не будут осведомлены о том, что происходит прослушивание. Конечно, если переданное сообщение было зашифровано таким образом, что его невозможно легко расшифровать, не имея подходящего ключа, перехват бесполезен: злоумышленник увидит только неразборчивые данные. (Между прочим, одного факта, что сообщение передается, иногда может быть достаточно, чтобы злоумышленник мог сделать выводы. Например, если во время мирового кризиса количество трафика в Белый дом

внезапно резко падает, в то время как количество трафика в определенный горный курорт Колорадо увеличивается на ту же величину, может быть полезна информация.)

Второй тип атаки, с которой нужно иметь дело, – это модификация сообщения. Изменить открытый текст легко; модифицировать зашифрованный текст, который был должным образом зашифрован, намного сложнее, потому что злоумышленнику сначала придется расшифровать сообщение, прежде чем он сможет существенно его изменить. Кроме того, он также должен будет правильно зашифровать его, иначе получатель может заметить, что сообщение было подделано.

Третий тип атаки – это когда злоумышленник вставляет зашифрованные сообщения в систему связи, пытаясь убедить R в том, что эти сообщения получены от S. Опять же, шифрование может помочь защитить от таких атак. Обратите внимание, что если нарушитель может изменять сообщения, он также может вставлять сообщения.

Существует фундаментальное различие между различными криптографическими системами в зависимости от того, одинаковы или нет ключи шифрования и дешифрования. В симметричной криптосистеме один и тот же ключ используется для шифрования и дешифрования сообщения:

$$P = D_K(E_K(P)).$$

В **симметричных криптосистемах** (symmetric cryptosystem) используется секретный (общий) ключ. И отправитель, и получатель совместно применяют один и тот же ключ, и для обеспечения работы защиты этот общий ключ должен храниться в секрете; никто больше не должен знать этот ключ. Мы будем использовать обозначения $K_{A,B}$ для обозначения ключа, общего для A и B.

В **асимметричной криптосистеме** (asymmetric cryptosystem) ключи шифрования и дешифрования различны, но вместе образуют уникальную пару. Другими словами, существует отдельный ключ K_E для шифрования и один ключ для дешифрования, K_D, такой, что

$$P = D_{K_D}(E_{K_E}(P)).$$

Один из ключей в асимметричной криптосистеме хранится в секрете; другой является открытым. По этой причине асимметричные криптосистемы также называют **системами с открытым ключом** (public-key systems). В дальнейшем мы будем использовать K_A^+ для обозначения открытого ключа, принадлежащего A, и K_A^- в качестве его соответствующего закрытого ключа.

Какой из ключей шифрования или дешифрования, который фактически известен, нужен, зависит от того, как эти ключи используются. Например, если Алиса хочет отправить конфиденциальное сообщение Бобу, она должна использовать открытый ключ Боба для шифрования сообщения. Поскольку Боб – единственный, кто имеет связанный и закрытый ключ дешифрования, он также является единственным человеком, который может расшифровать сообщение.

С другой стороны, предположим, что Боб хочет знать наверняка, что сообщение, которое он только что получил, на самом деле пришло от Алисы. В этом случае Алиса может сохранить свой ключ шифрования закрытым для шифрования отправляемых ею сообщений. Если Боб может успешно дешифровать сообщение с помощью открытого ключа Алисы (а открытый текст в сообщении имеет достаточно информации, чтобы сделать его значимым для Боба), он знает, что сообщение должно быть получено от Алисы, поскольку ключ дешифрования однозначно связан с ключом шифрования.

Одно из последних применений криптографии в распределенных системах – использование **хеш-функций** (hash function). Хеш-функция H принимает сообщение m произвольной длины в качестве входных данных и создает битовую строку h, имеющую фиксированную длину, в качестве выходных данных:

$$h = H(m).$$

Хеш h в некоторой степени сопоставим с дополнительными битами, которые добавляются к сообщению в системах связи для обеспечения обнаружения ошибок, таких как проверка циклическим избыточным кодом (cyclic-redundancy check, CRC).

Хеш-функции, которые используются в криптографических системах, обладают рядом существенных свойств. Во-первых, они являются **односторонними функциями** (one-way functions), что означает, что в вычислительном отношении невозможно найти вход m, который соответствует известному выходу h. С другой стороны, вычислить h из m легко. Во-вторых, они имеют свойство **слабого конфликтного сопротивления** (weak collision resistance), означающее, что, учитывая вход m и связанный с ним выход $h = H(m)$, в вычислительном отношении невозможно найти другой, отличный вход $m' \neq m$, такой, что $H(m) = H(m')$. Наконец, криптографические хеш-функции также обладают сильным свойством конфликтного сопротивления, что означает, что если дан только H, в вычислительном отношении невозможно найти любые два различных входных значения m и m′, так что $H(m) = H(m')$.

Аналогичные свойства должны применяться к любой функции шифрования E и используемым ключам. Кроме того, для любой функции шифрования E_K в вычислительном отношении невозможно найти ключ K при известном открытом тексте P и связанном зашифрованном тексте $C = E_K(P)$. По аналогии с конфликтным сопротивлением, когда дан открытый текст P и ключ K, фактически невозможно найти другой ключ K′ – такой, что $E_K(P) = E_{K'}(P)$.

Искусство и наука разработки алгоритмов для криптографических систем имеют долгую и увлекательную историю [Kahn, 1967], и создание безопасных систем часто удивительно сложно или даже невозможно [Schneier, 2000]. Подробное обсуждение любого из этих алгоритмов выходит за рамки данной книги. Информацию о криптографических алгоритмах можно найти в [Ferguson et al., 2010], [Menezes et al., 1996] и [Schneier, 1996]. Рисунок 9.5 суммирует обозначения и сокращения, которые мы используем в математических выражениях для всей этой книги.

Обозначение	Описание
$K_{A,B}$	Секретный ключ, общий для A и B
K_A^+	Открытый ключ A
K_A^-	Закрытый ключ A
$K(d)$	Некоторые данные d, зашифрованные ключом K

Рис. 9.5 ❖ Обозначения, используемые в этой главе

9.2. Безопасные каналы

Модель клиент-сервер – это удобный способ организации распределенной системы.

В этой модели серверы могут быть распределены и реплицированы, но также могут выступать в качестве клиентов по отношению к другим серверам. Рассматривая вопросы безопасности в распределенных системах, еще раз полезно вспомнить о клиентах и серверах. В частности, обеспечение безопасности распределенной системы сводится к двум основным проблемам. Первая проблема заключается в том, как обеспечить безопасность связи между клиентами и серверами. Безопасная связь требует аутентификации взаимодействующих сторон. Во многих случаях это также требует обеспечения целостности сообщения и, возможно, конфиденциальности. Как часть этой проблемы мы также должны рассмотреть возможность защиты связи в группе серверов.

Вторая проблема связана с авторизацией: как только сервер принял запрос от клиента, как он может узнать, авторизован этот клиент для выполнения этого запроса или нет? Авторизация связана с проблемой контроля доступа к ресурсам.

Вопрос о защите связи между клиентами и серверами можно рассматривать с точки зрения создания безопасного канала связи между взаимодействующими сторонами [Voydock and Kent, 1983]. Безопасный канал защищает отправителей и получателей от перехвата, модификации и изготовления сообщений. Однако не обязательно защищает от прерывания. Защита сообщений от перехвата обеспечивается за счет конфиденциальности: безопасный канал гарантирует, что его сообщения не могут быть подслушаны злоумышленниками. Защита от модификации и фабрикации злоумышленниками осуществляется посредством протоколов для взаимной аутентификации и целостности сообщений. Подробное и формальное описание логики, лежащей в основе аутентификации, можно найти в [Lampson et al., 1992].

Аутентификация

Прежде чем углубляться в детали различных протоколов аутентификации, стоит отметить, что аутентификация и целостность сообщений не могут обойтись друг без друга. Рассмотрим, например, распределенную систему, которая поддерживает аутентификацию двух взаимодействующих сторон,

но не предоставляет механизмов для обеспечения целостности сообщений. В такой системе Боб может точно знать, что Алиса является отправителем сообщения m. Однако если Боб не имеет гарантии того, что m не было изменено во время передачи, какая ему польза лишь от знания, что это Алиса отправила (оригинальную версию) m?

Аналогично предположим, что поддерживается только целостность сообщения, но не существует механизмов аутентификации. Когда Боб получает сообщение о том, что он только что выиграл $1 000 000 в лотерее, как он может быть счастлив, если не может проверить, что сообщение было отправлено организаторами этой лотереи?

Следовательно, аутентификация и целостность сообщения должны сочетаться. Во многих протоколах комбинация работает примерно следующим образом. Снова предположим, что Алиса и Боб хотят общаться и что Алиса берет на себя инициативу в настройке канала. Алиса начинает с отправки сообщения Бобу или иным способом доверенной третьей стороне, которая поможет настроить канал. Как только канал настроен, Алиса точно знает, что разговаривает с Бобом, а Боб точно знает, что он разговаривает с Алисой, они могут обмениваться сообщениями.

Чтобы впоследствии обеспечить целостность сообщений данных, которыми обмениваются после аутентификации, обычной практикой является использование криптографии с секретным ключом посредством сеансовых ключей. **Сеансовый ключ** (session key) – это общий (секретный) ключ, который используется для шифрования сообщений на предмет целостности и, возможно, также конфиденциальности. Такой ключ обычно используется только до тех пор, пока существует канал. Когда канал закрыт, связанный с ним сеансовый ключ уничтожается.

Аутентификация на основе общего секретного ключа

Давайте начнем с рассмотрения протокола аутентификации на основе общего секретного ключа, который уже используется Алисой и Бобом. Как им двоим на самом деле удалось получить общий ключ безопасным способом, пока не важно. В описании протокола Алиса и Боб обозначаются аббревиатурой A и B соответственно, а их общий ключ обозначается как $K_{A,B}$. Протокол использует общий подход, при котором одна сторона запрашивает у другой ответ, который может быть правильным, только если другая знает общий секретный ключ. Такие решения также известны как **протоколы «вызов-ответ»** (challenge-response protocols).

В случае аутентификации на основе общего секретного ключа протокол выполняется, как показано на рис. 9.6. Сначала Алиса отправляет свою идентификацию Бобу (сообщение 1), указывая, что она хочет установить канал связи между ними. Боб соответственно отправляет **вызов** (challenge) R_B Алисе, показанный на рис. 9.6, как сообщение 2.

Такой вызов может принимать форму случайного числа. Алиса должна зашифровать запрос с помощью секретного ключа $K_{A,B}$, которым она делится с Бобом, и вернуть зашифрованный вызов Бобу. Этот ответ показан как сообщение 3 на рис. 9.6, содержащее $K_{A,B}(R_B)$.

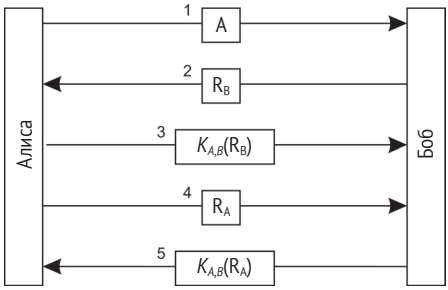


Рис. 9.6 ❖ Аутентификация на основе общего секретного ключа

Когда Боб получает ответ $K_{A,B}(R_B)$ на свой вызов R_B, он может снова расшифровать сообщение, используя общий ключ, чтобы посмотреть, содержит ли оно R_B. Если это так, то он знает, что Алиса находится на другой стороне, потому что никто больше не мог зашифровать R_B с $K_{A,B}$. Другими словами, Боб теперь убедился, что он действительно разговаривает с Алисой. Однако обратите внимание, что Алиса еще не подтвердила, что это действительно Боб с другой стороны канала. Поэтому она отправляет вызов R_A (сообщение 4), на который Боб отвечает, возвращая $K_{A,B}(R_A)$, показанный как сообщение 5. Когда Алиса расшифровывает его с помощью $K_{A,B}$ и видит свой R_A, она знает, что разговаривает с Бобом.

Примечание 9.2 (дополнительно: о разработке протоколов безопасности)

Одной из сложных проблем безопасности является разработка работающих протоколов. Чтобы проиллюстрировать, насколько легко все может пойти не так, рассмотрим «оптимизацию» протокола аутентификации, при котором количество сообщений было уменьшено с пяти до трех, как показано на рис. 9.7. Основная идея заключается в том, что если Алиса в конце концов захочет направить Бобу вызов, она может также отправить вызов вместе со своей идентификацией при настройке канала. Точно так же Боб возвращает свой ответ на этот вызов вместе со своим собственным вызовом в одном сообщении.

К сожалению, такой протокол больше не работает. Это может быть легко раскрыто тем, что известно как **атака отражения** (reflection attack). Чтобы объяснить, как работает такая атака, рассмотрим злоумышленника по имени Чак, которого мы обозначаем как C в наших протоколах. Цель Чака – установить канал с Бобом, чтобы Боб поверил, что разговаривает с Алисой. Чак может установить это, если он правильно ответит на вызов, отправленный Бобом, например вернув зашифрованную версию номера, отправленного Бобом. Без знания $K_{A,B}$ только Боб может сделать такое шифрование, и это именно тот трюк, с помощью которого Чак обманывает Боба.

Атака проиллюстрирована на рис. 9.8. Чак начинает с отправки сообщения, содержащего идентификацию Алисы A, а также вызов R_C. Боб возвращает свой вызов R_B и ответ $K_{A,B}(R_C)$ в одном сообщении. В этот момент Чаку нужно будет доказать, что он знает секретный ключ, вернув $K_{A,B}(R_B)$ Бобу. К сожалению, у него нет $K_{A,B}$. Вместо этого он пытается настроить второй канал, чтобы позволить Бобу выполнить шифрование для него.

Поэтому Чак отправляет A и R_B в одном сообщении, как и раньше, но теперь делает вид, что хочет второй канал. Это показано как сообщение 3 на рис. 9.8. Боб, не зная,

что он сам использовал R_B раньше как вызов, отвечает $K_{A,B}(R_B)$ и другим вызовом R_{B2}, показанным как сообщение 4. В этот момент Чак имеет $K_{A,B}(R_B)$ и завершает настройку первого сеанса, возвращая сообщение 5, содержащее ответ $K_{A,B}(R_B)$, который первоначально был запрошен из запроса, отправленного в сообщении 2.

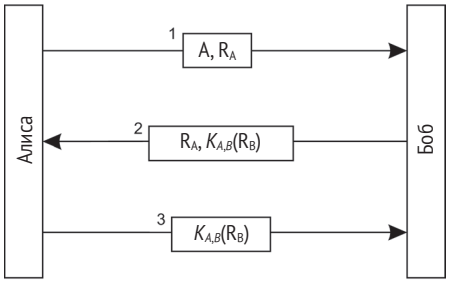


Рис. 9.7 ❖ Аутентификация на основе общего секретного ключа, но с использованием трех вместо пяти сообщений

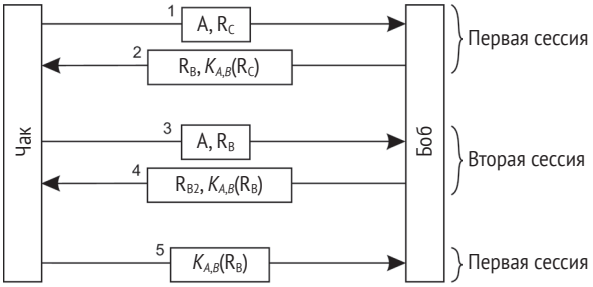


Рис. 9.8 ❖ Атака отражения

Как объяснено в [Kaufman et al., 2003], одна из ошибок, допущенных во время адаптации первоначального протокола, заключалась в том, что обе стороны в новой версии протокола использовали один и тот же вызов в двух разных прогонах протокола.

Лучшее решение – всегда применять разные вызовы для инициатора и респондента. Например, если Алиса всегда использует нечетное число, а Боб – четное число, Боб узнал бы, что происходит что-то подозрительное при получении R_B в сообщении 3 на рис. 9.8. (К сожалению, это решение подвержено другим атакам, в частности известной как «человек посередине атаки», что объясняется в [Ferguson et al., 2010].) В общем, позволяя двум сторонам настройку безопасного канала, делать несколько вещей одинаково – не очень хорошая идея.

Еще один принцип, который нарушается в адаптированном протоколе, заключается в том, что Боб выдавал ценную информацию в форме ответа $K_{A,B}(R_C)$, не зная точно, кому он ее давал. Этот принцип не был нарушен в первоначальном протоколе, в котором Алисе сначала нужно было подтвердить свою личность, после чего Боб хотел передать свою зашифрованную информацию.

Есть много принципов, которые разработчики криптографических протоколов последовательно изучают на протяжении многих лет. Одним из важных уроков является то, что разработка протоколов безопасности, которые делают то, что они должны делать, часто намного сложнее, чем кажется. Кроме того, настройка существующего протокола для улучшения его производительности может легко повлиять на его корректность. Больше о принципах разработки протоколов можно найти в [Abadi and Needham, 1996].

Аутентификация с использованием центра распределения ключей

Одной из проблем использования общего секретного ключа для аутентификации является масштабируемость. Если распределенная система содержит N хостов и каждому хосту требуется совместно использовать секретный ключ с каждым из остальных $N - 1$ хостов, системе в целом необходимо управлять $N(N - 1)/2$ ключами, и каждый хост может управлять $N - 1$ ключами. Для больших N это приведет к проблемам. Альтернативой является использование централизованного подхода с помощью **Центра распространения ключей** (Key Distribution Center, KDC). Центр KDC разделяет секретный ключ с каждым из хостов, но не требуется, чтобы пара хостов имела общий секретный ключ. Другими словами, использование KDC требует, чтобы мы управляли N ключами вместо $N(N - 1)/2$, что явно является улучшением.

Если Алиса хочет установить безопасный канал с Бобом, она может сделать это с помощью (доверенного) KDC. Вся идея в том, что KDC раздает ключ Алисе и Бобу, который они могут использовать для общения, как показано на рис. 9.9.

Алиса сначала отправляет сообщение в KDC, указывая, что она хочет поговорить с Бобом. KDC возвращает сообщение, содержащее общий секретный ключ $K_{A,B}$, который она может использовать. Сообщение шифруется секретным ключом $K_{A,KDC}$, который является общим для Алисы и KDC. Кроме того, KDC отправляет $K_{A,B}$ также Бобу, но теперь шифруется секретным ключом $K_{B,KDC}$, который является общим для Боба и KDC.

Основным недостатком этого подхода является то, что Алиса может захотеть начать настройку безопасного канала с Бобом еще до того, как Боб получил общий ключ от KDC. Кроме того, KDC требуется заполучить Боба в цикле, передав ему ключ. Эти проблемы можно обойти, если KDC просто передает $K_{B,KDC}(K_{A,B})$ обратно Алисе и позволяет ей позаботиться о соединении с Бобом.

Это приводит к протоколу, показанному на рис. 9.10. Сообщение $K_{B,KDC}(K_{A,B})$ также известно как **билет** (ticket). Работа Алисы – передать этот билет Бобу. Обратите внимание, что Боб по-прежнему единственный, кто может разумно использовать билет, поскольку он единственный, кроме KDC, который знает, как расшифровать содержащуюся в нем информацию.

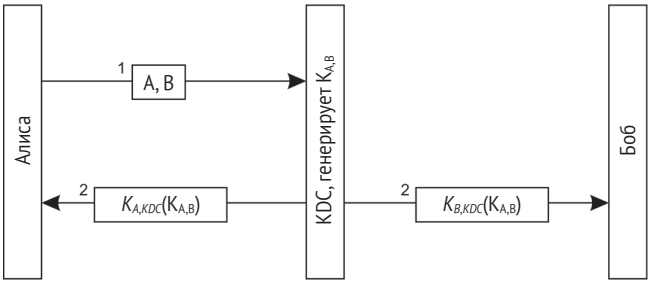


Рис. 9.9 ❖ Принцип использования Центра распределения ключей (KDC)

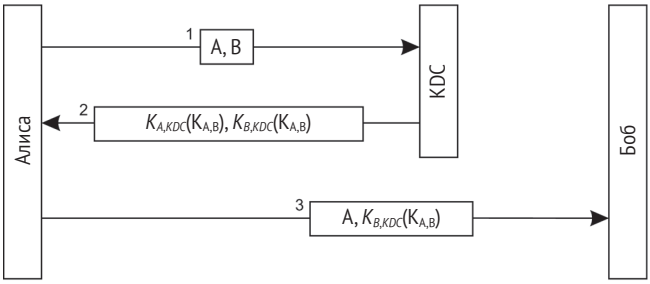


Рис. 9.10 ❖ Использование билета и разрешение Алисе установить соединение с Бобом

Примечание 9.3 (дополнительно: протокол Нидхэма–Шредера)

Протокол, показанный на рис. 9.10, на самом деле является вариантом хорошо известного примера протокола аутентификации с использованием KDC, известного как **протокол аутентификации Нидхэма–Шредера** (Needham-Schroeder), названный в честь его изобретателей [Needham and Schroeder, 1978].

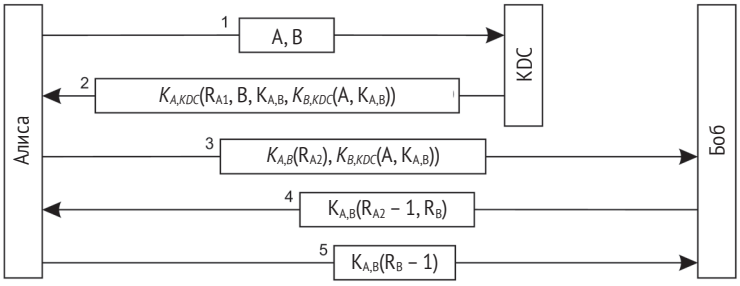


Рис. 9.11 ❖ Протокол аутентификации Нидхэма–Шредера

Протокол Нидхэма–Шредера, показанный на рис. 9.11, является так называемым **протоколом многоадресного вызова-ответа** (multiway challenge-response protocol) и работает следующим образом.

Когда Алиса хочет установить безопасный канал с Бобом, она отправляет запрос в KDC, содержащий запрос R_A, ее идентификацию A и, конечно, идентификацию Боба. KDC отвечает, давая ей билет $K_{B,KDC}(K_{A,B})$, вместе с секретным ключом $K_{A,B}$, которым она впоследствии может поделиться с Бобом.

Вызов R_{A1}, который Алиса отправляет в KDC вместе со своим запросом на установку канала для Боба, называется **одноразовым номером** (nonce). Одноразовый номер – это случайное число, которое используется только один раз, например выбранное из очень большого набора. Основная цель одноразового номера состоит в том, чтобы однозначно связать два сообщения друг с другом, в данном случае это сообщение 1 и сообщение 2. В частности, снова включив R_{A1} в сообщение 2, Алиса будет точно знать, что сообщение 2 отправляется в ответ на сообщение 1 и что оно не является, например, воспроизведением более старого сообщения.

Чтобы понять возникшую проблему, предположим, что мы не использовали одноразовые номера и что Чак украл один из старых ключей Боба, скажем $K^{old}_{B,KDC}$. Кроме того, Чак перехватил старый ответ $K_{A,KDC}(B, K_{A,B}, K^{old}_{B,KDC}(A, K_{A,B}))$, который KDC возвратил по предыдущему запросу Алисы, чтобы поговорить с Бобом. Тем временем Боб будет согласовывать новый общий секретный ключ с KDC. Однако Чак терпеливо ждет, пока Алиса снова не попросит установить безопасный канал с Бобом. В этот момент он воспроизводит старый ответ и обманывает Алису, заставляя ее поверить, что она разговаривает с Бобом, потому что он может расшифровать билет и доказать, что знает общий секретный ключ $K_{A,B}$. Очевидно, что это неприемлемо и должно быть защищено. При включении одноразового номера такая атака невозможна, поскольку воспроизведение более старого сообщения (имеющего другой одноразовый номер) будет немедленно обнаружено.

Сообщение 2 также содержит B, идентификатор Боба. Включив B, KDC защищает Алису от следующей атаки. Предположим, что B не было в сообщении 2. В этом случае Чак может изменить сообщение 1, заменив идентификатор Боба своим собственным идентификатором, скажем C. Затем KDC решит, что Алиса хочет установить безопасный канал с Чаком, и ответит соответствующим образом. Как только Алиса хочет связаться с Бобом, Чак перехватывает сообщение и обманывает Алису, заставляя ее поверить, что она разговаривает с Бобом. Копируя личность другой стороны из сообщения 1 в сообщение 2, Алиса немедленно обнаружит, что ее запрос был изменен.

После того как KDC передал билет Алисе, можно установить безопасный канал между Алисой и Бобом. Алиса начинает с отправки сообщения 3, которое содержит билет Бобу и запрос R_{A2}, зашифрованный общим ключом $K_{A,B}$, который только что сгенерировал KDC. Затем Боб расшифровывает билет, чтобы найти общий ключ, и возвращает ответ $R_{A2}-1$ вместе с запросом R_B для Алисы.

Следующее замечание – относительно сообщения 4 в порядке его передачи. В общем, возвращая $R_{A2} - 1$, а не только R_{A2}, Боб не только доказывает, что он знает общий секретный ключ, но и что он фактически расшифровал вызов. Опять же, это связывает сообщение 4 с сообщением 3 так же, как одноразовый R_A связал сообщение 2 с сообщением 1. Таким образом, протокол более защищен от повторов. Однако в этом особом случае было бы достаточно просто вернуть $K_{A,B}(R_{A2}, R_B)$ по той простой причине, что это сообщение еще нигде не использовалось ранее в протоколе. $K_{A,B}(R_{A2}, R_B)$ уже доказывает, что Боб был способен расшифровать вызов, отправленный сообщением 3. Сообщение 4, показанное на рис. 9.11, обязано историческим причинам.

Протокол Нидхэма–Шредера, представленный здесь, все еще имеет слабое место в том, что если Чак когда-нибудь получит старый ключ $K^{old}_{A,B}$, он сможет воспроизвести сообщение 3 и заставить Боба установить канал. Боб тогда поверит, что разговаривает с Алисой, в то время как Чак будет на другом конце канала. В этом случае нам нуж-

но связать сообщение 3 с сообщением 1, то есть сделать ключ зависимым от первоначального запроса Алисы о настройке канала с Бобом. Решение показано на рис. 9.12.

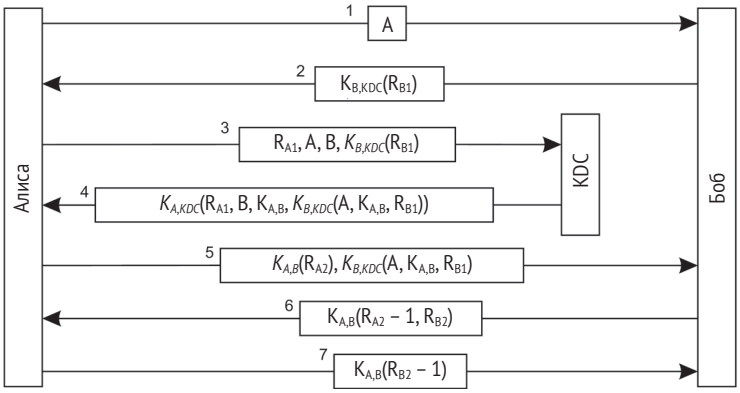


Рис. 9.12 ❖ Защита от злонамеренного повторного использования ранее сгенерированного сеансового ключа в протоколе Нидхэма–Шредера

Хитрость заключается в том, чтобы включить одноразовый номер в запрос, отправленный Алисой в KDC. Однако одноразовый номер должен исходить от Боба: это подтверждает Бобу, что тот, кто захочет установить с ним безопасный канал, получит соответствующую информацию от KDC. Поэтому Алиса сначала просит Боба отправить ей одноразовый номер R_{B1}, зашифрованный с помощью ключа, общего для Боба и KDC. Алиса включила этот одноразовый номер в свой запрос к KDC, который впоследствии расшифрует его и поместит результат в сгенерированный билет. Таким образом, Боб будет точно знать, что ключ сеанса привязан к исходному запросу Алисы на разговор с Бобом.

Аутентификация с использованием криптографии с открытым ключом

Давайте теперь посмотрим на аутентификацию с помощью криптосистемы с открытым ключом, которая не требует KDC. Опять же, рассмотрим ситуацию, когда Алиса хочет установить безопасный канал для Боба и оба имеют открытый ключ друг друга. Типичный протокол аутентификации, основанный на криптографии с открытым ключом, показан на рис. 9.13, который мы объясним ниже.

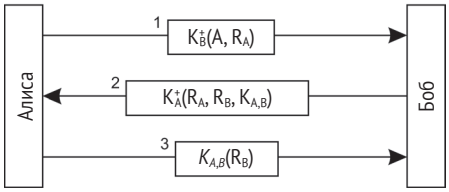


Рис. 9.13 ❖ Взаимная аутентификация в криптосистеме с открытым ключом

Алиса начинает с отправки запроса R_A Бобу, зашифрованному его открытым ключом K_B^+. Задача Боба – расшифровать сообщение и вернуть вызов Алисе. Поскольку Боб – единственный человек, который может расшифровать сообщение (используя закрытый ключ, связанный с открытым ключом, который использовала Алиса), Алиса будет знать, что она разговаривает с Бобом. Обратите внимание, что важно, чтобы Алиса гарантированно использовала открытый ключ Боба, а не открытый ключ кого-то, выдающего себя за Боба. Как такие гарантии могут быть предоставлены, обсуждается далее в этой главе.

Когда Боб получает запрос Алисы на настройку канала, он возвращает расшифрованный вызов вместе со своим собственным запросом R_B для аутентификации Алисы. Кроме того, он генерирует сеансовый **ключ** $K_{A,B}$, который можно использовать для дальнейшей связи. Ответ Боба на вызов Алисы, его собственный вызов и ключ сеанса помещаются в сообщение, зашифрованное открытым ключом K_B^+, принадлежащим Алисе, показанное на рис. 9.13 как сообщение 2. Только Алиса будет способна расшифровать это сообщение, используя закрытый ключ K_A^-.

Наконец, Алиса возвращает свой ответ на вызов Боба, используя сессионный ключ $K_{A,B}$, сгенерированный Бобом. Таким образом она докажет, что может расшифровать сообщение 2 и что она на самом деле Алиса, с которой разговаривает Боб.

Целостность и конфиденциальность сообщений

Помимо аутентификации, безопасный канал также должен обеспечивать гарантии целостности и конфиденциальности сообщений. Целостность сообщения означает, что сообщения защищены от скрытой модификации; конфиденциальность гарантирует, что сообщения не могут быть перехвачены и прочитаны злоумышленниками. Конфиденциальность легко устанавливается путем простого шифрования сообщения перед его отправкой. Шифрование может осуществляться либо через секретный ключ, предоставленный получателю, либо, альтернативно, через открытый ключ получателя. Однако защитить сообщение от модификаций несколько сложнее, как мы увидим далее.

Цифровые подписи

Целостность сообщения часто выходит за рамки фактической передачи по безопасному каналу. Рассмотрим ситуацию, когда Боб только что продал Алисе коллекционный предмет с виниловой пластинкой за 500 долларов. Вся сделка была совершена по электронной почте.

В конце Алиса отправляет Бобу сообщение, подтверждающее, что она купит предмет за 500 долларов. В дополнение к аутентификации есть по крайней мере две проблемы, о которых нужно заботиться относительно целостности сообщения.

1. Алиса должна быть уверена, что Боб не будет злонамеренно увеличивать сумму в 500 долларов, упомянутых в ее сообщении, и утверждать, что она обещала более 500 долларов.

2. Боб должен быть уверен, что Алиса не сможет отрицать, что отправила сообщение, например потому, что у нее появились другие соображения.

Эти две проблемы могут быть решены, если Алиса подписывает сообщение в цифровой форме таким образом, что ее подпись однозначно связана с его содержанием. Уникальная связь между сообщением и его подписью предотвращает то, что модификации сообщения останутся незамеченными. Кроме того, если подпись Алисы может быть проверена на подлинность, она не сможет впоследствии отрицать тот факт, что она подписала сообщение.

Существует несколько способов размещения цифровых подписей. Одной из популярных форм является использование криптосистемы с открытым ключом, как показано на рис. 9.14. Когда Алиса отправляет Бобу сообщение m, она шифрует его своим закрытым ключом K_A^- и отправляет его Бобу. Если она также хочет сохранить содержимое сообщения в секрете, она может использовать открытый ключ Боба и отправить $K_B^+(m, K_A^-(m))$, который объединяет m и версию, подписанную Алисой.

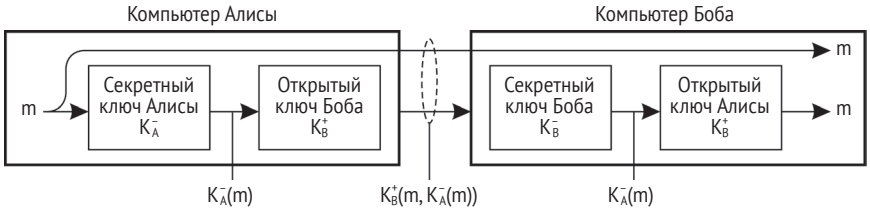


Рис. 9.14 ❖ Цифровая подпись сообщения с использованием криптографии с открытым ключом

Когда сообщение приходит к Бобу, он может расшифровать его, используя открытый ключ Алисы. Если он может быть уверен, что открытый ключ действительно принадлежит Алисе, то расшифровка подписанной версии m и успешное сравнение ее с m может означать только то, что он получен от Алисы. Алиса защищена от любых злонамеренных модификаций m Бобом, потому что Бобу всегда нужно будет доказывать, что модифицированная версия m также была подписана Алисой. Другими словами, дешифрованное сообщение само по себе никогда не считается доказательством. В интересах Боба также сохранить подписанную версию m, чтобы защитить себя от отказа Алисы.

Существует несколько проблем с этой схемой, хотя сам по себе протокол правильный. Во-первых, действительность подписи Алисы сохраняется только до тех пор, пока секретный ключ Алисы остается секретом. Если Алиса хочет выйти из сделки даже после отправки Бобу своего подтверждения, она может заявить, что ее личный ключ был украден до того, как сообщение было отправлено.

Другая проблема возникает, когда Алиса решает изменить свой закрытый ключ. Само по себе это может быть не такой уж плохой идеей, так как время от времени смена ключей обычно помогает предотвратить вторжение. Однако как только Алиса сменила ключ, ее заявление, отправленное Бобу, становится бесполезным. В таких случаях может потребоваться центральный

орган, который отслеживает изменение ключей в дополнение к использованию меток времени при подписании сообщений.

Примечание 9.4 (дополнительная информация: подписание дайджеста сообщения)

Другая проблема с этой схемой заключается в том, что Алиса шифрует все сообщение своим закрытым ключом. Такое шифрование может быть дорогостоящим с точки зрения требований к обработке (или даже математически неосуществимым, поскольку мы предполагаем, что сообщение, интерпретируемое как двоичное число, ограничено заранее определенным максимумом) и на самом деле не нужно. Напомним, что нам нужно уникально связать подпись с единственным конкретным сообщением. Более дешевой и, возможно, более элегантной схемой является использование **дайджеста сообщения** (message digest).

Дайджест сообщения – это битовая строка фиксированной длины h, которая была вычислена из сообщения m произвольной длины с помощью криптографической хеш-функции H. Если m изменить на m′, его хеш H(m′) будет отличаться от h = H(m), чтобы можно было легко обнаружить, что произошла модификация.

Для цифровой подписи сообщения Алиса может сначала вычислить дайджест сообщения и затем зашифровать дайджест своим закрытым ключом, как показано на рис. 9.15. Зашифрованный дайджест отправляется вместе с сообщением Бобу. Обратите внимание, что само сообщение отправляется в виде открытого текста: каждый может его прочитать. Если требуется конфиденциальность, то сообщение также должно быть зашифровано открытым ключом Боба.

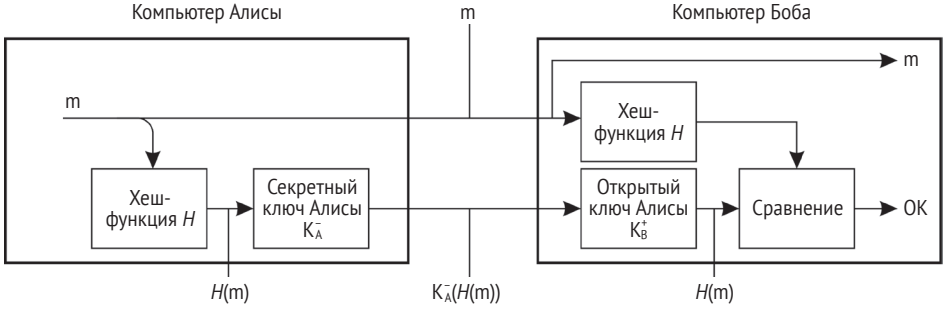


Рис. 9.15 ❖ Цифровая подпись сообщения с использованием дайджеста сообщения

Когда Боб получает сообщение и его зашифрованный дайджест, ему нужно просто расшифровать дайджест с помощью открытого ключа Алисы и отдельно рассчитать дайджест сообщения. Если дайджест, полученный из сообщения, и дешифрованный дайджест совпадают, Боб знает, что сообщение было подписано Алисой.

Сессионные ключи

Во время установления безопасного канала после завершения фазы аутентификации взаимодействующие стороны обычно используют уникальный общий сеансовый ключ для конфиденциальности. Ключ сеанса безопасно

сбрасывается, когда канал больше не используется. Альтернативой было бы использовать те же ключи для конфиденциальности, что и те, которые используются для настройки безопасного канала. Вместе с тем есть ряд важных преимуществ использования сеансовых ключей [Kaufman et al., 2003].

Во-первых, когда ключ используется часто, его легче обнаружить. В некотором смысле криптографические ключи подвержены «износу», как и обычные ключи. Основная идея состоит в том, что если злоумышленник может перехватить много данных, которые были зашифрованы с использованием одного и того же ключа, становится возможным организовать атаки, чтобы найти определенные характеристики используемых ключей и, возможно, выявить открытый текст или сам ключ. По этой причине гораздо безопаснее использовать ключи аутентификации как можно реже. Кроме того, такими ключами часто обмениваются с использованием некоторого относительно дорогостоящего внешнего механизма, такого как обычная почта или телефон. Обмен ключами таким образом должен быть сведен к минимуму.

Другой важной причиной создания уникального ключа для каждого безопасного канала является обеспечение защиты от атак воспроизведения, с которыми мы уже встречались ранее. Используя уникальный ключ сеанса каждый раз, когда устанавливается безопасный канал, взаимодействующие стороны, по крайней мере, защищены от воспроизведения всего сеанса. Для защиты воспроизведения отдельных сообщений из предыдущего сеанса обычно необходимы дополнительные меры, такие как включение меток времени или порядковых номеров в качестве части содержимого сообщения.

Предположим, что целостность и конфиденциальность сообщения были достигнуты с использованием того же ключа, который применялся для установления сеанса. В этом случае всякий раз, когда ключ скомпрометирован, злоумышленник может расшифровать сообщения, переданные во время старого разговора, что явно не является желательной функцией. Вместо этого гораздо безопаснее использовать ключи для каждого сеанса, потому что если такой ключ скомпрометирован, в худшем случае затрагивается только один сеанс. Сообщения, отправленные во время других сеансов, остаются конфиденциальными.

С этим связано и то, что Алиса может захотеть обменяться некоторыми конфиденциальными данными с Бобом, но она не доверяет ему настолько сильно, что предоставит ему информацию в виде данных, которые были зашифрованы с помощью долговременных ключей. Она может захотеть зарезервировать такие ключи для конфиденциальных сообщений, которыми она обменивается со сторонами, которым действительно доверяет. В таких случаях достаточно использовать относительно дешевый сеансовый ключ для общения с Бобом.

В общем, ключи аутентификации часто устанавливаются таким образом, что их замена оказывается относительно дорогой. Поэтому сочетание таких долговечных ключей с гораздо более дешевыми и более временными сеансовыми ключами часто является хорошим выбором для реализации безопасных каналов для обмена данными.

Безопасное групповое общение

Пока что мы сконцентрировались на создании безопасного канала связи между двумя сторонами. Однако в распределенных системах часто необходимо обеспечить безопасную связь между более чем двумя сторонами. Типичным примером является реплицированный сервер, для которого все коммуникации между репликами должны быть защищены от модификации, изготовления и перехвата, как в случае двухсторонних безопасных каналов. В этом разделе мы подробнее рассмотрим безопасное групповое общение.

Конфиденциальное групповое общение

Сначала рассмотрим проблему защиты связи между группой из N пользователей от прослушивания. Чтобы обеспечить конфиденциальность, простая схема состоит в том, чтобы позволить всем членам группы использовать один и тот же секретный ключ, который применяется для шифрования и дешифрования всех сообщений, передаваемых между членами группы. Поскольку секретный ключ в этой схеме является общим для всех участников, необходимо, чтобы все участники были надежными, чтобы действительно хранить ключ в секрете. Одно это предварительное условие делает использование единого общего секретного ключа для конфиденциальной групповой связи более уязвимым для атак, по сравнению с двухсторонними безопасными каналами.

Альтернативным решением является использование отдельного общего секретного ключа между каждой парой членов группы. Как только один из членов обнаруживает утечку информации, другие могут просто прекратить отправлять сообщения этому участнику, но все равно использовать ключи, которые они применяли, для связи друг с другом. Однако вместо того, чтобы поддерживать один ключ, теперь необходимо сохранить $N(N-1)/2$ ключей, что само по себе может быть сложной задачей.

Использование криптосистемы с открытым ключом может улучшить ситуацию. В этом случае каждый член имеет свою собственную (открытый ключ, закрытый ключ) пару, в которой открытый ключ может применяться всеми членами для отправки конфиденциальных сообщений. Тогда необходимо всего N пар ключей. Если один участник перестает быть заслуживающим доверия, он просто удаляется из группы, не имея возможности скомпрометировать другие ключи.

Безопасные реплицированные серверы

Теперь рассмотрим совершенно другую проблему: клиент отправляет запрос группе реплицируемых серверов. Серверы могут быть реплицированы по причинам отказоустойчивости или производительности, но в любом случае клиент ожидает, что ответ будет заслуживающим доверия. Другими словами, независимо от того, подвержена ли группа серверов византийским сбоям, клиент ожидает, что возвращенный ответ не был поврежден.

Решение для защиты клиента от таких ситуаций заключается в сборе ответов со всех серверов и аутентификации каждого из них. Если большинство ответов не повреждено (то есть от аутентифицированных серверов), клиент также может доверять ответу. К сожалению, такой подход выявляет репликацию серверов, нарушая прозрачность репликации.

Примечание 9.5 (дополнительно: секретный обмен)

Сутью безопасных и прозрачно реплицируемых серверов является секретный обмен. Когда несколько пользователей (или процессов) разделяют секрет, ни один из них не знает весь секрет. Вместо этого секрет может быть раскрыт, только если они все собираются вместе. Такие схемы могут быть чрезвычайно полезными. Возьмем, к примеру, запуск ядерной ракеты. Подобный акт обычно требует разрешения как минимум двух человек. Каждый из них имеет закрытый ключ, который должен использоваться в сочетании с другим для фактического запуска ракеты. Использование только одного ключа не действует.

В случае безопасных реплицируемых серверов мы ищем решение, с помощью которого не больше, чем k из N серверов могут дать неправильный ответ, а из этих k серверов не больше, чем $c \leq k$ фактически были повреждены. Обратите внимание, что это требование делает службу k-отказоустойчивой, как описано в разделе 8.2. Разница заключается в том, что мы теперь классифицируем поврежденный сервер как неисправный.

Следуя работе [Reiter, 1994], рассмотрим ситуацию, в которой серверы активно реплицируются. Другими словами, запрос отправляется на все серверы одновременно, а затем обрабатывается каждым из них. Каждый сервер выдает ответ, который он возвращает клиенту. Для надежно реплицируемой группы серверов мы требуем, чтобы каждый сервер сопровождал свой ответ цифровой подписью. Если r_i является ответом от сервера S_i, пусть $md(r_i)$ обозначает дайджест сообщения, вычисленный сервером S_i. Этот дайджест подписан S_i секретным ключом K_i^-.

Предположим, что мы хотим защитить клиента от максимально поврежденных серверов. Другими словами, группа серверов должна быть способна допускать повреждение не более чем на c серверах и при этом иметь возможность создавать ответ, которому клиент может доверять. Если сигнатуры отдельных серверов можно объединить так, что для построения действительной подписи для ответа необходимо как минимум $c + 1$ подписей, то это решило бы нашу проблему. Другими словами, мы хотим, чтобы реплицированные серверы генерировали секретную действительную подпись со свойством, что одних поврежденных серверов недостаточно для создания этой подписи.

В качестве примера рассмотрим группу из пяти реплицированных серверов, которые должны иметь возможность переносить два поврежденных сервера и попрежнему выдавать ответ, которому клиент может доверять. Каждый сервер S_i отправляет свой ответ r_i клиенту вместе со своей сигнатурой $sig(S_i, r_i) = K_i^-(md(r_i))$. Следовательно, клиент в конечном итоге получит пять триплетов $(r_i, md(r_i), sig(S_i, r_i))$, из которых он должен получить правильный ответ. Эта ситуация показана на рис. 9.16.

Каждый дайджест $md(r_i)$ также рассчитывается клиентом. Если r_i неверно, то обычно это можно определить, вычисляя $K_i^+(K_i^-(md(r_i)))$. Однако данный метод больше не может быть применен, потому что ни одному отдельному серверу нельзя доверять. Вместо этого клиент использует специальную, общеизвестную функцию дешифрования D, которая принимает набор

$$V = \{sig(S, r), sig(S', r'), sig(S'', r'')\}$$

из *трех* подписей в качестве входных данных и создает один дайджест в качестве выходных данных:

$d_{out} = D(V) = D(sig(S, r), sig(S', r'), sig(S'', r''))$.

Подробнее по поводу *D* см. [Reiter, 1994]. Существует 5!/(3!2!) = 10 возможных комбинаций трех подписей, которые клиент может использовать в качестве ввода для *D*. Если одна из этих комбинаций дает правильный дайджест $md(r_i)$ для некоторого ответа r_i, тогда клиент может считать r_i правильным. В частности, можно доверять тому, что ответ был получен как минимум тремя честными серверами.

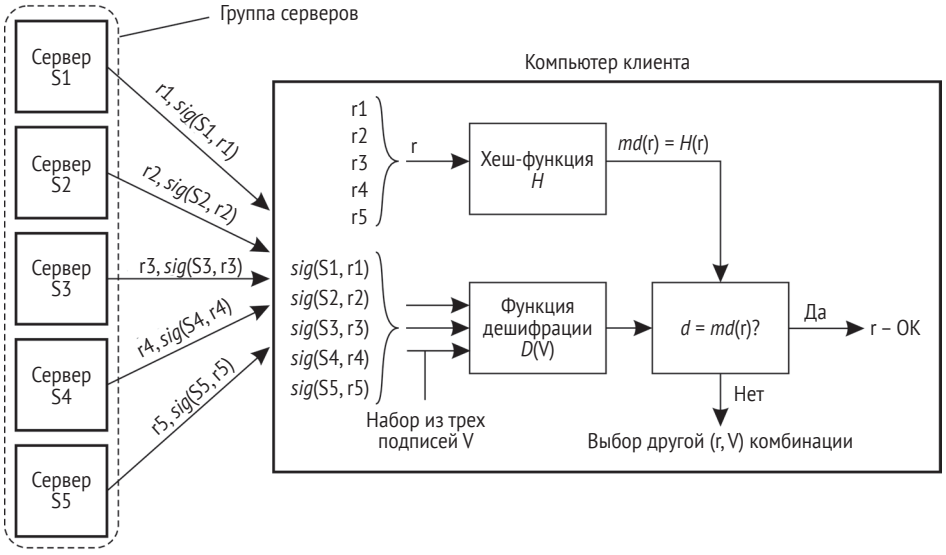


Рис. 9.16 ❖ Совместное использование секретной подписи в группе реплицируемых серверов

Чтобы улучшить прозрачность репликации, мы можем позволить каждому серверу S_i транслировать сообщение, содержащее его ответ r_i, на другие серверы вместе со связанной сигнатурой $sig(S_i, r_i)$. Когда сервер получил по крайней мере $c + 1$ таких сообщений, включая собственное сообщение, он пытается вычислить действительную подпись для одного из ответов. Если это удается, скажем для ответа r и набора V из сигнатур $c + 1$, сервер отправляет клиенту r и V как одно сообщение. Клиент может впоследствии проверить правильность r, проверив его сигнатуру, то есть $md(r) = D(V)$.

То, что мы только что описали, также известно как **(m,n)-пороговая схема** ((m,n)-threshold scheme), в нашем примере $m = c + 1$ и $n = N$, количество серверов. В (*m*, *n*)-пороговой схеме сообщение было разделено на *n* частей, известных как **тени** (shadows), поскольку любые *m* теней могут использоваться для восстановления исходного сообщения, но использование $m - 1$ или меньшего количества сообщений невозможно. Существует несколько способов построения (*m*, *n*)-пороговых схем. Подробности можно найти в [Schneier, 1996].

Пример: система Kerberos

К настоящему времени должно быть ясно, что включение безопасности в распределенные системы не является тривиальным. Проблемы вызваны тем, что вся система должна быть защищена; если какая-то часть небезопасна, вся система может быть взломана. Чтобы помочь в создании распределенных систем, которые могут обеспечить множество политик безопасности, был разработан ряд вспомогательных систем, которые можно использовать в качестве основы для дальнейшей разработки. Важной системой, которая широко используется, является **Kerberos** [Steiner et al., 1988; Kohl et al., 1994]. Система Kerberos была разработана в Массачусетском технологическом институте (Massachusetts Institute of Technology, M.I.T.) и основана на **протоколе аутентификации Нидхэма–Шредера** (Needham-Schroeder authentication protocol). Подробное описание системы Kerberos можно найти в [Neuman et al., 2005]. Практическая информация о запуске Kerberos описана в [Garman, 2003]. Общедоступная реализация Kerberos, известная как Shishi, описана в [Josefsson, 2015].

Систему Kerberos можно рассматривать как систему безопасности, которая помогает клиентам в настройке безопасного канала с любым сервером, являющимся частью распределенной системы. Безопасность основана на общих секретных ключах. Есть два разных компонента. **Сервер аутентификации** (Authentication Server, AS) отвечает за обработку запроса на вход от пользователя. Сервер AS аутентифицирует пользователя и предоставляет ключ, который можно использовать для настройки защищенных каналов с серверами. Настройка безопасных каналов осуществляется **Службой предоставления билетов** (Ticket Granting Service, TGS). Служба TGS раздает специальные сообщения, известные как билеты, которые используются, чтобы убедить сервер в том, что клиент действительно тот, кем он себя называет. Ниже приведены конкретные примеры билетов.

Давайте посмотрим, как Алиса входит в распределенную систему, которая использует систему Kerberos, и как она может установить безопасный канал с сервером Боба. Чтобы Алиса могла войти в систему, она может использовать любую доступную рабочую станцию. Рабочая станция отправляет свое имя в незашифрованном виде в AS, которая возвращает сеансовый ключ $K_{A,TGS}$ и билет, который ей нужно будет передать в TGS.

Билет, возвращаемый AS, содержит идентификационные данные Алисы и сгенерированный секретный ключ, который Алиса и TGS могут использовать для связи друг с другом. Сам билет будет передан Алисе в TGS. Поэтому важно, чтобы никто, кроме TGS, не мог его прочитать. По этой причине билет зашифрован секретным ключом KAS, разделяемым TGS между AS и TGS.

Эта часть процедуры входа в систему показана как сообщения 1, 2 и 3 на рис. 9.17 соответственно. Сообщение 1 на самом деле не является сообщением, но соответствует Алисе, набирающей свое имя пользователя на рабочей станции. Сообщение 2 содержит это имя и отправляется в AS. Сообщение 3 содержит сеансовый ключ $K_{A,TGS}$ и билет $K_{AS,TGS}(A, K_{A,TGS})$.

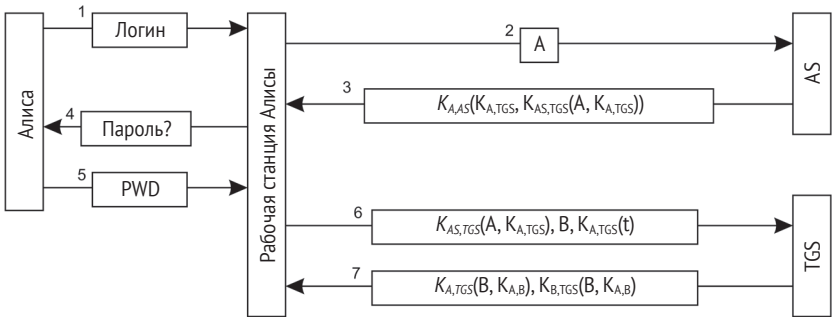


Рис. 9.17 ❖ Аутентификация в системе Kerberos

Чтобы обеспечить конфиденциальность, сообщение 3 шифруется секретным ключом $K_{A,AS}$, совместно используемым между Алисой и AS.

Когда рабочая станция получает ответ от AS, она запрашивает у Алисы пароль (показано как сообщение 4), который использует для последующего генерирования общего ключа $K_{A,AS}$. (Относительно просто взять пароль строки символов, применить криптографический хеш, а затем взять первые 56 бит в качестве секретного ключа.) Обратите внимание, что этот подход имеет не только то преимущество, что пароль Алисы никогда не передается в виде открытого текста по сети, но также и то, что рабочая станция даже не должна временно его хранить. Более того, как только она сгенерирует общий ключ $K_{A,AS}$, рабочая станция найдет сеансовый ключ $K_{A,TGS}$ и может забыть пароль Алисы и использовать только общий секретный ключ $K_{A,AS}$.

После этой части аутентификации Алиса может считать, что вошла в систему через текущую рабочую станцию. Полученный от AS билет хранится временно (обычно в течение 8–24 ч) и будет использоваться для доступа к удаленным службам. Конечно, если Алиса покидает свою рабочую станцию, она должна уничтожить любые кешированные билеты. Если она хочет поговорить с Бобом, она просит TGS сгенерировать сеансовый ключ для Боба, показанный как сообщение 6 на рис. 9.17. Тот факт, что у Алисы есть билет $K_{AS,TGS}(A, K_{A,TGS})$, доказывает, что она Алиса. TGS отвечает сессионным ключом $K_{A,B}$, снова инкапсулированным в билет, который Алисе позже придется передать Бобу.

Сообщение 6 также содержит метку времени t, зашифрованную с помощью секретного ключа, совместно используемого Алисой и TGS. Эта временная метка используется для предотвращения повторного злонамеренного воспроизведения сообщения 6 Чаком и попытки настроить канал для Боба. TGS проверит отметку времени перед возвратом билета Алисе. Если он отличается более чем на несколько минут от текущего времени, запрос на билет отклоняется.

Эта схема устанавливает так называемый единый вход. Пока Алиса не меняет рабочие станции, ей не нужно проходить аутентификацию на любом другом сервере, который является частью распределенной системы. Эта функция важна, когда приходится иметь дело со многими различными службами, которые распределены по нескольким машинам. В принципе, серверы

каким-то образом делегировали аутентификацию клиента в AS и TGS и будут принимать запросы от любого клиента, у которого есть действующий билет. Конечно, такие службы, как удаленный вход в систему, потребуют, чтобы связанный пользователь имел учетную запись, но это не зависит от аутентификации через систему Kerberos.

Настройка безопасного канала с Бобом теперь проста и показана на рис. 9.18. Сначала Алиса отправляет Бобу сообщение, содержащее билет, который она получила от TGS вместе с зашифрованной меткой времени. Когда Боб расшифровывает билет, он замечает, что разговаривает с ним Алиса, потому что только TGS мог построить билет.

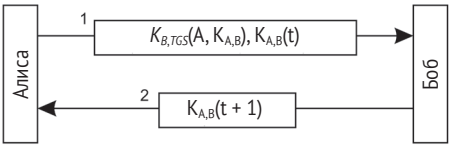


Рис. 9.18 ❖ Настройка безопасного канала в системе Kerberos

Она также находит секретный ключ $K_{A,B}$, что позволяет ей проверить метку времени. В этот момент Боб знает, что он говорит с Алисой, а не с кем-то, кто злонамеренно воспроизводит сообщение 1. Отвечая $K_{A,B}(t + 1)$, Боб доказывает Алисе, что он действительно Боб.

9.3. Контроль доступа

В модели клиент-сервер, которую мы использовали до сих пор, когда клиент и сервер настроили безопасный канал, клиент может выдавать запросы, которые должны выполняться сервером. Запросы включают выполнение операций над контролируемыми сервером ресурсами. Общая ситуация – это ситуация с объектным сервером, который контролирует несколько объектов. Запрос от клиента обычно включает в себя вызов метода конкретного объекта. Такой запрос может быть выполнен, только если у клиента есть достаточные **права доступа** (access right) для этого вызова. Формально проверка прав доступа называется **контролем доступа** (access contol), тогда как авторизация касается предоставления прав доступа. Эти два термина тесно связаны друг с другом и часто используются взаимозаменяемо.

Общие вопросы управления доступом

Чтобы понять различные проблемы, связанные с контролем доступа, в целом принята простая модель, показанная на рис. 9.19. Она состоит из **субъектов** (subjects), которые выдают запрос на доступ к **объекту** (object). Объект очень похож на объекты, которые мы обсуждали до сих пор. Его можно предста-

вить как инкапсулирующее собственное состояние и выполняющее операции в этом состоянии. Операции объекта, которые субъекты могут запросить, выполняются через интерфейсы. Субъекты лучше всего рассматривать как процессы, действующие от имени пользователей, но они также могут быть объектами, которые нуждаются в услугах других объектов для выполнения своей работы.

Рис. 9.19 ❖ Общая модель управления доступом к объектам

Контроль доступа к объекту – это все, что касается защиты объекта от вызовов субъектами, которым не разрешено иметь конкретные (или даже любые) из выполняемых методов. Кроме того, защита может включать проблемы управления объектами, такие как создание, переименование или удаление объектов. Защита часто обеспечивается программой, называемой **контрольным монитором** (reference monitor). В контрольном мониторе записано, что субъект может делать, и определено, разрешено ли субъекту выполнять конкретную операцию. Этот монитор вызывается (например, базовой операционной системой) каждый раз, когда объект вызывается. Следовательно, крайне важно, чтобы контрольный монитор сам по себе был защищен от взлома: злоумышленник не должен иметь доступа к нему.

Матрица контроля доступа

Общим подходом к моделированию прав доступа субъектов по отношению к объектам является построение **матрицы контроля доступа** (access control matrix). Каждый предмет представлен в этой матрице строкой; каждый объект представлен столбцом. Если матрица обозначена M, то запись M[s, o] точно указывает, какие операции субъекты могут запросить, чтобы они были выполнены над объектом o. Другими словами, всякий раз, когда субъект запрашивает вызов метода m объекта o, монитор ссылок должен проверить, присутствует ли m в M[s, o]. Если m не указан в M [s, o], вызов завершится неудачно.

Учитывая, что системе может легко понадобиться поддержка тысяч пользователей и миллионов объектов, которые нуждаются в защите, реализация матрицы контроля доступа в качестве истинной матрицы не является подходящим способом. Многие записи в матрице будут пустыми: один субъект, как правило, будет иметь доступ к относительно небольшому количеству объектов.

Соответственно, для реализации матрицы управления доступом используются другие, более эффективные способы.

Один из широко применяемых подходов состоит в том, что каждый объект поддерживает список прав доступа субъектов, которые хотят получить до-

ступ к объекту. По сути, это означает, что матрица распределена по столбцам по всем объектам, а пустые записи пропущены. Этот тип реализации приводит к тому, что называется **списком контроля доступа** (Access Control List, ACL). Предполагается, что каждый объект имеет свой собственный ACL.

Другой подход заключается в распределении матрицы по строкам путем предоставления каждому субъекту **списка возможностей** (list of capabilities), которые он имеет для каждого объекта. Другими словами, возможность соответствует записи в матрице управления доступом. Отсутствие возможности для конкретного объекта означает, что субъект не имеет прав доступа к этому объекту.

Возможность можно сравнить с билетом: ее владельцу предоставляются определенные права, связанные с этим билетом. Также ясно, что билет должен быть защищен от изменений его владельцем. Один подход, который особенно применим в распределенных системах, заключается в защите (списка) возможностей с помощью сигнатуры. Мы вернемся к этим и другим вопросам позже при обсуждении управления безопасностью.

Разница между тем, как контроль доступа ACL и список возможностей используются для защиты доступа к объекту, показана на рис. 9.20. Когда клиент отправляет запрос на сервер, серверный контрольный монитор, используя списки ACL, проверяет, знает ли он клиента и разрешено ли выполнять запрошенную операцию, как показано на рис. 9.20a. Однако при использовании списка возможностей клиент просто отправляет свой запрос на сервер. Сервер не интересуется, знает ли он клиента; возможность говорит об этом достаточно. Следовательно, серверу нужно только проверить, действительна ли эта возможность и указана ли запрошенная операция в списке возможностей. Этот подход к защите объектов с помощью возможностей показан на рис. 9.20b.

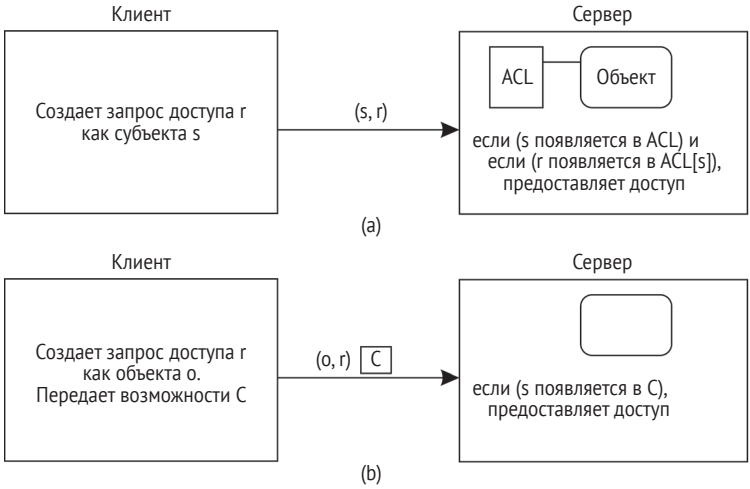


Рис. 9.20 ❖ Сравнение ACL и списка возможностей для защиты объектов: a) использование ACL; b) использование списка возможностей

Примечание 9.6 (дополнительно: домен защиты)

Система ACL и возможности помогают эффективно реализовать матрицу контроля доступа, игнорируя все пустые записи. Тем не менее список ACL или список возможностей может стать достаточно большим, если не будут приняты дополнительные меры.

Одним из общих способов сокращения ACL-списков является использование **доменов защиты** (protection domain). Формально домен защиты – это набор пар (объект, права доступа). Каждая пара указывает для данного объекта, какие именно операции разрешено выполнять [Saltzer and Schroeder, 1975]. Запросы на выполнение операции всегда выдаются в домене. Следовательно, всякий раз, когда субъект запрашивает выполнение операции над объектом, контрольный монитор сначала ищет область защиты, связанную с этим запросом. Затем, с учетом домена, контрольный монитор может проверить, разрешено ли выполнение запроса. Существуют различные варианты использования доменов защиты.

Одним из подходов является создание **групп** пользователей. Рассмотрим, например, веб-страницу во внутренней сети компании. Такая страница должна быть доступна каждому сотруднику, но никому другому. Вместо добавления записи для каждого возможного сотрудника в ACL для этой веб-страницы может быть решено создать отдельную группу Employee, содержащую всех текущих сотрудников. Всякий раз, когда пользователь получает доступ к веб-странице, монитор ссылок должен только проверить, является ли этот пользователь сотрудником. Какие пользователи принадлежат к группе Employee, хранится в отдельном списке (который, конечно же, защищен от несанкционированного доступа).

Подход можно сделать более гибким, введя иерархические группы. Например, если организация имеет три разных филиала, например в Амстердаме, Нью-Йорке и Сан-Франциско, она может захотеть разделить свою группу сотрудников на подгруппы, по одной для каждого города, что приведет к организации, как показано на рис. 9.21.

Доступ к веб-страницам внутренней сети организации должен быть разрешен всем сотрудникам. Однако, например, изменение веб-страниц, связанных с амстердамским филиалом, должно разрешаться только подмножеству сотрудников в Амстердаме. Если пользователь Дик из Амстердама хочет прочитать веб-страницу из интрасети, монитору ссылок необходимо сначала найти подмножества Employee_AMS, Employee_NYC и Employee_SF, которые совместно составляют набор группы Employee. Затем он должен проверить, содержит ли один из этих наборов Дика. Преимущество наличия иерархических групп состоит в том, что управление членством в группах является относительно простым и что очень большие группы могут при этом быть эффективно построены. Очевидным недостатком является то, что поиск члена может быть довольно дорогостоящим, если база данных членства распределена.

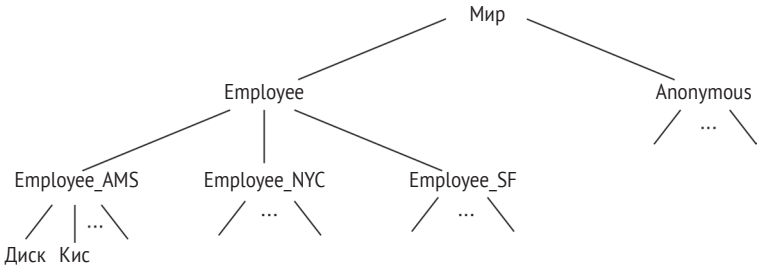


Рис. 9.21 ❖ Иерархическая организация доменов защиты как групп пользователей

Вместо того чтобы позволить контрольному монитору выполнять всю работу, альтернативой является предоставление каждому субъекту сертификата, в котором перечислены группы, к которым он принадлежит. Таким образом, всякий раз, когда Дик хочет прочитать веб-страницу из внутренней сети компании, он передает свой сертификат контрольному монитору, заявляя, что является членом Employee_AMS.

Чтобы гарантировать, что сертификат является подлинным и не был подделан, он должен быть защищен, например с помощью цифровой подписи. Сертификаты считаются сопоставимыми с возможностями.

Применительно к наличию групп в качестве доменов защиты можно также реализовать домены защиты в качестве ролей. В управлении доступом на основе ролей пользователь всегда входит в систему с определенной ролью, которая часто связана с функцией, выполняемой пользователем в организации [Sandhu et al., 1996]. Пользователь может иметь несколько функций. Например, Дик мог одновременно быть руководителем отдела, руководителем проекта и членом комитета по подбору персонала. В зависимости от роли, которую он выполняет при входе в систему, ему могут быть назначены различные привилегии. Другими словами, его роль определяет область защиты (то есть группу), в которой он будет действовать.

При назначении ролей пользователям и требовании, чтобы пользователи брали на себя роль при входе в систему, пользователи также должны при необходимости изменить свои роли. Например, может потребоваться, чтобы Дик как глава департамента иногда переходил на должность менеджера проекта. Обратите внимание, что такие изменения трудно выразить при реализации доменов защиты только в виде групп.

Помимо использования доменов защиты, эффективность может быть дополнительно повышена путем (иерархической) группировки объектов на основе операций, которые они предоставляют. Например, вместо рассмотрения отдельных объектов, чтобы достичь иерархии, объекты группируются в соответствии с предоставляемыми ими интерфейсами, возможно, с использованием подтипов (также называемых наследованием интерфейса, см. [Gamma et al., 1994]). В этом случае, когда субъект запрашивает выполнение операции на объекте, монитор ссылок проверяет, к какому интерфейсу относится операция для этого объекта. Затем он проверяет, разрешено ли субъекту вызывать операцию, принадлежащую этому интерфейсу, а не может ли она вызвать операцию для конкретного объекта.

Объединение доменов защиты и группирование объектов также возможны. Используя оба метода, наряду с конкретными структурами данных и ограниченными операциями над объектами в [Gladney, 1997] описывается, как реализовать списки ACL для очень больших наборов объектов, которые используются в цифровых библиотеках.

Брандмауэры

Мы показали, как можно установить защиту с использованием криптографических методов в сочетании с некоторой реализацией матрицы контроля доступа. Эти подходы работают до тех пор, пока все взаимодействующие стороны играют в соответствии с одним и тем же набором правил. Такие правила могут применяться при разработке автономной распределенной системы, которая более или менее изолирована от остального мира. Однако дело становится более сложным, когда необходимо общение с остальным миром, например отправка почты, доступ к веб-сайтам или предоставление локальных ресурсов.

Чтобы защитить ресурсы в этих условиях, нужен другой подход. На практике получается, что внешний доступ к любой части распределенной системы контролируется специальным эталонным монитором, известным как **брандмауэр** (firewall) [Cheswick and Bellovin, 2000; Zwicky et al., 2000]. Брандмауэры образуют один из наиболее часто используемых механизмов защиты в сетевых системах. По сути, брандмауэр отключает любую часть распределенной системы от внешнего мира, как показано на рис. 9.22. Все исходящие, но особенно все входящие пакеты направляются через специальный компьютер и проверяются до их прохождения. Несанкционированный трафик отбрасывается и не может продолжаться. Важным аспектом является то, что сам брандмауэр должен быть надежно защищен от любых угроз безопасности: он никогда не должен выходить из строя. Не менее важно то, что правила, которые предписывают, что может пройти, являются строгими и определяют намерения. Как сообщается в работе [Wool, 2010], правильная настройка брандмауэра является серьезной проблемой.

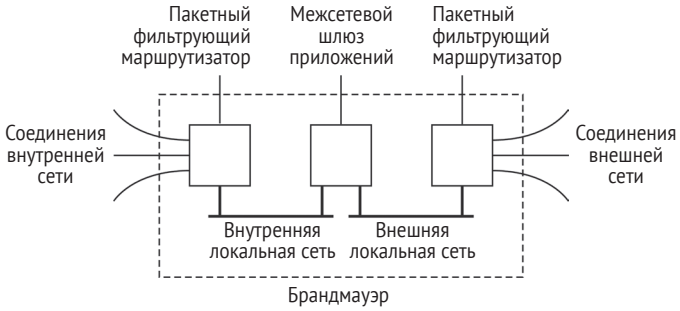


Рис. 9.22 ❖ Обычная реализация брандмауэра

Брандмауэры в основном бывают двух разных типов, которые часто объединяются. Важным типом брандмауэра является **шлюз с фильтрацией пакетов** (packet-filtering gateway). Этот тип брандмауэра работает как маршрутизатор и принимает решение о том, передавать или нет сетевой пакет на основе адреса источника и назначения, который содержится в заголовке пакета. Как правило, шлюз фильтрации пакетов, показанный на внешней локальной сети на рис. 9.22, защищает от входящих пакетов, тогда как шлюз на внутренней локальной сети фильтрует исходящие пакеты.

Например, чтобы защитить внутренний веб-сервер от запросов хостов, которые не находятся во внутренней сети, шлюз с фильтрацией пакетов может принять решение отбросить все входящие пакеты, адресованные веб-серверу.

Более тонкой является ситуация, когда сеть компании состоит из нескольких локальных сетей. Каждая ЛВС может быть защищена с помощью шлюза с пакетной фильтрацией, который настроен для прохождения входящего трафика только в том случае, если он исходил от хоста в одной из других локальных сетей. Таким образом, можно настроить частную виртуальную сеть.

Другой тип брандмауэра – это **шлюз уровня приложения** (application-level gateway). В отличие от шлюза фильтрации пакетов, который проверяет только заголовок сетевых пакетов, этот тип брандмауэра фактически проверяет содержимое входящего или исходящего сообщений. Типичным примером является почтовый шлюз, который отбрасывает входящую или исходящую почту, превышающую определенный размер. Существуют более сложные почтовые шлюзы, которые, например, способны фильтровать спам по электронной почте.

Другим примером шлюза уровня приложения является шлюз, позволяющий внешний доступ к серверу цифровой библиотеки, но предоставляющий только рефераты документов. Если внешний пользователь хочет большего, запускается протокол электронных платежей. Пользователи внутри брандмауэра имеют прямой доступ к библиотечному сервису.

Специальный вид шлюза уровня приложения – это так называемый **прокси-шлюз** (proxy gateway). Этот тип брандмауэра работает в качестве внешнего интерфейса для определенного типа приложений и обеспечивает передачу только тех сообщений, которые соответствуют определенным критериям. Рассмотрим, например, веб-сайт. Многие веб-страницы содержат скрипты или апплеты, которые должны выполняться в браузере пользователя. Чтобы предотвратить такую загрузку кода во внутреннюю локальную сеть, весь веб-трафик может быть направлен через веб-прокси шлюз. Этот шлюз принимает регулярные HTTP-запросы либо изнутри, либо за пределами брандмауэра. Другими словами, он представляется пользователям как обычный веб-сервер. Однако он фильтрует весь входящий и исходящий трафик, либо отбрасывая определенные запросы и страницы, либо изменяя страницы, когда они содержат исполняемый код.

Безопасный мобильный код

Важной проблемой в современных распределенных системах является возможность переноса кода между хостами, а не просто перенос пассивных данных. Тем не менее мобильный код вводит ряд серьезных угроз безопасности. Во-первых, хосты должны быть защищены от вредоносных агентов или загруженных программ. Последнее становится все более важным ввиду популярности смартфонов. Большинство пользователей распределенных систем не являются специалистами по системным технологиям и не смогут определить, можно ли доверять программе, которую они получают с другого хоста, чтобы не повредить их устройство. Во многих случаях даже специалисту может быть трудно обнаружить, что программа вообще загружается.

Если не предпринять меры безопасности, как только вредоносная программа установится на компьютере, она может легко повредить хост. Мы столкнулись с проблемой контроля доступа: программе не должен быть разрешен несанкционированный доступ к ресурсам хоста. Как мы увидим, защитить хост от загруженных вредоносных программ не всегда легко. Проблема не столько в том, чтобы избежать загрузки программ. Вместо

этого мы ищем поддержку мобильного кода, который может предоставить доступ к локальным ресурсам гибким, но полностью контролируемым образом.

Одним из подходов к защите от потенциально вредоносного кода является создание **песочницы** (sandbox). Песочница – это метод, с помощью которого загруженная программа выполняется таким образом, что каждая из ее инструкций может полностью контролироваться. Если предпринята попытка выполнить инструкцию, запрещенную хостом, выполнение программы будет остановлено. Аналогично выполнение останавливается, когда инструкция обращается к определенным регистрам или областям в памяти, которые хост не разрешил.

Один из подходов к реализации песочницы заключается в проверке исполняемого кода при его загрузке и вставке дополнительных инструкций для ситуаций, которые могут быть проверены только во время выполнения [Wahbe et al., 1993]. Вопросы становятся намного проще при работе с интерпретируемым кодом. Давайте кратко рассмотрим подход, принятый в Java (см. также [Oaks, 2001]). Каждая Java-программа состоит из нескольких классов, из которых создаются объекты. Здесь нет глобальных переменных и функций; все должно быть объявлено как часть класса. Выполнение программы начинается с метода, называемого main. Программа на Java компилируется в набор инструкций, которые интерпретируются так называемой **виртуальной машиной Java** (Java Virtual Machine, JVM). Чтобы клиентский процесс выполнял JVM, клиенту необходима загрузка и выполнение скомпилированной программы Java. JVM будет обрабатывать фактическое выполнение загруженной программы путем интерпретации каждой из ее инструкций, начиная с инструкций, которые содержат main. Обратите внимание, что эта модель соответствует **процессу виртуальной машины** (process virtual machine), как обсуждалось в разделе 3.2.

В изолированной программной среде Java защита начинается с гарантии того, что компоненту, который обрабатывает передачу программы на клиентский компьютер, можно доверять. Загрузка в Java осуществляется набором **загрузчиков классов** (class loaders). Каждый загрузчик классов отвечает за выборку указанного класса с сервера и установку его в адресное пространство клиента, чтобы JVM могла создавать из него объекты. Загрузчики классов создаются на основе существующих доверенных загрузчиков, которые автоматически применяют определенные политики, связанные с песочницей.

Второй компонент песочницы Java состоит из **верификатора байтового кода** (byte code verifier), который проверяет, соответствует ли загруженный класс правилам безопасности песочницы. В частности, верификатор проверяет, что класс не содержит недопустимых инструкций или инструкций, которые могут как-то повредить стек либо память. Не все классы проверяются. На рис. 9.23 показано, что проверяются только те, которые были загружены клиенту с внешнего сервера. В этом случае классы, расположенные на компьютере клиента, являются доверенными. Проверка того, проверены ли локальные классы, – это еще одна политика, которую можно установить.

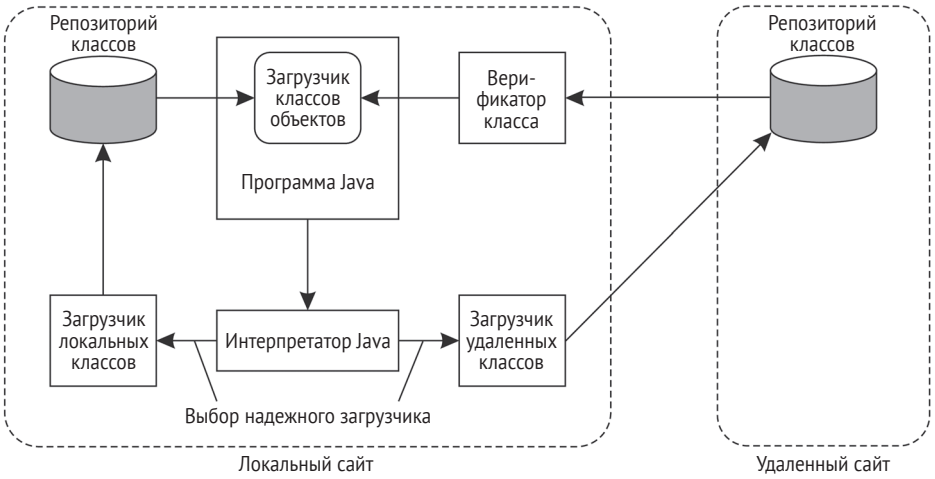


Рис. 9.23 ❖ Организация песочницы Java

Наконец, когда класс надежно загружен и проверен, JVM может создавать из него объекты и выполнять методы этих объектов. Чтобы фактически предотвратить объекты от несанкционированного доступа к ресурсам клиента, для выполнения различных проверок во время выполнения процедур используется **менеджер безопасности** (security manager). Java-программы, предназначенные для загрузки, вынуждены использовать менеджер безопасности; нет никакого способа, которым они могут обойти это. Менеджер безопасности, таким образом, играет роль эталонного монитора, как это обсуждалось ранее. Язык Java чрезвычайно гибок в настройке песочницы: менеджер безопасности использует обширный набор разрешений, которые могут быть установлены клиентом. Роль менеджера безопасности заключается в проверке разрешений, например файловых операций.

Следующим шагом в направлении повышения гибкости является требование аутентификации каждой загруженной программы и последующее применение определенной политики безопасности в зависимости от того, откуда пришла программа. Требовать, чтобы программы могли проходить проверку подлинности, относительно просто: мобильный код может быть подписан, как и любой другой документ. Этот подход к подписи кода часто применяется в качестве альтернативы песочнице. Фактически принимается только код с доверенных серверов.

Примечание 9.7 (дополнительно: применение политики безопасности)

Трудной частью является обеспечение соблюдения политики безопасности. В работе [Wallach et al., 1997] авторы предлагают три механизма в случае программ на Java. Первый подход основан на использовании ссылок на объекты в качестве возможностей. Чтобы получить доступ к локальному ресурсу, такому как файл, программе должна быть предоставлена ссылка на конкретный объект, который обрабатывает файловые операции при загрузке. Если такая ссылка не указана, то к файлам нет доступа. Этот принцип показан на рис. 9.24.

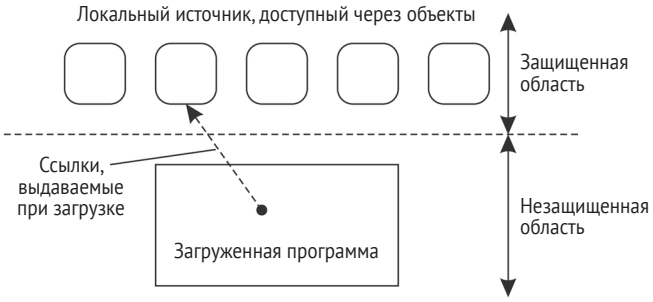


Рис. 9.24 ❖ Принцип использования ссылок на объекты Java в качестве возможностей

Все интерфейсы к объектам, которые реализуют файловую систему, изначально скрыты от программы, просто не раздавая никаких ссылок на эти интерфейсы. Строгая проверка типов в Java гарантирует, что невозможно создать ссылку на один из этих интерфейсов во время выполнения. Кроме того, мы можем использовать свойство Java, чтобы определенные переменные и методы были полностью внутренними для класса. В частности, программе может быть запрещено создавать свои собственные объекты обработки файлов, по сути скрывая операцию, которая создает новые объекты из данного класса. (В терминологии Java конструктор делается закрытым для своего связанного класса.)

Вторым механизмом реализации политики безопасности является **(расширенная) интроспекция (самоанализ) стека** ((extended) stack introspection). По сути, любому вызову метода m локального ресурса предшествует вызов специальной процедуры `enable_privilege`, который проверяет, имеет ли право вызывающая сторона вызывать m для этого ресурса. Если вызов авторизован, вызывающему предоставляются временные привилегии на время вызова. Прежде чем вернуть контроль вызывающему, когда m завершается, вызывается специальная процедура `disable_privilege`, чтобы отключить эти привилегии.

Для обеспечения выполнения вызовов для `enable_privilege` и `disable_privilege` может потребоваться разработчик интерфейсов для локальных ресурсов, чтобы вставить эти вызовы в подходящие места. Однако гораздо лучше позволить интерпретатору Java обрабатывать вызовы автоматически. Это стандартный подход, применяемый, например, в веб-браузерах для работы с апплетами Java. Элегантное решение заключается в следующем. При каждом вызове локального ресурса интерпретатор Java автоматически вызывает `enable_privilege`, который впоследствии проверяет, разрешен ли вызов. Если это так, вызов `disable_privilege` помещается в стек, чтобы гарантировать, что привилегии будут отключены при возврате вызова метода. Такой подход не позволяет вредоносным программистам обойти правила.

Еще одним важным преимуществом использования стека является то, что он позволяет гораздо лучше проверять привилегии. Предположим, что программа вызывает локальный объект O1, который, в свою очередь, вызывает объект O2. Хотя O1 может иметь разрешение на вызов O2, если вызывающему O1 не доверяют вызывать конкретный метод, принадлежащий O2, эта цепочка вызовов не должна быть разрешена. Самоанализ стека позволяет легко проверять такие цепочки, поскольку интерпретатору нужно просто проверить каждый кадр стека, начиная сверху, чтобы увидеть, есть ли кадр с активированными правами доступа (в этом случае вызов разрешен), или если есть кадр, который явно запрещает доступ к текущему ресурсу (в этом случае вызов немедленно прекращается). Данный подход показан на рис. 9.25.

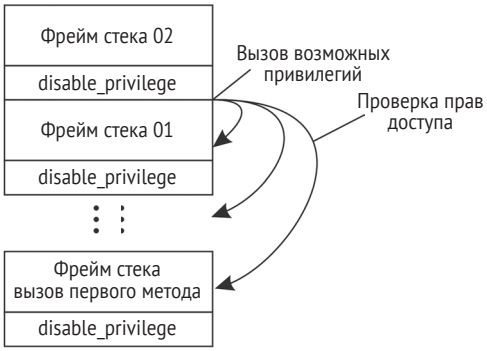


Рис. 9.25 ❖ Принцип самоанализа стека

По сути, интроспекция стека позволяет прикреплять привилегии к классам или методам и проверять эти привилегии для каждого вызывающего отдельно. Таким образом, можно реализовать домены защиты на основе классов, как подробно объясняется в [Gong and Schemers, 1998].

Третий подход к реализации политики безопасности заключается в **управлении пространством имен** (name space management). Чтобы предоставить программам доступ к локальным ресурсам, им в первую очередь необходимо получить доступ, включив соответствующие файлы, которые содержат классы, реализующие эти ресурсы. Включение требует, чтобы интерпретатору было присвоено имя, которое затем преобразовывает его в класс и впоследствии загружается во время выполнения программы. Чтобы обеспечить политику безопасности для конкретной загруженной программы, одно и то же имя может быть разрешено для разных классов в зависимости от того, откуда была загружена программа. Как правило, разрешение имен обрабатывается загрузчиками класса, которые необходимо адаптировать для реализации этого подхода. Подробности того, как это можно сделать, можно найти в [Wallach et al., 1997].

Описанный выше подход связывает привилегии с классами или методами, основанными на происхождении загруженной программы. Благодаря интерпретатору Java можно применять политики безопасности с помощью механизмов, описанных выше. В этом смысле архитектура безопасности становится сильно зависимой от языка, и ее необходимо будет заново разрабатывать для других языков. Независимые от языка решения, такие как, например, описанные в [Jaeger et al., 1999], требуют более общего подхода к обеспечению безопасности, а также сложнее в реализации. В этих случаях требуется поддержка со стороны защищенной операционной системы, которая знает о загруженном мобильном коде и обеспечивает выполнение всех обращений к локальным ресурсам для запуска через ядро, где выполняется последующая проверка.

Отказ в обслуживании

Контроль доступа обычно заключается в тщательном обеспечении доступа к ресурсам только авторизованным процессам. Особенно раздражающим типом атаки, связанной с контролем доступа, является злонамеренное предотвращение доступа авторизованных процессов к ресурсам. Защита от атак

типа **«отказ в обслуживании», DoS-атаки** (denial-of-service (DoS) attacks) становится все более важной, поскольку распределенные системы открываются через интернет. В тех случаях, когда DoS-атаки, исходящие из одного или нескольких источников, часто могут быть обработаны достаточно эффективно, вопросы становятся намного сложнее, когда приходится иметь дело с **распределенным отказом в обслуживании** (distributed denial of service, DDoS).

В DDoS-атаках огромная коллекция процессов совместно пытается отключить сетевой сервис. В этих случаях мы часто видим, что злоумышленникам удалось захватить большую группу машин, которые по незнанию участвуют в атаке. В работе [Smetters and Jacobson, 2009] различают атаки, направленные на истощение полосы пропускания, и атаки, направленные на истощение ресурсов.

Истощение полосы пропускания может быть достигнуто простым отправлением множества сообщений на одну машину. В результате обычные сообщения вряд ли смогут добраться до получателя. Атаки истощения ресурсов концентрируются на том, чтобы заставить получателя использовать ресурсы для бесполезных в других отношениях сообщений. Хорошо известной атакой с истощением ресурсов является TCP SYN-flooding. В этом случае злоумышленник пытается инициировать огромное количество соединений (то есть отправлять SYN (синхронные) пакеты как часть трехстороннего рукопожатия), но никогда не будет отвечать на подтверждения от получателя. В результате на сервере быстро заканчиваются дескрипторы сокетов, что не позволяет устанавливать какие-либо дальнейшие соединения.

Не существует единого метода защиты от DDoS-атак. Одна из проблем заключается в том, что злоумышленники используют невинных жертв, тайно устанавливая программное обеспечение на их машины, эффективно создавая так называемые **бот-сети** (botnets) [Silva et al., 2013]. В этих случаях единственное решение состоит в том, чтобы машины постоянно отслеживали свое состояние, проверяя файлы на предмет загрязнения. Учитывая легкость, с которой вирус может распространяться через интернет, полагаться только на это невозможно.

Гораздо лучше постоянно отслеживать сетевой трафик, например начиная с выходных маршрутизаторов, где пакеты покидают сеть организации. Опыт показывает, что, отбрасывая пакеты, чей адрес источника не принадлежит сети организации, можно предотвратить хаос. Как правило, чем больше пакетов можно отфильтровать вблизи источников, тем лучше.

В качестве альтернативы также можно сосредоточиться на входных маршрутизаторах, то есть там, где потоки попадают в сеть организации. Проблема состоит в том, что обнаружение атаки на входящем маршрутизаторе бывает слишком поздно, поскольку сеть, вероятно, уже будет недоступна для обычного трафика. Лучше, чтобы маршрутизаторы в интернете, например в сетях интернет-провайдеров, начинали отбрасывать пакеты, когда они подозревают, что атака продолжается. В целом необходимо применять много методов, поскольку новые атаки продолжают появляться. В работе [Zargar et al., 2013] дается практический обзор современного состояния атак и решений, связанных с отказом в обслуживании с сильным акцентом на атаки на уровне приложений (которые становятся все более распространенными).

Альтернативный обзор, концентрирующийся больше на решениях сетевого уровня, дан в [Peng et al., 2007].

9.4. Безопасное наименование

Тема, которой уделяется все больше внимания, связана с безопасными именами. Проще говоря: когда клиент получает объект на основе какого-либо имени, как он узнает, что вернул правильный объект? Вся проблема довольно фундаментальна: при разрешении имени в DNS как клиент узнает, что ему возвращен правильный адрес? При поиске объекта с использованием некоторой комбинации запроса URL и базы данных как получатель узнает, что он возвратил то, что было запрошено? Точнее, у нас есть три проблемы, о которых следует беспокоиться [Smetters and Jacobson, 2009]:

1) **достоверность** (Validity): возвращает ли объект полную неизмененную копию того, что было сохранено на сервере?
2) **происхождение** (Provenance): можно ли доверять серверу, который вернул объект как подлинного поставщика? Например, это может быть случай, когда клиенту возвращается кешированная версия исходного объекта;
3) **релевантность** (Relevance): уместно ли то, что было возвращено с учетом того, что было задано?

Частичное и хорошо известное решение для безопасного именования заключается в надежной привязке имени объекта к его содержимому с помощью хеширования. Проще говоря: примите *hash*(O) в качестве имени объекта O. Это форма **самоопределяющегося имени** (self-certifying name) и, как минимум, позволяет клиенту проверить правильность. Самоопределяющиеся имена были впервые введены в файл-серверах коллективного доступа (shared file server, SFS). Если мы можем предположить, что каждый объект O имеет известный и сертифицированный открытый ключ K_O^+, мы также можем принять *hash*(K_O^+) в качестве имени, имея преимущество в том, что в определенной степени происхождение можно проверить. Этот подход был исследован в GlobeDoc [Popescu et al., 2005].

Проблема с самоопределяющимися именами состоит в том, что они действуют как чистые идентификаторы и, в принципе, не очень дружелюбны к человеку. Кроме того, если мы просто примем хеш объекта в качестве имени, то каждое изменение этого объекта приведет к другому имени. Эта проблема несколько исправлена подходом в GlobeDoc, но затем нам нужно убедиться, что мы используем правильный открытый ключ, а также убедиться, что нам возвращается текущая версия объекта.

На практике, если предположить, что существует некоторая форма инфраструктуры открытых ключей, как мы обсудим позже в этой главе, в сочетании с возможностью эффективной проверки с помощью исходного сервера объекта, это должно решить проблему.

В результате как только мы захотим использовать дружественное для человека наименование объектов, мы сталкиваемся с проблемой безопасной

привязки такого имени, возможно, самоопределяющегося. Это порождает проблему невозможности проверки на релевантность. Возвращает ли процесс разрешения имен самоопределяемый, но нечитаемый человеком идентификатор того объекта, который мы искали?

Выход из положения, широко обсуждаемый в [Ghodsi et al., 2011], позволяет имени объекта O принимать форму (*hash*(K_O^+), *label*), где *label* (метка) – это понятное человеку имя, которое можно использовать для поиска объекта. Метка может быть просто тегом, но также и глобально уникальным иерархически организованным именем, таким как URL. Когда объект получен, он будет подписан сервером, как обсуждалось ранее. Получатель берет открытый ключ объекта, проверяет, что именно он использовался в имени, и впоследствии проверяет, является ли объект подлинным. Очевидно, что метка должна быть частью содержимого объекта, в противном случае будет невозможно проверить, что метка также принадлежит объекту. Например, серверу происхождения может потребоваться отдельно подписать метку тем же ключом, который используется для самого объекта.

9.5. Управление безопасностью

Теперь мы подробнее рассмотрим управление безопасностью. Во-первых, нам необходимо рассмотреть общее управление криптографическими ключами и в особенности способы распространения открытых ключей. Оказывается, сертификаты играют здесь важную роль. Во-вторых, мы обсуждаем проблему безопасного управления группой серверов, концентрируясь на проблеме добавления нового члена группы, которому доверяют текущие участники. Очевидно, что перед лицом распределенных и реплицируемых служб важно, чтобы безопасность не была скомпрометирована путем допуска в группу вредоносного процесса. В-третьих, мы обращаем внимание на управление авторизацией, рассматривая возможности и так называемые сертификаты атрибутов. Важной проблемой в распределенных системах в отношении управления авторизацией является то, что один процесс может делегировать некоторые или все свои права доступа другому процессу. Надежное делегирование прав имеет свои тонкости, о которых мы и поговорим в этом разделе.

Управление ключами

До сих пор мы описывали различные криптографические протоколы, в которых (неявно) предполагали, что различные ключи были легкодоступны. Например, в случае криптосистем с открытым ключом мы предполагали, что отправитель сообщения имел в своем распоряжении открытый ключ получателя, чтобы он мог зашифровать сообщение для обеспечения конфиденциальности. Аналогично в случае аутентификации с использованием центра распространения ключей (KDC) мы предполагали, что каждая сторона уже поделилась секретным ключом с KDC.

Однако создание и распространение ключей не являются тривиальным вопросом. Например, о рассылке секретных ключей по незащищенному каналу не может быть и речи, и во многих случаях нам необходимо прибегать к внеполосным методам. Кроме того, необходимы механизмы для отзыва ключей, то есть для предотвращения использования ключа после его взлома или аннулирования.

Установка ключей

Давайте начнем с рассмотрения того, как можно установить сеансовые ключи. Когда Алиса хочет установить безопасный канал с Бобом, она может сначала использовать открытый ключ Боба для инициирования связи, как показано на рис. 9.13. Если Боб соглашается, он может впоследствии сгенерировать ключ сеанса и вернуть его Алисе, зашифрованный открытым ключом Алисы. Зашифрованный общий ключ сеанса перед его передачей может безопасно передаваться по сети.

Аналогичная схема может быть использована для генерации и распространения сеансового ключа, когда Алиса и Боб уже совместно используют секретный ключ. Тем не менее оба метода требуют, чтобы у связывающихся сторон уже были средства, доступные для установления безопасного канала. Другими словами, определенная форма установления и распределения ключей уже должна была иметь место. Тот же аргумент применяется, когда общий секретный ключ устанавливается с помощью доверенной третьей стороны, такой как KDC.

Элегантной и широко применяемой схемой для установки общего ключа по небезопасному каналу является **обмен ключами Диффи–Хеллмана** (Diffie-Hellman key exchange) [Diffie and Hellman, 1976]. Протокол работает следующим образом. Предположим, что Алиса и Боб хотят установить общий секретный ключ. Первое требование состоит в том, чтобы они согласовали два больших числа, n и g, которые подчиняются ряду математических свойств (которые мы здесь не обсуждаем). И n, и g могут быть обнародованы; нет необходимости прятать их от посторонних. Алиса выбирает большое случайное число, скажем x, которое она хранит в секрете. Точно так же Боб выбирает свой собственный большой секрет, скажем y. На данный момент достаточно информации для создания секретного ключа, как показано на рис. 9.26.

Алиса начинает с отправки $g^x \bmod n$ Бобу вместе с n и g. Важно отметить, что эта информация может быть отправлена в виде открытого текста, поскольку фактически невозможно вычислить x с учетом $g^x \bmod n$.

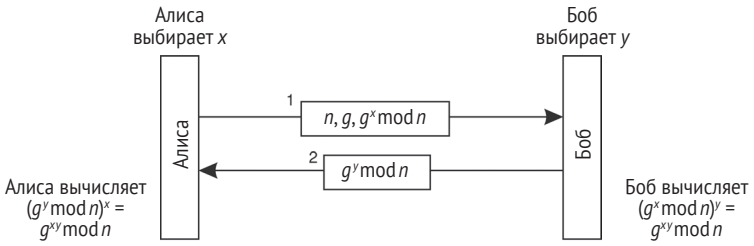


Рис. 9.26 ❖ Принцип обмена ключами Диффи–Хеллмана

Когда Боб получает сообщение, он вычисляет $(g^x \bmod n)^y$, который математически равен $g^{xy} \bmod n$. Кроме того, он отправляет $g^y \bmod n$ Алисе, которая затем может вычислить $(g^y \bmod n)^x = g^{xy} \bmod n$. И Алиса, и Боб, и только эти двое, теперь будут иметь общий секретный ключ $g^{xy} \bmod n$. Обратите внимание, что ни одному из них не нужно было сообщать свои закрытые числа (x и y соответственно) другому.

Ключи Диффи–Хеллмана можно рассматривать как криптосистему с открытым ключом. В случае Алисы x – это ее закрытый ключ, а $g^x \bmod n$ – ее открытый ключ. Для работы системы Диффи–Хеллмана необходимо надежное распространение открытого ключа

Распространение ключей

Одним из наиболее сложных аспектов в управлении ключами является фактическое распространение начальных ключей. В симметричной криптосистеме начальный общий секретный ключ должен передаваться по безопасному каналу, который обеспечивает аутентификацию, а также конфиденциальность, как показано на рис. 9.27. Если Алисе и Бобу не доступны ключи для настройки такого безопасного канала, необходимо распространить ключ вне диапазона. Другими словами, Алисе и Бобу придется связываться друг с другом, используя другой канал связи.

В случае криптосистемы с открытым ключом нам необходимо распространить открытый ключ таким образом, чтобы получатели могли быть уверены, что ключ действительно связан с заявленным закрытым ключом. Другими словами, как показано на рис. 9.27, хотя сам открытый ключ может быть отправлен в виде открытого текста, необходимо, чтобы канал, по которому он отправляется, мог обеспечивать аутентификацию. Закрытый ключ, конечно, должен быть отправлен по безопасному каналу, обеспечивая аутентификацию, а также конфиденциальность.

Когда дело доходит до распространения ключей, распространение открытых ключей с проверкой подлинности является, пожалуй, самым интересным. На практике распространение открытых ключей осуществляется посредством **сертификатов открытых ключей** (public-key certificates). Такой сертификат состоит из открытого ключа вместе со строкой, идентифицирующей объект, с которым связан этот ключ. Объект может быть пользователем, но также хостом или каким-то специальным устройством. Открытый ключ и идентификатор были вместе подписаны **органом сертификации** (certification authority), и эта подпись помещается в сертификат (идентификационные данные органа сертификации, естественно, являются частью сертификата). Подписание происходит с помощью закрытого ключа K_{CA}^-, который принадлежит органу по сертификации. Соответствующий открытый ключ K_{CA}^+ предполагается общеизвестным. Например, открытые ключи различных органов сертификации встроены в большинство веб-браузеров и поставляются с двоичными файлами.

Использование сертификата с открытым ключом работает следующим образом. Предположим, что клиент желает удостовериться, что открытый ключ, найденный в сертификате, действительно принадлежит идентифици-

рованному объекту. Затем он использует открытый ключ соответствующего органа по сертификации для проверки подписи сертификата. Если подпись на сертификате соответствует (открытый ключ, идентификатор), клиент соглашается с тем, что открытый ключ действительно принадлежит идентифицированному объекту.

Важно отметить, что, принимая сертификат в порядке, клиент фактически полагает, что сертификат не был подделан. В частности, клиент должен предположить, что открытый ключ K^{+}_{CA} действительно принадлежит соответствующему органу сертификации. В случае сомнений должна быть возможность проверить достоверность K^{+}_{CA} с помощью другого сертификата, полученного от иного, возможно, более доверенного органа сертификации.

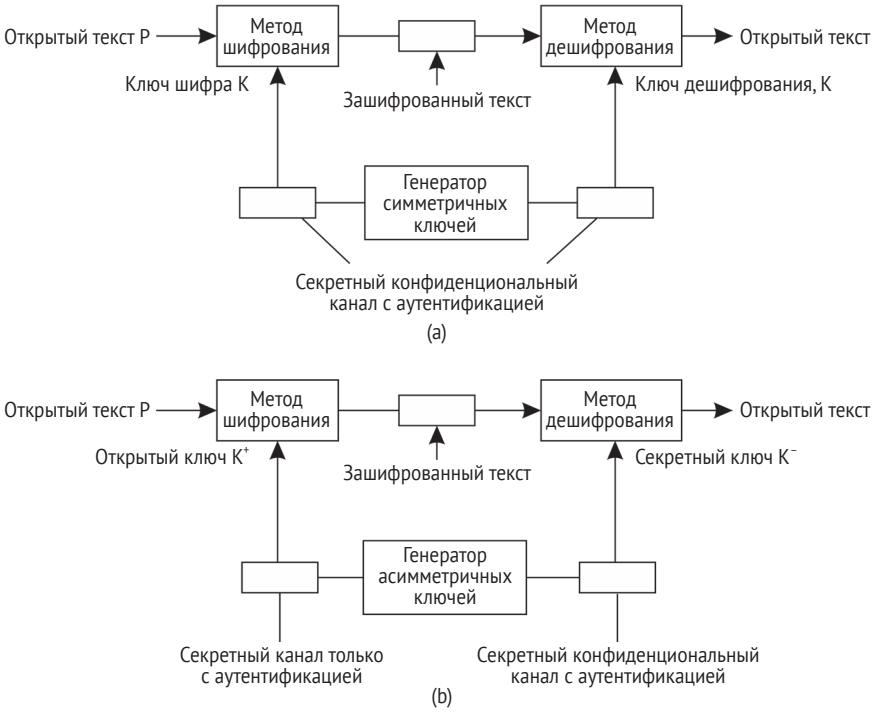


Рис. 9.27 ❖ Распространение:
a) секретных ключей; b) открытых ключей (см. также [Menezes et al., 1996])

Примечание 9.8 (дополнительная информация: срок действия сертификатов)

Важным вопросом, касающимся сертификатов, является их долговечность. Сначала давайте рассмотрим ситуацию, в которой орган по сертификации выдает пожизненные сертификаты. По существу, в сертификате говорится, что открытый ключ всегда будет действителен для объекта, идентифицируемого сертификатом. Понятно, что такое утверждение совсем не то, что мы хотим. Если закрытый ключ идентифицируемого объекта когда-либо скомпрометирован, ни один ничего не подозревающий клиент не сможет использовать открытый ключ (не говоря уже о вре-

доносных клиентах). В этом случае нам нужен механизм для отзыва сертификата, сделав публично известной информацию, что сертификат больше не действителен.

Есть несколько способов отозвать сертификат. Один из распространенных подходов – использование **списка отзыва сертификатов** (Certificate Revocation List, CRL), регулярно публикуемого органом по сертификации. Всякий раз, когда клиент проверяет сертификат, он должен будет проверить CRL, чтобы увидеть, был сертификат отозван или нет. Это означает, что клиент должен будет, по крайней мере, обращаться к органу сертификации каждый раз, когда публикуется новый CRL. Обратите внимание, что если CRL публикуется ежедневно, то для отзыва сертификата также потребуется день. Между тем скомпрометированный сертификат может быть ложно использован до тех пор, пока он не будет опубликован в следующем CRL. Следовательно, время между публикацией списков отзыва сертификатов не может быть слишком продолжительным.

Альтернативный подход заключается в ограничении срока действия каждого сертификата. По сути, этот подход аналогичен выдаче аренды, как мы обсуждали в разделе 7.4. Срок действия сертификата автоматически истекает через некоторое время. Если по какой-либо причине сертификат должен быть отозван до истечения срока его действия, орган по сертификации все еще может опубликовать его в CRL. Однако этот подход все равно заставит клиентов проверять последний CRL при проверке сертификата.

Наконец, крайний случай – сократить время жизни сертификата почти до нуля. Фактически это означает, что сертификаты больше не используются; вместо этого клиент всегда должен будет обратиться в орган по сертификации, чтобы проверить действительность открытого ключа. Как следствие центр сертификации должен постоянно находиться в сети.

На практике сертификаты выдаются с ограниченным сроком действия. В случае интернет-приложений время истечения часто составляет целый год [Stein, 1998]. Такой подход требует, чтобы CRL публиковались регулярно, но также проверялись при проверке сертификатов. Практика показывает, что клиентские приложения почти никогда не обращаются к CRL и просто предполагают, что сертификат действует до тех пор, пока не отозван.

Безопасное управление группами

Многие системы безопасности используют специальные службы, такие как центры распространения ключей (KDC) или центры сертификации (CA). Эти сервисы демонстрируют сложность проблемы обеспечения безопасности в распределенных системах. Во-первых, необходимо доверие. Чтобы повысить доверие к службам безопасности, необходимо обеспечить высокую степень защиты от всех видов угроз безопасности. Например, как только CA был скомпрометирован, становится невозможным проверить достоверность открытого ключа, делая всю систему безопасности бесполезной.

С другой стороны, также необходимо, чтобы многие службы безопасности обеспечивали высокую доступность. Например, в случае KDC каждый раз, когда два процесса хотят установить безопасный канал, по крайней мере одному из них потребуется связаться с KDC для получения общего секретного ключа. Если KDC недоступен, безопасная связь не может быть установлена, если не доступен альтернативный метод для установления ключа, такой как обмен ключами по Диффи–Хеллману.

Решением задачи высокой доступности является репликация. С другой стороны, репликация делает сервер более уязвимым для атак безопасности. Мы уже обсуждали, как безопасное групповое общение может происходить путем обмена секретом между членами группы. По сути, ни один член группы не способен компрометировать сертификаты, что делает саму группу очень защищенной.

Примечание 9.9 (дополнительно: управление членством в группе)

Остается рассмотреть, как на самом деле управлять группой реплицированных серверов. В работе [Reiter et al., 1994] авторы предлагают следующее решение. Проблема, которая должна быть решена, состоит в том, чтобы гарантировать, что когда процесс просит присоединиться к группе G, целостность группы не ставится под угрозу. Предполагается, что группа G использует секретный ключ CK_G, общий для всех членов группы, для шифрования групповых сообщений. Кроме того, он также использует для связи с членами группы пару открытый/закрытый ключ (K_G^+, K_G^-).

Всякий раз, когда процесс P хочет присоединиться к группе G, он отправляет запрос на присоединение JR (join request), идентифицирующий G и P, местное время процесса P *T*, сгенерированный панелью ответа RP (reply pad) и сгенерированный секретный ключ $K_{P,G}$. RP и $K_{P,G}$ совместно шифруются с использованием открытого ключа группы K_G^+, как показано в сообщении 1 на рис. 9.28. Использование RP и $K_{P,G}$ объясняется более подробно ниже. Запрос на присоединение JR подписывается P и отправляется вместе с сертификатом, содержащим открытый ключ P. Мы использовали широко применяемое обозначение $[M]_A$ для обозначения того, что сообщение M было подписано субъектом A.

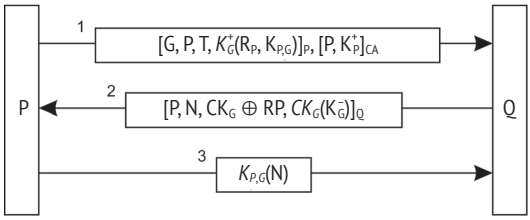


Рис. 9.28 ❖ Безопасный прием нового члена группы

Когда член группы Q получает такой запрос на присоединение, он сначала аутентифицирует P, после чего происходит связь с другими членами группы, чтобы увидеть, может ли P быть допущен в качестве члена группы. Аутентификация P осуществляется обычным способом с помощью сертификата. Временная метка *T* используется, чтобы удостовериться, что сертификат был еще действителен во время его отправки. (Обратите внимание, что мы должны быть уверены, что время также не было подделано.) Член группы Q проверяет подпись органа по сертификации и впоследствии извлекает из сертификата общедоступную информацию о ключе P, чтобы проверить действительность JR. В этот момент специфичный для группы протокол показывает, все ли члены группы согласны принять P.

Если P разрешено присоединиться к группе, Q возвращает сообщение GA о входе в группу, показанное как сообщение 2 на рис. 9.28, идентифицирующее P и содержащее одноразовый номер *N*.

Панель ответа RP используется для шифрования ключа связи группы CK_G. Кроме того, P также понадобится закрытый ключ группы K_G^-, который зашифрован с помощью CK_G. Сообщение GA впоследствии подписывается Q с использованием ключа $K_{P,G}$.

Процесс P теперь может аутентифицировать Q, потому что только истинный член группы может открыть секретный ключ $K_{P,G}$. Одноразовый номер *N* в этом протоколе не используется для безопасности; вместо этого, когда P отправляет обратно *N*, зашифрованный с помощью $K_{P,G}$ (сообщение 3), Q знает, что P получил все необходимые ключи и, следовательно, теперь действительно присоединился к группе.

Обратите внимание, что вместо использования панели ответа RP, P и Q могли также сами зашифровать CKG, применяя открытый ключ P. Однако, поскольку RP используется только один раз, и именно для шифрования ключа связи группы в сообщении GA, использование RP более безопасно. Если закрытый ключ P когда-либо будет раскрыт, станет возможным также выявить CK_G, что поставит под угрозу секретность всех групповых коммуникаций.

Управление авторизацией

Управление безопасностью в распределенных системах также связано с управлением правами доступа. До сих пор мы почти не касались вопроса о том, как права доступа первоначально предоставляются пользователям или группам пользователей и как они впоследствии поддерживаются не поддающимся подделке способом. Настало время исправить это упущение.

В нераспределенных системах управление правами доступа относительно просто. Когда новый пользователь добавляется в систему, ему предоставляются начальные права, например на создание файлов и подкаталогов в определенном каталоге, создание процессов, использование времени ЦП и т. д. Другими словами, полная учетная запись для пользователя настраивается для одной конкретной машины, в которой все права были заранее определены системными администраторами.

В распределенной системе все осложняется тем, что ресурсы распределены по нескольким машинам. Если следовать этому подходу, необходимо создать учетную запись для каждого пользователя на каждом компьютере. По сути, такой подход и используется в сетевых операционных системах. Задачу можно немного упростить, создав одну учетную запись на центральном сервере. Этот сервер опрашивают каждый раз, когда пользователь обращается к определенным ресурсам или машинам.

Возможности и атрибуты

Гораздо лучшим подходом, который широко применяется в распределенных системах, является использование возможностей. Возможность – это не поддающаяся обработке структура данных для конкретного ресурса, точно определяющая права доступа, которые держатель возможности имеет в отношении этого ресурса. Существуют разные реализации возможностей. Здесь мы кратко обсудим реализацию, используемую в операционной системе Amoeba [Tanenbaum et al., 1986]. Несмотря на то что это пример из давнего

времени, простота делает его отличным кандидатом для понимания основных принципов.

Система Amoeba (Амёба) была одной из первых объектно-ориентированных распределенных систем. Объект находится на сервере, а прозрачный доступ клиентам предоставляется с помощью прокси. Чтобы вызвать операцию над объектом, клиент передает возможность своей локальной операционной системе, которая затем находит сервер, на котором находится объект, и потом выполняет процедуру удаленного вызова (RPC) для этого сервера.

Возможность представляет собой 128-битный идентификатор, внутренне организованный, как показано на рис. 9.29. Первые 48 бит инициализируются сервером объекта при создании объекта и фактически образуют машинно независимый идентификатор сервера объекта, называемый **портом сервера** (**server port**).

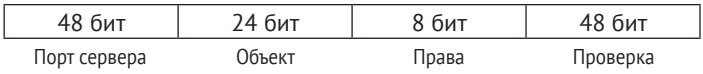


Рис. 9.29 ❖ Возможность в системе Amoeba

Следующие 24 бита используются для идентификации объекта на данном сервере. (Обратите внимание, что порт сервера вместе с идентификатором объекта образует 72-разрядный уникальный системный идентификатор.) Следующие 8 бит используются для указания прав доступа держателя возможности. Наконец, 48-битное поле проверки применяется, чтобы сделать возможность не поддающейся подделке, как мы объясним ниже.

Когда объект создается, его сервер выбирает случайное поле проверки и сохраняет его как в возможности, так и внутри своих собственных таблиц. Все правильные биты в новой возможности изначально включены, и именно эта возможность владельца возвращается клиенту. Когда возможность отправляется обратно на сервер в запросе на выполнение операции, проверяется поле проверки.

Примечание 9.10 (дополнительно: ограниченные возможности)

Чтобы создать ограниченную возможность, клиент может передать ее обратно на сервер вместе с битовой маской для новых прав. Сервер берет исходное поле проверки из своих таблиц, *XOR* (эксклюзивное «или») присваивает ему новые права (которые должны быть подмножеством прав в возможности) и пропускает результат через одностороннюю функцию.

Затем сервер создает новую возможность с тем же значением в поле объекта, но с новыми битами прав в поле прав и выходом односторонней функции в поле проверки. Новая возможность потом возвращается вызывающей стороне. Клиент, если пожелает, может отправить эту новую возможность другому процессу.

Способ создания ограниченных возможностей показан на рис. 9.30. В этом примере владелец отключил все права, кроме одного. Например, ограниченная возможность может разрешить чтение объекта, но не более того.

Значение поля прав различно для каждого типа объекта, поскольку сами юридические операции также различаются от типа к типу объекта.

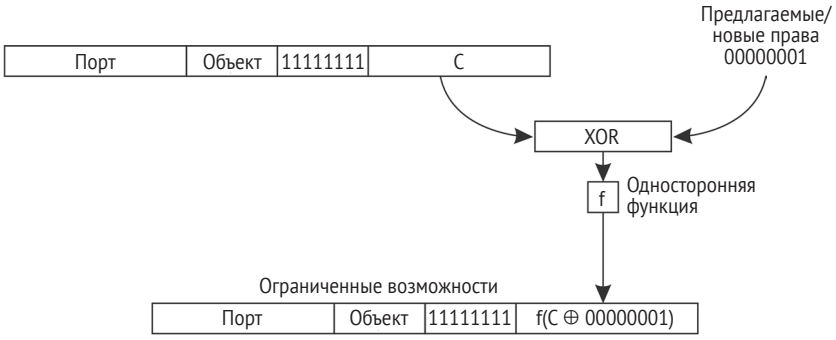


Рис. 9.30 ❖ Генерация ограниченной возможности из способности владельца

Когда ограниченная возможность возвращается на сервер, сервер видит из поля прав, что владелец не обладает этой возможностью, потому что по крайней мере один бит отключен. Затем сервер выбирает исходное случайное число из своих таблиц, выполняет *XOR* с полем прав из возможности и пропускает результат через одностороннюю функцию. Если результат совпадает с полем проверки, возможность принимается.

Из этого алгоритма должно быть очевидно, что пользователь, который пытается добавить права, которых у него нет, просто лишается этой возможности. Обращение контрольного поля в ограниченной возможности получить аргумент (C *XOR* 00000001) на рис. 9.30 невозможно, потому что функция f является односторонней функцией. Именно с помощью этой криптографической техники возможности защищены от взлома. Обратите внимание, что f по сути делает то же самое, что и вычисление дайджеста сообщения, как обсуждалось ранее. Изменение чего-либо в исходном сообщении (например, инвертирование бита) будет немедленно обнаружено.

Обобщение возможностей, которое иногда используется в распределенных системах, является **атрибутом сертификата** (attribute certificate). В отличие от рассмотренных выше сертификатов, которые используются для проверки действительности открытого ключа, сертификаты атрибутов используются для перечисления определенных пар (*атрибут, значение*), которые применяются к идентифицированному объекту. В частности, сертификаты атрибутов могут использоваться для перечисления прав доступа, которые имеет владелец сертификата в отношении идентифицированного ресурса.

Как и другие сертификаты, сертификаты атрибутов выдаются специальными органами сертификации, обычно называемыми **органами сертификации атрибутов** (attribute certification authoritise). По сравнению с возможностями системы Amoeba, такие полномочия соответствуют серверу объекта. В целом, однако, орган по сертификации атрибутов и сервер, управляющий объектом, для которого был создан сертификат, не обязательно должны совпадать. Права доступа, указанные в сертификате, подписаны органом по сертификации атрибутов. Более подробно об эволюции управления доступом на основе возможностей обсуждается в [Karp et al., 2010] и [Franqueira and Wieringa, 2012].

Делегирование (прав)

Теперь рассмотрим следующую проблему. Пользователь хочет напечатать большой файл, для которого у него есть права доступа только для чтения. Чтобы не беспокоить окружающих, пользователь отправляет запрос на сервер печати с просьбой начать печать файла не ранее 2 часов утра. Вместо того чтобы отправлять весь файл на принтер, пользователь передает имя файла на принтер, чтобы он мог скопировать его в каталог спулинга, если это действительно необходимо.

Хотя эта схема, кажется, должна работать, есть одна главная проблема: принтер, как правило, не имеет соответствующих разрешений на доступ к указанному файлу. Другими словами, если никакие специальные меры не предпринимаются, как только сервер печати захочет прочитать файл, чтобы распечатать его, система откажет серверу в доступе к файлу. Эту проблему можно было бы решить, если бы пользователь временно **делегировал** свои права доступа к файлу на сервер печати.

Делегирование прав доступа является важным аспектом реализации защиты, в частности в компьютерных системах и распределенных системах. Основная идея проста: передавая определенные права доступа от одного процесса другому, становится легче распределять работу между несколькими процессами, не оказывая негативного влияния на защиту ресурсов. В случае распределенных систем процессы могут выполняться на разных компьютерах и даже в разных административных доменах. Делегирование может помочь избежать больших накладных расходов, поскольку защита часто может осуществляться локально.

Есть несколько способов реализовать делегирование. Общий подход, описанный Нейманом [Neuman, 1993] и реализованный в системе Kerberos, заключается в использовании прокси. **Прокси** (proxy) в контексте безопасности в компьютерных системах – это токен, который позволяет его владельцу работать с теми же или ограниченными правами и привилегиями, что и субъект, предоставивший токен. (Обратите внимание, что это понятие прокси отличается от прокси в качестве синонима заглушки на стороне клиента. Хотя мы стараемся избегать перегрузки терминов, здесь мы делаем исключение, поскольку термин «прокси» в приведенном выше определении слишком широко используется, не следует это игнорировать.) Процесс может создать прокси в лучшем случае с правами и привилегиями, которые он имеет сам. Если процесс создает новый прокси на основе того, который у него есть в настоящее время, производный прокси будет иметь как минимум те же ограничения, что и исходный, и, возможно, больше.

Прежде чем рассматривать общую схему делегирования, рассмотрим следующие два подхода. Во-первых, делегирование относительно просто, если Алиса всех знает. Если она хочет делегировать права Бобу, ей просто нужно создать сертификат, говорящий «Алиса говорит, что у Боба есть права R», например $[A, B, R]_A$. Если Боб хочет передать некоторые из этих прав Чарли, он попросит Чарли связаться с Алисой и попросить у нее соответствующий сертификат.

Во втором простом случае Алиса может просто сконструировать сертификат, говорящий: «Носитель этого сертификата имеет права R». Однако в этом

случае нам необходимо защитить сертификат от нелегального копирования, как это делается для безопасного обмена возможностями между процессами. Прокси в схеме Неймана позволяет избежать проблемы, связанной с тем, что Алиса должна знать всех, кому необходимо делегировать права.

Прокси в схеме Неймана состоит из двух частей, как показано на рис. 9.31. Пусть A будет процессом, который создал прокси. Первая часть прокси-сервера представляет собой набор $C = R, S^{+}_{\text{рогху}}$, состоящий из набора R прав доступа, которые были делегированы A, вместе с общеизвестной частью секрета, который используется для аутентификации владельца сертификата. Мы объясним использование $C = R, S^{+}_{\text{рогху}}$ ниже. Сертификат имеет сигнатуру подписи *sig*(A, C) от A, чтобы защитить его от изменений. Вторая часть содержит другую часть секрета, обозначенную как $S^{-}_{\text{рогху}}$. Важно, чтобы $S^{-}_{\text{рогху}}$ был защищен от разглашения при делегировании прав другому процессу.

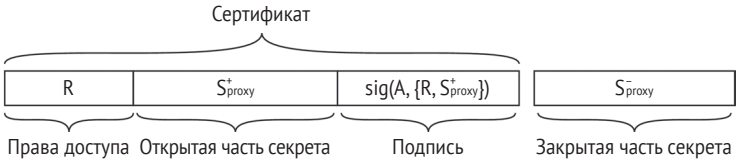


Рис. 9.31 ❖ Общая структура прокси, используемая для делегирования

Другой подход к работе с прокси-сервером заключается в следующем. Если Алиса хочет передать некоторые из своих прав Бобу, она составляет список прав (R), которые Боб может использовать. Подписывая список, она не позволяет Бобу в это вмешиваться. Однако наличие только подписанного списка прав часто недостаточно. Если Боб хочет воспользоваться своими правами, ему, возможно, придется доказать, что он действительно получил список от Алисы или, скажем, не украл его у кого-то еще. Поэтому Алиса задает очень неприятный вопрос ($S^{+}_{\text{рогху}}$), ответ на который знает только она ($S^{-}_{\text{рогху}}$). Любой желающий может легко проверить правильность ответа на вопрос. Вопрос добавляется в список до того, как Алиса добавляет свою подпись.

При делегировании некоторых своих прав Алиса передает подписанный список прав вместе с неприятным вопросом Бобу. Она также дает Бобу ответ, гарантируя, что никто не сможет его перехватить. Теперь у Боба есть список прав, подписанный Алисой, который он может передать, скажем, Чарли, когда это необходимо. Чарли задаст ему неприятный вопрос внизу списка. Если Боб знает ответ на него, Чарли будет знать, что Алиса действительно передала перечисленные права Бобу.

Важным свойством этой схемы является то, что с Алисой не нужно консультироваться. Фактически Боб может решить передать (некоторые из) права в списке Дейву. При этом он также сообщит Дэйву ответ на вопрос, чтобы Дейв смог доказать, что список был передан ему кем-то, имеющим на это право. Алисе вообще не нужно знать о Дейве.

Протокол для делегирования и реализации прав показан на рис. 9.32. Предположим, что Алиса и Боб имеют общий секретный ключ $K_{A,B}$, который можно использовать для шифрования сообщений, которые они отправляют

друг другу. Затем Алиса сначала отправляет Бобу сертификат C = R, $S^{+}_{\text{рогху}}$, подписанный *sig*(A, C) (и снова обозначенный как $[R, S^{+}_{\text{рогху}}]_A$). Нет необходимости шифровать это сообщение: оно может быть отправлено в виде открытого текста. Только зашифрованная часть секрета должна быть зашифрована и показана как $K_{A,B}(S^{-}_{\text{рогху}})$ в сообщении 1.

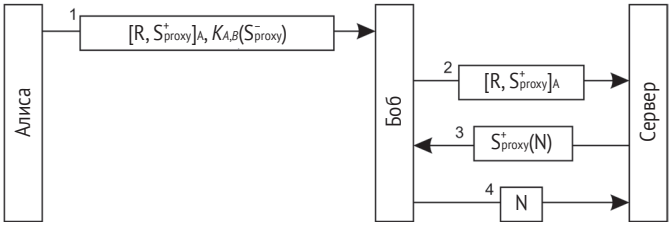


Рис. 9.32 ❖ Использование прокси для делегирования и подтверждения прав доступа

Теперь предположим, что Боб хочет, чтобы операция выполнялась на объекте, который находится на конкретном сервере. Также предположим, что Алиса уполномочена на выполнение этой операции и что она передала эти права Бобу. Поэтому Боб передает свои учетные данные на сервер в виде подписанного сертификата $[R, S^{+}_{\text{рогху}}]_A$.

В этот момент сервер сможет проверить, что C не был подделан: любые изменения в списке прав или неприятный вопрос будут замечены, потому что оба были совместно подписаны Алисой. Однако сервер еще не знает, является ли Боб законным владельцем сертификата. Чтобы убедиться в этом, сервер должен использовать секрет, поставляемый с C.

Существует несколько способов реализации $S^{+}_{\text{рогху}}$ и $S^{-}_{\text{рогху}}$. Например, предположим, что $S^{+}_{\text{рогху}}$ – это открытый ключ, а $S^{-}_{\text{рогху}}$ – соответствующий закрытый ключ. Z может затем бросить вызов Бобу, отправив ему одноразовый номер *N*, зашифрованный с помощью $S^{+}_{\text{рогху}}$. Расшифровывая $S^{+}_{\text{рогху}}(N)$ и возвращая *N*, Боб доказывает, что он знает секрет и, таким образом, является законным владельцем сертификата. Есть и другие способы реализации безопасного делегирования, но основная идея всегда такова: покажите, что знаете секрет.

9.6. Резюме

Безопасность играет чрезвычайно важную роль в распределенных системах. Распределенная система должна обеспечивать механизмы, которые позволяют применять различные политики безопасности. Разработка и правильное применение этих механизмов обычно делают безопасность сложной инженерной задачей.

Можно выделить три важные проблемы. Первая проблема состоит в том, что распределенная система должна предлагать средства для установления безопасных каналов между процессами. Безопасный канал, в принципе,

предоставляет средства для взаимной аутентификации взаимодействующих сторон и защиты сообщений от несанкционированного доступа во время их передачи. Безопасный канал обычно также обеспечивает конфиденциальность, не позволяющую никому, кроме сообщающихся сторон, читать сообщения, проходящие через канал.

При проектировании важно в основном использовать симметричную криптосистему (основанную на общих секретных ключах) или ее комбинацию с криптосистемой с открытым ключом.

Вторая проблема в безопасных распределенных системах – это контроль доступа или авторизация. Авторизация имеет дело с защитой ресурсов таким образом, что только процессы, которые имеют надлежащие права доступа, могут осуществлять фактический доступ и использовать эти ресурсы. Контроль доступа всегда осуществляется после аутентификации процесса. С контролем доступа связано предотвращение отказа в обслуживании, что является сложной проблемой для систем, доступных через интернет.

Существует два способа реализации контроля доступа. В первом каждый ресурс может поддерживать список контроля доступа, в котором точно перечислены права доступа каждого пользователя или процесса. В качестве альтернативы процесс может иметь сертификат, в котором точно указываются его права на определенный набор ресурсов. Основное преимущество использования сертификатов состоит в том, что процесс может легко передать свой билет другому процессу, то есть делегировать свои права доступа. Сертификаты, однако, имеют тот недостаток, что их часто трудно отозвать.

Особое внимание требуется при работе с контролем доступа в случае мобильного кода. Было сделано несколько предложений, из которых «песочница» в настоящее время является наиболее широко применяемой. Однако «песочницы» имеют ряд ограничений, и были разработаны более гибкие подходы, учитывающие реальные проблемы защиты.

Третья проблема безопасных распределенных систем касается управления. Существуют два важных аспекта: управление ключами и управление авторизацией. Управление ключами включает распространение криптографических ключей, в котором сертификаты, выданные доверенными третьими сторонами, играют важную роль. В отношении управления авторизацией важны сертификаты атрибутов и делегирование.

Наконец, особое внимание требует обработка безопасных имен. Практическое решение – присвоить объекту имя, взяв хеш его открытого ключа вместе с удобочитаемой меткой (которая также должна быть надежно привязана к объекту).

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Мартен ван Стин, Эндрю С. Таненбаум

Распределенные системы

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Яроцкий В. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 47,45. Тираж 200 экз.

Отпечатано в ПАО «Т8 Издательские Технологии»
109316, Москва, Волгоградский проспект, д. 42, корпус 5

Веб-сайт издательства: **www.dmkpress.com**