

ЛАРРИ ПЕТЕРСОН
БРЮС ДЭВИ

КОМПЬЮТЕРНЫЕ СЕТИ

6-е издание

системный
подход



ИЗДАТЕЛЬСТВО «АСТ»
МОСКВА

УДК 004.7
ББК 32.973.202
П29

*Книга «Computer Networks: A Systems Approach, Sixth Edition»
на английском языке доступна для свободного скачивания по ссылке:
<https://www.dbooks.org/computer-networks-0128182008>.
Эта открытая книга распространяется по лицензии
Creative Commons (CC BY).
Исходный код книги доступен по адресу
<https://github.com/SystemsApproach/book>
на условиях лицензии Creative Commons (CC BY 4.0)*

Петерсон, Ларри.

П29 Компьютерные сети: системный подход / Ларри Петерсон, Брюс Дэви. — 6-е изд. — Москва : Издательство АСТ, 2026. — 480 с. : ил. — (Программирование для всех).

ISBN 978-5-17-162231-2.

В этой книге, ставшей мировым бестселлером, авторы на примере реальных заданий по проектированию компьютерных сетей подробно объясняют последние технологические новации в данной сфере, используя системно-ориентированный подход. Здесь приведены актуальные на сегодняшний день примеры поэтапной разработки и создания компьютерных сетей, из текста исключены анахронизмы (например, dial-up modem), рассмотрено использование широко распространенных приложений (Netflix, Spotify) для актуализации затронутых проблем, а также обновлены цифры, отражающие современное состояние сетевых технологий. Данное издание поможет начинающим IT-инженерам и программистам получить представление о том, как отдельные компоненты сети вписываются в более крупную и сложную систему взаимодействий, кроме того, книга способствует формированию инженерной интуиции, что в нашу эпоху стремительных технологических изменений имеет решающее значение для развития способности специалиста данной отрасли принимать обоснованные решения о том, как проектировать или выбирать системы следующего поколения.

**УДК 004.7
ББК 32.973.202**

ISBN 978-5-17-162231-2

© Larry L. Peterson, Bruce S. Davie, 2019
Перевод на русский язык: ООО «Интеджер».
Издание на русском языке: ООО «Издательство АСТ».

Содержание

Раздел 1. Основы	7
Глава 1.1. Приложения	8
Глава 1.2. Требования	10
Глава 1.3. Архитектура	23
Глава 1.4. Программное обеспечение	31
Глава 1.5. Производительность	36
Раздел 2. Прямые ссылки	46
Глава 2.1. Технологический ландшафт	47
Глава 2.2. Кодирование	50
Глава 2.3. Фрейминг (кадрирование)	53
Глава 2.4. Обнаружение ошибок	59
Глава 2.5. Надежная передача	65
Глава 2.6. Сети с множественным доступом	76
Глава 2.7. Беспроводные сети	83
Глава 2.8. Сети доступа	94
Раздел 3. Межсетевое взаимодействие	102
Глава 3.1. Основы коммутации	103
Глава 3.2. Коммутируемый Ethernet	115
Глава 3.3. Интернет (IP)	124
Глава 3.4. Маршрутизация	152
Глава 3.5. Реализация	169
Раздел 4. Продвинутое множество взаимодействий	178
Глава 4.1. Глобальный интернет	178
Глава 4.2. IP-версия 6	190
Глава 4.3. Многоадресная рассылка (multicast)	198
Глава 4.4. Маршрутизация с множеством протокольных меток (MPLS)	209
Глава 4.5. Маршрутизация среди мобильных устройств	219
Раздел 5. Сквозные протоколы	226
Глава 5.1. Простой демultipлексор (UDP)	227
Глава 5.2. Надежный побайтовый поток (TCP)	229
Глава 5.3. Вызов удаленной процедуры	256
Глава 5.4. Транспортный протокол реального времени (RTP)	271
Раздел 6. Контроль перегрузок	283
Глава 6.1. Вопросы распределения ресурсов	284
Глава 6.2. Дисциплины очередей	292
Глава 6.3. Управление перегрузками TCP	297
Глава 6.4. Расширенное управление перегрузками	307
Глава 6.5. Качество обслуживания	318
Раздел 7. Сквозные данные	338
Глава 7.1. Форматирование представления	339
Глава 7.2. Мультимедийные данные	351
Раздел 8. Безопасность сети	369
Глава 8.1. Доверие и угрозы	370
Глава 8.2. Криптографические строительные блоки	371
Глава 8.3. Предраспределение ключей	379
Глава 8.4. Протоколы аутентификации	386
Глава 8.5. Примеры систем	391
Раздел 9. Приложения	411
Глава 9.1. Традиционные приложения	411
Глава 9.2. Мультимедийные приложения	432
Глава 9.3. Инфраструктурные приложения	444
Глава 9.4. Накладные (оверлейные) сети	456

Предисловие

Прошло почти десять лет с тех пор, как вышло 5-е издание книги «Компьютерные сети: системный подход». За это время многое изменилось, особенно заметным стало стремительное развитие облачных технологий и приложений для смартфонов. Во многом это напоминает огромное влияние, которое оказывала Всемирная паутина на Интернет, когда вышло 1-е издание книги в 1996 году.

Шестое издание адаптировано к современности, но в нем сохранен *системный подход* в подаче материала. Мы обновили и улучшили это последнее издание книги четырьмя способами:

- Мы обновили примеры, чтобы они отражали современное состояние компьютерных сетей. Этот процесс включает в себя удаление анахронизмов (например, dial-up modem), использование популярных приложений (Netflix, Spotify) для актуализации рассматриваемых проблем, а также обновление цифр для отражения современных технологий (например, 10-Gbps Ethernet).
- Мы связали точки между оригинальными исследованиями, которые привели к развитию таких технологий, как многоадресная рассылка и потоковое видео в реальном времени, уже ставших обыденностью в облачных приложениях, такими как GoToMeeting, Netflix и Spotify. Это соответствует акцентированию нашего внимания на процессе проектирования, а не только на конечном результате, что особенно важно сегодня, когда большая часть информации в Интернете доступна в основном в виде проприетарных коммерческих сервисов.
- Мы рассмотрели Интернет в более широком контексте «облака» и, что не менее важно, в контексте коммерческих сил, которые формируют «облако». Это оказало минимальное влияние на технические детали, представленные в книге, но данная тема обсуждается в новом разделе «Перспектива» после каждого раздела. Мы надеемся, что одним из побочных эффектов этого обсуждения станет понимание непрерывной эволюции Интернета и тех возможностей для инноваций, которые он предоставляет.
- Важные принципы проектирования сетей представлены в книге в виде серии *основных выводов*. Каждый вывод — это краткое изложение либо общего правила проектирования системы, либо фундаментальной концепции сетевого взаимодействия, основанной на примерах, которые представлены в издании. С педагогической точки зрения эти выводы соответствуют высокоуровневым *целям обучения*, поставленным в книге.

В частности, 6-е издание включает следующие основные изменения:

- Новый подраздел «Перспектива» в разделе 1 представляет вновь затронутую тему *облачных технологий*.
- В новой главе 2.8 описывается *сеть доступа*, включая пассивные оптические сети (PON) и сети радиодоступа 5G (RAN).
- Переработаны темы глав 3.1 (*основы коммутации*) и 3.2 (*коммутуруемый Ethernet*), включая расширенное освещение виртуальных локальных сетей (VLAN).
- Глава 3.5 обновлена и включает описания *коммутаторов White-Box* и *программно-определяемых сетей (SDN)*.
- Новый подраздел «Перспектива» в разделе 3 описывает *VXLAN* и роль *оверлеев* в облаке.
- Переработаны темы глав 4.1 (*глобальная сеть Интернет*) и 4.2 (*IP версия 6*).
- Новый подраздел «Перспектива» в разделе 4 описывает, как *облачные технологии* влияют на структуру Интернета.
- Глава 5.2 расширена за счет включения в нее обсуждения протокола *QUIC*.
- Глава 5.3 расширена за счет включения описания *gRPC*.
- Главы 6.3 и 6.4 обновлены и включают описания *TCP CUBIC*, *DCTCP* и *BBR*.
- Глава 6.4 расширена за счет включения описания *активного управления очередями (AQM)*.

- Глава 7.1 расширена за счет включения описания *буферов протокола (Protocol Buffers)*.
- Глава 7.2 расширена за счет включения описания *адаптивной потоковой передачи HTTP*.
- В новой главе 8.1 представлена двойственность *угроз и доверия*.
- Переработаны темы в главах 8.3 (*предраспределение ключей*) и 8.4 (*протоколы аутентификации*).
- Новый подраздел «Перспектива» после раздела 8 описывает *децентрализованное управление идентификацией* и роль *блокчейна*.
- Глава 9.1 обновлена и включает описание *HTTP/2*, а также рассмотрение *REST, gRPC и облачных сервисов*.
- Глава 9.3 расширена и включает описание современных систем управления сетью, в том числе использование *OpenConfig* и *gNMI*.

Организация подачи материала

Чтобы построить обучающий сетевой курс на основе материала этой книги, важно понять общую организацию подачи материала, который состоит из трех основных частей:

- Концептуальный и фундаментальный материал, то есть основополагающие идеи, лежащие в основе сетевого взаимодействия.
- Основные протоколы и алгоритмы, иллюстрирующие применение основополагающих идей на практике.
- Продвинутый материал, который может вписаться (или не вписаться) в один семестровый курс.

Эту характеристику можно применить на уровне разделов: раздел 1 является основополагающим, разделы 2, 3, 5 и 9 — основными, а разделы 4, 6, 7 и 8 охватывают более сложные темы.

Эту характеристику можно применить и на уровне глав, где каждая глава развивается от базовых понятий к конкретным технологиям и продвинутым методам. Например, раздел 3 начинается с представления основ коммутируемых сетей (3.1), затем рассматриваются особенности коммутируемого Ethernet и IP-интернета (3.2 – 3.4), и завершается факультативным рассмотрением SDN (3.5). Аналогичным образом раздел 6 начинается с фундаментальных идей (6.1 – 6.2), затем рассматривается алгоритм избежания перегрузок в TCP (6.3), и завершается дополнительным продвинутым материалом (6.4 – 6.5).

Благодарности

Мы хотели бы поблагодарить следующих людей за помощь в создании нового материала:

Ларри Бракмо (управление перегрузками TCP)
 Кармело Касконе (коммутаторы White-Box)
 Чарльз Чан (коммутаторы White-Box)
 Джуд Нельсон (децентрализованная идентификация)
 Огуз Сунай (сотовые сети)
 Томас Вачуска (сетевое управление)

...а также некоторых отдельных лиц (пользователей GitHub) за их разнообразный вклад и исправление ошибок:

Мохаммед Аль-Амин
 Энди Бавьер
 Мануэль Берфельде
 Крис Голдсуорси
 Джон Хартман
 Диего Лопес Леон
 Маттео Скандоло

Майк Ваврзоняк
(spacewander)
Арно (arvdrpoo)
Десмонд (kingdido999)
Гуо (ZJUGuoShuai)
Хеллман (eshellman)
Дао (vertextao)
Майк Аппельман
Сет (springbov)

Наконец, мы хотели бы поблагодарить следующих рецензентов за их многочисленные полезные замечания и предложения. Они оказали существенное влияние на результат:

Марк Дж. Инделикато, Рочестерский технологический институт
Майкл Йоншик Чой, Иллинойский технологический институт
Сарвеш Кулкарни, Университет Вилланова
Александр Л. Виджесинха, Университет Таусона

Ларри и Брюс, ноябрь 2019 г.

Раздел 1.

Основы

Я должен создать свою систему или быть поработанным чужой; я не буду рассуждать и сравнивать: мое дело — создавать.

Уильям Блейк

Задача: создание сети

Представьте, что вы хотите создать компьютерную сеть, которая имеет потенциал для глобального роста и может поддерживать такие разнообразные приложения, как видеоконференции, видео по запросу, электронная торговля, распределенные вычисления и цифровые библиотеки. Какие доступные технологии могут послужить основными «строительными блоками», и какую программную архитектуру вы бы разработали, чтобы интегрировать эти блоки в эффективный коммуникационный сервис? Ответ на этот вопрос является основной целью данной книги — описать доступные «строительные блоки», а затем показать, как их можно использовать для построения сети с нуля.

Прежде чем мы сможем понять, как проектировать компьютерную сеть, нужно выяснить, что именно она собой представляет. Когда-то термин «*сеть*» означал набор последовательных линий, используемых для подключения терминалов к мейнфреймам. Другими важными сетями являются телефонная сеть для передачи голоса и сеть кабельного телевидения, используемая для распространения видеосигналов. Главное, что объединяет эти сети, заключается в том, что они специализированы для обработки одного определенного типа данных (нажатия клавиш, голоса или видео) и обычно подключаются к устройствам специального назначения (терминалы, телефонные трубки и телевизоры).

Что отличает компьютерную сеть от других типов сетей? Вероятно, самой важной характеристикой является универсальность. Компьютерные сети строятся в основном из программируемого оборудования общего назначения, и они не оптимизированы для какой-то конкретной задачи, такой как телефонные звонки или передача телевизионных сигналов. Вместо этого они способны передавать множество различных типов данных и поддерживают широкий и постоянно растущий спектр приложений. Современные компьютерные сети практически полностью взяли на себя функции, ранее выполняемые специализированными сетями. В этом разделе рассматриваются некоторые типичные приложения компьютерных сетей и обсуждаются требования, которые должен учитывать проектировщик сети, желающий поддерживать такие приложения.

Как только мы определим требования, как нам действовать дальше? К счастью, мы не будем строить первую сеть. Другие разработчики, в первую очередь сообщество исследователей, ответственных за Интернет, уже прошли этот путь до нас. Мы будем использовать богатый опыт, накопленный при разработке Интернета, для управления нашим проектированием. Этот опыт воплощен в *сетевой архитектуре*, которая определяет доступные аппаратные и программные компоненты и показывает, как их можно организовать для формирования полной сетевой системы.

Помимо понимания того, как создаются сети, еще более важно знать, как они эксплуатируются и управляются, а также как разрабатываются сетевые приложения. Почти у всех теперь есть компьютерные сети в домах, офисах и даже в автомобилях, поэтому эксплуатация сетей больше не является делом только для нескольких специалистов. А с распространением смартфонов гораздо больше людей нынешнего поколения разрабатывают сетевые приложения, чем в прошлом. Поэтому нам необходимо рассматривать сети с разных точек зрения: строителей, операторов, разработчиков приложений.

Чтобы начать путь к пониманию того, как строить, эксплуатировать и программировать сеть, в этом разделе мы затронем четыре основных момента. Во-первых, были исследованы требования, которые различные приложения и сообщества людей предъявля-

ют к сети. Во-вторых, мы ввели идею сетевой архитектуры, которая закладывает основу для остальной части книги. В-третьих, мы познакомим вас с некоторыми ключевыми элементами реализации компьютерных сетей. И, наконец, мы определили ключевые метрики, используемые для оценки производительности компьютерных сетей.

Глава 1.1. Приложения

Большинство людей знакомы с Интернетом по его приложениям: Всемирная паутина, электронная почта, социальные сети, потоковое воспроизведение музыки и фильмов, видеоконференции, обмен мгновенными сообщениями, файлообменники — и это лишь несколько примеров. Иными словами, мы взаимодействуем с Интернетом как *пользователи* сети. Пользователи Интернета представляют собой самый многочисленный класс людей, которые так или иначе взаимодействуют с Интернетом, но есть и несколько других важных групп.

Существует группа людей, которые *создают* приложения — группа, значительно расширившаяся в последние годы благодаря мощным платформам для программирования и новым устройствам, таким как смартфоны, которые создают новые возможности для быстрой разработки приложений и вывода их на глобальный рынок.

Есть и те, кто *занимается эксплуатацией* или *управлением* сетями — в основном это работа «за кулисами», но критически важная и зачастую очень сложная. С распространением домашних сетей все больше и больше людей становятся, пусть и в небольшой степени, сетевыми операторами.

Наконец, есть те, кто *проектирует* и *создает* устройства и протоколы, из которых в совокупности состоит Интернет. Эта последняя группа является традиционной целевой аудиторией учебников по сетям, таких как этот. Однако на протяжении всей книги мы также будем рассматривать перспективы разработчиков приложений и операторов сетей.

Учет этих точек зрения позволит нам лучше понять разнообразные требования, которым должна соответствовать сеть. Разработчики приложений также смогут создавать приложения, которые будут работать лучше, если они будут понимать, как работает и взаимодействует с приложениями основная технология. Итак, прежде чем мы начнем разбираться, как построить сеть, давайте более внимательно рассмотрим типы приложений, которые поддерживают современные сети.

Глава 1.1.1. Классы приложений

Всемирная паутина (World Wide Web) — это интернет-приложение, которое превратило Интернет из малоизвестного инструмента, использовавшегося в основном учеными и инженерами, в массовое явление, которым он является сегодня. Сама паутина стала такой мощной платформой, что многие люди путают ее с Интернетом, и утверждать, что паутина — это одно приложение, несколько преувеличенно.

В своей базовой форме паутина представляет собой интуитивно простой интерфейс. Пользователи просматривают страницы, заполненные текстовыми и графическими объектами, и кликают на объекты, о которых хотят узнать больше, после чего появляется соответствующая новая страница. Большинство людей также знают, что каждый выбираемый объект на странице связан с идентификатором для следующей страницы или объекта, которые будут просмотрены. Этот идентификатор, называемый унифицированным указателем ресурса (URL), обеспечивает способ идентификации всех возможных объектов, которые можно просмотреть в вашем веб-браузере. Например,

`http://www.cs.princeton.edu/llp/index.html`

является URL-адресом страницы, предоставляющей информацию об одном из авторов этой книги: строка `http` указывает на то, что для загрузки страницы следует использовать

протокол передачи гипертекста (HTTP), www.cs.princeton.edu — это имя машины, которая обслуживает страницу, а [/llp/index.html](http://www.cs.princeton.edu/llp/index.html) однозначно идентифицирует домашнюю страницу Ларри на этом сайте.

Однако большинство пользователей Интернета не знают, что при щелчке по одному такому URL может быть обменено более десятка сообщений по Интернету, и даже намного больше, если веб-страница сложная и содержит множество встроенных объектов. Этот обмен сообщениями включает до шести сообщений для преобразования имени сервера (www.cs.princeton.edu) в его IP-адрес (128.112.136.35), три сообщения для установки соединения по протоколу управления передачей (TCP) между вашим браузером и этим сервером, четыре сообщения для отправки браузером HTTP-запроса «GET» и ответа сервера с запрашиваемой страницей (и для подтверждения получения этого сообщения каждой из сторон) и четыре сообщения для завершения соединения TCP. Конечно, это не включает миллионы сообщений, которыми обмениваются узлы Интернета в течение дня, просто чтобы сообщить друг другу, что они существуют и готовы обслуживать веб-страницы, переводить имена в адреса и пересылать сообщения к их конечному пункту назначения.

Еще один широко распространенный класс приложений Интернета — это передача потокового аудио и видео. Такие услуги, как видео по запросу и интернет-радио, используют эту технологию. Хотя мы часто начинаем с веб-сайта для инициации потоковой сессии, доставка аудио и видео имеет некоторые важные отличия от получения простой веб-страницы с текстом и изображениями. Например, вам зачастую не нужно загружать весь видеофайл (этот процесс может занять несколько минут), прежде чем смотреть первую сцену. Потоковое аудио и видео предполагает более своевременную передачу сообщений от отправителя к получателю, и у получателя отображается видео или воспроизводится аудио практически по мере его поступления.

Заметьте, что разница между потоковыми приложениями и более традиционной доставкой текста, графики и изображений заключается в том, что люди потребляют аудио- и видеопотоки непрерывно, и прерывистость — в форме пропущенных звуков или задержек видео — неприемлема. Напротив, обычная (не потоковая) страница может доставляться и читаться частями. Это различие влияет на то, как сеть поддерживает эти разные классы приложений.

Другой класс приложений — это приложения для реального времени с аудио и видео. Они имеют значительно более жесткие временные ограничения, чем потоковые приложения. При использовании приложений для голосовой связи по IP, таких как Skype, или приложений для видеоконференций, взаимодействие между участниками должно происходить своевременно. Когда человек на одном конце «делает жест», это действие должно быть отображено на другом конце как можно быстрее¹.

Когда один человек пытается перебить другого, перебиваемый должен услышать это как можно скорее и решить, прервать ли свое повествование или продолжать говорить вместе с перебивающим. Слишком большая задержка в такой среде делает систему непригодной для использования. В отличие от этого, при видео по запросу, если проходит несколько секунд с момента запуска видео пользователем до отображения первого изображения, услуга все еще считается удовлетворительной. Кроме того, интерактивные приложения обычно предполагают аудио- и/или видеопотоки в обоих направлениях, в то время как потоковое приложение, скорее всего, передает видео или аудио только в одном направлении.

Средства видеоконференций, работающие через Интернет, появились еще в начале 1990-х годов, но широкое распространение получили в последние несколько лет, когда на рынке появилось несколько коммерческих видеоприложений. Пример одной из таких систем показан на рис. 1.1. Как при загрузке веб-страницы требуется нечто большее, чем кажется на первый взгляд, так же происходит и в случае с видеоприложениями. Напри-

¹ Не совсем «как можно быстрее». . . Исследования человеческого фактора показывают, что 300 мс — это разумная верхняя граница того, сколько задержки сигнала по пути туда и обратно можно терпеть в телефонном разговоре, прежде чем люди начнут жаловаться, а задержка в 100 мс вполне приемлема.

мер, размещение видеоконтента в сети с относительно низкой пропускной способностью или обеспечение синхронизации видео и аудио и их своевременной передачи — все это проблемы, о которых приходится беспокоиться разработчикам сетей и протоколов. Эти и многие другие вопросы, связанные с мультимедийными приложениями, мы рассмотрим далее.

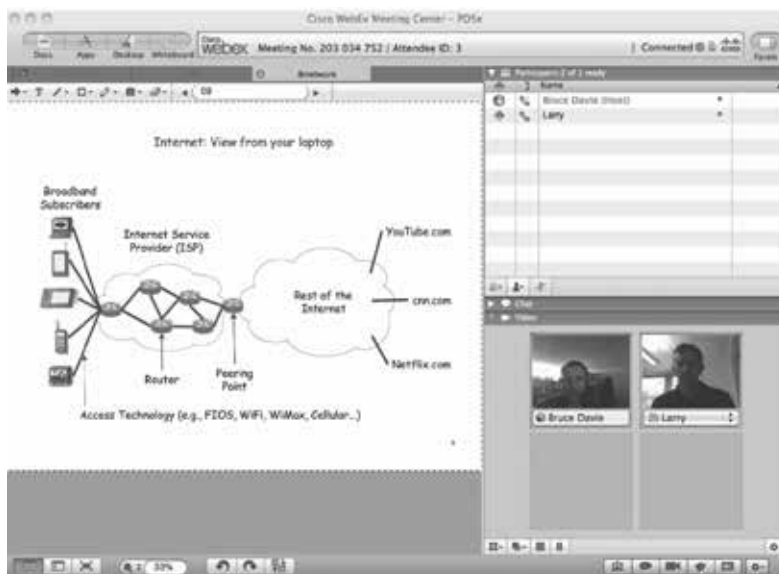


Рисунок 1.1 Мультимедийное приложение, включающее видеоконференции.

Хотя это всего лишь два примера, загрузка страниц из Сети и участие в видеоконференции демонстрируют разнообразие приложений, которые могут быть созданы на базе Интернета, и указывают на сложность его конструкции. Позже в книге мы разработаем более полную таксономию типов приложений, которая поможет направить рассмотрение ключевых проектных решений при создании, эксплуатации и использовании сетей с таким широким спектром приложений. В конце книги мы вновь рассмотрим эти два конкретных приложения, а также несколько других, которые иллюстрируют широту возможностей современного Интернета.

На данный момент этого краткого обзора нескольких типичных приложений достаточно, чтобы начать рассматривать проблемы, которые необходимо решить, если мы хотим создать сеть, поддерживающую такое разнообразие приложений.

Глава 1.2. Требования

Мы поставили перед собой амбициозную цель: объяснить, как построить компьютерную сеть с нуля. Наш подход к достижению этой цели будет заключаться в том, чтобы начать с основных принципов и затем задавать вопросы, которые мы бы естественным образом задали при построении реальной сети. На каждом этапе мы будем использовать современные протоколы для иллюстрирования различных вариантов проектных решений, доступных нам, но мы не будем принимать эти существующие артефакты как непреложную истину. Вместо этого мы будем задавать (и отвечать на) вопрос о том, почему сети спроектированы именно таким образом. Хотя есть соблазн удовлетвориться простым пониманием того, как сети устроены сегодня, важно усвоить основные концепции, потому что сети постоянно меняются по мере развития технологий и создания новых приложений. Наш опыт показывает, что как только вы поймете фундаментальные идеи, любой новый протокол, с которым вы столкнетесь, будет относительно прост в усвоении.

Глава 1.2.1. Заинтересованные стороны

Как мы уже отмечали, человек, который изучает сети, может рассматривать их с нескольких точек зрения. Когда мы готовили первое издание этой книги, большинство людей вообще не имело доступа к Интернету, а те, кто имел, получали его на работе, в университете или через модемное подключение дома. Набор популярных приложений можно было пересчитать по пальцам. Таким образом, как и большинство книг того времени, наша книга была сосредоточена на точке зрения тех, кто разрабатывает сетевое оборудование и протоколы. Мы продолжаем фокусироваться на этой точке зрения и надеемся, что после прочтения данной книги вы будете знать, как разрабатывать сетевое оборудование и протоколы будущего.

Однако мы также хотим рассмотреть точки зрения еще двух заинтересованных сторон: тех, кто разрабатывает сетевые приложения, и тех, кто управляет ими или эксплуатирует их. Давайте рассмотрим, как эти три заинтересованные стороны могут формулировать свои требования к сети:

- *Разработчик приложений* перечислил бы услуги, которые нужны его приложению: например, гарантию, что каждое сообщение, отправленное приложением, будет доставлено без ошибок в течение определенного времени, или возможность плавного переключения между различными соединениями с сетью по мере перемещения пользователя.
- *Оператор сети* перечислил бы характеристики системы, которая легко администрируется и управляется: например, возможность легко устранить неисправности, добавлять новые устройства в сеть и правильно настраивать их, а также удобство учета использования.
- *Проектировщик сети* перечислил бы свойства экономичного проектирования: например, эффективное использование сетевых ресурсов и их справедливое распределение между различными пользователями. Вопросы производительности также, вероятно, будут важны.

В этом разделе мы попытались сформулировать требования различных заинтересованных сторон в высокоуровневое введение в основные аспекты, определяющие проектирование сетей, и тем самым выявить вызовы, рассматриваемые в остальной части этой книги.

Глава 1.2.2. Масштабируемое подключение

Начнем с очевидного: сеть должна обеспечивать соединение между большим количеством компьютеров. Иногда достаточно построить ограниченную сеть, которая соединяет только несколько выбранных машин. Фактически по соображениям конфиденциальности и безопасности многие частные (корпоративные) сети имеют цель ограничить количество подключенных машин. В отличие от этого, другие сети (ярким примером которых является Интернет) разработаны таким образом, чтобы иметь возможность подключить все компьютеры в мире. Система, спроектированная для поддержания роста до произвольно больших размеров, называется *масштабируемой*. Используя Интернет в качестве модели, эта книга рассматривает проблему масштабируемости.

Чтобы более полно усвоить требования к соединению, мы должны выяснить, как компьютеры соединяются в сети. Соединение происходит на множестве различных уровней. На самом низком уровне сеть может состоять из двух или более компьютеров, напрямую соединенных каким-либо физическим средством, таким как коаксиальный кабель или оптоволокно. Мы называем такое физическое средство связи каналом и часто именуют компьютеры, которые оно соединяет, *узлами*. (Иногда узел представляет собой более специализированное устройство, а не компьютер, но мы пренебрегаем этим различием ради достижения целей нашего обсуждения.) Как показано на рис. 1.2, физические каналы иногда ограничены парой узлов (такой канал называется «точка-точка»), тогда

как в других случаях более двух узлов могут использовать один физический канал (такой канал называется «*множественным доступом*»). Беспроводные каналы, как те, которые предоставляются сетями сотовой связи и Wi-Fi, являются важным классом каналов с множественным доступом. Каналы с множественным доступом ограничены размером: как географическим расстоянием, которое они могут преодолеть, так и количеством узлов, которые они могут соединить. По этой причине они часто реализуют так называемую «*последнюю милю*», соединяя конечных пользователей с остальной частью сети.

Если бы компьютерные сети были ограничены ситуациями, когда все узлы напрямую связаны друг с другом через общую физическую среду, то либо сети были бы весьма ограничены в количестве подключаемых компьютеров, либо количество проводов, выходящих из каждого узла, быстро стало бы как неуправляемым, так и очень дорогим. Но соединение между двумя узлами не обязательно означает прямое физическое соединение между ними — косвенное соединение может быть достигнуто между набором «сотрудничающих» узлов. Рассмотрим следующие два примера того, как набор компьютеров может быть косвенно связан.

На рис. 1.3 показана пара узлов, каждый из которых подключен к одному или нескольким каналам «точка-точка». Те узлы, которые подключены как минимум к двум каналам, используют программное обеспечение, которое пересылает данные, полученные по одному каналу, через другой. Если организовать эти узлы систематически, такие пересылающие узлы формируют *коммутируемую сеть*. Существует множество типов коммутируемых сетей, из которых два наиболее распространенных — *коммутация каналов* и *коммутация пакетов*. Первая наиболее известна в телефонной системе, тогда как вторая используется в подавляющем большинстве компьютерных сетей и будет основным объектом изучения в этой книге. (Коммутация каналов, однако, снова набирает популярность в области оптических сетей, что становится важным, поскольку спрос на пропускную способность сети постоянно растет.) Важной особенностью пакетных сетей является то, что узлы в такой сети передают друг другу дискретные блоки данных. Думайте об этих блоках данных как о фрагментах данных приложения, таких как файл, электронное письмо или изображение. Мы называем каждый блок данных либо *пакетом*, либо *сообщением*, и пока будем использовать эти термины взаимозаменяемо.

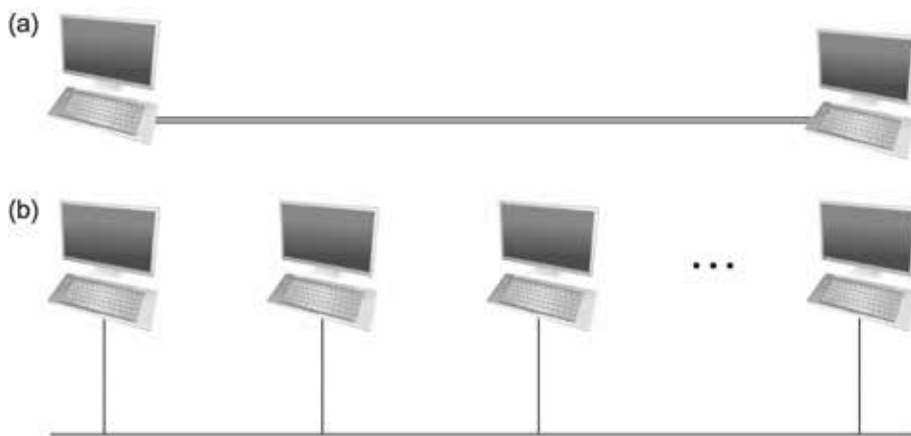


Рисунок 1.2. Прямые каналы связи: (a) точка-точка; (b) множественный доступ.

В сетях с коммутацией пакетов обычно используют стратегию, называемую «*хранение и пересылка*» (*store-and-forward*). Как следует из названия, каждый узел в сети хранения и пересылки сначала получает полный пакет по какому-либо каналу, сохраняет его во внутренней памяти, а затем пересылает полный пакет следующему узлу. В отличие

от нее, сеть с коммутацией каналов сначала устанавливает выделенный канал через последовательность соединений и затем позволяет исходному узлу передавать поток битов по этому каналу к узлу назначения. Основная причина использования в компьютерной сети коммутации пакетов, а не коммутации каналов, заключается в эффективности, о которой будет рассказано в следующей главе.

Облако на рис. 1.3 различает узлы внутри, которые *реализуют сеть* (они обычно называются *коммутаторами*, и их основная функция — хранить и пересылать пакеты), и узлы вне облака, которые *используют сеть* (они традиционно называются *хостами*, и они поддерживают пользователей и запускают прикладные программы). Также отметим, что облако является одним из самых важных символов в компьютерных сетях. В общем, мы используем облако, чтобы обозначить любой тип сети, будь то отдельное соединение «точка-точка», канал с множественным доступом или коммутируемая сеть. Таким образом, всякий раз, когда вы видите облако, используемое на рисунке, вы можете думать о нем как о заместителе любой из сетевых технологий, рассматриваемых в этой книге.¹

Второй способ, при котором набор компьютеров может быть косвенно подключен, показан на рис. 1.4. В этой ситуации несколько независимых сетей (облаков) соединены для формирования интерсети, или просто — интернета. Мы принимаем условность Интернета, обозначая общий термин «интерсеть» с маленькой буквы «и» как «интернет», а TCP/IP Интернет, который мы используем каждый день, с большой буквы «И» как «Интернет». Узел, подключенный к двум или более сетям, обычно называется *маршрутизатором* или *шлюзом*, и он выполняет ту же роль, что и коммутатор — пересылает сообщения из одной сети в другую. Обратите внимание, что интерсеть сама по себе может рассматриваться как другой тип сети, что означает, что интерсеть может быть построена из нескольких интерсетей. Таким образом, мы можем рекурсивно строить сети произвольно большого размера, соединяя облака для формирования более крупных облаков. Можно обоснованно утверждать, что идея соединения значительно различающихся сетей была фундаментальной инновацией Интернета и что успешный рост Интернета до глобальных размеров и миллиардов узлов был результатом удачных решений, принятых ранними архитекторами Интернета, которые мы обсудим позже.

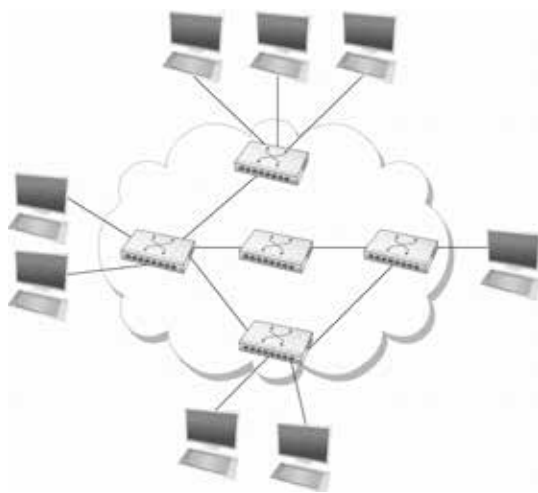


Рисунок 1.3. Коммутируемая сеть.

¹ Использование облаков для представления сетей предшествует термину облачных вычислений по крайней мере на пару десятилетий, но есть тесная связь между этими двумя понятиями, которые мы рассматриваем в «Перспективах» после каждого раздела.



Рисунок 1.4. Объединение сетей.

Тот факт, что набор хостов напрямую или косвенно подключен друг к другу, не означает, что мы успешно обеспечили связь между хостами. Последним требованием является то, что каждый узел должен иметь возможность указать, с каким из других узлов в сети он хочет общаться. Это достигается путем присвоения каждому узлу *адреса*. Адрес — это строка байтов, которая идентифицирует узел; то есть сеть может использовать адрес узла для его отличия от других узлов, подключенных к сети. Когда исходный узел хочет, чтобы сеть доставила сообщение определенному узлу-получателю, он указывает адрес узла-получателя. Если отправляющий и принимающий узлы не подключены напрямую, то коммутаторы и маршрутизаторы сети используют этот адрес для принятия решения о том, как перенаправить сообщение к узлу-получателю. Процесс систематического определения того, как перенаправлять сообщения к узлу-получателю на основе его адреса, называется *маршрутизацией*.

В этом кратком введении в адресацию и маршрутизацию предполагалось, что узел-источник хочет отправить сообщение одному узлу назначения (*одноадресная рассылка*). Хотя это наиболее распространенный сценарий, также возможно, что узел-источник захочет передать сообщение всем узлам сети. Или же узел-источник может захотеть отправить сообщение некоторому подмножеству других узлов, но не всем.

Основные выводы

Основная идея, которую следует извлечь из этого обсуждения, заключается в том, что *сеть* можно определить рекурсивно как состоящую из двух или более узлов, соединенных физической линией, или как две или более сети, соединенные узлом. Другими словами, сеть может быть построена из вложенных сетей, где на нижнем уровне сеть реализована с помощью какого-либо физического носителя. Одними из ключевых задач обеспечения сетевой связи являются определение адреса для каждого узла, доступного в сети (будь то логический или физический), и использование таких адресов для пересылки сообщений к соответствующему(им) узлу(ам)-получателю(ям).

Глава 1.2.3. Экономически эффективное совместное использование ресурсов

Как было сказано выше, данная книга сосредоточена на сетях с пакетной коммутацией. В этой главе объясняется ключевое требование компьютерных сетей — *экономическая эффективность*, которая приводит нас к выбору стратегии пакетной коммутации.

При наличии нескольких узлов, косвенно соединенных через вложенные сети, имеет-ся возможность того, чтобы любая пара хостов отправляла сообщения друг другу через по-

следовательность каналов связи и узлов. Конечно, мы хотим делать больше, чем просто поддерживать одну пару взаимодействующих хостов — мы хотим предоставить всем парам хостов возможность обмениваться сообщениями. Вопрос в том, как все хосты, которые хотят общаться, могут совместно использовать сеть, особенно если они хотят использовать ее одновременно? И, как будто эта проблема была недостаточно сложной, также надо учесть, как несколько хостов могут совместно задействовать один и тот же канал связи, когда они все хотят использовать его одновременно.

Чтобы объяснить, как хосты совместно используют сеть, нам нужно ввести фундаментальное понятие *мультиплексирования* (multiplexing), которое означает, что системный ресурс делится между несколькими пользователями. На интуитивном уровне мультиплексирование можно объяснить на примере многозадачной компьютерной системы, где один физический процессор используется (мультиплексируется) между несколькими задачами, каждая из которых считает, что у нее есть свой собственный процессор. Точно так же данные, отправляемые несколькими пользователями, могут быть мультиплексированы через физические каналы связи, которые составляют сеть.

Чтобы понять, как это может работать, рассмотрим простую сеть, изображенную на рис. 1.5, где три хоста на левой стороне сети (отправители S1 – S3) отправляют данные трем хостам на правой стороне (получатели R1 – R3), используя коммутируемую сеть, содержащую только один физический канал связи. (Для простоты предположим, что хост S1 отправляет данные хосту R1 и так далее.) В этой ситуации три потока данных, соответствующие трем парам хостов, мультиплексируются на один физический канал связи коммутатором 1, а затем *демультиплексируются* (demultiplexed) обратно в отдельные потоки коммутатором 2. Обратите внимание, что мы намеренно не уточняем, что именно означает «поток данных». Для целей этого обсуждения предположим, что у каждого хоста на левой стороне есть большой объем данных, которые он хочет отправить своему «партнеру» на правой стороне.

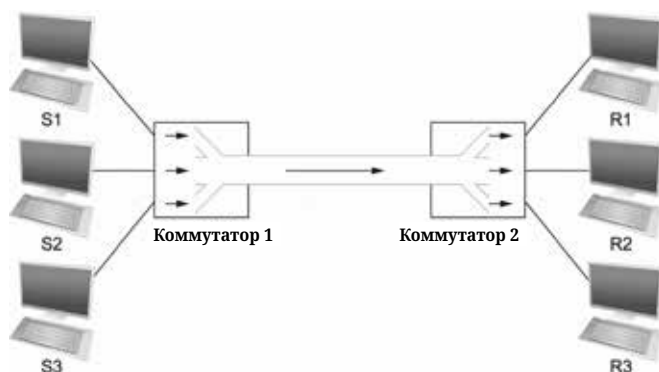


Рисунок 1.5. Мультиплексирование нескольких логических потоков по одному физическому каналу.

Существует несколько различных методов для мультиплексирования некоторого количества потоков на один физический канал связи. Один из распространенных методов — это *синхронное временное разделение мультиплексирования* (synchronous time-division multiplexing, STDM). Идея STDM заключается в разделении времени на равные отрезки и круговом способе предоставления каждому потоку возможности отправлять свои данные по физическому каналу. Другими словами, во время отрезка 1 передаются данные от S1 к R1; в отрезке 2 передаются данные от S2 к R2; в отрезке 3 S3 отправляет данные R3. Затем первый поток (S1 к R1) снова начинает свою передачу, и процесс повторяется. Другой метод — это *мультиплексирование по частоте* (frequency-division multiplexing, FDM). Идея FDM заключается в передаче каждого потока по физическому каналу на раз-

ных частотах, так же, как сигналы различных ТВ-станций передаются на разных частотах в эфире или по коаксиальным телевизионным кабелям.

Несмотря на простоту, и STDM, и FDM имеют два недостатка. Во-первых, если у одного из потоков нет данных для отправки, его доля физического канала (его временной отрезок или частота) остается неиспользованной, даже если у других потоков есть данные для передачи. Например, S3 должен был подождать своей очереди за S1 и S2 в примере, который был рассмотрен ранее, даже если S1 и S2 ничего не отправляли. Для компьютерной коммуникации время простоя канала может быть очень большим (например, учтите время, которое вы тратите на чтение веб-страницы (оставляя канал простаивающим) по сравнению с временем, которое вы тратите на загрузку страницы). Во-вторых, как STDM, так и FDM ограничены ситуациями, когда максимальное количество потоков фиксировано и известно заранее. Непрактично изменять размер отрезка или добавлять дополнительные отрезки в случае STDM или добавлять новые частоты в случае FDM.

Форма мультиплексирования, которая устраняет эти недостатки и используется чаще в этой книге, называется *статистическим мультиплексированием* (statistical multiplexing). Хотя название полностью не отражает концепции, статистическое мультиплексирование на самом деле имеет довольно простой смысл и основано на двух ключевых идеях. Во-первых, оно похоже на STDM в том смысле, что физический канал используется последовательно во времени — сначала передаются данные одного потока, затем данные другого потока и так далее. В отличие от STDM, данные передаются из каждого потока по требованию, а не в предопределенный временной интервал. Таким образом, если только у одного потока есть данные для отправки, он может передавать эти данные без ожидания своего временного отрезка и, следовательно, без необходимости ожидать, как во время отрезков, которые были выделены для других потоков, те потоки простаивают. Именно избегание времени простоя придает эффективность коммутации пакетов.

Однако в текущем определении статистическое мультиплексирование не имеет механизма, который гарантирует, что все потоки в конечном итоге получат свой черед передавать данные по физическому каналу. То есть, как только поток начинает отправлять данные, нам нужен какой-то способ ограничить передачу, чтобы другие потоки также могли получить свой черед. Для учета этой необходимости статистическое мультиплексирование определяет верхнюю границу размера блока данных, который каждому потоку разрешается передавать в данный момент времени. Этот блок данных ограниченного размера обычно называется *пакетом*, чтобы отличать его от произвольно большого сообщения, которое может передать прикладная программа. Поскольку пакетная коммутация в сети ограничивает максимальный размер пакетов, отправитель может не иметь возможности отправить полное *сообщение* одним пакетом. В этом случае исходное сообщение может потребоваться разбить на несколько пакетов, а получатель должен будет воссоздать исходное сообщение из этих пакетов.

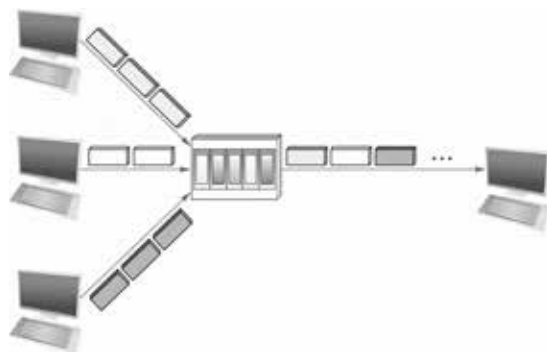


Рисунок 1.6. Коммутатор, мультиплексирующий пакеты из нескольких источников на один общий канал.

Иными словами, каждый поток отправляет последовательность пакетов через физический канал, и на каждом этапе принимается решение, какой пакет следует отправить дальше. Заметьте, что если только у одного потока есть данные для отправки, то он может отправлять последовательность пакетов один за другим. Однако если более одного потока имеют данные для передачи, их пакеты чередуются на канале. На рис. 1.6 показано, как коммутатор мультиплексирует пакеты от нескольких источников на один общий канал.

Решение о том, какой пакет отправить следующим по общему каналу, может приниматься различными способами. Например, в сети, состоящей из коммутаторов, соединенных линиями связи, как на рис. 1.5, решение принимается коммутатором, передающим пакеты по общему каналу. (Как мы увидим позже, не во всех сетях с коммутацией пакетов используются коммутаторы, и они могут использовать другие механизмы для определения того, чей пакет попадет на линию связи следующим). Каждый коммутатор в сети с коммутацией пакетов принимает это решение независимо, на основе каждого пакета. Один из вопросов, который встает перед проектировщиком сети — как сделать так, чтобы это решение было справедливым. Например, коммутатор может быть спроектирован таким образом, чтобы обслуживать пакеты по принципу «первый пришел — первый ушел» (FIFO). Другой подход заключается в передаче пакетов от каждого из различных потоков, которые в настоящее время отправляют данные через коммутатор, по принципу круговой выборки. Это может быть сделано для того, чтобы определенные потоки получали определенную долю пропускной способности канала или чтобы их пакеты не задерживались в коммутаторе дольше определенного времени. Иногда говорят, что сеть, которая пытается распределить полосу пропускания между определенными потоками, поддерживает *качество обслуживания* (QoS).

Также обратите внимание на рис. 1.6, что, поскольку коммутатору приходится мультиплексировать три входящих потока пакетов на один исходящий канал, существует вероятность того, что коммутатор будет получать пакеты быстрее, чем может вместить общий канал. В этом случае коммутатор вынужден буферизировать эти пакеты в своей памяти. Если коммутатор получает пакеты быстрее, чем может отправить их в течение длительного периода времени, то в конце концов у него закончится буферное пространство, и некоторые пакеты придется отбросить. Когда коммутатор работает в таком состоянии, говорят, что он *перегружен*.

Основные выводы

В заключение следует сказать, что статистическое мультиплексирование определяет экономически эффективный способ для множества пользователей (например, использование потоков данных между хостами) совместно использовать ресурсы сети (каналы и узлы) с тонкой настройкой. Оно использует пакет как минимальную единицу для выделения каналов сети различным потокам, позволяя каждому коммутатору распределять использование физических каналов на основе пакетов. Справедливое распределение пропускной способности каналов между разными потоками и управление перегрузкой при ее возникновении являются основными задачами статистического мультиплексирования.

Глава 1.2.4. Поддержка общих служб

Ранее мы сосредоточились на вызовах, связанных с обеспечением экономически выгодного подключения группы хостов, но рассматривать компьютерную сеть как доставку пакетов между несколькими компьютерами было бы слишком просто. Если быть более точным, сеть стоит рассматривать как средство для взаимодействия набора прикладных процессов, распределенных по этим компьютерам. Другими словами, следующим требованием компьютерной сети является предоставление возможности для прикладных программ, запущенных на подключенных к сети хостах, взаимодействовать

между собой в значимом смысле. С точки зрения разработчика приложений сеть должна упростить его жизнь.

Когда двум прикладным программам нужно взаимодействовать друг с другом, должно происходить много сложных вещей, кроме простой отправки сообщения с одного хоста на другой. Один из вариантов заключается в том, чтобы разработчики приложений встроили всю эту сложную функциональность в каждую программу. Однако поскольку многие приложения требуют общих сервисов, гораздо логичнее реализовать эти общие сервисы один раз и затем позволить разработчику приложений строить приложение, используя эти сервисы. Задача для дизайнера сети заключается в определении правильного набора общих сервисов. Цель состоит в том, чтобы скрыть сложность сети от приложения, не ограничивая разработчика приложений.

Интуитивно мы рассматриваем сеть как логические каналы, по которым процессы на уровне приложений могут взаимодействовать друг с другом; каждый канал предоставляет набор услуг, необходимых данному приложению. Другими словами, подобно тому, как мы используем облако для абстрактного представления связи между множеством компьютеров, мы теперь думаем о канале как о соединении одного процесса с другим. На рисунке 1.7 показана пара процессов на уровне приложений, общающихся по логическому каналу, который, в свою очередь, реализован поверх облака, соединяющего набор хостов. Мы можем представить канал как трубу, соединяющую два приложения, так что приложение-отправитель может поместить данные в один конец и ожидать, что они будут доставлены по сети приложению, находящемуся на другом конце трубы.

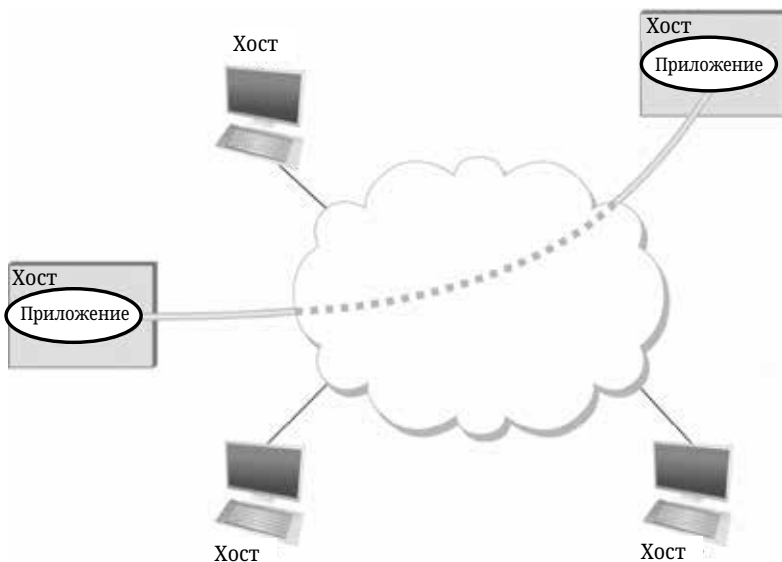


Рисунок 1.7. Процессы, взаимодействующие по абстрактному каналу.

Как и любая абстракция, логические каналы «процесса-к-процессу» реализуются поверх коллекции физических «хост-к-хосту» каналов. Это является основой слоистости, краеугольным камнем сетевых архитектур, которые обсуждаются в следующей главе.

Основная задача здесь — определить, какую функциональность должны предоставлять каналы приложениям. Например, нужно ли приложению гарантировать, что сообщения, отправленные по каналу, будут доставлены, или допустимо, если некоторые сообщения не придут? Необходимо ли, чтобы сообщения приходили к процессу-получателю в том же порядке, в котором они были отправлены, или получатель не заботится о порядке, в котором приходят сообщения? Нужно ли сети обеспечивать защиту от прослушивания канала третьими лицами или конфиденциальность не представляет интереса?

В общем случае сеть предоставляет различные типы каналов, и каждое приложение выбирает тип, который лучше всего соответствует его потребностям. В остальной части этой главы иллюстрируется подход к определению полезных каналов.

Определение общих шаблонов коммуникации

Проектирование абстрактных каналов начинается с понимания потребностей в коммуникации представительной коллекции приложений, затем извлечения их общих требований к коммуникации и, наконец, включения функциональности, которая удовлетворяет эти требования в сети.

Одним из первых приложений, поддерживаемых в любой сети, является программа доступа к файлам, такая как *Протокол передачи файлов* (File Transfer Protocol, FTP) или *Сетевая файловая система* (Network File System, NFS). Хотя многие детали могут отличаться, например, передаются ли по сети полные файлы или только отдельные блоки файла для чтения/записи в данный момент, коммуникационная составляющая удаленного доступа к файлам характеризуется парой процессов: один запрашивает чтение или запись файла, а второй процесс выполняет этот запрос. Хост, запрашивающий доступ к файлу, называется *клиентом*, а хост, обеспечивающий доступ к файлу, называется *сервером*.

Чтение файла включает отправку клиентом небольшого запроса на сервер и ответ сервера большим сообщением, содержащим данные файла. Запись работает наоборот: клиент отправляет серверу большое сообщение с данными для записи на диск, и сервер отвечает небольшим сообщением, подтверждающим, что запись на диск выполнена.

Цифровая библиотека является более сложным приложением по сравнению с приложением по передаче файлов, но она требует аналогичных коммуникационных услуг. Например, *Ассоциация вычислительной техники* (Association for Computing Machinery, ACM) управляет крупной цифровой библиотекой литературы по компьютерным наукам на сайте по адресу:

<http://portal.acm.org/dl.cfm>

Эта библиотека имеет широкий спектр функций поиска и просмотра, чтобы помочь пользователям найти статьи, которые они ищут, но в конечном итоге большая часть ее работы заключается в ответе на запросы пользователей на получение файлов, таких как электронные копии журнальных статей.

Используя доступ к файлам, цифровую библиотеку и два видеоприложения, описанные во введении (видеоконференции и «видео по требованию») в качестве представительного образца, мы можем предоставить следующие два типа каналов: каналы «запрос/ответ» и каналы *поточковых сообщений*. Канал «запрос/ответ» будет использоваться для приложений передачи файлов и цифровой библиотеки. Он будет гарантировать, что каждое сообщение, отправленное одной стороной, будет получено другой стороной и что будет доставлена только одна копия каждого сообщения. Канал «запрос/ответ» также может защищать конфиденциальность и целостность данных, передаваемых между клиентскими и серверными процессами, чтобы неавторизованные стороны не могли читать или изменять данные.

Канал потоковых сообщений мог бы использоваться как приложениями «видео по требованию», так и видеоконференциями, предоставляя поддержку как одностороннего, так и двустороннего трафика, а также различные характеристики задержки. Канал потоковых сообщений может не гарантировать доставку всех сообщений, поскольку видеоприложение может функционировать достаточно хорошо, даже если некоторые видеок кадры не будут получены. Однако он должен гарантировать, что все доставленные сообщения приходят в том же порядке, в котором они были отправлены, чтобы избежать отображения кадров в неправильной последовательности. Как и канал «запрос/ответ», канал потоковых сообщений может нуждаться в защите конфиденциальности и целостности видеоданных. Наконец, каналу потоковых сообщений может потребоваться поддержка

многоадресной передачи данных, чтобы несколько сторон могли участвовать в телеконференции или просмотре видео.

Хотя для проектировщика сети типично стремление к наименьшему числу абстрактных типов каналов, которые могут обслуживать наибольшее число приложений, есть опасность в попытке обойтись слишком малым числом абстракций каналов. Проще говоря, если у вас есть молоток, то все выглядит как гвоздь. Например, если у вас есть только каналы потока сообщений и запросов/ответов, то возникает соблазн использовать их для следующего приложения, даже если ни один из этих типов не обеспечивает именно ту семантику, которая нужна приложению. Таким образом, разработчики сетей, вероятно, будут изобретать новые типы каналов и добавлять опции к существующим каналам до тех пор, пока прикладные программисты будут изобретать новые приложения.

Также следует отметить, что независимо от того, какую функциональность предоставляет данный канал, возникает вопрос, где эта функциональность реализуется. Во многих случаях проще всего рассматривать подключение от хоста к хосту в основной сети просто как *передачу битов*, с любой семантикой высокого уровня, предоставляемой на конечных хостах. Преимущество такого подхода заключается в том, что он сохраняет коммутаторы в средней части сети как можно более простыми (они просто пересылают пакеты), но это требует от конечных хостов взять на себя значительную часть ответственности за поддержку семантически богатых каналов между процессами. Альтернативой является внедрение дополнительной функциональности в коммутаторы, что позволяет конечным хостам быть «простыми» устройствами (например, телефонными аппаратами). Мы будем рассматривать вопрос о том, как различные сетевые сервисы распределяются между коммутаторами и конечными хостами, как постоянно появляющуюся проблему в сетевом проектировании.

Надежная доставка сообщений

Как подразумевают рассмотренные выше примеры, надежная доставка сообщений является одной из наиболее важных функций, которую может предоставить сеть. Однако сложно определить, как обеспечить эту надежность, не понимая, как могут происходить ошибки в работе сети. Во-первых, важно осознать, что компьютерные сети существуют не в идеальном мире. Машины выходят из строя и перезагружаются, волокна могут быть оборваны, электрические помехи могут повредить биты в передаваемых данных, коммутаторы могут исчерпать буферное пространство, а программное обеспечение, управляющее аппаратным обеспечением, может содержать ошибки и иногда отправлять пакеты в никуда. Таким образом, одним из основных требований к сети является восстановление от определенных видов отказов, чтобы приложения не занимались этими проблемами.

Существует три основных класса отказов, о которых должны беспокоиться проектировщики сетей. Во-первых, при передаче пакета по физическому каналу могут возникать битовые ошибки; то есть единица может превращаться в ноль или наоборот. Иногда ошибки касаются отдельных битов, но чаще всего возникают *серийные ошибки* — несколько последовательных битов повреждаются. Ошибки битов обычно возникают из-за внешних воздействий, таких как удары молнии, скачки напряжения и микроволновые печи, которые мешают передаче данных. Хорошая новость в том, что такие ошибки битов довольно редки и в среднем затрагивают лишь один из каждых 10^6 до 10^7 битов на обычном медном кабеле и один из каждых 10^{12} до 10^{14} битов на обычном оптоволоконном кабеле. Как мы увидим далее, существуют техники, которые с высокой вероятностью обнаружат эти ошибки битов. Обнаружив ошибку, иногда можно ее исправить — если мы знаем, какой бит или биты повреждены, мы можем просто их поменять, — в других случаях повреждения настолько серьезны, что необходимо отбросить весь пакет. В таком случае отправитель может быть вынужден повторно отправить пакет.

Второй класс отказов связан с пакетами, а не с отдельными битами; другими словами, сеть теряет полный пакет. Одной из причин такого отказа может быть наличие неисправимой ошибки бита в пакете, из-за чего его приходится отбросить. Однако более вероятной причиной является перегрузка одного из узлов, обрабатывающего пакет — например, коммутатора, пересылающего его с одной линии на другую — из-за чего у него нет места для хранения пакета, и он вынужден его отбросить. Это и есть проблема, о которой мы говорили ранее — проблема перегрузки. Реже встречается ситуация, когда программное обеспечение, работающее на одном из узлов, обрабатывающих пакет, допускает ошибку. Например, оно может неправильно переслать пакет по неправильной линии, и пакет, таким образом, никогда не достигнет своего конечного пункта назначения. Как мы увидим, одной из основных сложностей при работе с потерянными пакетами является различие между пакетом, который действительно потерян, и пакетом, который просто задерживается в пути к месту назначения.

Третий класс отказов связан с уровнем узлов и линков. Другими словами, физическое соединение оборвано, или компьютер, к нему подключенный, вышел из строя. Это может произойти из-за сбоя программного обеспечения, отключения питания или неосмотрительных действий оператора экскаватора. Часто встречаются отказы из-за неправильной конфигурации устройства сети. Хотя все эти отказы в конечном итоге могут быть устранены, они могут оказать огромное влияние на сеть на продолжительное время. Тем не менее они не обязательно должны полностью парализовать сеть. Например, в пакетно-коммутируемой сети иногда можно обойти неисправный узел или связь. Одной из сложностей при работе с этим третьим классом отказов является различие между компьютером, который вышел из строя, и компьютером, который просто медленно работает, или, в случае со связью, между оборванной связью и связью, которая очень нестабильна и поэтому вводит высокое количество ошибок битов.

Основные выводы

Основная идея, которую следует вынести из этого обсуждения, заключается в том, что определение полезных каналов включает как понимание требований приложений, так и осознание ограничений обычной технологии. Задача состоит в том, чтобы заполнить разрыв между тем, чего ожидает приложение, и тем, что может предоставить обычная технология. Этот разрыв иногда называют *семантическим разрывом*.

Глава 1.2.5. Управляемость

Последнее требование, которое, как кажется, часто игнорируется или оставляется на последний момент (как и в нашем случае), заключается в том, что сетью необходимо управлять. Управление сетью включает обновление оборудования по мере роста сети для обработки большего трафика или подключения большего числа пользователей, устранение неполадок в сети, когда что-то идет не так или производительность не соответствует ожиданиям, а также добавление новых функций для поддержки новых приложений. Исторически сложилось так, что управление сетью было трудоемким процессом, и хотя вряд ли удастся полностью исключить участие людей, этот аспект все чаще решается с помощью автоматизации и самовосстанавливающихся систем.

Это требование частично связано с вопросом масштабируемости, обсуждавшимся ранее, — по мере того как Интернет масштабировался до поддержки миллиардов пользователей и по крайней мере сотен миллионов узлов, проблемы поддержания его корректной работы и правильной настройки новых устройств по мере их добавления становились все более сложными. Настройка одного маршрутизатора в сети часто является задачей для квалифицированного специалиста; настройка тысяч маршрутизаторов и выяснение причин неправильного поведения такой крупной сети могут стать задачей, выходящей за рамки возможностей одного человека. Поэтому автоматизация становится столь важной.

Один из способов облегчить управление сетью — избегать изменений. Как только сеть начинает работать, ее просто не стоит трогать. Такое мышление выявляет основное противоречие между *стабильностью* и *скоростью внедрения новых функций* (скорость, с которой новые возможности вводятся в сеть). Придерживание стабильности было подходом, который телекоммуникационная отрасль (не говоря уже об университетских системных администраторах и корпоративных ИТ-отделах) применяла на протяжении многих лет, делая ее одной из самых медлительных и избегающих рисков отраслей. Но недавний взрыв облачных технологий изменил эту динамику, сделав необходимым приведение стабильности и скорости внедрения функций в более сбалансированное состояние. Влияние облачных технологий на сеть является темой, которая поднимается снова и снова на протяжении всей книги и которой мы уделяем особое внимание в подразделе «Перспектива» после каждого раздела. На данный момент достаточно сказать, что управление быстро развивающейся сетью является центральной задачей в сетевых технологиях сегодня.

Глава 1.3. Архитектура

Предыдущая глава установила довольно значительный набор требований к проектированию сети — компьютерная сеть должна обеспечивать общую, экономически эффективную, справедливую и надежную связь между большим числом компьютеров. Как будто этого было недостаточно, сети не остаются неизменными, они должны эволюционировать, чтобы учитывать изменения как в базовых технологиях, на которых они основаны, так и в требованиях, предъявляемых к ним прикладными программами. Более того, сети должны быть управляемыми людьми с разным уровнем навыков. Проектирование сети, отвечающей этим требованиям, — непростая задача.

Чтобы справиться с этой сложностью, разработчики сетей создали общие схемы, обычно называемые *сетевыми архитектурами*, которые направляют проектирование и внедрение сетей. Эта глава более подробно определяет, что мы имеем в виду под сетевой архитектурой, вводя основные идеи, общие для всех сетевых архитектур. Также вводятся две из наиболее широко известных архитектур — OSI (или 7-уровневая) архитектура и Интернет-архитектура.

Глава 1.3.1. Многослойность и протоколы

Абстракция — это сокрытие деталей реализации за четко определенным интерфейсом. Она является фундаментальным инструментом, используемым системными разработчиками для управления сложной системой. Идея абстракции заключается в том, чтобы определить модель, которая может захватывать какой-то важный аспект системы, инкапсулировать эту модель в объекте, который предоставляет интерфейс, которым могут управлять другие компоненты системы, и скрывать детали того, как объект реализован, от пользователей объекта. Задача состоит в том, чтобы определить абстракции, которые одновременно предоставляют услугу, полезную во многих ситуациях, и которые могут быть эффективно реализованы в базовой системе. Именно это мы делали, когда вводили идею канала в предыдущем разделе: мы предоставляли абстракцию для приложений, которая скрывает сложность сети от разработчиков приложений.

Прикладные программы
Каналы процесса к процессу
Соединение между хостами
Аппаратное обеспечение

Рисунок 1.8. Пример многоуровневой сетевой системы.

Абстракции естественным образом приводят к созданию уровней, особенно в сетевых системах. Основная идея заключается в том, что вы начинаете с услуг, предоставляемых базовым оборудованием, а затем добавляете последовательность уровней, каждый из которых предоставляет более высокий (более абстрактный) уровень услуг. Услуги, предоставляемые на высоких уровнях, реализуются с использованием услуг, предоставляемых низкими уровнями. Опираясь на рассмотрение требований, приведенное в предыдущей главе, можно представить простую сеть, имеющую два уровня абстракции, расположенных между прикладной программой и базовым оборудованием, как показано на рис. 1.8. В этом случае уровень, непосредственно находящийся над оборудованием, может обеспечивать подключение между хостами, абстрагируя тот факт, что между любыми двумя хостами может быть произвольная и сложная топология сети. Следующий уровень строится на доступной услуге связи между хостами и предоставляет поддержку каналов от процесса к процессу, абстрагируя тот факт, что, например сеть иногда теряет сообщения.

Многослойность обеспечивает две полезные функции. Во-первых, она разбивает задачу построения сети на более мелкие компоненты. Вместо реализации монолитного программного обеспечения, которое делает все, что вам когда-либо понадобится, вы можете реализовать несколько уровней, каждый из которых решает определенную часть задачи. Во-вторых, она обеспечивает более модульный дизайн. Если вы решите добавить новую услугу, вам потребуется изменить функциональность только на одном уровне, повторно используя функции, предоставляемые на всех остальных уровнях.

Представление системы в виде линейной последовательности уровней является упрощением. Зачастую на каждом уровне системы предоставляется множество абстракций, каждая из которых предоставляет разные услуги для более высоких уровней, но строится на одних и тех же низкоуровневых абстракциях. Чтобы понять это, рассмотрим два типа каналов, обсуждавшихся в предыдущей главе. Один обеспечивает услугу «запрос/ответ», а другой поддерживает услугу потоковой передачи сообщений. Эти два канала могут быть альтернативными предложениями на каком-то уровне многоуровневой сетевой системы, как показано на рис. 1.9.

Используя это рассмотрение уровней в качестве основы, мы теперь готовы более точно обсудить архитектуру сети. Для начала, абстрактные объекты, составляющие уровни сетевой системы, называются *протоколами*. То есть протокол предоставляет услугу связи, которую объекты более высокого уровня (например, прикладные процессы или протоколы более высокого уровня) используют для обмена сообщениями. Например, мы можем представить сеть, поддерживающую протокол запрос/ответ и протокол потоковой передачи сообщений, соответствующие каналам запрос/ответ и потоковой передачи сообщений, обсуждавшимся выше.

Прикладные программы	
Канал запрос/ответ	Канал потоковой передачи данных
Соединение между хостами	
Аппаратное обеспечение	

Рисунок 1.9. Многоуровневая система с альтернативными абстракциями, доступными на данном уровне.

Каждый протокол определяет два различных интерфейса. Во-первых, он определяет *сервисный интерфейс* для других объектов на том же компьютере, которые хотят использовать его коммуникационные услуги. Этот интерфейс сервиса определяет операции, которые локальные объекты могут выполнять с протоколом. Например, протокол запрос/ответ будет поддерживать операции, с помощью которых приложение может отправ-

лять и получать сообщения. Реализация протокола HTTP может поддерживать операцию для получения страницы HTML с удаленного сервера. Приложение, такое как веб-браузер, будет вызывать такую операцию каждый раз, когда браузеру нужно получить новую страницу (например, когда пользователь нажимает на ссылку на текущей странице).

Во-вторых, протокол определяет *пиринговый (или одноранговый) интерфейс* (peer interface) для своего аналога (peer) на другой машине. Этот второй интерфейс определяет форму и значение сообщений, обмен которых между аналогами протокола реализует коммуникационную услугу. Это определяет способ, которым протокол запрос/ответ на одной машине общается со своим аналогом на другой машине. В случае с HTTP, например, спецификация протокола подробно описывает, как форматируется команда *GET*, какие аргументы можно использовать с этой командой и как веб-сервер должен отвечать, когда получает такую команду.

Вкратце, протокол определяет коммуникационную услугу, которую он предоставляет локально (сервисный интерфейс), вместе с набором правил, регулирующих обмен сообщениями между аналогами протокола для реализации этой услуги (интерфейс для аналогов). Эта ситуация иллюстрируется на рис. 1.10.

За исключением аппаратного уровня, где аналоги напрямую общаются друг с другом через физическую среду, одноранговое общение является косвенным, то есть каждый протокол общается со своим аналогом, передавая сообщения некоторому протоколу более низкого уровня, который, в свою очередь, доставляет сообщение своему аналогу. Кроме того, на любом данном уровне может быть несколько протоколов, каждый из которых предоставляет различные коммуникационные услуги. Поэтому мы представляем набор протоколов, составляющих сетевую систему, в виде *графа протоколов*. Узлы графа соответствуют протоколам, а ребра представляют отношения «зависимости». Например, на рис. 1.11 иллюстрируется граф протоколов для гипотетической многослойной системы, которую мы обсуждали — протоколы RRP (Request/Reply Protocol, протокол запрос/ответ) и MSP (Message Stream Protocol, протокол потока сообщений) реализуют два различных типа каналов процесс-процесс, и оба зависят от протокола Host-to-Host Protocol (NHP, протокол хоста к хосту), который предоставляет услугу соединения хост-хост.

В этом примере предположим, что программа доступа к файлам на хосте 1 хочет отправить сообщение своему аналогу на хосте 2, используя коммуникационную услугу, предлагаемую RRP. В этом случае программа доступа к файлам просит RRP отправить сообщение от ее имени. Для общения со своим аналогом RRP вызывает услуги NHP, который, в свою очередь, передает сообщение своему аналогу на другой машине. Как только сообщение поступает в экземпляр NHP на хосте 2, NHP передает сообщение вверх к RRP, который, в свою очередь, доставляет сообщение программе доступа к файлам. В этом конкретном случае говорится, что приложение использует услуги стека протоколов RRP/NHP.

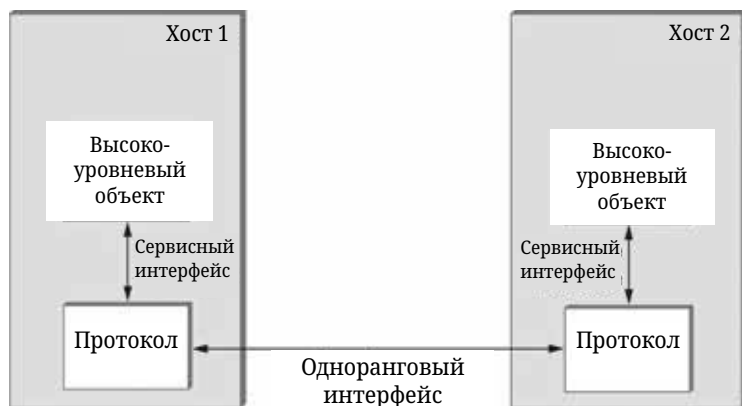


Рисунок 1.10. Сервисные интерфейсы и одноранговые (пиринговые) интерфейсы.

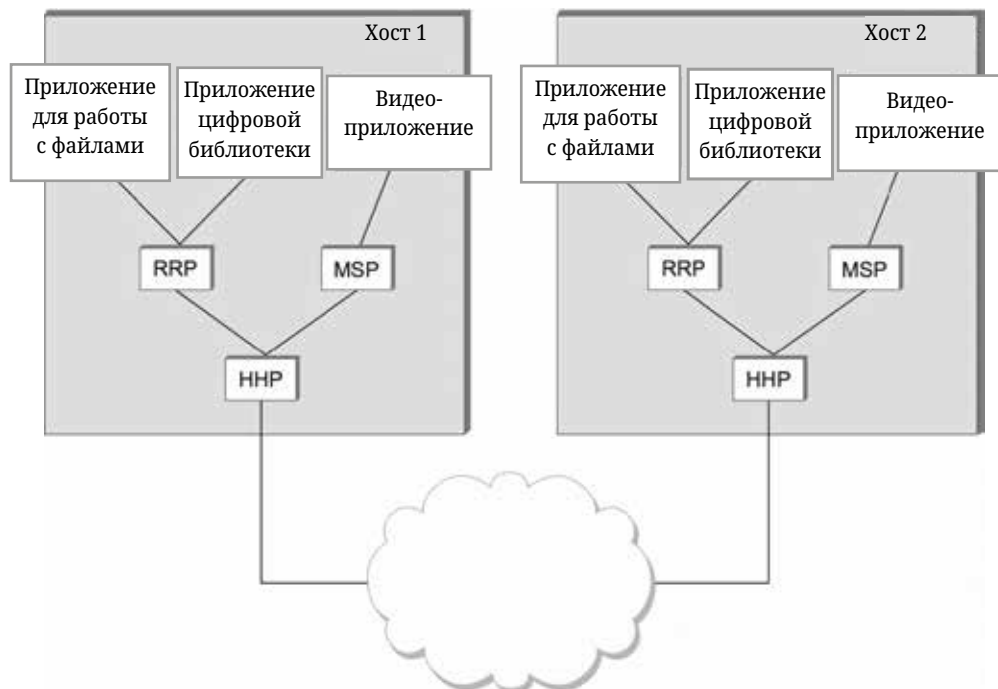


Рисунок 1.11. Пример графа протокола.

Заметьте, что термин *протокол* используется в двух разных значениях. Иногда он относится к абстрактным интерфейсам — то есть к операциям, определяемым интерфейсом сервиса, и форме и значению сообщений, обмениваемых между партнерами, а иногда он относится к модулю, который фактически реализует эти два интерфейса. Чтобы различать интерфейсы и модуль, который реализует эти интерфейсы, мы обычно называем первые, то есть интерфейсы, *спецификацией протокола*. Спецификации обычно выражаются с помощью комбинации псевдокода, диаграмм переходов состояний, изображений форматов пакетов и других абстрактных обозначений. Должно быть так, что данный протокол может быть реализован различными способами разными программистами, при условии, что каждый придерживается спецификации. Сложность заключается в том, чтобы обеспечить успешный обмен сообщениями между двумя разными реализациями одной и той же спецификации. Считается, что два или более протокольных модуля, которые точно реализуют спецификацию протокола, *взаимодействуют друг с другом*.

Мы можем представить себе множество различных протоколов и графов протоколов, которые удовлетворяют требованиям связи коллекции приложений. К счастью, существуют стандартизирующие организации, такие как *рабочая группа инженеров Интернета* (Internet Engineering Task Force, IETF) и *международная организация по стандартизации* (International Standards Organization, ISO), которые устанавливают правила для конкретного графа протоколов. Набор правил, регулирующих форму и содержание графа протоколов, мы называем *сетевой архитектурой*. Хотя это выходит за рамки данного обсуждения, стандартизирующие организации разработали четко определенные процедуры для внедрения, проверки и, наконец, утверждения протоколов в своих архитектурах. Вскоре мы кратко опишем архитектуры, определяемые IETF и ISO, но сначала нам нужно понять две дополнительные вещи о механике многослойности протоколов.

Глава 1.3.2. Инкапсуляция

Рассмотрим, что происходит, когда одна из прикладных программ отправляет сообщение своему партнеру, передавая его через RRP. С точки зрения RRP, сообщение, которое оно получает от приложения, представляет собой не интерпретированную строку байтов. RRP не важно, что эти байты представляют собой массив целых чисел, электронное письмо, цифровое изображение или что-то еще; оно просто отвечает за отправку их своему партнеру. Однако RRP должно передать управляющую информацию своему партнеру, инструктируя его, как обрабатывать сообщение при получении. RRP делает это, прикрепляя *заголовок* к сообщению. Заголовок — это небольшая по объему структура данных (от нескольких байтов до нескольких десятков байтов), которая используется для обмена информацией между партнерами. Как следует из названия, заголовки прикрепляются к началу сообщения. Однако в некоторых случаях эта информация для одноранговой сети отправляется в конце сообщения, и тогда она называется *трейлером*. Точный формат заголовка, прикрепленного RRP, определяется его спецификацией протокола. Остальная часть сообщения — это данные, передаваемые от имени приложения, она называется *телом сообщения* или *полезной нагрузкой*. Мы говорим, что данные приложения *инкапсулированы* в новом сообщении, созданном RRP.

Этот процесс инкапсуляции затем повторяется на каждом уровне протокольного графа. Например, ННР инкапсулирует сообщение RRP, прикрепляя свой собственный заголовок. Если теперь предположить, что ННР отправляет сообщение своему партнеру по сети, то, когда сообщение прибывает на конечный хост, оно будет обработано в обратном порядке: ННР сначала интерпретирует заголовок ННР в начале сообщения (то есть выполняет любые действия, соответствующие содержимому заголовка) и передает тело сообщения (но не заголовок ННР) вверх к RRP, который выполняет действия, указанные в заголовке RRP, прикрепленном его партнером, и передает тело сообщения (но не заголовок RRP) приложению. Сообщение, переданное от RRP приложению на хосте 2, является точно таким же сообщением, которое приложение передало RRP на хосте 1; приложение не видит никаких заголовков, которые были прикреплены для реализации сервисов связи нижнего уровня. Этот процесс иллюстрируется на рис. 1.12. Заметьте, что в этом примере узлы в сети (например, коммутаторы и маршрутизаторы) могут проверять заголовок ННР в начале сообщения.



Рисунок 1.12. Высокоуровневые сообщения инкапсулируются внутри низкоуровневых сообщений.

Заметьте, когда мы говорим, что протокол низкого уровня не интерпретирует сообщение, переданное ему протоколом высокого уровня, мы имеем в виду, что он не знает, как извлечь какой-либо смысл из данных, содержащихся в сообщении. Однако иногда протокол низкого уровня применяет некоторое простое преобразование к данным, переданным ему, например, сжимает или шифрует их. В этом случае протокол преобразует все тело сообщения, включая как исходные данные приложения, так и все заголовки, прикрепленные к этим данным протоколами более высокого уровня.

Глава 1.3.3. Мультиплексирование и демultipлексирование

Напомним, что фундаментальная идея коммутации пакетов заключается в мультиплексировании множества потоков данных через одну физическую связь. Эта же идея применяется к всему протокольному графу, а не только к узлам коммутации. На рис. 1.11, например, можно рассматривать RRP как реализацию логического канала связи, по которому сообщения от двух различных приложений мультиплексируются на исходном хосте, а затем демultipлексируются обратно к соответствующему приложению на конечном хосте.

Практически это означает, что заголовок, который RRP прикрепляет к своим сообщениям, содержит идентификатор, фиксирующий, к какому приложению принадлежит сообщение. Мы называем этот идентификатор *ключом демultipлексирования RRP*, или сокращенно *демукс-ключом*. На исходном хосте RRP включает соответствующий демукс-ключ в свой заголовок. Когда сообщение доставляется на RRP на конечном хосте, оно снимает заголовок, проверяет демукс-ключ и демultipлексирует сообщение к правильному приложению.

RRP не уникален в своей поддержке мультиплексирования; почти каждый протокол реализует этот механизм. Например, NRP имеет свой собственный демукс-ключ, чтобы определить, какие сообщения передавать вверх к RRP, а какие — к MSP. Однако среди протоколов — даже тех, что находятся внутри одной сетевой архитектуры — нет единого соглашения о том, что именно представляет собой демукс-ключ. Некоторые протоколы используют 8-битное поле (что означает, что они могут поддерживать только 256 протоколов высокого уровня), другие используют 16- или 32-битные поля. Также некоторые протоколы имеют одно поле демultipлексирования в своем заголовке, в то время как другие имеют два поля демultipлексирования. В первом случае один и тот же демукс-ключ используется с обеих сторон связи, а во втором случае каждая сторона использует разные ключи для идентификации протокола высокого уровня (или прикладной программы), к которому должно быть доставлено сообщение.

Глава 1.3.4. 7-уровневая модель OSI

ISO была одной из первых организаций, которая формально определила общий способ подключения компьютеров. Их архитектура, называемая архитектурой *взаимодействия открытых систем* (Open Systems Interconnection, OSI) и проиллюстрированная на рис. 1.13, определяет разделение сетевой функциональности на семь уровней, где один или несколько протоколов реализуют функциональность, назначенную данному уровню. В этом смысле представленная схема не является графом протоколов как таковым, а скорее *эталонной моделью* для графа протоколов. Она часто называется моделью из 7 уровней. Хотя сегодня не существует сети, основанной на OSI, терминология, которую она определила, все еще широко используется, поэтому стоит ее кратко рассмотреть.

Начиная с самого нижнего уровня и поднимаясь вверх, *физический* уровень обрабатывает передачу сырых битов по коммуникационной линии. *Канальный* уровень затем собирает поток битов в более крупную сущность, называемую *кадром*. Сетевые адаптеры, вместе с драйверами устройств, работающими в операционной системе узла, обычно реализуют уровень канала данных. Это означает, что кадры (не сырые биты) фактически до-

ставляются на узлы. *Сетевой* уровень обрабатывает маршрутизацию между узлами в сети с коммутацией пакетов. На этом уровне единица данных, обмениваемая между узлами, обычно называется *пакетом*, а не кадром, хотя по сути они одно и то же. Нижние три уровня реализуются на всех сетевых узлах, включая коммутаторы в сети и хосты, подключенные к внешней части сети. *Транспортный* уровень затем реализует то, что мы до этого момента называли каналом процесс-процесс. Здесь единица данных, обмениваемая между узлами, обычно называется *сообщением*, а не пакетом или кадром. Транспортный уровень и уровни выше него обычно работают только на конечных хостах, а не на промежуточных коммутаторах или маршрутизаторах.

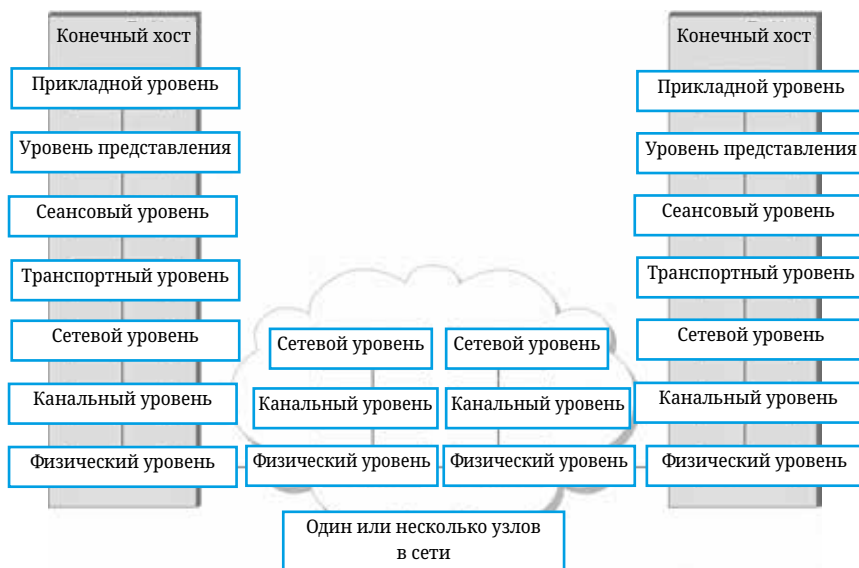


Рисунок 1.13. 7-уровневая модель OSI.

Перепрыгнув к верхнему (седьмому) уровню и опускаясь вниз, мы находим *прикладной* уровень. Протоколы уровня приложения включают такие вещи, как *протокол передачи гипертекста* (Hypertext Transfer Protocol, HTTP), который является основой Всемирной паутины и позволяет веб-браузерам запрашивать страницы с веб-серверов. Ниже находится уровень *представления*, который занимается форматом данных, обмениваемых между партнерами. Например, определяет, является ли длина целого числа 16, 32 или 64 бита, передается ли старший байт первым либо последним или как форматируется видеопоток. Наконец, *сеансовый* уровень предоставляет пространство имен, которое используется для объединения потенциально различных транспортных потоков, являющихся частью одного приложения. Например, он может управлять аудиопотоком и видеопотоком, которые объединяются в приложении для видеоконференций.

Глава 1.3.5. Архитектура Интернета

Архитектура Интернета, которая иногда также называется архитектурой TCP/IP по названию двух основных протоколов, показана на рис. 1.14. Альтернативное представление дано на рисунке 1.15. Архитектура Интернета эволюционировала благодаря опыту работы над более ранней сетью с коммутацией пакетов под названием ARPANET. Как Интернет, так и ARPANET были профинансированы *Агентством перспективных исследовательских проектов* (Advanced Research Projects Agency, ARPA), одним из агентств по финансированию исследований и разработок Министерства обороны США. Интернет и ARPANET суще-

ствовали до архитектуры OSI, и опыт, полученный при их создании, оказал значительное влияние на эталонную модель OSI.

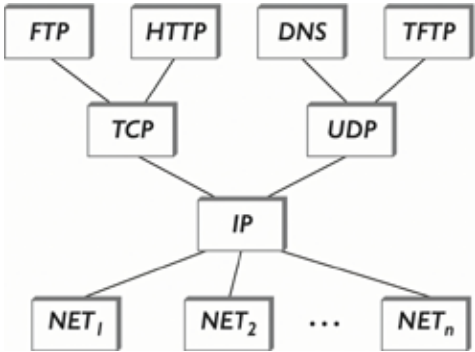


Рисунок 1.14. Граф интернет-протокола.

Хотя 7-уровневую модель OSI можно, при некотором воображении, применить к Интернету, вместо нее часто используется более простая стековая модель. На самом нижнем уровне находится множество сетевых протоколов, обозначаемых как NET1, NET2 и так далее. На практике эти протоколы реализуются комбинацией аппаратного обеспечения (например, сетевой адаптер) и программного обеспечения (например, драйвер сетевого устройства). Например, на этом уровне могут находиться протоколы Ethernet или беспроводные протоколы (такие как стандарты Wi-Fi 802.11). (Эти протоколы, в свою очередь, могут включать несколько подуровней, но архитектура Интернета ничего о них не предполагает.) Следующий уровень состоит из одного протокола — *Интернет-протокола* (Internet Protocol, IP). Этот протокол поддерживает взаимосвязь множества сетевых технологий в одну логическую интернетсеть. Уровень выше IP содержит два основных протокола — *протокол управления передачей* (Transmission Control Protocol, TCP) и *протокол пользовательских дейтаграмм* (User Datagram Protocol, UDP). TCP и UDP предоставляют альтернативные логические каналы для приложений: TCP обеспечивает надежный канал передачи байтового потока, а UDP предоставляет ненадежный канал доставки дейтаграмм (*дейтаграмму* можно рассматривать как синоним сообщения). В терминологии Интернета TCP и UDP иногда называют *сквозными* протоколами, хотя их также корректно называть *транспортными* протоколами.

Приложение	
TCP	UDP
IP	
Подсеть	

Рисунок 1.15. Альтернативное представление архитектуры Интернета.
Уровень «подсети» исторически назывался уровнем «сети», а сейчас часто называется «уровнем 2» (под влиянием модели OSI).

Над транспортным уровнем работает ряд прикладных протоколов, таких как HTTP, FTP, Telnet (удаленный вход) и протокол простой передачи почты (Simple Mail Transfer Protocol, SMTP), которые обеспечивают взаимодействие популярных приложений. Чтобы понять разницу между протоколом уровня приложения и самим приложением, подумайте обо всех различных веб-браузерах, которые существуют или существовали (Firefox,

Chrome, Safari, Netscape, Mosaic, Internet Explorer). Существует также множество различных реализаций веб-серверов. Причина, по которой вы можете использовать любую из этих программ для доступа к определенному сайту в Интернете, заключается в том, что все они соответствуют одному и тому же протоколу уровня приложения: HTTP. Путаница возникает из-за того, что один и тот же термин иногда применяется и к приложению, и к протоколу уровня приложения, который оно использует (например, FTP часто используется как название приложения, которое реализует протокол FTP).

Большинство людей, активно работающих в сфере сетевых технологий, знакомы как с архитектурой Интернета, так и с 7-уровневой архитектурой OSI, и существует общее согласие относительно того, как соотносятся уровни в разных архитектурах. Считается, что прикладной уровень Интернета находится на 7-м уровне, транспортный уровень — на 4-м, уровень IP (интернет-работы или просто сети) — на 3-м, а уровень каналов или подсетей ниже IP — на 2-м.

IETF и стандартизация

- Хотя мы называем ее «архитектурой Интернета», а не «архитектурой IETF», справедливо сказать, что именно IETF является основным органом стандартизации, ответственным за ее определение, а также за спецификацию многих ее протоколов, таких как TCP, UDP, IP, DNS и BGP. Но архитектура Интернета также включает множество протоколов, определенных другими организациями, включая стандарты Ethernet и Wi-Fi IEEE 802.11, веб-спецификации HTTP/HTML W3C, стандарты сетей 4G и 5G 3GPP и стандарты кодирования видео H.232 ITU-T, чтобы назвать лишь несколько.
- Помимо определения архитектур и спецификации протоколов существуют также другие организации, которые поддерживают более значимую цель интероперабельности (или просто взаимодействия). Один из примеров — IANA (Администрация адресного пространства Интернета (Internet Assigned Numbers Authority), которая, как следует из ее названия, отвечает за выдачу уникальных идентификаторов, необходимых для работы протоколов. В свою очередь IANA является департаментом в составе ICANN (Корпорации по управлению доменными именами и IP-адресами в Интернете, Internet Corporation for Assigned Names and Numbers) — некоммерческой организации, которая отвечает за общее руководство Интернетом.
- Архитектура Интернета имеет три особенности, которые стоит выделить. Во-первых, как иллюстрируется на рис. 1.15, архитектура Интернета не подразумевает строгого разделения по уровням. Приложение может обходить определенные транспортные уровни и напрямую использовать IP или одну из базовых сетей. Фактически программисты могут свободно определять новые абстракции каналов или приложения, работающие поверх любых существующих протоколов.
- Во-вторых, если внимательно посмотреть на граф протоколов на рис. 1.14, можно заметить форму песочных часов — широкую сверху, узкую посередине и снова широкую внизу. Эта форма на самом деле отражает центральную философию архитектуры. То есть IP служит основной точкой архитектуры — он определяет общий метод обмена пакетами среди множества сетей. Выше IP может быть бесконечное количество транспортных протоколов, каждый из которых предлагает различную абстракцию канала для приложений. Таким образом, вопрос доставки сообщений от хоста к хосту полностью отделен от вопроса предоставления полезного сервиса связи «процесс-процесс».
- Ниже IP архитектура позволяет использовать бесконечное множество различных сетевых технологий, начиная от Ethernet и беспроводных сетей и одиночных каналов «точка-точка».
- Третьей особенностью архитектуры Интернета (или, точнее, культуры IETF) является то, что для официального включения нового протокола в архитектуру должна существовать как спецификация протокола, так и по крайней мере одна (а предпочтительно две) представительных реализаций этой спецификации. Наличие рабочих реализаций

- требуется для принятия стандартов IETF. Эта культурная установка дизайнерского сообщества помогает гарантировать, что протоколы архитектуры могут быть эффективно реализованы. Возможно, ценность, которую интернет-культура придает рабочему программному обеспечению, лучше всего выражена цитатой на футболках, часто носимых на встречах IETF:

Мы отвергаем королей, президентов и голосования.

Мы верим в грубый консенсус и работающий код.

Дэвид Кларк

Основные выводы

Из этих трех атрибутов архитектуры Интернета философия дизайна песочных часов достаточно важна, чтобы повторить ее еще раз. Узкая талия песочных часов представляет собой минимальный и тщательно отобранный набор глобальных возможностей, который позволяет сосуществовать, обмениваться возможностями и быстро развиваться как приложениям более высокого уровня, так и коммуникационным технологиям более низкого уровня. Модель с узкой талией имеет решающее значение для способности Интернета адаптироваться к новым требованиям пользователей и меняющимся технологиям.

Глава 1.4. Программное обеспечение

Сетевые архитектуры и спецификации протоколов — это важные вещи, но хорошего плана недостаточно, чтобы объяснить феноменальный успех Интернета: количество компьютеров, подключенных к Интернету, росло экспоненциально более трех десятилетий (хотя точные цифры трудно определить). Количество пользователей Интернета, по оценкам, составляло около 4,1 миллиарда к концу 2018 года — примерно половина населения мира.

Что объясняет успех Интернета? Безусловно, существует множество факторов (включая хорошую архитектуру), но одним из факторов, который сделал Интернет таким невероятно успешным, является тот факт, что большая часть его функциональности обеспечивается программным обеспечением, работающим на компьютерах общего назначения. Важность этого заключается в том, что новая функциональность может быть легко добавлена с помощью «просто небольшого программирования». В результате новые приложения и сервисы появляются с невероятной скоростью.

С этим связано и значительное увеличение вычислительной мощности доступных компьютеров. Хотя компьютерные сети всегда были способны в принципе передавать любую информацию, такую как цифровые голосовые образцы, оцифрованные изображения и так далее, этот потенциал не был особо нужным, если компьютеры, отправляющие и принимающие эти данные, были слишком медленными, чтобы делать что-то значительное с этой информацией. Практически все современные компьютеры способны воспроизводить оцифрованное аудио и видео с такой скоростью и разрешением, которые вполне пригодны для использования.

С тех пор как вышло первое издание этой книги, написание сетевых приложений стало повседневным занятием, а не работой лишь для нескольких специалистов. Многие факторы способствовали этому, включая улучшение инструментов, делающих эту работу проще, и открытие новых рынков, таких как приложения для смартфонов.

Важно отметить, что знание того, как реализовать сетевое программное обеспечение, является неотъемлемой частью понимания компьютерных сетей, и хотя маловероятно, что вам поручат реализовать низкоуровневый протокол, такой как IP, велика вероятность, что вам придется реализовать протокол на уровне приложения — ту самую «убий-

ственную программу», которая принесет вам невообразимую славу и богатство. Чтобы помочь вам начать, эта глава вводит некоторые понятия, связанные с реализацией сетевого приложения поверх Интернета. Как правило, такие программы одновременно являются приложением (то есть предназначены для взаимодействия с пользователями) и протоколом (то есть общаются с аналогичными программами по сети).

Глава 1.4.1. Интерфейс программирования приложений (сокеты)

Начинать реализацию сетевого приложения следует с интерфейса, предоставляемого сетью. Поскольку большинство сетевых протоколов реализовано в программном обеспечении (особенно те, которые находятся на высоких уровнях протокольного стека), и практически все компьютерные системы реализуют свои сетевые протоколы как часть операционной системы, когда мы говорим об «интерфейсе, предоставляемом сетью», мы обычно имеем в виду интерфейс, который операционная система (ОС) предоставляет своей сетевой подсистеме. Этот интерфейс часто называют сетевым *программным интерфейсом приложения* (application programming interface, API).

Хотя каждая операционная система вольна определять свой собственный сетевой API (и большинство из них так и сделали), со временем некоторые из этих API получили широкую поддержку; то есть они были перенесены на операционные системы, отличные от их родных систем. Именно это произошло с *интерфейсом сокетов*, первоначально предоставленным в дистрибутиве Unix от Berkeley, который теперь поддерживается практически во всех популярных операционных системах и является основой для интерфейсов, специфичных для языков программирования, таких как Java или Python. Мы используем Linux и C для всех примеров кода в этой книге, поскольку Linux является операционной системой с открытым исходным кодом, а C остается предпочтительным языком для сетевых внутренних процессов. (C также имеет преимущество, поскольку раскрывает все низкоуровневые детали, что полезно для понимания базовых идей.)

Благодаря сокетам произошел взрывной рост приложений

- Важность Socket API трудно переоценить. Он определяет точку разделения между приложениями, работающими поверх Интернета, и деталями реализации самого Интернета.
- В результате того, что сокеты предоставляют хорошо определенный и стабильный интерфейс, написание интернет-приложений превратилось в многомиллиардную индустрию. Начиная с простейшей парадигмы «клиент/сервер» и горстки простых приложений, таких как электронная почта, передача файлов и удаленный вход, теперь каждый имеет доступ к бесконечному количеству облачных приложений со своих смартфонов.

В этом разделе заложены основы, и мы вновь вспомним, как просто программа-клиент открывает сокет, чтобы обмениваться сообщениями с программой-сервером, но сегодня поверх Socket API выстроена богатая программная экосистема. Этот слой включает в себя множество облачных инструментов, которые упрощают реализацию масштабируемых приложений. Мы возвращаемся к теме взаимодействия облака и сети в каждой главе, начиная с подраздела «*Перспектива*» после раздела 1.

Прежде чем описывать интерфейс сокетов, важно держать в голове два разных аспекта. Каждый протокол предоставляет определенный набор сервисов, а API предоставляет синтаксис, с помощью которого эти сервисы можно вызывать на конкретной компьютерной системе. Реализация затем отвечает за отображение осязаемого набора операций и объектов, определенных API, на абстрактный набор сервисов, определенных протоколом. Если интерфейс определен хорошо, то с помощью его синтаксиса можно будет вызывать сервисы множества разных протоколов. Такой универсальности, безусловно, стремились достичь при создании Socket API, хотя он далеко не идеален.

Главная абстракция Socket API, что неудивительно, это сокет. Сокет — это точка, в которой локальный процесс приложения подключается к сети. Интерфейс определяет, какие

операции нужны для создания сокета, подключения сокета к сети, отправки/получения сообщений через сокет и закрытия сокета. Чтобы упростить обсуждение, мы ограничимся показом того, как сокеты используются с TCP.

Первым шагом будет создание сокета, которое выполняется с помощью следующей операции:

```
int socket(int domain, int type, int protocol);
```

Причина, по которой эта операция принимает три аргумента, заключается в том, что интерфейс сокетов был разработан достаточно универсальным, чтобы поддерживать любой набор базовых протоколов. В частности, аргумент `domain` определяет семейство протоколов, которое будет использоваться: `PF_INET` обозначает семейство Internet, `PF_UNIX` — средства передачи данных Unix, а `PF_PACKET` — прямой доступ к сетевому интерфейсу (т. е. в обход стека протоколов TCP/IP). Аргумент `type` указывает на семантику обмена данными. `SOCK_STREAM` используется для обозначения потока байтов. `SOCK_DGRAM` — альтернативный вариант, обозначающий сервис, ориентированный на сообщения, например, предоставляемый UDP. Аргумент `protocol` определяет конкретный протокол, который будет использоваться. В нашем случае этот аргумент — `UNSPEC`, поскольку сочетание `PF_INET` и `SOCK_STREAM` подразумевает TCP. Наконец, возвращаемое значение `socket` — это *дескриптор* (handle) для вновь созданного сокета, то есть идентификатор, по которому мы можем обращаться к сокету в будущем. Он указывается в качестве аргумента при последующих операциях с этим сокетом.

Следующий шаг зависит от того, являетесь ли вы клиентом или сервером. На серверной машине приложение выполняет *пассивное* открытие — сервер говорит, что готов принимать соединения, но на самом деле не устанавливает соединение. Для этого сервер вызывает следующие три операции:

```
int bind(int socket, struct sockaddr *address, int addr_len);  
int listen(int socket, int backlog);  
int accept(int socket, struct sockaddr *address, int *addr_len);
```

Операция `bind`, как следует из ее названия, привязывает вновь созданный сокет (`socket`) с указанным адресом (`address`). Это сетевой адрес *локального* участника — сервера. Обратите внимание, что при использовании интернет-протоколов адрес — это структура данных, которая включает в себя как IP-адрес сервера, так и номер TCP-порта. Порты используются для косвенной идентификации процессов. Они представляют собой разновидность *демукса-ключей*. Номер порта обычно представляет собой некоторое известное число, характерное для предоставляемой услуги; например, веб-серверы обычно принимают соединения по порту 80.

Операция `listen` затем определяет, сколько соединений может ожидать указанный сокет (`socket`). Наконец, операция `accept` выполняет пассивное открытие. Это блокирующая операция, которая не возвращается до тех пор, пока удаленный участник не установит соединение, а когда она завершается, то возвращает новый сокет, который относится к только что установленному соединению, а аргумент `address` содержит адрес удаленного участника. Обратите внимание, что когда `accept` возвращается, исходный сокет, который был указан в качестве аргумента, все еще существует и соответствует пассивному `open`; он используется в последующих вызовах `accept`.

На клиентской машине процесс приложения выполняет *активное* открытие, то есть сообщает, с кем он хочет общаться, вызывая следующую единственную операцию:

```
int connect(int socket, struct sockaddr *address, int addr_len);
```

Эта операция не возвращается до тех пор, пока TCP не установит успешное соединение, после чего приложение может начать отправку данных. В этом случае `address` содержит адрес удаленного участника. На практике клиент обычно указывает только адрес уда-

ленного участника и позволяет системе заполнить локальную информацию. В то время как сервер обычно прослушивает сообщения на известном порту, клиент обычно не заботится о том, какой порт он использует для себя; ОС просто выбирает неиспользуемый порт.

После установления соединения процессы приложения вызывают следующие две операции для отправки и получения данных:

```
int send(int socket, char *message, int msg_len, int flags);
int recv(int socket, char *buffer, int buf_len, int flags);
```

Первая операция отправляет данное сообщение через указанный сокет, в то время как вторая операция принимает сообщение с указанного сокета в заданный буфер. Обе операции принимают набор флагов, которые контролируют определенные детали операции.

Глава 1.4.2. Пример приложения

Теперь мы покажем реализацию простой клиент/серверной программы, которая использует Socket API для отправки сообщений по TCP-соединению. Программа также использует другие сетевые утилиты Linux, которые мы будем вводить по мере необходимости. Наше приложение позволяет пользователю на одной машине вводить и отправлять текст пользователю на другой машине. Это упрощенная версия программы Linux talk, которая похожа на программу, лежащую в основе приложений для мгновенного обмена сообщениями.

Клиент

Мы начнем с клиентской части, которая принимает в качестве аргумента имя удаленной машины. Она вызывает утилиту Linux для преобразования этого имени в IP-адрес удаленного узла. Следующим шагом будет построение структуры данных адреса (sin), ожидаемой интерфейсом сокета. Обратите внимание, что эта структура данных указывает, что мы будем использовать сокет для подключения к Интернету (AF_INET). В нашем примере мы используем TCP-порт 5432 в качестве известного порта сервера; так получилось, что этот порт не был назначен ни одному другому интернет-сервису. Последним шагом в установке соединения является вызов операции socket и connect. После возврата операции соединение устанавливается, и клиентская программа переходит в свой основной цикл, который считывает текст со стандартного ввода и отправляет его через сокет.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#define SERVER_PORT 5432
#define MAX_LINE 256

int main(int argc, char *argv[])
{
    FILE *fp;
    struct hostent *hp;
    struct sockaddr_in sin;
    char *host;
    char buf[MAX_LINE];
    int s;
    int len;

    if (argc==2) {
```

```

    host = argv[1];
}
else {
    fprintf(stderr, «usage: simplex-talk host\n»);
    exit(1);
}

/* перевод имени хоста в IP-адрес узла */
hp = gethostbyname(host);
if (!hp) {
    fprintf(stderr, «simplex-talk: unknown host: %s\n», host);
    exit(1);
}

/* создание структуры данных адреса */
bzero((char *)&sin, sizeof(sin));
sin.sin_family = AF_INET;
bcopy(hp->h_addr, (char *)&sin.sin_addr, hp->h_length);
sin.sin_port = htons(SERVER_PORT);

/* активное открытие */
if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror(«simplex-talk: socket»);
    exit(1);
}
if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
    perror(«simplex-talk: connect»);
    close(s);
    exit(1);
}

/* основной цикл: получение и отправка строк текста */
while (fgets(buf, sizeof(buf), stdin)) {
    buf[MAX_LINE-1] = '\0';
    len = strlen(buf) + 1;
    send(s, buf, len, 0);
}
}

```

Сервер

Сервер работает так же просто. Сначала он создает структуру данных адреса, заполняя ее собственным номером порта (SERVER_PORT). Не указывая IP-адрес, программа сервера готова принимать соединения на любом из IP-адресов локального хоста. Далее сервер выполняет предварительные шаги, связанные с пассивным открытием; он создает сокет, привязывает его к локальному адресу и устанавливает максимальное количество разрешенных ожидающих соединений. Наконец, основной цикл ждет, когда удаленный хост попытается подключиться, и когда это происходит, он получает и выводит символы, которые поступают по соединению.

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>

```

```
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 5432
#define MAX_PENDING 5
#define MAX_LINE 256

int main() {
    struct sockaddr_in sin;
    char buf[MAX_LINE];
    int buf_len, addr_len;
    int s, new_s;

    /* создаем структуру данных адреса */
    bzero((char *)&sin, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;
    sin.sin_port = htons(SERVER_PORT);

    /* настраиваем пассивное открытие */
    if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        perror(«simplex-talk: socket»);
        exit(1);
    }
    if ((bind(s, (struct sockaddr *)&sin, sizeof(sin))) < 0) {
        perror(«simplex-talk: bind»);
        exit(1);
    }

    listen(s, MAX_PENDING);

    /* ждем подключения, затем получаем и выводим текст */
    while (1) {
        if ((new_s = accept(s, (struct sockaddr *)&sin, &addr_len)) < 0) {
            perror(«simplex-talk: accept»);
            exit(1);
        }
        while ((buf_len = recv(new_s, buf, sizeof(buf), 0)) > 0) {
            fputs(buf, stdout);
        }
        close(new_s);
    }
}
```

Глава 1.5. Производительность

До этого момента мы сосредотачивались главным образом на функциональных аспектах сетей. Однако, как и от любой другой компьютерной системы, от компьютерных сетей также ожидается хорошая производительность. Это потому, что эффективность распределенных по сети вычислений часто напрямую зависит от того, насколько эффективно сеть

передает данные для этих вычислений. Хотя старая поговорка программиста «сначала сделай правильно, а потом делай быстро» остается верной, в сетях часто необходимо проектировать с учетом производительности. Поэтому важно понимать различные факторы, влияющие на производительность сети.

Глава 1.5.1. Пропускная способность и задержка

Производительность сети измеряется двумя основными способами: *пропускной способностью (bandwidth)* (также называемой *throughput*) и *задержкой (latency или delay)* (также называемой *delay*). Пропускная способность сети определяется количеством бит, которые могут быть переданы по сети за определенный период времени. Например, сеть может иметь пропускную способность 10 миллионов бит/секунду (Мбит/с), и это означает, что она способна передавать 10 миллионов бит каждую секунду. Иногда полезно думать о пропускной способности в терминах того, сколько времени требуется для передачи каждого бита данных. Например, в сети с пропускной способностью 10 Мбит/с требуется 0,1 микросекунды (мкс) для передачи каждого бита.

Пропускная способность (bandwidth) и пропускная способность (throughput) — это тонко различающиеся термины. Во-первых, пропускная способность буквально измеряет ширину полосы частот. Например, устаревшие телефонные линии поддерживали полосу частот от 300 до 3300 Гц; говорилось, что их пропускная способность составляет 3300 Гц – 300 Гц = 3000 Гц. Если вы видите, что словосочетание «*пропускная способность*» (bandwidth) используется в ситуации, когда она измеряется в герцах, то, вероятно, речь идет о диапазоне сигналов, которые могут быть приняты.

Когда мы говорим о пропускной способности коммуникационного канала, мы обычно имеем в виду количество бит в секунду, которые могут быть переданы по каналу. Это также иногда называют *скоростью передачи данных*. Мы можем сказать, что пропускная способность Ethernet-соединения составляет 10 Мбит/с. Однако полезно также различать максимальную скорость передачи данных, доступную на канале, и количество бит в секунду, которые мы можем фактически передать по каналу на практике. Мы склонны использовать слово *throughput (пропускная способность)* для обозначения *измеренной производительности системы*. Таким образом, из-за различных неэффективностей реализации пара узлов, соединенных каналом с пропускной способностью 10 Мбит/с, может достигать throughput всего 2 Мбит/с. Это означало бы, что приложение на одном хосте может передавать данные на другой хост со скоростью 2 Мбит/с.

Наконец, мы часто говорим о *требованиях* приложения к пропускной способности. Это количество бит в секунду, которые необходимо передать по сети для приемлемой работы. Для некоторых приложений это может быть «сколько получится»; для других — фиксированное число (желательно не больше доступной пропускной способности канала (bandwidth)); а для третьих — число, которое варьируется с течением времени. Мы подробнее рассмотрим эту тему позже в данной главе.

Хотя можно говорить о пропускной способности всей сети в целом, иногда нужно быть более точным, фокусируясь, например, на пропускной способности одного физического канала или логического канала «процесс-процесс». На физическом уровне пропускная способность постоянно улучшается, и этому не видно конца. Интуитивно, если представить одну секунду времени как расстояние, которое можно измерить линейкой, а пропускную способность — как количество бит, помещающихся в это расстояние, то каждый бит можно представить как импульс определенной ширины. Например, каждый бит на канале с пропускной способностью 1 Мбит/с имеет ширину 1 мкс, в то время как каждый бит на канале с пропускной способностью 2 Мбит/с имеет ширину 0,5 мкс, как показано на рис. 1.16. Чем более совершенна технология передачи и приема, тем уже может быть каждый бит, а значит, выше пропускная способность. Для логических каналов «процесс-процесс» пропускная способность также зависит от других факторов,

включая то, сколько раз программное обеспечение, реализующее канал, должно обрабатывать и, возможно, преобразовывать каждый бит данных.

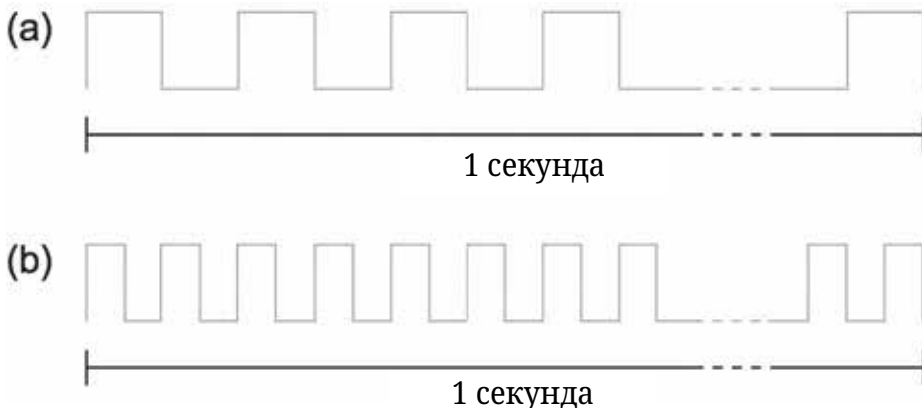


Рисунок 1.16. Передаваемые с определенной пропускной способностью биты можно рассматривать как имеющие некоторую ширину:

- (a) биты, передаваемые на скорости 1 Мбит/с
(каждый бит имеет ширину 1 микросекунда);*
- (b) биты, передаваемые на скорости 2 Мбит/с
(каждый бит имеет ширину 0,5 микросекунды).*

Второй показатель производительности, задержка (латентность или latency), соответствует времени, необходимому для передачи сообщения от одного конца сети до другого. (Как и в случае с пропускной способностью (или bandwidth), мы можем фокусироваться на задержке одного канала или всего маршрута от начала до конца.) Задержка измеряется исключительно во времени. Например, трансконтинентальная сеть может иметь задержку 24 миллисекунды (мс); то есть сообщение проходит от одного побережья Северной Америки до другого за 24 мс. Существует много ситуаций, в которых важнее знать, сколько времени требуется, чтобы отправить сообщение от одного конца сети до другого и обратно, а не только в одну сторону. Это называется *временем кругового путешествия* (round-trip-time, RTT) сети.

Мы часто думаем о задержке как о трехкомпонентной величине. Во-первых, существует задержка из-за скорости распространения света. Эта задержка возникает потому, что ничто, включая бит в проводе, не может двигаться быстрее скорости света. Если вы знаете расстояние между двумя точками, вы можете рассчитать задержку по скорости света, хотя следует быть осторожным, потому что свет распространяется в разных средах с разной скоростью: он движется со скоростью 3.0×10^8 м/с в вакууме, 2.3×10^8 м/с в медном кабеле и 2.0×10^8 м/с в оптическом волокне. Во-вторых, есть время, необходимое для передачи единицы данных. Это зависит от пропускной способности сети и размера пакета, в котором передаются данные. В-третьих, могут быть задержки в очередях внутри сети, так как коммутаторы пакетов обычно должны хранить пакеты некоторое время перед тем, как переслать их на исходящую линию связи. Итак, можно определить общую задержку следующим образом:

$$\begin{aligned} \text{Latency} &= \text{Propagation} + \text{Transmit} + \text{Queue} \\ \text{Propagation} &= \text{Distance} / \text{SpeedOfLight} \\ \text{Transmit} &= \text{Size} / \text{Bandwidth} \end{aligned}$$

где *Distance* — это длина провода, по которому будут передаваться данные, *SpeedOfLight* — это скорость света по этому проводу, *Size* — это размер пакета, а *Bandwidth* — это пропускная способность, с которой передается пакет. Обратите внимание, что если сообщение содержит только один бит и мы говорим о единственном канале (а не о всей сети), то компоненты *Transmit* и *Queue* не имеют значения, и задержка соответствует только задержке распространения.

Пропускная способность и задержка вместе определяют характеристики производительности данного канала или соединения. Их важность, однако, зависит от приложения. Для некоторых приложений задержка важнее пропускной способности. Приведем пример: клиент отправляет серверу сообщение размером 1 байт и получает в ответ сообщение размером 1 байт, время проведения данных манипуляций зависит от задержки. При условии, что в подготовке ответа не задействованы серьезные вычисления, производительность приложения будет значительно отличаться на трансконтинентальном канале с временем кругового прохода (*RTT*) 100 мс и на канале в пределах одной комнаты с *RTT* 1 мс. Однако, если пропускная способность канала составляет 1 Мбит/с или 100 Мбит/с, разница субъективно незначительна, так как в первом случае время передачи байта (*Transmit*) составляет 8 мкс, а во втором — 0.08 мкс.

В противоположность этому рассмотрим программу цифровой библиотеки, которой нужно получить изображение размером 25 мегабайт (МБ) — чем больше доступная пропускная способность, тем быстрее она сможет вернуть изображение пользователю. Здесь производительность определяется пропускной способностью канала. Чтобы это понять, предположим, что пропускная способность канала составляет 10 Мбит/с. Передача изображения займет 20 секунд ($25 \times 106 \times 8 \text{ бит} / (10 \times 106 \text{ Мбит/с}) = 20 \text{ секунд}$), что делает отличие относительно неважным, находится ли изображение на другой стороне канала с задержкой 1 мс или 100 мс; разница между временем ответа 20.001 секунды и 20.1 секунды незначительна.

Рис. 1.17 дает представление о том, как задержка или пропускная способность могут доминировать в производительности в различных ситуациях. График показывает, сколько времени требуется для передачи объектов разного размера (1 байт, 2 КБ, 1 МБ) через сети с *RTT* в диапазоне от 1 до 100 мс и скоростью соединения 1.5 или 10 Мбит/с. Мы используем логарифмические шкалы для отображения производительности. Для объекта размером 1 байт (например, нажатия клавиши) задержка остается почти равной *RTT*, так что невозможно отличить сеть с пропускной способностью 1.5 Мбит/с от сети с 10 Мбит/с. Для объекта размером 2 КБ (например, сообщения электронной почты) скорость соединения имеет большое значение при *RTT* 1 мс, но незначительное при *RTT* 100 мс. А для объекта размером 1 МБ (например, цифрового изображения) *RTT* не имеет значения — производительность определяется скоростью соединения по всему диапазону *RTT*.

Примечание: на протяжении этой книги мы используем термины «задержка (*latency*)» и «задержка (*delay*)» в общем смысле для обозначения времени выполнения определенной функции, такой как доставка сообщения или перемещение объекта. Когда мы говорим о конкретном времени, которое требуется сигналу для распространения от одного конца канала до другого, мы используем термин «задержка распространения».

К слову, компьютеры становятся настолько быстрыми, что иногда полезно мыслить, по меньшей мере в переносном смысле, в терминах *инструкций на милю*. Подумайте, что происходит, когда компьютер, способный выполнять 100 миллиардов инструкций в секунду, отправляет сообщение по каналу с временем отклика 100 мс (для упрощения расчетов предположим, что сообщение проходит расстояние в 5000 миль). Если этот компьютер простаивает все 100 мс в ожидании ответного сообщения, то он потеряет возможность выполнить 10 миллиардов инструкций, или 2 миллиона инструкций на милю. Чтобы оправдать эту потерю, отправка через сеть должна была быть целесообразной.

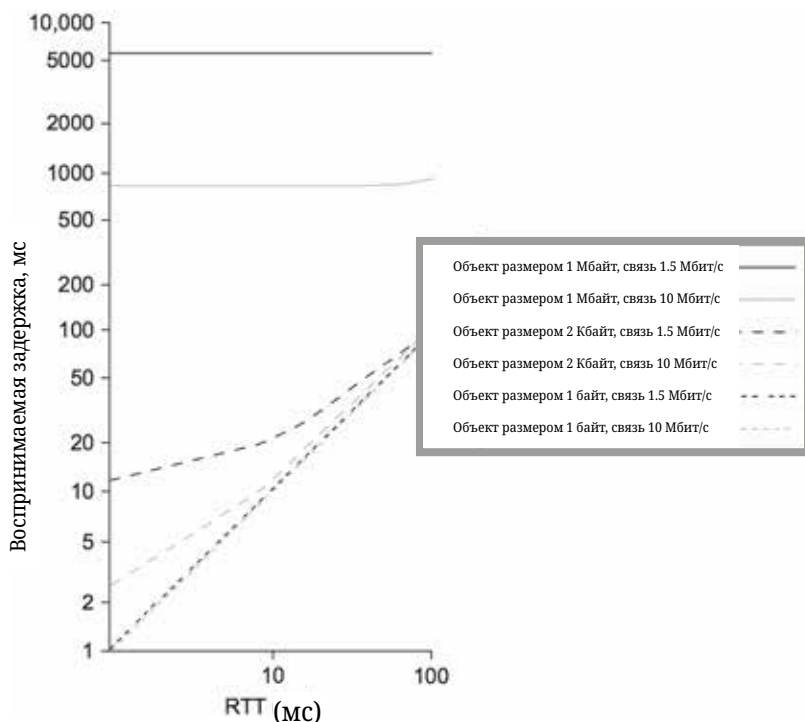


Рисунок 1.17. Зависимость воспринимаемой задержки (времени отклика) от времени прохождения в оба конца для различных размеров объектов и скоростей соединения.

Глава 1.5.2. Производство задержки и пропускной способности

Также полезно говорить о производстве этих двух метрик, которое часто называют *произведением задержки и пропускной способности*. Интуитивно, если мы представим канал между парой процессов как пустую трубу (см. рис. 1.18), где задержка соответствует длине трубы, а пропускную способность определяет диаметр трубы, то произведение задержки и пропускной способности дает объем трубы — максимальное количество бит, которые могут находиться в транзите через трубу в любой момент времени. Другими словами, если задержка (измеряемая во времени) соответствует длине трубы, то, учитывая ширину каждого бита (также измеряемую во времени), можно вычислить, сколько бит умещается в трубе. Например, трансконтинентальный канал с односторонней задержкой 50 мс и пропускной способностью 45 Мбит/с способен содержать

$$50 \times 10^{-3} \text{ сек} \times 45 \times 10^6 \text{ бит/сек} = 2,25 \times 10^6 \text{ битов}$$

или примерно 280 КБ данных. Другими словами, этот пример канала (трубы) вмещает столько байт, сколько могла бы вместить память персонального компьютера начала 1980-х годов.

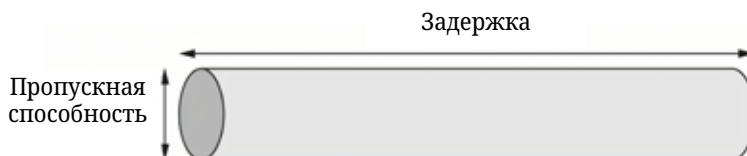


Рисунок 1.18. Сеть как труба.

Произведение задержки и пропускной способности важно знать при построении высокопроизводительных сетей, поскольку оно соответствует количеству битов, которые отправитель должен передать до того, как первый бит достигнет получателя. Если отправитель ожидает, что получатель как-то сигнализирует о начале получения битов, и этот сигнал занимает еще какое-то время задержки канала, чтобы вернуться к отправителю, тогда отправитель может отправить данные объемом до одного $RTT \times$ пропускная способность, прежде чем получить от получателя подтверждение, что все в порядке. Биты в канале называются «в полете», и это означает, что если получатель скажет отправителю прекратить передачу, он может получить до одного $RTT \times$ пропускная способность данных до того, как отправитель сможет отреагировать. В нашем примере это количество соответствует $5,5 \times 10^6$ битам (671 КБ) данных. С другой стороны, если отправитель не заполняет канал полностью, то есть не отправляет данные объемом, равным произведению $RTT \times$ пропускная способность, он остановится и будет ждать сигнала, то есть он не будет полностью использовать сеть.

Обратите внимание, что в основном нас интересует сценарий RTT , который мы просто называем произведением задержки и пропускной способности, без явного указания, что «задержка» означает RTT (то есть умножьте одностороннюю задержку на два). Обычно понятно из контекста, означает ли «задержка» в произведении задержки и пропускной способности одностороннюю задержку или RTT . Таблица 1.1 показывает некоторые примеры произведений $RTT \times$ пропускная способность для типичных сетевых каналов.

Таблица 1.1.
Примеры произведений задержки и пропускной способности.

Тип канала	Пропускная способность	Одностороннее расстояние	Время RTT	Произведение RTT и пропускной способности
Беспроводной LAN	54 Мбит/с	50 м	0,33 мкс	18 бит
Спутниковый канал	1 Гбит/с	35000 км	230 мс	230 Мбит
Оптоволоконный канал через страну	10 Гбит/с	4000 км	40 мс	400 Мбит

Глава 1.5.3. Высокоскоростные сети

Постоянное увеличение пропускной способности заставляет сетевых дизайнеров задуматься о том, что происходит на пределе возможностей, или, иными словами, как бесконечная доступная пропускная способность влияет на проектирование сетей.

Хотя высокоскоростные сети приводят к значительному увеличению пропускной способности для приложений, во многих отношениях их влияние на наше представление о сетях проявляется в том, что не меняется с увеличением пропускной способности: скорость света. Как сказал Скотти из «Звездного пути»: «Ты не можешь изменить законы физики». Другими словами, «высокая скорость» не означает, что задержка улучшается с той же скоростью, что и пропускная способность; трансконтинентальное время RTT для канала с пропускной способностью 1 Гбит/с такое же, как и для канала с пропускной способностью 1 Мбит/с, то есть 100 мс.

Чтобы оценить значимость постоянно растущей пропускной способности при фиксированной задержке, рассмотрим, что требуется для передачи файла размером 1 МБ по сети с пропускной способностью 1 Мбит/с и по сети с пропускной способностью 1 Гбит/с, обе из которых имеют RTT 100 мс. В случае сети с пропускной способностью 1 Мбит/с требует-

ся 80 RTT для передачи файла; за каждый RTT передается 1,25 % файла. Напротив, тот же файл размером 1 МБ даже близко не заполняет пропускную способность сети с пропускной способностью 1 Гбит/с за один RTT, так как произведение задержки и пропускной способности составляет 12,5 МБ.

Рис. 1.19 иллюстрирует разницу между двумя сетями. По сути, файл размером 1 МБ выглядит как поток данных, который нужно передать по сети с пропускной способностью 1 Мбит/с, тогда как в сети с пропускной способностью 1 Гбит/с он выглядит как единый пакет. Чтобы лучше понять этот пример, рассмотрим, каким образом файл размером 1 МБ для сети с пропускной способностью 1 Гбит/с эквивалентен пакету размером 1 КБ для сети с пропускной способностью 1 Мбит/с.

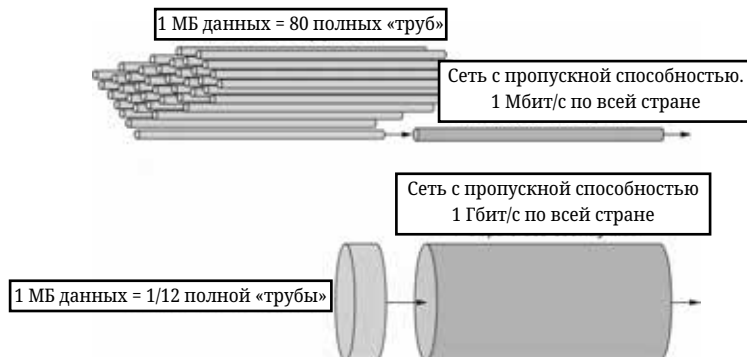


Рисунок 1.19.: Взаимосвязь между пропускной способностью и задержкой.
Файл размером 1 Мбайт.

Другой способ представить подобную ситуацию заключается в том, что больше данных может быть передано за каждый RTT в высокоскоростной сети, настолько, что один RTT становится значительным промежутком времени. Таким образом, если вы не задумываетесь о разнице между передачей файла, занимающей 101 RTT, а не 100 RTT (относительная разница всего в 1 %), то разница между 1 RTT и 2 RTT внезапно становится значительной — увеличивает на 100 %. Другими словами, задержка, а не пропускная способность, становится на первое место в наших размышлениях о проектировании сети.

Возможно, лучший способ понять соотношение между пропускной способностью и задержкой — вернуться к основам. Эффективная сквозная пропускная способность, которая может быть достигнута в сети, определяется простым соотношением:

$$\text{Пропускная способность} = \text{Размер передачи} / \text{Время передачи}$$

где «Время передачи» включает не только элементы одного пути, описанные ранее в этой главе, но и любое дополнительное время, затраченное на запрос или настройку передачи. Обычно мы представляем это соотношение так:

$$\text{Время передачи} = \text{RTT} + 1/\text{Пропускная способность} \times \text{Размер передачи}.$$

Мы используем это вычисление, чтобы учесть сообщение-запрос, отправленное по сети, и данные, отправленные обратно. Например, рассмотрим ситуацию, когда пользователь хочет получить файл размером 1 МБ по сети с пропускной способностью 1 Гбит/с и временем кругового путешествия (RTT) 100 мс. Это включает время передачи 1 МБ ($1 / 1 \text{ Гбит/с} \times 1 \text{ МБ} = 8 \text{ мс}$) и 100 мс RTT, что дает общее время передачи 108 мс. Это означает, что эффективная пропускная способность будет:

$$1 \text{ МБ} / 108 \text{ мс} = 74,1 \text{ Мбит/с}$$

а не 1 Гбит/с. Очевидно, что передача большего объема данных поможет улучшить эффективную пропускную способность, где на пределе возможностей бесконечно большой размер передачи приведет к тому, что эффективная пропускная способность приблизится к пропускной способности сети. С другой стороны, необходимость выждать более одного RTT — например, для повторной передачи потерянных пакетов — ухудшит эффективную пропускную способность для любой передачи конечного размера и будет наиболее заметной для небольших передач.

Глава 1.5.4. Требования в производительности приложений

Мы рассматривали производительность с акцентом на сеть; то есть мы говорили о том, что будет поддерживать данный канал или соединение. Неявное предположение заключалось в том, что потребности программ приложений просты — им нужно столько пропускной способности, сколько может предоставить сеть. Это, безусловно, верно для упомянутой ранее программы цифровой библиотеки, которая загружает изображение размером 250 МБ; чем больше доступной пропускной способности, тем быстрее программа сможет вернуть изображение пользователю.

Однако некоторые приложения могут предоставить верхний предел пропускной способности, который им нужен. Видеоприложения — яркий пример. Предположим, кто-то хочет транслировать видео, которое составляет четверть размера стандартного телевизионного экрана; то есть оно имеет разрешение 352 на 240 пикселей. Если каждый пиксель представлен 24 битами информации, как это бывает при 24-битном цвете, тогда размер каждого кадра будет $(352 \times 240 \times 24) / 8 = 247,5$ КБ. Если приложению нужно поддерживать частоту кадров 30 кадров в секунду, тогда оно может запросить скорость передачи данных 75 Мбит/с. Для приложения не важно, что сеть может предоставлять большую пропускную способность, потому что у него есть лишь определенное количество данных для передачи в заданный период времени.

К сожалению, ситуация не так проста, как предполагает этот пример. Поскольку разница между любыми двумя соседними кадрами в видеопотоке часто незначительна, сжать видео возможно, передавая только различия между соседними кадрами. Каждый кадр также можно сжать, потому что не все детали изображения легко воспринимаются человеческим глазом. Сжатое видео не передается с постоянной скоростью, а изменяется со временем в зависимости от таких факторов, как количество действий и деталей на картинке, а также используемый алгоритм сжатия. Поэтому можно сказать, какова будет средняя потребность в пропускной способности, но мгновенная скорость может быть больше или меньше.

Ключевой вопрос заключается в том, за какой временной интервал вычисляется среднее значение. Предположим, что это видеоприложение можно сжать до такой степени, что оно будет требовать только 2 Мбит/с в среднем. Если оно передает 1 мегабит за 1-секундный интервал и 3 мегабита в следующем 1-секундном интервале, то за 2-секундный интервал средняя скорость будет равняться 2 Мбит/с; однако это будет слабым утешением для канала, рассчитанного на поддержку не более 2 мегабит в одну секунду. Очевидно, что только знания средних потребностей в пропускной способности приложения не всегда достаточно.

Однако, как правило, можно установить верхний предел того, какой большой всплеск данных приложение подобного рода может передать. Всплеск может быть описан некоторой пиковой скоростью, которая поддерживается в течение определенного времени. Альтернативно он может быть описан как количество байтов, которое может быть отправлено с пиковой скоростью перед возвратом к средней или какой-либо более низкой скорости. Если эта пиковая скорость выше доступной пропускной способности канала, то избыточные данные придется где-то буферизовать, чтобы передать их позже. Зная, какого размера всплеск может быть отправлен, сетевой дизайнер может выделить достаточную емкость буфера для удержания этого всплеска.

Аналогично тому, как потребности приложения в пропускной способности могут быть чем-то иным, чем «все, что оно может получить», требования приложения к задержке могут быть сложнее, чем просто «как можно меньшая задержка». В случае задержки иногда не так важно, является ли однонаправленная задержка в сети 100 мс или 500 мс, как то, насколько сильно задержка варьируется от пакета к пакету. Вариация задержки называется *джиттером* (jitter).

Рассмотрим ситуацию, когда источник отправляет пакет каждые 33 мс, как это было бы в случае видеоприложения, передающего кадры 30 раз в секунду. Если пакеты прибывают на место назначения с интервалом ровно 33 мс, то можно сделать вывод, что задержка, испытываемая каждым пакетом в сети, была точно такой же. Однако если интервал между прибытием пакетов на место назначения — иногда называемый *межпакетным интервалом* (inter-packet gap) — изменяется, то задержка, испытываемая последовательностью пакетов, также должна была быть переменной, и говорят, что сеть ввела джиттер в поток пакетов, как показано на рис. 1.20. Такие вариации, как правило, не вводятся на одном физическом канале, но они могут происходить, когда пакеты испытывают разные задержки в очереди в многоходовой сети с коммутацией пакетов. Эта задержка в очереди соответствует компоненту задержки, определенному ранее, который варьируется со временем.



Рисунок 1.20. Джиттер, вызванный сетью.

Чтобы понять значимость джиттера (jitter), представьте, что пакеты, передаваемые по сети, содержат видеокдры, и для отображения этих кадров на экране приемник должен получать новый кадр каждые 33 мс. Если кадр прибывает раньше, его можно просто сохранить до времени отображения. К сожалению, если кадр прибывает поздно, приемник не успеет вовремя обновить экран, и качество видео пострадает; видео не будет плавным. Обратите внимание, что не обязательно устранять джиттер, важно знать, насколько он велик. Причина в том, что если приемник знает верхние и нижние пределы задержки, которую может испытывать пакет, он может задержать время начала воспроизведения видео (то есть отображения первого кадра) достаточно долго, чтобы в будущем у него всегда был кадр для отображения, когда это необходимо. Приемник задерживает кадр, эффективно сглаживая джиттер, путем хранения его в буфере.

Перспектива: скорость внедрения новых функций

Этот раздел знакомит с некоторыми участниками компьютерных сетей — проектировщиками сетей, разработчиками приложений, конечными пользователями и операторами сетей, чтобы помочь понять технические требования, формирующие то, как сети проектируются и строятся. Предполагается, что все решения по проектированию являются чисто техническими, но, конечно, это обычно не так. Многие другие факторы, от рыночных сил до государственной политики и этических соображений, также влияют на то, как сети проектируются и строятся.

Из них самым влиятельным является рынок, и он соответствует взаимодействию между операторами сетей (например, AT&T, Comcast, Verizon, DT, NTT, China Unicom), поставщиками сетевого оборудования (например, Cisco, Juniper, Ericsson, Nokia, Huawei,

NEC), поставщиками приложений и услуг (например, Facebook, Google, Amazon, Microsoft, Apple, Netflix, Spotify) и, конечно, абонентами и клиентами (то есть отдельными людьми, а также предприятиями). Границы между этими участниками не всегда четкие, многие компании играют несколько ролей. Наиболее заметным примером этого феномена являются крупные облачные провайдеры, которые (а) создают свое собственное сетевое оборудование, используя товарные компоненты, (б) развертывают и эксплуатируют свои собственные сети и (в) предоставляют конечным пользователям услуги и приложения поверх своих сетей.

Когда вы учитываете эти и другие факторы в процессе технического проектирования, вы осознаете, что в учебной версии истории есть несколько неявных предположений, которые нуждаются в переоценке. Одно из них заключается в том, что проектирование сети — это одноразовая деятельность. Построить один раз и использовать вечно (с учетом обновлений аппаратного обеспечения, чтобы пользователи могли наслаждаться преимуществами последних улучшений производительности). Второе заключается в том, что задача построения сети в значительной степени отделена от задачи эксплуатации сети. Ни одно из этих предположений не совсем верно.

Дизайн сети явно развивается, и мы задокументировали эти изменения с каждым новым изданием учебника за эти годы. Делать это на временной шкале, измеряемой годами, исторически было достаточно, но каждый, кто загружал и использовал последнюю версию приложения для смартфона, знает, насколько редко по сегодняшним стандартам что-либо измеряется годами. Проектирование с учетом эволюции должно быть частью процесса принятия решений.

Что касается второго пункта, компании, которые строят сети, почти всегда являются теми же, кто их эксплуатирует. Их коллективно называют *операторами сетей*, и они включают компании, перечисленные выше. Но если снова обратиться к облачным технологиям для вдохновения, мы увидим, что разработка и эксплуатация не являются правдой только на уровне компании, но также это способ, которым самые быстро развивающиеся облачные компании организуют свои инженерные команды: вокруг модели DevOps.

Что все это значит? То, что компьютерные сети сейчас находятся в середине крупной трансформации, с операторами сетей, пытающимися одновременно ускорить темпы инноваций (иногда это называется скоростью внедрения новых функций) и при этом продолжать предлагать надежность в своих услугах (сохранять стабильность). И они все чаще делают это, перенимая лучшие практики облачных провайдеров, которые можно обобщить двумя основными темами: (1) использовать товарное оборудование и перемещать всю интеллектуальную часть в программное обеспечение, и (2) принимать гибкие инженерные решения, которые разрушают барьеры между разработкой и эксплуатацией.

Эта трансформация иногда называется «облачиванием» (cloudification) или «софтверизацией» (softwarization) сети, и хотя у Интернета всегда была мощная программная экосистема, она исторически была ограничена приложениями, работающими *поверх* сети (например, использование Socket API, описанное в *главе 1.4*). Что изменилось, так это то, что сегодня эти же инженерные практики, вдохновленные сетью, применяются к внутренним частям сети. Этот новый подход, известный как *программно-определяемые сети* (Software Defined Networks, SDN), является революционным не столько в том, как мы решаем фундаментальные технические задачи кадрирования, маршрутизации, фрагментации/сборки, планирования пакетов, управления перегрузками, безопасности и так далее, сколько в том, как быстро сеть развивается для поддержки новых функций.

Раздел 2.

Прямые ссылки

Заглядывать слишком далеко вперед — ошибка.

За один раз можно справиться только с одним звеном в цепи судьбы.

Уинстон Черчилль

Проблема: подключение к сети

В первом разделе мы узнали, что сети состоят из связей, соединяющих узлы. Одна из фундаментальных проблем, с которой мы сталкиваемся, — как соединить два узла вместе. Мы также ввели абстракцию «облако», чтобы представить сеть без раскрытия всех ее внутренних сложностей. Таким образом, мы также должны решить аналогичную проблему подключения хоста к облаку. По сути, это проблема, с которой сталкивается каждый интернет-провайдер (Internet Service Provider, ISP), когда хочет подключить нового клиента к своей сети.

Независимо от того, хотим ли мы создать тривиальную сеть из двух узлов с одной связью или подключить миллиардный хост к существующей сети, такой как Интернет, нам нужно решить общий набор задач. Во-первых, нам необходима некоторая физическая среда для подключения. Среда может быть проводом, куском оптического волокна или менее осязаемой средой (например, воздухом), через которую можно передавать электромагнитное излучение (например, радиоволны). Она может охватывать небольшую площадь (например, офисное здание) или широкую территорию (например, межконтинентальную).

Однако подключение двух узлов с помощью подходящей среды — это только первый шаг. Пять дополнительных проблем должны быть решены, прежде чем узлы смогут успешно обмениваться пакетами, и после их решения мы обеспечим *уровень 2 (L2)* подключения (используя терминологию из архитектуры OSI).

Первая — это *кодирование* битов на передающем носителе таким образом, чтобы они могли быть поняты принимающим узлом. Вторая — это вопрос разделения последовательности передаваемых по каналу битов на полные сообщения, которые могут быть доставлены конечному узлу. Это проблема *кадрирования*, и сообщения, доставляемые конечным узлам, часто называются *фреймами* (или иногда *пакетами*). Третья — так как фреймы иногда повреждаются во время передачи, необходимо обнаружить эти ошибки и предпринять соответствующие меры; это проблема *обнаружения ошибок*. Четвертая проблема заключается в том, чтобы сделать канал надежным, несмотря на то, что он время от времени повреждает фреймы. Наконец, в тех случаях, когда канал разделяется несколькими узлами — как это часто бывает с беспроводными каналами, например — необходимо регулировать доступ к этому каналу. Это проблема управления *доступом к среде передачи данных*.

Хотя эти пять проблем — кодирование, кадрирование, обнаружение ошибок, надежная доставка и управление доступом — можно обсуждать в абстрактном виде, на самом деле они вполне реальны и решаются по-разному различными сетевыми технологиями. В данном разделе эти вопросы рассматриваются в контексте конкретных сетевых технологий: волоконно-оптические линии связи «точка-точка» (где SONET является распространенным примером); сети с множественным доступом с обнаружением несущей (Carrier Sense Multiple Access, CSMA) (классический Ethernet и Wi-Fi являются самыми известными примерами); волоконная оптика до дома (где PON является доминирующим стандартом); мобильная беспроводная связь (где 4G быстро переходит в 5G).

Цель этого раздела — одновременно разобрать доступные технологии канального уровня и исследовать эти пять фундаментальных проблем. Мы рассмотрим, что необходимо для того, чтобы сделать самые разные физические среды и каналные технологии полезными «строительными блоками» для создания надежных масштабируемых сетей.

Глава 2.1. Технологический ландшафт

Прежде чем погружаться в проблемы, изложенные ранее, полезно сначала ознакомиться с различными технологиями каналов связи. Это связано, в частности, с разнообразными условиями, в которых пользователи пытаются подключить свои устройства.

На одном конце спектра операторы сетей, создающие глобальные сети, вынуждены иметь дело с каналами связи, охватывающими сотни или тысячи километров и соединяющими маршрутизаторы размером с холодильник. На другом конце спектра обычный пользователь встречает каналы связи в основном как способ подключения компьютера к существующему Интернету. Иногда этот канал будет беспроводным (Wi-Fi) в кафе; иногда это будет Ethernet-канал в офисном здании или университете; иногда это будет смартфон, подключенный к сотовой сети; для все большего числа людей это оптоволоконный канал, предоставляемый интернет-провайдером; многие другие используют какой-либо медный провод или кабель для подключения. К счастью, существует множество общих стратегий, используемых на этих, казалось бы, разных типах каналов связи, чтобы они все могли быть надежными и полезными для вышележащих уровней в стеке протоколов. В данном разделе рассматриваются эти стратегии.

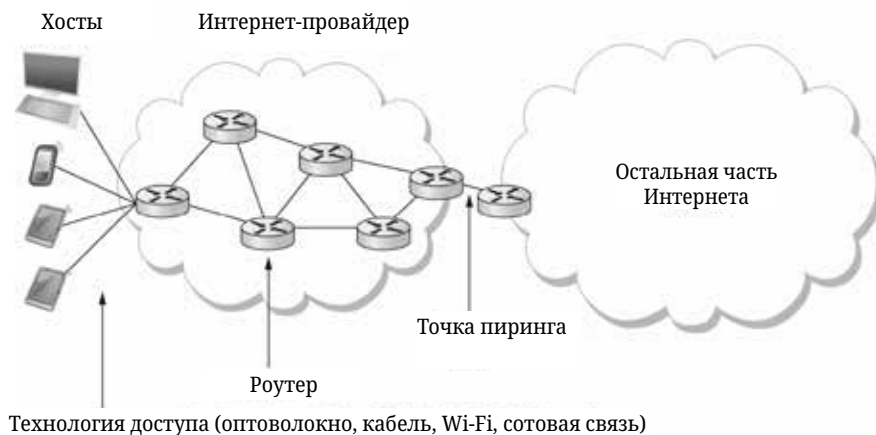


Рисунок 2.1. Представление Интернета у конечного пользователя.

Рис. 2.1 иллюстрирует различные типы каналов связи, которые можно найти в современном Интернете. Слева мы видим разнообразные пользовательские устройства — от смартфонов и планшетов до полноценных компьютеров, подключенных к интернет-провайдеру различными способами. Хотя эти каналы могут использовать разные технологии, на этом рисунке они все выглядят одинаково — прямая линия, соединяющая устройство с маршрутизатором. Существуют каналы, которые соединяют маршрутизаторы внутри интернет-провайдера, а также каналы, которые соединяют интернет-провайдера с «остальной частью Интернета», состоящей из множества других интернет-провайдеров и хостов, к которым они подключены.

Эти каналы все выглядят одинаково не только потому, что мы не очень хорошие художники, но и потому, что часть роли сетевой архитектуры состоит в предоставлении общей абстракции чего-то такого сложного и разнообразного, как канал связи. Идея заключается в том, что вашему ноутбуку или смартфону не важно, к какому именно типу канала они подключены — единственное, что имеет значение, это наличие соединения с Интернетом. Аналогично маршрутизатору не важно, какой тип канала соединяет его с другими маршрутизаторами — он может отправить пакет по каналу с довольно высокой вероятностью того, что пакет достигнет другого конца канала.

Как сделать так, чтобы все эти разные типы каналов связи выглядели одинаково для конечных пользователей и маршрутизаторов? По сути, нам нужно справиться со всеми физическими ограничениями и недостатками каналов, существующих в реальном мире. Мы набросали некоторые из этих вопросов ранее, но прежде чем мы сможем обсудить их, нам нужно вспомнить некоторые простые законы физики. Все эти каналы сделаны из какого-то физического материала, который может передавать сигналы, такие как радиоволны или другие виды электромагнитного излучения, но то, что нам действительно нужно, это передавать биты. В следующих главах этого раздела мы рассмотрим, как кодировать биты для передачи по физическому носителю, а затем и другие упомянутые выше вопросы. К концу этого раздела мы поймем, как отправлять полные пакеты через практически любой тип канала, независимо от используемого физического носителя.

Один из способов охарактеризовать каналы связи — по среде, которую они используют: обычно это медный провод в какой-то форме, такой как витая пара (некоторые Ethernet и стационарные телефоны) и коаксиальный кабель (кабельное телевидение); оптоволокно, которое используется как для подключения до дома, так и для многих дальних соединений в магистральной Интернет; или воздух/свободное пространство для беспроводных соединений.

Еще одна важная характеристика канала — это частота, измеряемая в герцах, с которой электромагнитные волны колеблются. Расстояние между двумя соседними максимумами или минимумами волны, обычно измеряемое в метрах, называется *длиной волны*. Так как все электромагнитные волны распространяются со скоростью света (которая, в свою очередь, зависит от среды), эта скорость, деленная на частоту волны, равна ее длине волны. Мы уже видели пример телефонной линии для передачи голоса, которая переносит непрерывные электромагнитные сигналы в диапазоне от 300 Гц до 3300 Гц; 300-герцовая волна, распространяющаяся через медь, будет иметь длину волны:

$$\begin{aligned} \text{скорость света в меди} / \text{частота} \\ &= 2/3 \times 3 \times 10^8 / 300 \\ &= 667 \times 10^3 \text{ метров} \end{aligned}$$

Вообще электромагнитные волны охватывают гораздо более широкий диапазон частот: от радиоволн до инфракрасного света, видимого света, рентгеновских и гамма-лучей. На рис. 2.2 показан электромагнитный спектр и указаны среды, которые обычно используются для передачи различных диапазонов частот.

Что не показывает рис. 2.2 — это место сотовой сети в данном спектре. Это несколько сложнее иллюстрировать, потому что конкретные частотные диапазоны, лицензированные для сотовых сетей, варьируются по всему миру, и еще сложнее потому, что операторы сетей часто одновременно поддерживают как старые/устаревшие технологии, так и технологии новые/следующего поколения, каждая из которых занимает разные частотные диапазоны. В общих чертах, традиционные сотовые технологии охватывают диапазон от 700 МГц до 2400 МГц, с новыми выделениями в среднем спектре на уровне 6 ГГц и выделениями миллиметровых волн (mmWave) выше 24 ГГц. Этот диапазон миллиметровых волн, вероятно, станет важной частью мобильной сети 5G.

Итак, мы понимаем, что канал связи — это физическая среда, передающая сигналы в виде электромагнитных волн. Такие каналы обеспечивают основу для передачи всевозможной информации, включая тот тип данных, который нас интересует — двоичные данные (1 и 0). Мы говорим, что двоичные данные *закодированы* в сигнале. Задача, как закодировать двоичные данные на электромагнитные сигналы — сложная тема. Чтобы сделать ее проще, можно разделить ее на два уровня. Нижний уровень занимается *модуляцией* — изменением частоты, амплитуды или фазы сигнала для передачи информации. Простой пример модуляции — это изменение мощности (амплитуды) одной длины волны. Интуитивно это эквивалентно включению и выключению света. Поскольку вопрос модуляции вторичен для нашего обсуждения каналов связи как строительного блока компьютерных сетей, мы просто предполагаем, что можно передавать пару различных

сигналов — представим их как «высокий» сигнал и «низкий» сигнал, — и рассматриваем только верхний уровень, который занимается гораздо более простой проблемой кодирования двоичных данных на эти два сигнала. В следующей главе такие способы кодирования будут рассмотрены подробно.

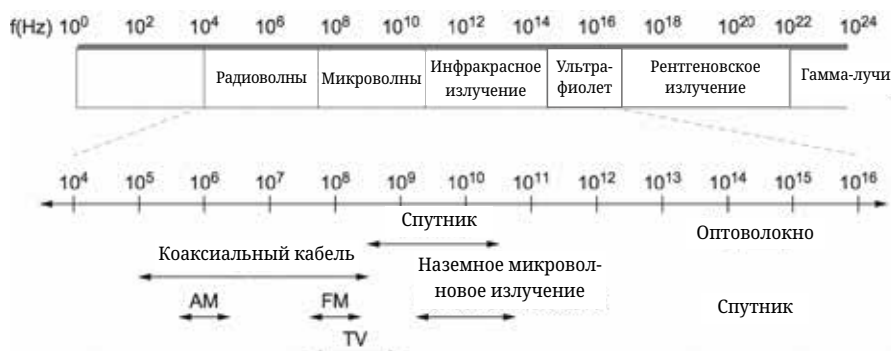


Рисунок 2.2. Электромагнитный спектр.

Еще один способ классификации каналов связи — в зависимости от того, как они используются. Различные экономические и разворачиваемые вопросы влияют на то, где встречаются разные типы каналов. Большинство потребителей взаимодействуют с Интернетом либо через беспроводные сети (которые они встречают в кафе, аэропортах, университетах и т.д.), либо через так называемые «последние мили» (или, альтернативно, *сети доступа*), предоставляемые интернет-провайдером, как показано на рис. 2.1. Сводная информация об этих типах каналов приведена в таблице 2.1. Их обычно выбирают, потому что это экономически эффективное решение, чтобы можно было охватить миллионы потребителей. DSL (Digital Subscriber Line, цифровая абонентская линия), например, — это старая технология, которая была развернута по уже существующим витым медным парам, использовавшимся для обычных телефонных служб; G.Fast — это технология на основе меди, обычно используемая в многоквартирных домах, а PON (Passive Optical Network, пассивная оптическая сеть) — более новая технология, которая часто используется для подключения домов и предприятий через недавно развернутое оптоволокно.

Таблица 2.1.
Общие услуги, доступные для подключения «последней мили» к вашему дому.

Услуга	Пропускная способность
DSL (медь)	до 100 Мбит/с
G.Fast (медь)	до 1 Гбит/с
PON (оптика)	до 10 Гбит/с

И, конечно же, есть *мобильная* или *сотовая сеть* (также известная как 4G, но она быстро развивается в 5G), которая соединяет наши мобильные устройства с Интернетом. Эта технология также может служить единственным подключением к Интернету в наших домах или офисах, но при этом предоставляет дополнительное преимущество возможности поддерживать подключение к Интернету при перемещении с места на место.

Эти примеры технологий являются распространенными вариантами подключения сети доступа («последней мили») к вашему дому или предприятию, но их недостаточно для создания полноценной сети с нуля. Для этого вам также понадобятся *магистральные линии связи*, соединяющие города. Современные магистральные линии почти пол-

ностью оптоволоконные, и в них обычно используется технология SONET (Synchronous Optical Network), которая изначально была разработана для удовлетворения жестких требований к управлению, предъявляемых операторами телефонной связи.

Наконец, в дополнение к каналам связи «последней мили», магистральным и мобильным, существуют каналы связи, которые находятся внутри здания или кампуса — их принято называть *локальными вычислительными сетями* (LAN). Ethernet и его беспроводной родственник Wi-Fi являются доминирующими технологиями в этой области.

Этот обзор типов соединений далеко не исчерпывающий, но он должен дать вам представление о разнообразии типов соединений и некоторых причинах этого разнообразия. В следующих главах мы рассмотрим, как протоколы сетевого взаимодействия могут использовать это разнообразие и представлять сеть в единой и последовательной форме для высоких уровней, несмотря на всю низкоуровневую сложность и экономические факторы.

Глава 2.2. Кодирование

Первый шаг в превращении узлов и каналов связи в полезные «строительные блоки» — понять, как соединить их таким образом, чтобы биты могли быть переданы от одного узла к другому. Как упоминалось в предыдущей главе, сигналы распространяются по физическим каналам. Задача заключается в кодировании двоичных данных, которые исходный узел хочет отправить, в сигналы, которые способны нести каналы связи, а затем в декодировании сигнала обратно в соответствующие двоичные данные на принимающем узле. Мы не рассматриваем детали модуляции и предполагаем, что работаем с двумя дискретными сигналами: высоким и низким. На практике эти сигналы могут соответствовать двум разным напряжениям на медном канале, двум разным уровням мощности на оптическом канале или двум разным амплитудам в радиотрансляции.

Большинство функций, обсуждаемых в этой главе, выполняются сетевым адаптером — устройством, которое соединяет узел с каналом связи. Сетевой адаптер содержит компонент сигнализации, который фактически кодирует биты в сигналы на отправляющем узле и декодирует сигналы в биты на принимающем узле. Таким образом, как показано на рис. 2.3, сигналы перемещаются по каналу между двумя компонентами сигнализации, а биты — между сетевыми адаптерами.

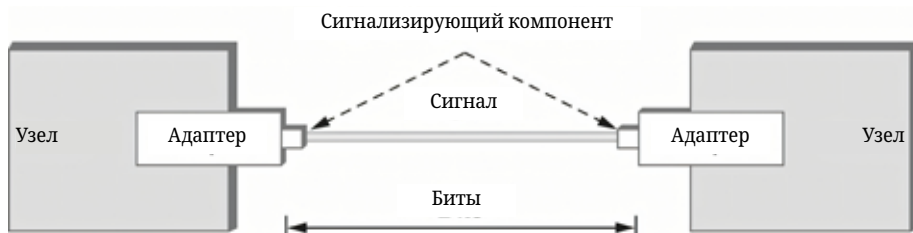


Рисунок 2.3. Сигналы проходят между сигнализирующими компонентами; биты проходят между адаптерами.

Давайте вернемся к проблеме кодирования битов в сигналы. Очевидное решение заключается в отображении значения 1 на высокий сигнал и значения 0 на низкий сигнал. Именно такое отображение используется в схеме кодирования, загадочно названной «без возврата к нулю» (non-return to zero, NRZ). Например, на рис. 2.4 схематически изображен сигнал, закодированный с использованием NRZ (внизу), соответствующий передаче определенной последовательности битов (вверху).

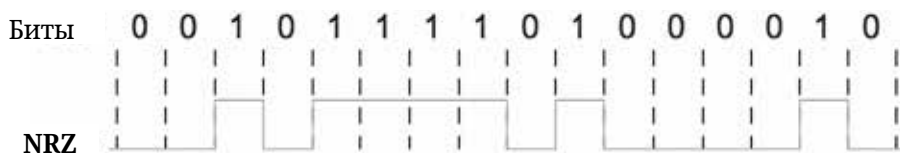


Рисунок 2.4. NRZ-кодирование битового потока.

Проблема NRZ заключается в том, что последовательность из нескольких последовательных 1 означает, что сигнал остается высоким на линии связи в течение длительного периода времени; аналогично несколько последовательных 0 означают, что сигнал остается низким в течение длительного времени. Есть две фундаментальные проблемы, вызванные длинными строками 1 или 0. Первая заключается в том, что это приводит к ситуации, известной как *блуждание базовой линии*. В частности, приемник хранит среднее значение сигнала, который он принимал до сих пор, и затем использует это среднее значение для различия низких и высоких сигналов. Если сигнал значительно ниже этого среднего значения, приемник делает вывод, что он только что увидел 0. Аналогично сигнал, значительно превышающий среднее значение, интерпретируется как 1. Проблема, конечно, в том, что слишком большое количество последовательных 1 или 0 приводит к изменению среднего значения, что затрудняет обнаружение значительных изменений в сигнале.

Вторая проблема заключается в том, что для *восстановления тактового сигнала* необходимы частые переходы от высокого уровня к низкому и *наоборот*. Интуитивно проблема восстановления тактовой частоты заключается в том, что процессы кодирования и декодирования управляются тактовым сигналом — каждый такт передатчик передает бит, а приемник восстанавливает бит. Часы передатчика и приемника должны быть точно синхронизированы, чтобы приемник корректно восстанавливал те же биты, которые передает передатчик. Если часы приемника немного быстрее или медленнее часов передатчика, приемник неправильно декодирует сигнал. Можно представить, что тактовый сигнал отправляется приемнику по отдельному проводу, но этого обычно избегают из-за удвоения стоимости кабельной системы. Вместо этого приемник извлекает тактовый сигнал из принятого сигнала — процесс восстановления тактовой частоты. Каждый раз, когда сигнал меняется, например, при переходе с 1 на 0 или с 0 на 1, приемник знает, что это граница тактового цикла, и может снова синхронизироваться. Однако длительный период времени без таких переходов приводит к смещению тактовой частоты. Таким образом, восстановление тактовой частоты зависит от наличия множества переходов в сигнале, независимо от того, какие данные отправляются.

Один из подходов к решению этой проблемы, называемый *инверсией без возврата к нулю* (non-return to zero inverted, NRZI), заключается в том, что передатчик совершает переход от текущего сигнала для кодирования 1 и остается на текущем сигнале для кодирования 0. Это решает проблему последовательных 1, но, очевидно, ничего не делает для последовательных 0. NRZI проиллюстрирован на рис. 2.5. Альтернативный метод, называемый *манчестерским кодированием*, более явно объединяет тактовый сигнал с данными, передавая результат операции исключающего ИЛИ между данными, закодированными по схеме NRZ, и тактовым сигналом. (Представьте себе локальный тактовый сигнал как внутренний сигнал, чередующийся от низкого уровня к высокому; пара низкий/высокий считается одним тактовым циклом.) Манчестерское кодирование также проиллюстрировано на рис. 2.5. Обратите внимание, что манчестерское кодирование приводит к тому, что 0 кодируется как переход от низкого уровня к высокому, а 1 кодируется как переход от высокого уровня к низкому. Поскольку и 0, и 1 приводят к переходу сигнала, тактовый сигнал может эффективно восстанавливаться на приемнике. (Существует также вариант манчестерского кодирования, называемый *дифференциальным манчестерским кодированием*, в котором 1 кодируется первой половиной сигнала,

равной последней половине сигнала предыдущего бита, а 0 кодируется первой половиной сигнала, противоположной последней половине сигнала предыдущего бита.)

Проблема манчестерской схемы кодирования заключается в том, что она удваивает частоту переходов сигнала на канале, а это означает, что у приемника вдвое меньше времени для обнаружения каждого импульса сигнала. Скорость, с которой сигнал изменяется, называется *скоростью передачи* (baud rate) канала. В случае манчестерского кодирования скорость передачи битов составляет половину скорости передачи сигналов, поэтому такое кодирование считается эффективным только на 50 %. Имейте в виду, что если бы приемник мог справиться с более высокой скоростью передачи сигналов, требуемой манчестерским кодированием на рис. 2.5, то и NRZ, и NRZI могли бы передавать вдвое больше битов за тот же период времени.

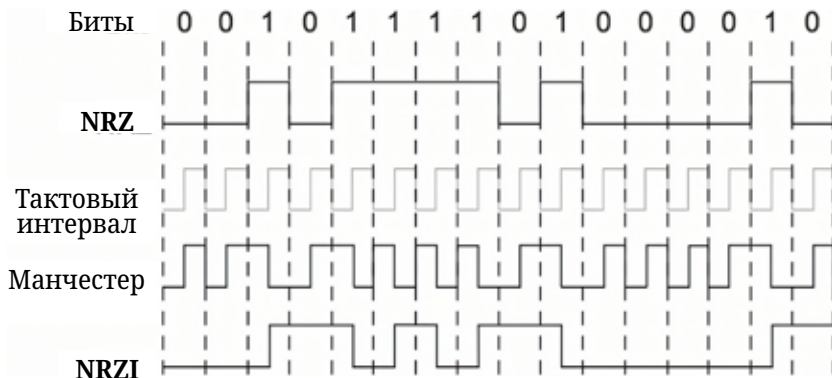


Рисунок 2.5. Различные стратегии кодирования.

Обратите внимание, что скорость передачи битов не обязательно меньше или равна скорости передачи сигналов (baud rate), как это предполагает манчестерское кодирование. Если схема модуляции способна использовать (и распознавать) четыре различных сигнала, а не только два (например, «высокий» и «низкий»), то можно кодировать два бита в каждом тактовом интервале, что приводит к скорости передачи битов, в два раза превышающей скорость передачи сигналов. Аналогично возможность модулировать между восемью различными сигналами означает возможность передавать три бита за тактовый интервал. В общем, важно помнить, что мы упростили модуляцию, которая гораздо сложнее, чем передача «высоких» и «низких» сигналов. Не редкость и варьирование комбинации фазы и амплитуды сигнала, что позволяет кодировать 16 или даже 64 различных паттерна (часто называемых *символами*) в каждом тактовом интервале. QAM (Quadrature Amplitude Modulation, *квадратурная амплитудная модуляция*) является широко используемым примером такой схемы модуляции.

Последнее кодирование, которое мы рассмотрим, называется 4B/5B и пытается решить проблему неэффективности манчестерского кодирования, не сталкиваясь с проблемой длительных периодов высоких или низких сигналов. Идея 4B/5B заключается во вставке дополнительных битов в поток битов, чтобы разбить длинные последовательности из 0 или 1. Если говорить конкретно, то каждые 4 бита реальных данных кодируются в 5-битный код, который затем передается приемнику; отсюда и название 4B/5B. 5-битные коды выбираются таким образом, что каждый из них имеет не более одного начального 0 и не более двух конечных 0. Таким образом, при передаче ни одна пара 5-битных кодов не приводит к передаче более чем трех последовательных 0. Полученные 5-битные коды затем передаются с использованием кодирования NRZI, что объясняет, почему код заботится только о последовательных 0 — NRZI уже решает проблему последовательных 1. Обратите внимание, что кодирование 4B/5B обеспечивает 80 % эффективности.

Таблица 2.2.
Кодирование 4В/5В.

4-битный символ данных	5-битный код
0000	11110
0001	01001
0010	10100
0011	10101
0100	01010
0101	01011
0110	01110
0111	01111
1000	10010
1001	10011
1010	10110
1011	10111
1100	11010
1101	11011
1110	11100
1111	11101

Таблица 2.2 показывает 5-битные коды, которые соответствуют каждому из 16 возможных 4-битных символов данных. Обратите внимание, что поскольку 5 бит достаточно для кодирования 32 различных кодов, и мы используем только 16 из них для данных, остается 16 кодов, которые мы можем использовать для других целей. Из них код 11111 используется, когда линия простаивает, код 00000 соответствует состоянию, когда линия мертва, а 00100 интерпретируется как остановка. Из оставшихся 13 кодов 7 являются недействительными, поскольку они нарушают правило «одного начального 0 и двух конечных 0», а остальные 6 представляют собой различные управляющие символы. Некоторые из протоколов кадрирования, описанные далее в этой главе, используют эти управляющие символы.

Глава 2.3. Фрейминг (кадрирование)

Теперь, когда мы рассмотрели, как передавать последовательность битов по каналу «точка-точка» (от адаптера к адаптеру), давайте рассмотрим сценарий, представленный на рис. 2.6. Напомним, что мы сосредотачиваемся на пакетно-коммутируемых сетях, а это означает, что блоки данных (на этом уровне называемые *кадрами*), а не потоки битов, обмениваются между узлами. Именно сетевой адаптер позволяет узлам обмениваться кадрами. Когда узел А хочет передать кадр узлу В, он сообщает своему адаптеру о необходимости передать кадр из памяти узла. Это приводит к отправке последовательности битов по каналу связи. Адаптер на узле В затем собирает последовательность битов, поступающих по каналу, и размещает соответствующий кадр в памяти узла В. Признание того, какая именно последовательность битов составляет кадр — то есть

определение, где кадр начинается и заканчивается, — является центральной задачей, стоящей перед адаптером.

Существует несколько способов решения проблемы фрейминга (или кадрирования). В этом разделе используются три различных протокола для иллюстрации различных подходов к разработке. Обратите внимание, что, хотя мы обсуждаем фрейминг в контексте каналов «точка-точка», эта проблема является фундаментальной и должна также решаться в сетях с множественным доступом, таких как Ethernet и Wi-Fi.



Рисунок 2.6. Поток битов между адаптерами, кадров между хостами.

Глава 2.3.1. Байт-ориентированные протоколы (PPP)

Один из самых старых подходов к кадрированию, имеющий корни в подключении терминалов к мейнфреймам, заключается в том, чтобы рассматривать каждый кадр как набор байтов (символов), а не набор битов. Ранние примеры таких *байт-ориентированных* протоколов включают Binary Synchronous Communication (BISYNC), разработанный IBM в конце 1960-х годов, и Digital Data Communication Message Protocol (DDCMP), использовавшийся в DECNET компании Digital Equipment Corporation. (Когда-то крупные компьютерные компании, такие как IBM и DEC, также строили частные сети для своих клиентов.) Широко используемый протокол «точка-точка» (Point-to-Point Protocol, PPP) является недавним примером этого подхода.

На высоком уровне существуют два подхода к байт-ориентированному кадрированию. Первый заключается в использовании специальных символов, известных как *символы-стражи* (sentinel characters), для указания начала и конца кадров. Идея состоит в том, чтобы обозначить начало кадра, отправляя специальный символ SYN (синхронизация). Данные кадра затем иногда содержатся между двумя другими специальными символами: STX (начало текста) и ETX (конец текста). BISYNC использовал этот подход. Проблема подхода со стражами, конечно, заключается в том, что один из специальных символов может появиться в данных кадра. Стандартный способ преодоления этой проблемы заключается в «экранировании» символа предшествующим ему символом DLE (data-link-escape) всякий раз, когда он появляется в теле кадра; символ DLE также экранируется (предшествующим ему дополнительным DLE) в теле кадра. (Программисты на C могут заметить, что это аналогично тому, как кавычка экранируется обратной косой чертой, когда она встречается внутри строки.) Этот подход часто называют *вставкой символов*, потому что именно это и происходит.

Альтернативой обнаружению конца кадра с помощью значения стража является включение количества байтов в кадре в начало кадра, в заголовок кадра. DDCMP использовал этот подход. Опасность такого подхода заключается в том, что ошибка передачи может повредить поле счетчика, в таком случае конец кадра не будет правильно обнаружен. (Похожая проблема существует с подходом на основе стража, если поле ETX становится поврежденным.) Если это произойдет, приемник будет накапливать столько байтов, сколько укажет неправильное поле счетчика, а затем использовать поле обнаружения ошибок, чтобы определить, что кадр испорчен. Это иногда называ-

ется *ошибкой кадрирования*. Приемник затем будет ждать, пока не увидит следующий символ SYN, чтобы начать собирать байты, составляющие следующий кадр. Таким образом, возможно, что ошибка кадрирования приведет к некорректному приему подряд идущих кадров.

Протокол PPP (протокол «точка-точка», Point-to-Point Protocol), который часто используется для передачи пакетов Интернет-протокола по различным видам точечных соединений, использует стражи (или сентинеллы) и вставку символов. Формат кадра PPP представлен на рис. 2.7.

Этот рисунок является первым из многих, которые вы увидите в данной книге, используемых для иллюстрации форматов кадров или пакетов, поэтому необходимо дать некоторые пояснения. Мы показываем пакет как последовательность помеченных полей. Над каждым полем указано число, обозначающее длину этого поля в битах. Обратите внимание, что пакеты передаются, начиная с крайнего левого поля.

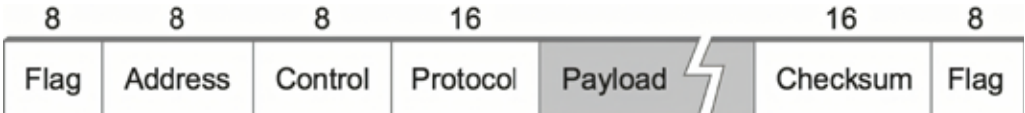


Рисунок 2.7. Формат кадра PPP.

Специальный символ начала текста, обозначаемый как поле Flag, находится по адресу 01111110. Поля Address и Control обычно содержат значения по умолчанию и поэтому не представляют интереса. Поле Protocol используется для демultipлексирования; оно идентифицирует протокол высокого уровня, например IP. Размер Payload-кадра может быть согласован, но по умолчанию он составляет 1500 байт. Поле Checksum имеет длину 2 (по умолчанию) или 4 байта. Обратите внимание, что, несмотря на общее название, это поле на самом деле является CRC, а не контрольной суммой (как описано в следующей главе).

Формат кадра PPP необычен тем, что несколько размеров полей согласуются, а не фиксируются. Это согласование проводится с помощью протокола под названием «протокол управления каналом» (Link Control Protocol, LCP). PPP и LCP работают вместе: LCP отправляет управляющие сообщения, инкапсулированные в кадры PPP — такие сообщения обозначаются идентификатором LCP в поле PPP, — а затем изменяет формат кадра PPP на основе информации, содержащейся в этих управляющих сообщениях. LCP также участвует в установлении соединения между двумя узлами, когда обе стороны обнаруживают возможность связи по каналу (например, когда каждый оптический приемник обнаруживает входящий сигнал от волокна, к которому он подключен).

Глава 2.3.2. Бит-ориентированные протоколы (HDLC)

В отличие от протоколов, ориентированных на байты, бит-ориентированный протокол не учитывает границы байтов — он просто рассматривает кадр как набор битов. Эти биты могут быть взяты из какого-то набора символов, например, ASCII; они могут быть значениями пикселей в изображении; или они могут быть инструкциями и операндами из исполняемого файла. Протокол управления синхронной передачей данных (The Synchronous Data Link Control, SDLC), разработанный IBM, является примером бит-ориентированного протокола; SDLC позже был стандартизирован ISO как протокол управления каналами данных высокого уровня (High-Level Data Link Control, HDLC). В следующем обсуждении мы используем HDLC в качестве примера; его формат кадра представлен на рис. 2.8.

HDLC обозначает начало и конец кадра специальной битовой последовательностью 01111110. Эта последовательность также передается в те моменты, когда канал бездействует, чтобы передатчик и приемник могли поддерживать синхронизацию своих часов. Таким образом, оба протокола, по сути, используют подход с сентинелом (или стражем).

Поскольку эта последовательность может появиться в любом месте в теле кадра — фактически биты 01111110 могут пересекать границы байтов, — бит-ориентированные протоколы используют аналог символа DLE, технику, известную как *вставка битов*.



Рисунок 2.8. Формат кадра HDLC.

Вставка битов в протоколе HDLC работает следующим образом. На стороне отправителя каждый раз, когда передаются пять последовательных 1 из тела сообщения (то есть за исключением случаев, когда отправитель пытается передать специальную последовательность 01111110), отправитель вставляет 0 перед передачей следующего бита. На стороне приемника, если приходят пять последовательных 1, приемник принимает решение на основе следующего бита, который он видит (то есть бит, следующий за пятью 1). Если следующий бит — 0, значит он был вставлен, и приемник его удаляет. Если следующий бит — 1, то верно одно из двух: либо это маркер конца кадра, либо в поток битов закралась ошибка. Смотря на *следующий* бит, приемник может различить эти два случая. Если он видит 0 (то есть последние 8 бит, которые он видел, это 01111110), то это маркер конца кадра; если он видит 1 (то есть последние 8 бит, которые он видел, это 01111111), то здесь должна быть ошибка, и весь кадр отбрасывается. В последнем случае приемник должен ждать следующего 01111110, прежде чем он сможет начать принимать снова, и, как следствие, существует вероятность, что приемник не сможет принять два последовательных кадра. Очевидно, что все еще существуют проблемы, из-за которых ошибки кадрирования могут остаться невыявленными, например, когда из-за ошибок генерируется целая ложная последовательность конца кадра, но эти сбои относительно маловероятны. Надежные способы обнаружения ошибок обсуждаются в следующей главе.

Интересной характеристикой вставки битов, как и вставки символов, является то, что размер кадра зависит от данных, передаваемых в полезной нагрузке кадра. На самом деле невозможно сделать все кадры точно одного размера, учитывая, что данные, которые могут быть переданы в любом кадре, произвольны. (Чтобы убедиться в этом, подумайте, что произойдет, если последний байт тела кадра будет символом ETX.) Форма кадрирования, которая обеспечивает одинаковый размер всех кадров, описана далее.

Глава 2.3.3. Кадрирование на основе тактового сигнала (SONET)

Третий подход к кадрированию представлен стандартом Synchronous Optical Network (SONET). За неимением широко принятого общего термина мы называем этот подход просто *кадрированием на основе тактового сигнала*. SONET был впервые предложен Bell Communications Research (Bellcore), а затем разработан под эгидой Американского национального института стандартов (ANSI) для цифровой передачи по оптоволоконным сетям; с тех пор он был принят ITU-T. На протяжении многих лет SONET оставался доминирующим стандартом для передачи данных на большие расстояния по оптическим сетям.

Важно отметить, что полная спецификация SONET значительно больше, чем эта книга. Таким образом, следующее обсуждение обязательно охватит только ключевые моменты стандарта. Также SONET решает как проблему кадрирования, так и проблему кодирования. Он также решает проблему, которая очень важна для телефонных компаний — мультиплексирование нескольких низкоскоростных каналов на один высокоскоростной канал. (Фактически многие аспекты дизайна SONET отражают тот факт, что телефонные компании должны заботиться о мультиплексировании большого количества каналов с пропускной способностью 64 кбит/с, которые традиционно используются для телефонных звонков). Мы начнем с подхода SONET к кадрированию и обсудим другие вопросы далее.

Как и в предыдущих схемах кадрирования, кадр SONET содержит некоторую специальную информацию, которая сообщает приемнику, где начинается и заканчивается кадр; однако на этом сходства заканчиваются. В частности, здесь не используется вставка битов, поэтому длина кадра не зависит от передаваемых данных. Закономерно возникает вопрос: как приемник узнает, где начинается и заканчивается каждый кадр? Рассмотрим этот вопрос для самого низкоскоростного канала SONET, который называется STS-1 и работает на скорости 51.84 Мбит/с. Кадр STS-1 показан на рисунке 2.9. Он представлен в виде 9 строк по 90 байтов каждая, и первые 3 байта каждой строки являются служебной информацией, а остальные предназначены для данных, передаваемых по каналу. Первые два байта кадра содержат специальную битовую последовательность, и именно эти байты позволяют приемнику определить, где начинается кадр. Однако, поскольку вставка битов не используется, нет никаких причин, почему бы эта последовательность не могла случайно появиться в полезной части кадра. Чтобы предотвратить это, приемник постоянно ищет специальную битовую последовательность, надеясь увидеть ее появление каждые 810 байтов, поскольку каждый кадр имеет длину $9 \times 90 = 810$ байтов. Когда специальная последовательность появляется в нужном месте достаточное количество раз, приемник заключает, что он синхронизирован, и может правильно интерпретировать кадр.

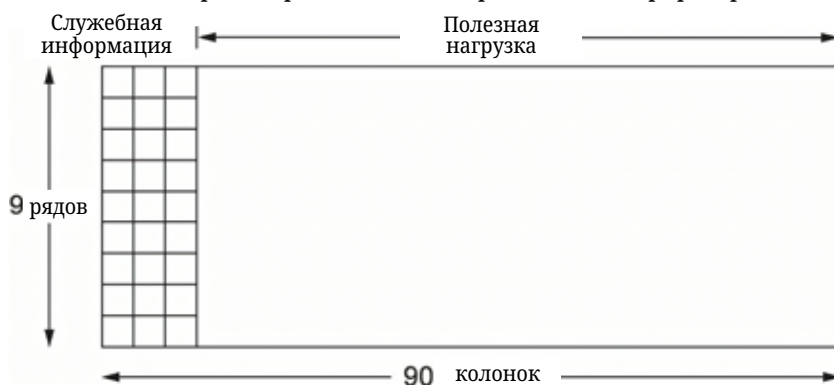


Рисунок 2.9. Кадр SONET STS-1.

Одна из вещей, которую мы не описываем из-за сложности SONET, — это детальное использование всех остальных байтов служебной информации. Часть этой сложности можно объяснить тем, что SONET работает через оптическую сеть оператора, а не только по одному каналу. (Напомним, что мы упрощаем и не учитываем тот факт, что операторы реализуют сеть, и вместо этого делаем акцент на том, что мы можем арендовать канал SONET у них и использовать этот канал для создания собственной пакетной сети.) Дополнительная сложность возникает из-за того, что SONET предоставляет значительно более широкий набор услуг, чем просто передача данных. Например, 64 кбит/с пропускной способности канала SONET зарезервированы для голосового канала, который используется для телефонного обслуживания.

Верхние байты кадра SONET кодируются с использованием NRZ, простого кодирования, описанного в предыдущей главе, где 1 соответствует высокому сигналу, а 0 — низкому. Однако, чтобы обеспечить достаточное количество переходов для восстановления тактового сигнала отправителем приемником, байты полезной нагрузки подвергаются *скремблированию*. Это делается путем вычисления, исключающего ИЛИ (XOR) данных, которые должны быть переданы с использованием известной битовой последовательности. Битовая последовательность длиной 127 бит содержит много переходов от 1 к 0, поэтому ее использование с данными передачи позволяет получить сигнал с достаточным количеством переходов для восстановления тактового сигнала.

SONET поддерживает мультиплексирование нескольких низкоскоростных каналов следующим образом. Определенный канал SONET работает на одной из конечного набо-

ра возможных скоростей, начиная с 51.84 Мбит/с (STS-1) и до 39,813,120 Мбит/с (STS-768)¹. Следует отметить, что все эти скорости являются целыми кратными STS-1. Значение скорости для кадрирования заключается в том, что один кадр SONET может содержать подкадры для нескольких низкоскоростных каналов. Вторая связанная особенность заключается в том, что каждый кадр имеет длину 125 мкс. Это означает, что на скорости STS-1 кадр SONET имеет длину 810 байт, а на скорости STS-3 каждый кадр SONET имеет длину 2430 байт. Обратите внимание на синергию между этими двумя особенностями: $3 \times 810 = 2430$, и это означает, что три кадра STS-1 точно укладываются в один кадр STS-3.

Интуитивно кадр STS-N можно рассматривать как состоящий из N кадров STS-1, байты которых чередуются; то есть сначала передается байт из первого кадра, затем байт из второго кадра и так далее. Причина чередования байтов из каждого кадра STS-N заключается в обеспечении равномерного поступления байтов каждого кадра STS-1 на приемник; то есть байты поступают на приемник с постоянной скоростью 51 Мбит/с, а не скапливаются все сразу в одной из N частей 125-мкс интервала.

Хотя сигнал STS-N можно рассматривать как результат мультиплексирования N кадров STS-1, данные из этих кадров STS-1 можно объединить в большую полезную нагрузку STS-N; такая связь обозначается как STS-Nc (конкатенированная). Одно из полей в верхней части используется для этой цели. На схеме на рис. 2.10 показан пример конкатенации в случае трех кадров STS-1, объединенных в один кадр STS-3c. Важность того, что SONET-связь обозначена как STS-3c, а не STS-3, заключается в том, что в первом случае пользователь может рассматривать ее как одну 155,25-Мбит/с трубу, в то время как STS-3 следует рассматривать как три отдельных 51,84-Мбит/с канала, которые случайно совместно используют один оптоволоконный кабель.

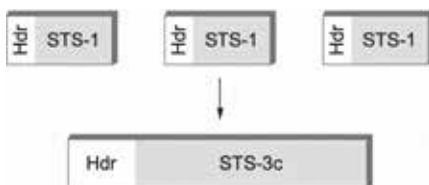


Рисунок 2.10. Три кадра STS-1, мультиплексированные на один кадр STS-3c.

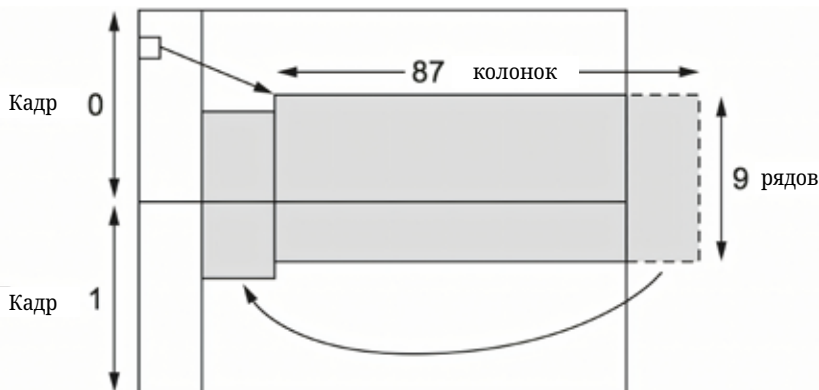


Рисунок 2.11. Кадры SONET вне фазы.

¹ STS означает Synchronous Transport Signal (синхронный транспортный сигнал), и это термин, который SONET использует для обозначения кадров. Существует параллельный термин — Optical Carrier (оптический носитель, OC), который используется для обозначения основного оптического сигнала, передающего кадры SONET. Мы говорим, что эти два термина параллельны, потому что STS-3 и OC-3, если использовать конкретный пример, оба подразумевают скорость передачи 155,52 Мбит/с. Поскольку мы сосредоточены здесь на кадрировании, мы будем использовать STS, но, скорее всего, вы услышите, что кто-то упоминает оптический канал по его названию «OC».

Следует отметить, что предыдущее описание SONET является слишком упрощенным, поскольку предполагает, что полезная нагрузка для каждого кадра полностью содержится внутри кадра. (Почему бы и нет?) Фактически мы должны рассматривать кадр STS-1, описанный выше, просто как заполнитель для кадра, где фактическая полезная нагрузка может *перемещаться* за пределы кадров. Эта ситуация иллюстрируется на рис. 2.11. Здесь показано, как полезная нагрузка STS-1 перемещается за пределы двух кадров STS-1, а также как полезная нагрузка сдвигается на несколько байт вправо и, следовательно, переносится на новую строку. Одно из полей в верхней части кадра указывает на начало полезной нагрузки. Значение этой возможности заключается в том, что это упрощает задачу синхронизации используемых часов по всей сети передатчиков и приемников, на что приемники тратят много времени.

Глава 2.4. Обнаружение ошибок

Как обсуждалось в первом разделе, в кадры иногда вводятся битовые ошибки. Это происходит, например, из-за электрических помех или теплового шума. Хотя ошибки редки, особенно на оптических линиях связи, необходим некоторый механизм для их обнаружения, чтобы можно было принять корректирующие меры. В противном случае конечный пользователь остается в недоумении, почему программа на языке С, которая успешно компилировалась всего минуту назад, теперь внезапно имеет синтаксическую ошибку, хотя единственное, что произошло в малом промежутке времени, это ее копирование через сетевую файловую систему.

Существует долгая история борьбы с битовыми ошибками в компьютерных системах, которая восходит как минимум к 1940-м годам. Код Хэмминга и коды Рида-Соломона — два известных способа, которые были разработаны для использования в считывателях перфокарт, при хранении данных на магнитных дисках и в ранней оперативной памяти. В этой главе описываются некоторые из наиболее часто используемых в сетях методов обнаружения ошибок.

Обнаружение ошибок — это только одна часть проблемы. Другая часть — исправление ошибок после их обнаружения. Существует два основных подхода, которые можно применить, когда получатель сообщения обнаруживает ошибку. Один из них — уведомить отправителя о том, что сообщение повреждено, чтобы отправитель мог повторно передать копию сообщения. Так как битовые ошибки редки, то, вероятно, повторно переданная копия будет без ошибок. Альтернативно некоторые типы алгоритмов обнаружения ошибок позволяют получателю восстановить правильное сообщение даже после его повреждения; такие алгоритмы полагаются на коды *исправления ошибок*, обсуждаемые ниже.

Одним из самых распространенных методов обнаружения ошибок при передаче данных является метод, известный как *циклическая избыточная проверка* (cyclic redundancy check, CRC). Он используется почти во всех протоколах канального уровня, обсуждаемых в этом разделе. В данной главе приводится основной алгоритм CRC, но перед обсуждением этого метода мы сначала опишем более простой алгоритм *контрольной суммы*, используемый в нескольких интернет-протоколах.

Основная идея любой схемы обнаружения ошибок заключается в добавлении избыточной информации в кадр, которая может быть использована для определения, были ли введены ошибки. В крайнем случае можно представить себе передачу двух полных копий данных. Если две копии идентичны у приемника, то, вероятно, обе правильны. Если они различаются, то ошибка была введена в одну (или обе) из них, и их нужно отбросить. Это довольно плохая схема обнаружения ошибок по двум причинам. Во-первых, она передает n избыточных битов для сообщения из n бит. Во-вторых, многие ошибки останутся необнаруженными — любые ошибки, которые случайно повредят одинаковые позиции битов в первой и второй копиях сообщения. В общем, цель кодов

обнаружения ошибок заключается в обеспечении высокой вероятности обнаружения ошибок при относительно малом количестве избыточных битов.

К счастью, мы можем обеспечить достаточно высокую вероятность обнаружения ошибок, передавая всего k избыточных битов для сообщения из n бит, где k значительно меньше, чем n . Например, в Ethernet кадр, несущий до 12 000 бит (1500 байт) данных, требует всего 32-битного кода CRC, или, как это обычно обозначается, использует CRC-32. Такой код обнаружит подавляющее большинство ошибок, как мы увидим ниже.

Мы называем дополнительные биты избыточными, потому что они не добавляют новой информации к сообщению. Вместо этого они напрямую получены из исходного сообщения с использованием некоторого определенного алгоритма. Как отправитель, так и получатель точно знают, что это за алгоритм. Отправитель применяет алгоритм к сообщению, чтобы сгенерировать избыточные биты. Затем он передает как сообщение, так и эти несколько дополнительных битов. Когда получатель применяет тот же алгоритм к полученному сообщению, он должен (при отсутствии ошибок) получить тот же результат, что и отправитель. Он сравнивает результат с тем, который был отправлен ему отправителем. Если они совпадают, можно заключить (с высокой вероятностью), что ошибок в сообщении во время передачи не произошло. Если они не совпадают, можно быть уверенным, что либо сообщение, либо избыточные биты были повреждены, и необходимо принять соответствующие меры — то есть отбросить сообщение или исправить его, если это возможно.

Обратите внимание на терминологию для этих дополнительных битов. Обобщенно их называют *кодами обнаружения ошибок*. В конкретных случаях, когда алгоритм для создания кода основан на сложении, их могут называть *контрольной суммой*. Мы убедимся, что контрольная сумма в Интернете имеет соответствующее название: это проверка ошибок, использующая алгоритм суммирования. К сожалению, словосочетание «*контрольная сумма*» часто используется неточно, чтобы обозначить любую форму кода обнаружения ошибок, включая CRC. Это может быть запутанным, поэтому мы призываем вас использовать словосочетание «*контрольная сумма*» только для обозначения кодов, которые действительно используют сложение, и использовать «*код обнаружения ошибок*» для обозначения общего класса кодов, описанных в этой главе.

Глава 2.4.1. Алгоритм контрольной суммы Интернета

Наш первый подход к обнаружению ошибок представлен контрольной суммой Интернета. Хотя она не используется на канальном уровне, она тем не менее обеспечивает ту же функциональность, что и CRC, поэтому мы обсуждаем ее здесь.

Идея контрольной суммы Интернета очень проста — вы складываете все слова, которые передаются, а затем передаете результат полученной суммы. Этот результат и является контрольной суммой. Получатель выполняет те же вычисления с полученными данными и сравнивает результат с полученной контрольной суммой. Если какие-либо переданные данные, включая саму контрольную сумму, повреждены, результаты не совпадут, и получатель узнает, что произошла ошибка.

Можно представить множество различных вариаций базовой идеи контрольной суммы. Точная схема, используемая интернет-протоколами, работает следующим образом. Рассмотрим данные, для которых вычисляется контрольная сумма, как последовательность 16-битных целых чисел. Складываем их, используя 16-битную арифметику с дополнением до единицы (объяснено ниже), а затем берем дополнение до единицы от результата. Это 16-битное число и есть контрольная сумма.

В арифметике с дополнением единицы отрицательное целое число ($-x$) представляется как дополнение x , то есть каждый бит x инвертируется. При сложении чисел в арифметике с дополнением единицы к результату необходимо добавить перенос из старшего бита. Рассмотрим, например, сложение -5 и -3 в арифметике дополнения единицы на 4-битных целых числах: $+5$ — это 0101, поэтому -5 — это 1010; $+3$ — это 0011, поэтому

–3 — это 1100. Если сложить 1010 и 1100, игнорируя перенос, то получится 0110. В арифметике с дополнением до единицы тот факт, что эта операция вызвала перенос из старшего бита, заставляет нас увеличить результат, что дает 0111, который, как и следовало ожидать, является дополнением до единицы представления –8 (полученного путем инвертирования битов в 1000).

Следующая процедура дает прямую реализацию алгоритма контрольной суммы Интернета. Аргумент `count` указывает длину `buf`, измеренную в 16-битных единицах. Предполагается, что `buf` уже дополнен нулями до границы 16 бит.

```
u_short
cksum(u_short *buf, int count)
{
    register u_long sum = 0;
    while (count--)
    {
        sum += *buf++;
        if (sum & 0xFFFF0000)
        {
            /* произошел перенос, поэтому оборачиваем */
            sum &= 0xFFFF;
            sum++;
        }
    }
    return ~(sum & 0xFFFF);
}
```

Этот код гарантирует, что при вычислениях используется арифметика с дополнением единицы, а не с дополнением двойки, которая используется в большинстве компьютеров. Обратите внимание на оператор `if` внутри цикла `while`. Если в верхних 16 битах суммы есть перенос, то мы увеличиваем сумму, как и в предыдущем примере.

По сравнению с нашим повторяющимся кодом этот алгоритм хорошо справляется с использованием малого количества избыточных битов — всего 16 для сообщения любой длины, но он не очень хорош для профессионального обнаружения ошибок. Например, пара односторонних ошибок, одна из которых увеличивает слово, а другая уменьшает другое слово на ту же величину, останется необнаруженной. Причина использования такого алгоритма, несмотря на его относительно слабую защиту от ошибок (по сравнению с CRC, например), проста: этот алгоритм намного легче реализовать в программном обеспечении. Опыт показывает, что контрольная сумма такой формы была адекватной, но одной из причин, по которой она является адекватной, является то, что эта контрольная сумма — последняя линия защиты в протоколе от конца до конца. Большинство ошибок обнаруживаются более сильными алгоритмами обнаружения ошибок, такими как CRC, на канальном уровне.

Глава 2.4.2. Циклический избыточный код

Теперь должно быть понятно, что основной целью при разработке алгоритмов обнаружения ошибок является максимизация вероятности обнаружения ошибок при использовании лишь небольшого количества избыточных битов. Циклические избыточные коды (CRC) используют довольно мощную математику для достижения этой цели. Например, 32-битный CRC обеспечивает сильную защиту от распространенных битовых ошибок в сообщениях длиной в тысячи байт. Теоретическая основа циклического избыточного кода коренится в разделе математики, называемом *конечными полями*. Хотя это может звучать устрашающе, основные идеи можно легко понять.

Для начала представьте $(n+1)$ -битное сообщение как представляемое многочленом степени n , то есть многочленом, наивысший член которого x^n . Сообщение представляет многочленом, используя значение каждого бита в сообщении в качестве коэффициента для каждого члена многочлена, начиная с самого значимого бита, чтобы представить наивысший член. Например, 8-битное сообщение, состоящее из битов 10011010, соответствует многочлену

$$M(x) = (1 \times x^7) + (0 \times x^6) + (0 \times x^5) + (1 \times x^4) + (1 \times x^3) + (0 \times x^2) + (1 \times x^1) + (0 \times x^0)$$

$$M(x) = x^7 + x^4 + x^3 + x^1$$

Таким образом, можно считать, что отправитель и получатель обмениваются многочленами друг с другом.

Для целей вычисления CRC отправитель и получатель должны согласовать *делящий многочлен* $C(x)$. $C(x)$ — это многочлен степени k . Например, предположим, что $C(x) = x^3 + x^2 + 1$. В этом случае $k = 3$. Ответ на вопрос «Откуда взялся $C(x)$?» в большинстве практических случаев звучит так: «Вы находите его в книге». На самом деле выбор $C(x)$ имеет значительное влияние на то, какие типы ошибок могут быть надежно обнаружены, как мы обсудим ниже. Существует несколько делящих многочленов, которые являются очень хорошим выбором для различных сред, и точный выбор обычно делается как составляющая проектирования протокола. Например, стандарт Ethernet использует хорошо известный многочлен степени 32.

Когда отправитель хочет передать сообщение $M(x)$ длиной в $n+1$ бит, фактически передается сообщение длиной в $n+1$ бит плюс k бит. Мы называем полное переданное сообщение, включая избыточные биты, $P(x)$. Мы собираемся сделать так, чтобы многочлен, представляющий $P(x)$, был точно делимым на $C(x)$; ниже мы объясним, как это достигается. Если $P(x)$ передается по каналу и в ходе передачи не было внесено ошибок, то получатель должен быть в состоянии разделить $P(x)$ на $C(x)$ точно, оставив остаток равным нулю. С другой стороны, если в $P(x)$ в ходе передачи была внесена ошибка, то, скорее всего, полученный многочлен уже не будет точно делимым на $C(x)$, и, таким образом, получатель получит ненулевой остаток, что укажет на наличие ошибки.

Для лучшего понимания данного алгоритма полезно немного знать о многочленной арифметике; она немного отличается от обычной целочисленной арифметики. Мы имеем дело со специальным классом многочленной арифметики, где коэффициенты могут быть только нулем или единицей, а операции над коэффициентами выполняются с использованием арифметики по модулю 2. Это называется «многочленная арифметика по модулю 2». Поскольку эта книга о сетях, а не математический текст, давайте сосредоточимся на ключевых свойствах этого типа арифметики для наших целей (что мы просим вас принять как данное):

- Любой многочлен $B(x)$ может быть разделен на делящий многочлен $C(x)$, если $B(x)$ имеет более высокую степень, чем $C(x)$.
- Любой многочлен $B(x)$ может быть разделен один раз на делящий многочлен $C(x)$, если $B(x)$ имеет ту же степень, что и $C(x)$.
- Остаток, полученный при делении $B(x)$ на $C(x)$, получается путем выполнения операции исключающего ИЛИ (XOR) на каждой паре соответствующих коэффициентов.

Например, многочлен $x^3 + 1$ можно разделить на $x^3 + x^2 + 1$ (потому что они оба имеют степень 3), и остаток будет $0 \times x^3 + 1 \times x^2 + 0 \times x^1 + 0 \times x^0 = x^2$ (полученный путем исключающего ИЛИ коэффициентов каждого члена). В терминах сообщений можно сказать, что 1001 можно разделить на 1101 и оставить остаток 0100. Вы должны видеть, что остаток представляет собой просто побитовое исключающее ИЛИ двух сообщений.

Теперь, когда мы знаем основные правила деления многочленов, мы можем выполнять длинное деление, которое необходимо для работы с более длинными сообщениями. Пример показан ниже.

Вспомним, что мы хотели создать многочлен для передачи, который получается из исходного сообщения $M(x)$, который длиннее $C(x)$ на k бит и точно делится на $C(x)$. Мы можем сделать это следующим образом:

1. Умножить $M(x)$ на x^k ; то есть добавить k нулей в конец сообщения. Назовем это сообщением с добавленными нулями $S(x)$.
2. Разделить $T(x)$ на $C(x)$ и найти остаток.
3. Вычесть остаток из $T(x)$.

Должно быть очевидно, что оставшееся на данном этапе сообщение точно делится на $C(x)$. Можно также отметить, что результирующее сообщение состоит из $M(x)$, за которым следует остаток, полученный на шаге 2, потому что, когда мы вычли остаток (который не может быть длиннее k бит), мы просто применили исключающее ИЛИ с k нулями, добавленными на шаге 1. Эта часть станет понятнее на примере.

Рассмотрим сообщение $x^7 + x^4 + x^3 + x^1$, или 10011010. Мы начинаем с умножения на x^3 , так как наш делящий многочлен имеет степень 3. Это дает 10011010000. Мы делим это на $C(x)$, что в данном случае соответствует 1101. Рис. 2.12 показывает операцию длинного деления многочленов. Учитывая правила многочленной арифметики, описанные выше, операция длинного деления происходит так же, как если бы мы делили целые числа. Таким образом, на первом шаге нашего примера мы видим, что делитель 1101 делится один раз на первые четыре бита сообщения (1001), так как они одной степени, и оставляет остаток 100 (1101 XOR 1001). Следующий шаг — это взять цифру из сообщения до тех пор, пока мы не получим еще один многочлен той же степени, что и $C(x)$, в данном случае 1001. Мы снова вычисляем остаток (100) и продолжаем, пока вычисление не будет завершено. Обратите внимание, что «результат» длинного деления, который появляется в верхней части вычисления, не имеет особого значения — важен остаток в конце.

В самом низу рис. 2.12 видно, что остаток в примере вычисления равен 101. Поэтому мы знаем, что 10011010000 минус 101 будет делиться на $C(x)$ без остатка, и это то, что мы отправляем. Операция вычитания в полиномиальной арифметике — это логическая операция исключающего ИЛИ (XOR), так что на самом деле мы отправляем 10011010101. Как отмечалось выше, эта информация оказывается исходным сообщением с добавленным к нему остатком от вычисления деления. Получатель делит полученный полином на $C(x)$, и, если результат равен 0, он заключает, что ошибок не было. Если результат ненулевой, может потребоваться отбросить поврежденное сообщение; в некоторых кодах небольшую ошибку можно исправить (например, если ошибка затронула только один бит). Код, который позволяет исправлять ошибки, называется *кодом, исправляющим ошибки* (ECC).

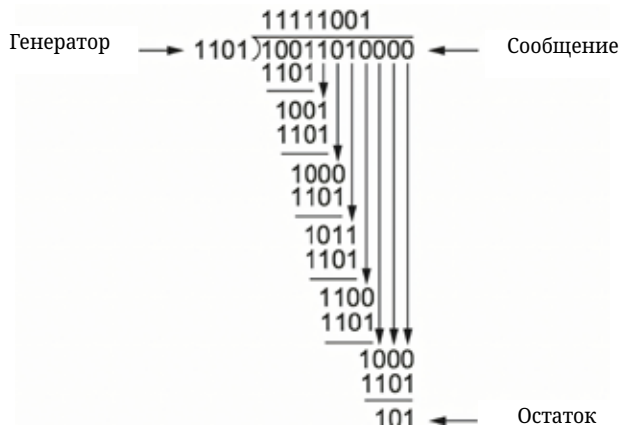


Рисунок 2.12. Вычисление CRC с помощью полиномиального длинного деления.

Теперь рассмотрим вопрос, откуда берется полином $C(x)$. Интуитивно идея состоит в том, чтобы выбрать этот полином так, чтобы вероятность его деления без остатка на сообщение с ошибками была очень мала. Если переданное сообщение — это $P(x)$, то введение ошибок можно представить как добавление другого полинома $E(x)$, так что получатель видит $P(x) + E(x)$. Единственный способ, при котором ошибка может остаться незамеченной, — это если полученное сообщение можно ровно разделить на $C(x)$, и поскольку мы знаем, что $P(x)$ можно ровно разделить на $C(x)$, это могло бы случиться только в том случае, если $E(x)$ можно ровно разделить на $C(x)$. Трюк заключается в том, чтобы выбрать $C(x)$ так, чтобы это было очень маловероятно для типичных ошибок.

Одним из распространенных типов ошибок является одноразрядная ошибка, которую можно выразить как $E(x) = x^i$, когда она затрагивает позицию бита i . Если мы выберем $C(x)$ таким образом, чтобы первый и последний члены (то есть члены x^k и x^0) были ненулевыми, то у нас уже есть двучлен, который не может делиться на одночлен $E(x)$ без остатка. Таким образом, подобный $C(x)$ может обнаруживать все одноразрядные ошибки. В общем, можно доказать, что следующие типы ошибок могут быть обнаружены с помощью $C(x)$ с указанными свойствами:

- Все одноразрядные ошибки, если члены x^k и x^0 имеют ненулевые коэффициенты
- Все двуразрядные ошибки, если $C(x)$ имеет множитель как минимум с тремя членами
- Любое нечетное количество ошибок, если $C(x)$ содержит множитель $(x + 1)$

Мы упомянули, что возможно использовать коды, которые не только обнаруживают наличие ошибок, но и позволяют их исправлять. Поскольку подробности таких кодов требуют еще более сложной математики, чем та, что необходима для понимания CRC (циклических избыточных кодов), мы не будем останавливаться на них. Однако стоит рассмотреть достоинства исправления ошибок по сравнению с их обнаружением.

На первый взгляд может показаться, что исправление всегда лучше, так как при обнаружении ошибок мы вынуждены отбросить сообщение и, в общем, попросить передать его заново. Это расходует полосу пропускания и может вызвать задержку в ожидании повторной передачи. Однако у исправления есть обратная сторона: оно, как правило, требует большего количества избыточных бит для отправки кода, исправляющего ошибки, который так же силен (то есть способен справляться с таким же количеством ошибок), как код, который только обнаруживает ошибки. Таким образом, хотя обнаружение ошибок требует отправки большего количества бит только при возникновении ошибок, исправление ошибок требует отправки большего количества бит *постоянно*. В результате исправление ошибок наиболее полезно, когда (1) ошибки довольно вероятны, как это может быть, например, в беспроводной среде, или (2) стоимость повторной передачи слишком высока, например, из-за задержки, связанной с повторной передачей пакета по спутниковому каналу.

Использование кодов исправления ошибок в сетях иногда называют *прямым исправлением ошибок* (forward error correction, FEC), потому что исправление ошибок обрабатывается «заранее» путем отправки дополнительной информации, а не ожиданием появления ошибок и их устранения путем повторной передачи. FEC часто используется в беспроводных сетях, таких как 802.11.

- Любая «пакетная» ошибка (то есть последовательность последовательных ошибочных битов), длина которой меньше k бит (большинство пакетных ошибок длиной более k бит также могут быть обнаружены).

Шесть версий $C(x)$ широко используются в протоколах канального уровня. Например, Ethernet использует CRC-32, который определяется следующим образом:

- $\text{CRC-32} = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$

Наконец, отметим, что алгоритм CRC, хотя и кажется сложным, легко реализуется в аппаратном обеспечении с использованием k -битного сдвигового регистра и логических элементов XOR. Количество бит в сдвиговом регистре равно степени порождающего многочлена (k). На рис. 2.13 показано оборудование, которое используется для порождающего многочлена $x^3 + x^2 + 1$ из нашего предыдущего примера. Сообщение подается слева, начиная с самого значащего бита и заканчивая строкой из k нулей, которая прикрепляется к сообщению, как в примере с делением в столбик. Когда все биты поданы и обработаны с использованием логических элементов XOR (исключающее ИЛИ), регистр содержит остаток — то есть CRC (самый значащий бит справа). Положение логических элементов XOR определяется следующим образом: если биты в сдвиговом регистре пронумерованы от 0 до $k - 1$ слева направо, то логический элемент XOR ставится перед битом n , если в порождающем многочлене есть член x^n . Таким образом, мы видим логический элемент XOR перед позициями 0 и 2 для порождающего многочлена $x^3 + x^2 + x^0$.

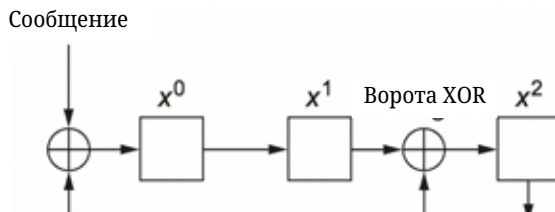


Рисунок 2.13. Вычисление CRC с помощью сдвигового регистра.

Глава 2.5. Надежная передача

Как мы видели в предыдущей главе, кадры иногда повреждаются во время передачи, и код ошибки, такой как CRC, используется для обнаружения таких ошибок. Хотя некоторые коды ошибок достаточно сильны, чтобы также исправлять ошибки, на практике накладные расходы обычно слишком велики, чтобы справиться с диапазоном битовых и пакетных ошибок, которые могут возникнуть на сетевом канале. Даже когда используются коды исправления ошибок (например, на беспроводных каналах), некоторые ошибки будут слишком серьезными для исправления. В результате некоторые поврежденные кадры должны быть отброшены. Протокол канального уровня, который хочет надежно передавать кадры, должен каким-то образом восстанавливать эти отброшенные (утерянные) кадры.

Стоит отметить, что надежность может обеспечиваться на канальном уровне, но многие современные канальные технологии не включают эту функцию. Более того, надежная доставка часто обеспечивается на более высоких уровнях, включая транспортный и иногда прикладной уровень. Точное место, где она должна обеспечиваться, является предметом споров и зависит от многих факторов. Мы описываем основы надежной доставки здесь, поскольку принципы одинаковы для всех уровней, но вам следует знать, что мы говорим не только о функции канального уровня.

Надежная доставка обычно достигается с помощью комбинации двух фундаментальных механизмов — *подтверждений* и *тайм-аутов*. Подтверждение (acknowledgement, ACK) — это небольшой управляющий кадр, который протокол отправляет своему аналогу, сообщая о получении ранее отправленного кадра. Под управляющим кадром мы понимаем заголовок без данных, хотя протокол может вставить ACK в кадр данных, который он отправляет в противоположном направлении. Получение подтверждения указывает отправителю исходного кадра, что его кадр был успешно доставлен. Если отправитель не получает подтверждение после разумного времени ожидания, он *повторно отправляет* исходный кадр. Это действие ожидания разумного времени называется *тайм-аутом*.

Общая стратегия использования подтверждений и тайм-аутов для реализации надежной доставки иногда называется *автоматическим повторным запросом* (automatic repeat request, ARQ). В этой главе описываются три различных алгоритма ARQ с использованием общих терминов; то есть мы не даем подробную информацию о полях заголовков конкретного протокола.

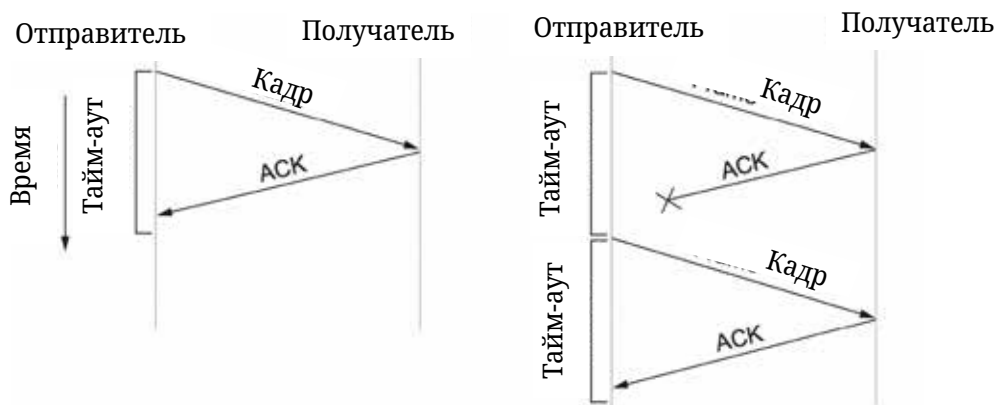
Глава 2.5.1. «Стоп-и-жди» (Stop-and-wait)

Самая простая схема ARQ — это алгоритм «*стоп-и-жди*» (stop-and-wait). Идея «стоп-и-жди» проста: после передачи одного кадра отправитель ждет подтверждения, прежде чем передавать следующий кадр. Если подтверждение не приходит после определенного периода времени, отправитель ждет тайм-аут и повторно отправляет исходный кадр.

Рис. 2.14 иллюстрирует временные диаграммы для четырех различных сценариев, вытекающих из этого базового алгоритма. Сторона отправителя представлена слева, сторона получателя изображена справа, и время течет сверху вниз. Рис. 2.14 (a) показывает ситуацию, в которой ACK получен до истечения таймера; (b) и (c) показывают ситуации, в которых исходный кадр и ACK, соответственно, были потеряны; и (d) показывает ситуацию, в которой тайм-аут срабатывает слишком рано. Напомним, что под «потерянным» мы подразумеваем, что кадр был поврежден во время передачи, это повреждение было обнаружено кодом ошибки на приемной стороне, и кадр был впоследствии отброшен.

Временные диаграммы пакетов, показанные в этом разделе, являются примерами часто используемого инструмента для обучения, объяснения и проектирования протоколов. Они полезны, потому что визуально отображают поведение распределенной системы во времени — что, возможно, будет довольно сложно проанализировать. При проектировании протокола вы часто должны быть готовы к непредвиденным ситуациям — система ломается, сообщение теряется или что-то, что, как вы ожидали, произойдет быстро, на самом деле занимает много времени. Такие диаграммы могут помочь нам понять, что может пойти не так в подобных случаях, и, следовательно, помочь разработчику протокола быть готовым ко всем возможным ситуациям.

В алгоритме остановки и ожидания есть одна важная тонкость. Предположим, отправитель посылает кадр, а получатель подтверждает его, но подтверждение либо потеряно, либо приходит с задержкой. Такая ситуация показана на временных шкалах (c) и (d) на рисунке 2.14. В обоих случаях отправитель задерживается и повторно передает исходный кадр, но получатель будет думать, что это следующий кадр, поскольку он правильно принял и подтвердил первый кадр. Это может привести к доставке дублирующих копий кадра. Для решения этой проблемы в заголовок протокола с остановкой и ожиданием обычно включают 1-битный номер последовательности, то есть:



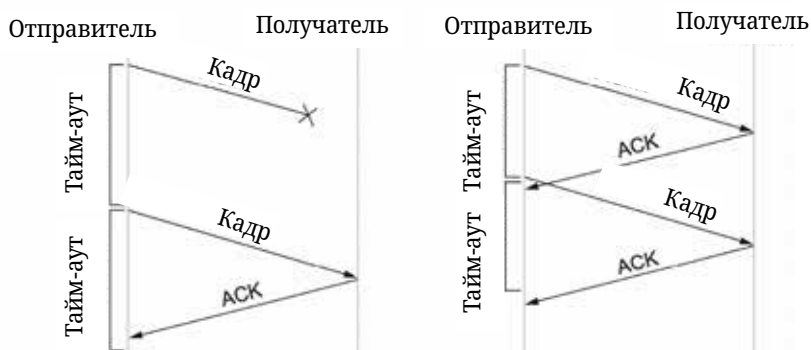


Рисунок 2.14. Временная шкала, показывающая четыре различных сценария для алгоритма остановки и ожидания. (a) ACK получен до истечения таймера; (b) исходный кадр потерян; (c) ACK потерян; (d) тайм-аут срабатывает слишком рано.

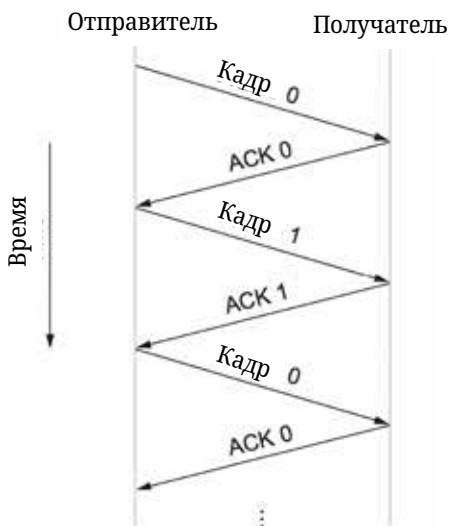


Рисунок 2.15. Временная шкала для остановки и ожидания с 1-битным порядковым номером.

Основной недостаток алгоритма «стоп-и-жди» заключается в том, что он позволяет отправителю иметь только один кадр в процессе передачи по каналу в любой момент времени, что может быть значительно ниже пропускной способности канала. Рассмотрим, например, канал с пропускной способностью 1,5 Мбит/с и временем кругового обхода 45 мс. У этого канала произведение задержки на пропускную способность составляет 67,5 Кбит, или примерно 8 КБ. Поскольку отправитель может отправлять только один кадр за время кругового обхода, и, предполагая, что размер кадра составляет 1 КБ, это предполагает максимальную скорость отправки

$$\text{биты-за-кадр} / \text{время-за-кадр} = 1024 \times 8 / 0,045 = 182 \text{ Кбит/с}$$

или примерно одну восьмую от пропускной способности канала. Таким образом, чтобы полностью использовать канал, мы хотели бы, чтобы отправитель мог передавать до восьми кадров, прежде чем ему придется ждать подтверждения.

Основные выводы

Значимость произведения задержки на пропускную способность заключается в том, что оно представляет количество данных, которые могут находиться в процессе передачи. Мы хотели бы иметь возможность отправлять такое количество данных без ожидания первого подтверждения. Принцип, который здесь работает, часто называется «*держать трубу полной*». Алгоритмы, представленные в следующих двух главах, делают именно это.

Глава 2.5.2. «Скользящее окно»

Рассмотрим сценарий, в котором произведение задержки на пропускную способность канала составляет 8 КБ, а размер кадров составляет 1 КБ. Мы хотели бы, чтобы отправитель был готов отправить девятый кадр практически в тот же момент, когда придет подтверждение для первого кадра. Алгоритм, который позволяет нам сделать это, называется «*скользящее окно*», и его иллюстративная временная шкала приведена на рис. 2.16.

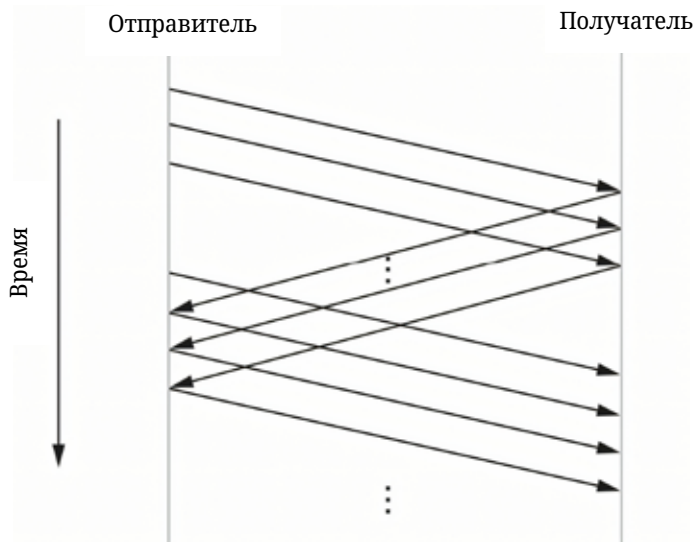


Рисунок 2.16. Временная шкала для алгоритма «скользящего окна».

Алгоритм «скользящего окна»

Алгоритм «скользящего (или раздвижного) окна» работает следующим образом. Во-первых, отправитель присваивает каждому кадру *порядковый номер*, обозначаемый как SeqNum. Пока давайте проигнорируем тот факт, что SeqNum реализован в заголовке конечного размера, и вместо этого предположим, что он может бесконечно расти. Отправитель хранит три переменные: *размер окна отправки*, обозначаемый как SWS (send window size), который дает верхнюю границу на количество полученных (неподтвержденных) кадров, которые отправитель может передать; LAR (last acknowledgement received) обозначает порядковый номер *последнего полученного подтверждения*; и LFS (last frame sent) обозначает порядковый номер *последнего отправленного кадра*. Отправитель также поддерживает следующее инвариантное условие:

$$\text{LFS} - \text{LAR} \leq \text{SWS}$$

Эта ситуация проиллюстрирована на рис. 2.17.

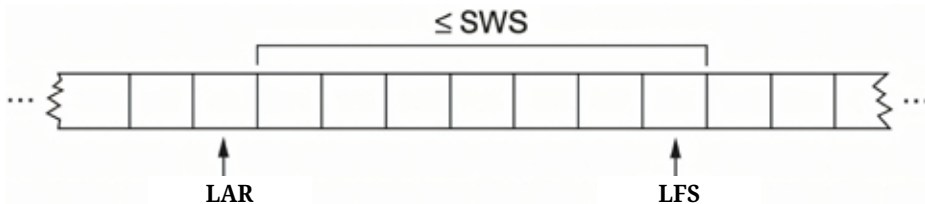


Рисунок 2.17. «Скользящее окно» на отправителе.

Когда приходит подтверждение, отправитель перемещает LAR вправо, что позволяет ему передать еще один кадр. Также отправитель связывает таймер с каждым переданным кадром и повторно передает кадр, если таймер истекает до получения ACK. Обратите внимание, что отправитель должен быть готов буферизовать до SWS кадров, так как он должен быть готов повторно передать их до тех пор, пока они не будут подтверждены.

Получатель хранит следующие три переменные: *размер окна приема*, обозначаемый как RWS (receive window size), который дает верхнюю границу на количество кадров, принятых вне очереди, которые получатель готов принять; LAF (largest acceptable frame) обозначает порядковый номер *самого большого приемлемого кадра*; и LFR (last frame received) обозначает порядковый номер *последнего полученного кадра*. Получатель также поддерживает следующее инвариантное условие:

$$LAF - LFR \leq RWS$$

Эта ситуация проиллюстрирована на рис. 2.18.



Рисунок 2.18. «Скользящее окно» на приемник.

Когда кадр с порядковым номером SeqNum прибывает, приемник выполняет следующие действия. Если $SeqNum \leq LFR$ или $SeqNum > LAF$, то кадр находится за пределами окна приемника и отбрасывается. Если $LFR < SeqNum \leq LAF$, то кадр находится в пределах окна приемника и принимается. Теперь приемник должен решить, отправлять ли подтверждение (ACK). Пусть SeqNumToAck обозначает наибольший порядковый номер, который еще не был подтвержден, так что все кадры с порядковыми номерами, меньшими или равными SeqNumToAck, были получены. Приемник подтверждает получение SeqNumToAck, даже если кадры с более высокими номерами были получены. Это подтверждение называется *кумулятивным*. Затем он устанавливает $LFR = SeqNumToAck$ и корректирует $LAF = LFR + RWS$.

Например, допустим, что $LFR = 5$ (то есть последнее подтверждение, отправленное приемником, было для порядкового номера 5), и $RWS = 4$. Это означает, что $LAF = 9$. Если кадры 7 и 8 придут, они будут буферизованы, так как находятся в пределах окна приемника. Однако подтверждение (ACK) отправлять не нужно, так как кадр 6 еще не прибыл. Кадры 7 и 8 считаются прибывшими вне очереди. (Технически приемник может повторно отправить подтверждение для кадра 5, когда придут кадры 7 и 8.) Если

кадр 6 все же прибывает — возможно, он задержался, потому что был потерян первый раз и должен был быть повторно передан, или он просто был задержан, — приемник подтверждает кадр 8, сдвигает LFR до 8 и устанавливает LAF до 12.¹ Если кадр 6 действительно был потерян, то на стороне отправителя произойдет тайм-аут, вызывая повторную передачу кадра 6.

Мы наблюдаем, что, когда происходит тайм-аут, количество данных в транзите уменьшается, так как отправитель не может продвинуть свое окно, пока кадр 6 не будет подтвержден. Это означает, что при возникновении потерь пакетов эта схема больше не поддерживает полное заполнение канала. Чем больше времени занимает обнаружение потери пакета, тем серьезнее становится эта проблема.

Обратите внимание, что в этом примере приемник мог бы отправить *отрицательное подтверждение* (negative acknowledgement, NAK) для кадра 6, как только прибыл кадр 7. Однако это излишне, так как механизм тайм-аута отправителя достаточен для обнаружения этой ситуации, а отправка NAK добавляет дополнительную сложность приемнику. Также, как уже упоминалось, было бы закономерно отправлять дополнительные подтверждения кадра 5, когда прибывают кадры 7 и 8; в некоторых случаях отправитель может использовать дублирующиеся ACK как подсказку, что кадр был потерян. Оба подхода помогают улучшить производительность, позволяя заранее обнаруживать потерю пакетов.

Еще одной вариацией этой схемы было бы использование *селективных подтверждений*. То есть приемник мог бы подтверждать именно те кадры, которые он получил, а не только кадры с наивысшим полученным номером по порядку. Таким образом, в приведенном выше примере приемник мог бы подтвердить получение кадров 7 и 8. Предоставление большего количества информации отправителю потенциально делает его задачу поддерживать заполнение канала проще, но усложняет реализацию.

Размер окна отправки (SWS) выбирается в зависимости от того, сколько кадров мы хотим иметь в передаче на линии в данный момент; SWS легко вычислить для данного произведения задержки на пропускную способность. С другой стороны, приемник может установить RWS на любое значение. Два распространенных варианта настроек — это $RWS = 1$, что означает, что приемник не будет буферизовать кадры, которые приходят вне очереди, и $RWS = SWS$, что означает, что приемник может буферизовать все кадры, которые отправляет отправитель. Нет смысла устанавливать $RWS > SWS$, так как невозможно, чтобы более чем SWS кадров пришли вне очереди.

Конечные порядковые числа и «скользящее окно»

Теперь вернемся к упрощению, которое мы ввели в алгоритм, — нашему предположению, что порядковые номера могут бесконечно увеличиваться. На практике, конечно, порядковый номер кадра указан в заголовке поля фиксированного размера. Например, 3-битное поле означает, что есть восемь возможных порядковых номеров, от 0 до 7. Это делает необходимым повторное использование порядковых номеров, или, другими словами, порядковые номера зацикливаются. Это вводит проблему установления различия между разными инкарнациями одних и тех же порядковых номеров, что подразумевает, что количество возможных порядковых номеров должно быть больше, чем количество допустимых кадров в ожидании. Например, алгоритм «стоп-и-жди» позволял одному кадру находиться в ожидании и имел два различных порядковых номера.

Предположим, у нас есть одно дополнительное число в нашем пространстве порядковых номеров по сравнению с потенциально ожидаемыми кадрами; то есть $SWS \leq \text{MaxSeqNum} - 1$, где MaxSeqNum — это количество доступных порядковых номеров. Является ли это достаточным? Ответ зависит от RWS. Если $RWS = 1$, то $\text{MaxSeqNum} \geq SWS + 1$ будет достаточным. Если RWS равно SWS, то наличие MaxSeqNum , только на одно большее,

¹ Хотя маловероятно, что пакет может задержаться или прийти не по порядку на канале «точка-точка», этот же алгоритм используется на многоходовых соединениях, где такие задержки возможны.

чем размер окна отправки, недостаточно. Чтобы это понять, рассмотрим ситуацию, когда у нас есть восемь порядковых номеров от 0 до 7, и $SWS = RWS = 7$. Предположим, что отправитель передает кадры с 0 по 6, они успешно принимаются, но АСК-ответы теряются. Приемник теперь ожидает кадры с номером 7 и с 0 по 5, но отправитель превышает время ожидания и отправляет кадры с 0 по 6. К сожалению, приемник ожидает вторую инкарнацию кадров с 0 по 5, но получает первую инкарнацию этих кадров. Это именно та ситуация, которой мы хотели избежать.

Оказывается, что при $RWS = SWS$ размер окна отправки может быть не более чем в два раза меньше, чем количество доступных номеров последовательности, или, выражаясь более точно,

$$SWS < (MaxSeqNum + 1) / 2$$

Интуитивно это означает, что протокол скользящего окна чередуется между двумя половинами пространства порядковых номеров, так же как «стоп-и-жди» чередуется между порядковыми номерами 0 и 1. Единственное различие в том, что он непрерывно скользит между двумя половинами, а не чередуется дискретно.

Обратите внимание, что это правило специфично для ситуации, когда $RWS = SWS$. Мы оставляем это как упражнение для определения более общего правила, которое работает для произвольных значений RWS и SWS . Также обратите внимание, что соотношение между размером окна и пространством порядковых номеров зависит от предположения, которое настолько очевидно, что его легко упустить из виду, а именно что кадры не переставляются во время передачи. Это не может произойти на прямом соединении «точка-точка», так как нет способа для одного кадра обогнать другой во время передачи. Однако мы увидим, что алгоритм «скользящего окна» используется в других средах, и нам придется разработать другое правило.

Реализация «скользящего окна»

Следующие процедуры иллюстрируют, как можно реализовать отправку и прием сторон алгоритма «скользящего окна». Процедуры взяты из рабочего протокола, названного соответственно протоколом «скользящего окна» (sliding window protocol, SWP). Чтобы не беспокоиться о соседних протоколах в графике протокола, мы обозначим протокол, находящийся выше SWP, как протокол высокого уровня (high-level protocol, HLP), а протокол, находящийся ниже SWP, как протокол канального уровня (link-level protocol, LLP).

Мы начнем с определения пары структур данных. Во-первых, заголовок кадра очень простой: он содержит порядковый номер (SeqNum) и номер подтверждения (AckNum). Он также содержит поле Flags, которое указывает, является ли кадр АСК либо содержит данные.

```
typedef u_char SwpSeqno;
typedef struct {
    SwpSeqno SeqNum; /* порядковый номер этого кадра */
    SwpSeqno AckNum; /* подтверждение принятого кадра */
    u_char Flags; /* до 8 бит флагов */
} SwpHdr;
```

Далее состояние алгоритма «скользящего окна» имеет следующую структуру. Для стороны отправки этого протокола состояние включает переменные LAR и LFS, как описано ранее в этой главе, а также очередь, которая содержит кадры, которые были переданы, но еще не подтверждены (sendQ). Состояние отправки также включает счетный семафор, называемый sendWindowNotFull. Мы увидим, как это используется, ниже, но в общем семафор — это примитив синхронизации, который поддерживает операции semWait и semSignal. Каждый вызов semSignal увеличивает семафор на 1, а каждый вызов semWait уменьшает его на 1, при этом вызывающий процесс блокируется (приостанавливается), если уменьшение семафора приводит к его значению меньше 0. Процесс, который блоки-

руется во время вызова `semWait`, будет разрешен к возобновлению, как только будет выполнено достаточное количество операций `semSignal`, чтобы повысить значение семафора выше 0.

Для стороны приема протокола состояние включает переменную NFE. Это *следующий ожидаемый кадр*, кадр с порядковым номером на один больше последнего принятого кадра (LFR), описанного ранее в этой главе. Также есть очередь, которая содержит кадры, полученные вне очереди (`recvQ`). Наконец, хотя это и не показано, размеры окон отправителя и приемника определяются константами SWS и RWS соответственно.

```
typedef struct {
    /* состояние на стороне отправителя: */
    SwpSeqno LAR; /* порядковый номер последнего полученного ACK */
    SwpSeqno LFS; /* последний отправленный кадр */
    Semaphore sendWindowNotFull;
    SwpHdr hdr; /* предварительно инициализированный заголовок */
    struct sendQ_slot {
        Event timeout; /* событие, связанное с тайм-аутом отправки */
        Msg msg;
    } sendQ[SWS];

    /* состояние на стороне получателя: */
    SwpSeqno NFE; /* порядковый номер следующего ожидаемого кадра */
    struct recvQ_slot {
        int received; /* является ли сообщение допустимым? */
        Msg msg;
    } recvQ[RWS];
} SwpState;
```

Сторона отправки SWP реализуется процедурой `sendSWP`. Эта процедура довольно проста. Сначала `semWait` вызывает блокировку этого процесса на семафоре до тех пор, пока не будет разрешено отправить еще один кадр. Как только процессу разрешено продолжить, `sendSWP` устанавливает порядковый номер в заголовке кадра, сохраняет копию кадра в очереди передачи (`sendQ`), планирует событие тайм-аута на случай, если кадр не будет подтвержден, и отправляет кадр на следующий нижний уровень протокола, который мы обозначаем как LINK.

Одна деталь, на которую стоит обратить внимание, это вызов `store_swp_hdr` непосредственно перед вызовом `msgAddHdr`. Эта процедура переводит структуру C, содержащую заголовок SWP (`state->hdr`), в строку байтов, которая может быть безопасно прикреплена к началу сообщения (`hbuf`). Эта процедура (не показана) должна перевести каждое целое число в заголовке в сетевой порядок байтов и удалить любое заполнение, которое компилятор добавил к структуре C. Вопрос порядка байтов является нетривиальной задачей, но пока достаточно предположить, что эта процедура размещает самый значительный бит многозначного целого числа в байт с наивысшим адресом.

Еще одна сложность в этой процедуре — использование `semWait` и семафора `sendWindowNotFull`. `sendWindowNotFull` инициализируется размером окна скользящего отправителя, SWS (эта инициализация не показана). Каждый раз, когда отправитель передает кадр, операция `semWait` уменьшает этот счет и блокирует отправителя, если счет становится равным 0. Каждый раз, когда принимается ACK, операция `semSignal`, вызываемая в `deliverSWP` (см. ниже), увеличивает этот счет, таким образом разблокируя любого ожидающего отправителя.

```
static int
sendSWP(SwpState *state, Msg *frame)
{
```

```

struct sendQ_slot *slot;
hbuf[HLEN];
/* ожидание открытия окна отправки */
semWait(&state->sendWindowNotFull);
state->hdr.SeqNum = ++state->LFS;
slot = &state->sendQ[state->hdr.SeqNum % SWS];
store_swp_hdr(state->hdr, hbuf);
msgAddHdr(frame, hbuf, HLEN);
msgSaveCopy(&slot->msg, frame);
slot->timeout = evSchedule(swpTimeout, slot, SWP_SEND_TIMEOUT);
return send(LINK, frame);
}

```

Прежде чем перейти к приемной стороне SWP, нам нужно устранить кажущееся несоответствие. С одной стороны, мы говорили, что протокол верхнего уровня вызывает услуги протокола нижнего уровня, вызывая операцию отправки, поэтому мы ожидали бы, что протокол, который хочет отправить сообщение через SWP, вызовет `send(SWP, packet)`. С другой стороны, процедура, которая реализует операцию отправки SWP, называется `sendSWP`, и ее первый аргумент — переменная состояния (`SwpState`). В чем дело? Ответ заключается в том, что операционная система предоставляет код-связку, который переводит общий вызов `send` в протокол-специфичный вызов `sendSWP`. Этот код-связка сопоставляет первый аргумент `send` (магическую переменную протокола SWP) с указателем функции на `sendSWP` и указателем на состояние протокола, которое необходимо SWP для выполнения своей работы. Причина, по которой протокол верхнего уровня косвенно вызывает протокол-специфичную функцию через общий вызов функции, заключается в том, что мы хотим ограничить количество информации, закодированной в протоколе верхнего уровня о протоколе нижнего уровня. Это облегчает изменение конфигурации графа протокола в будущем.

Теперь переходим к протокол-специфичной реализации операции `deliver SWP`, которая приведена в процедуре `deliverSWP`. Эта процедура на самом деле обрабатывает два разных типа входящих сообщений: ACK для ранее отправленных кадров с этого узла и кадры данных, прибывающие на этот узел. В определенном смысле половина этой процедуры, связанная с ACK, является аналогом отправляющей стороны алгоритма, приведенного в `sendSWP`. Решение о том, является ли входящее сообщение ACK или кадром данных, принимается путем проверки поля `Flags` в заголовке. Обратите внимание, что эта конкретная реализация не поддерживает объединение ACK с кадрами данных (`piggybacking`).

Если входящий кадр является ACK, `deliverSWP` просто находит слот в очереди передачи (`sendQ`), который соответствует этому ACK, отменяет событие тайм-аута и освобождает кадр, сохраненный в этом слоте. Эта работа выполняется в цикле, поскольку ACK может быть кумулятивным. Единственное, на что стоит обратить внимание в этом случае, — это вызов подпрограммы `swpInWindow`. Эта подпрограмма, приведенная ниже, обеспечивает, чтобы номер последовательности кадра, подтверждаемого ACK, находился в диапазоне номеров, которые отправитель в настоящее время ожидает получить.

Если входящий кадр содержит данные, `deliverSWP` сначала вызывает `msgStripHdr` и `load_swp_hdr` для извлечения заголовка из кадра. Подпрограмма `load_swp_hdr` является аналогом `store_swp_hdr`, обсуждавшейся ранее; она переводит строку байтов в структуру данных на языке C, содержащую заголовок SWP. Затем `deliverSWP` вызывает `swpInWindow`, чтобы убедиться, что номер последовательности кадра находится в ожидаемом диапазоне. Если это так, процедура проходит по набору полученных подряд кадров и передает их протоколу верхнего уровня, вызывая процедуру `deliverHLP`. Она также отправляет кумулятивный ACK обратно отправителю, но делает это, проходя по очереди приема (она не использует переменную `SeqNumToAck`, упомянутую в описании выше).

```

static int
deliverSWP(SwpState state, Msg *frame)
{
    SwpHdr hdr;
    char *hbuf;
    hbuf = msgStripHdr(frame, HLEN);
    load_swp_hdr(&hdr, hbuf);

    if (hdr->Flags & FLAG_ACK_VALID)
    {
        /* получено подтверждение - выполняем сторону ОТПРАВИТЕЛЯ */
        if (swpInWindow(hdr.AckNum, state->LAR + 1, state->LFS))
        {
            do
            {
                struct sendQ_slot *slot;
                slot = &state->sendQ[++state->LAR % SWS];
                evCancel(slot->timeout);
                msgDestroy(&slot->msg);
                semSignal(&state->sendWindowNotFull);
            } while (state->LAR != hdr.AckNum);
        }
    }
    if (hdr.Flags & FLAG_HAS_DATA)
    {
        struct recvQ_slot *slot;
        /* получен пакет данных - выполняем сторону ПОЛУЧАТЕЛЯ */
        slot = &state->recvQ[hdr.SeqNum % RWS];

        if (!swpInWindow(hdr.SeqNum, state->NFE, state->NFE + RWS - 1))
        {
            /* отбросить сообщение */
            return SUCCESS;
        }

        msgSaveCopy(&slot->msg, frame);
        slot->received = TRUE;

        if (hdr.SeqNum == state->NFE)
        {
            Msg m;
            while (slot->received)
            {
                deliver(HLP, &slot->msg);
                msgDestroy(&slot->msg);
                slot->received = FALSE;
                slot = &state->recvQ[++state->NFE % RWS];
            }
            /* отправить ACK: */
            prepare_ack(&m, state->NFE - 1);
            send(LINK, &m);
            msgDestroy(&m);
        }
    }
}

```

```

    }

    return SUCCESS;
}

```

Наконец, `swpInWindow` — это простая подпрограмма, которая проверяет, попадает ли данный номер последовательности в диапазон между минимальным и максимальным номером последовательности.

```

static bool
swpInWindow(SwpSeqno seqno, SwpSeqno min, SwpSeqno max)
{
    SwpSeqno pos, maxpos;
    pos = seqno - min;          /* pos *должна* быть в диапазоне [0..MAX) */
    maxpos = max - min + 1;     /* maxpos находится в диапазоне [0..MAX) */
    return pos < maxpos;
}

```

Порядок кадров и управление потоком

Протокол «скользящего окна» — пожалуй, самый известный алгоритм в компьютерных сетях. Однако легко запутаться в его использовании, так как он может выполнять три разные роли. Первая роль — это та, на которой мы сосредоточились в данной главе — надежная доставка кадров через ненадежное соединение. (В общем алгоритм можно использовать для надежной доставки сообщений через ненадежную сеть.) Это основная функция алгоритма.

Вторая роль, которую может выполнять алгоритм «скользящего окна», — это сохранение порядка, в котором передаются кадры. Это легко сделать на приемной стороне — так как каждый кадр имеет номер последовательности, приемник просто следит за тем, чтобы не передавать кадр на следующий, более высокий уровень протокола, пока он не передаст все кадры с меньшими номерами последовательности. То есть приемник буферизует (то есть не передает) кадры, которые пришли не по порядку. Версия алгоритма скользящего окна, описанная в этой главе, действительно сохраняет порядок кадров, хотя мы могли бы представить вариант, при котором приемник передает кадры следующему протоколу, не ожидая доставки всех предыдущих кадров. Вопрос, который мы должны задать себе, заключается в том, действительно ли нам нужно, чтобы протокол «скользящего окна» сохранял порядок кадров на уровне канала, или же эта функциональность должна быть реализована протоколом более высокого уровня в стеке.

Третья роль, которую иногда играет алгоритм «скользящего окна», — это поддержка управления *потоком* — механизма обратной связи, с помощью которого приемник может сдерживать отправителя. Такой механизм используется для предотвращения перегрузки приемника отправителем — то есть для предотвращения передачи большого объема данных, чем приемник способен обработать. Это обычно достигается путем дополнения протокола «скользящего окна» так, чтобы приемник не только подтверждал полученные кадры, но и информировал отправителя о количестве кадров, которые он готов принять. Количество кадров, которые приемник способен принять, соответствует объему свободного буфера. Как и в случае с упорядоченной доставкой, нам нужно убедиться, что управление потоком необходимо на уровне канала, прежде чем включать его в протокол «скользящего окна».

Основные выводы

Одной из важных концепций, которую следует вынести из этого обсуждения, является принцип проектирования системы, который мы называем *разделением задач*.

То есть нужно быть осторожным, чтобы различать разные функции, которые иногда

объединяются в одном механизме, и нужно убедиться, что каждая функция необходима и поддерживается наиболее эффективным способом. В данном конкретном случае надежная доставка, упорядоченная доставка и управление потоком иногда комбинируются в едином протоколе «скользящего окна», и мы должны задать себе вопрос, допустимо ли это делать на уровне канала.

Глава 2.5.3. Параллельные логические каналы

Протокол передачи данных, использовавшийся в оригинальной ARPANET, представляет интересную альтернативу протоколу «скользящего окна», так как он способен поддерживать заполненность канала, используя простой алгоритм «стоп-и-жди». Важным следствием этого подхода является то, что кадры, отправленные по данному каналу, не сохраняют никакого определенного порядка. Протокол также не предполагает управление потоком.

Идея, лежащая в основе протокола ARPANET, которая называется «параллельные логические каналы», заключается в мультиплексировании нескольких логических каналов на одном соединении «точка-точка» и запуске алгоритма «стоп-и-жди» на каждом из этих логических каналов. Нет никакой связи между кадрами, отправленными по любому из логических каналов, но поскольку разные кадры могут быть отправлены по каждому из нескольких логических каналов, отправитель может поддерживать полную загрузку канала.

Если говорить более точно, отправитель хранит три бита состояния для каждого канала: один логический, указывающий, занят ли канал в данный момент; бит последовательности, который будет использован в следующий раз при отправке кадра по этому логическому каналу; и следующий номер последовательности, ожидаемый для кадра, который прибудет на этот канал. Когда у узла есть кадр для отправки, он использует самый низкий свободный канал, и в противном случае ведет себя так же, как в алгоритме «стоп-и-жди».

На практике ARPANET поддерживала 8 логических каналов для каждого наземного соединения и 16 для каждого спутникового соединения. В случае наземного соединения заголовок каждого кадра включал 3-битный номер канала и 1-битный номер последовательности, что в сумме давало 4 бита. Это точно соответствует количеству битов, которые требуются протоколу «скользящего окна» для поддержки до 8 ожидающих кадров на канале, когда $RWS = SWS$.

Глава 2.6. Сети с множественным доступом

Разработанный в середине 1970-х годов исследователями в Xerox Palo Alto Research Center (PARC), Ethernet в конечном итоге стал доминирующей технологией локальных сетей, опередив множество конкурирующих технологий. Сегодня он в основном конкурирует с беспроводными сетями 802.11, но остается чрезвычайно популярным в университетских сетях и центрах обработки данных. Более общее название технологии, которая лежит в основе Ethernet, это Carrier Sense, Multiple Access with Collision Detect (CSMA/CD).

Как следует из названия CSMA, Ethernet является сетью с множественным доступом, что означает, что набор узлов отправляет и принимает кадры по общей линии связи. Таким образом, Ethernet можно представить как шину, к которой подключены несколько станций. «Обнаружение несущей» в CSMA/CD означает, что все узлы могут различать свободную и занятую линию связи, а «обнаружение коллизий» означает, что узел слушает, когда передает, и, следовательно, может обнаружить, когда кадр, который он передает, столкнулся (коллизировал) с кадром, переданным другим узлом.

Ethernet берет свое начало в ранней пакетной радиосети под названием Aloha, разработанной в Университете Гавайев для поддержки компьютерной связи между Гавайскими островами. Как и в сети Aloha, основная проблема, с которой сталкивается Ethernet, заключается в том, как справедливо и эффективно управлять доступом к общему средству

передачи (в Aloha средством передачи была атмосфера, а в Ethernet изначально был коаксиальный кабель). Основная идея как в сети Aloha, так и в Ethernet заключается в алгоритме, который контролирует, когда каждый узел может передавать данные.

Современные Ethernet-соединения в основном построены по принципу «точка-точка»; то есть они соединяют один хост с *Ethernet-коммутатором* или соединяют коммутаторы между собой. В результате алгоритм «множественного доступа» не используется в современных проводных сетях Ethernet, но его вариант теперь применяется в беспроводных сетях, таких как сети 802.11 (также известные как Wi-Fi). Из-за огромного влияния Ethernet было решено описать его классический алгоритм здесь, а затем объяснить, как он был адаптирован для Wi-Fi, в следующем разделе. Мы также обсудим Ethernet-коммутаторы в другом месте. Пока сосредоточимся на том, как работает одно соединение Ethernet.

Digital Equipment Corporation и Intel Corporation присоединились к Xerox для определения стандарта Ethernet на скорости 10 Мбит/с в 1978 году. Этот стандарт затем стал основой для стандарта IEEE 802.3, который дополнительно определяет более широкий набор физических сред, по которым может работать Ethernet, включая версии на 100 Мбит/с, 1 Гбит/с, 10 Гбит/с, 40 Гбит/с и 100 Гбит/с.

Глава 2.6.1. Физические свойства

Сегменты Ethernet изначально реализовывались с использованием коаксиального кабеля длиной до 500 м. (Современные сети Ethernet используют витые пары медных проводов, обычно определенного типа, известного как «Категория 5», или оптические волокна, и в некоторых случаях могут быть значительно длиннее 500 м.) Этот кабель был похож на тип, используемый для кабельного телевидения. Хосты подключались к сегменту Ethernet путем подсоединения к нему. *Трансивер* (или приемопередатчик), небольшое устройство, непосредственно прикрепленное к точке подключения, определял, когда линия была свободна, и управлял сигналом при передаче хостом. Он также принимал входящие сигналы. Трансивер, в свою очередь, подключался к Ethernet-адаптеру, который был подключен к хосту. Эта конфигурация показана на рис. 2.19.

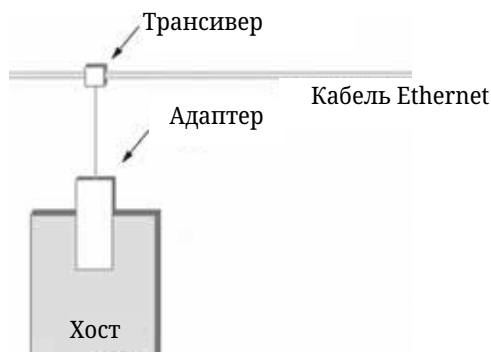


Рисунок 2.19. Приемопередатчик и адаптер Ethernet.

Несколько сегментов Ethernet могут быть соединены друг с другом с помощью *ретрансляторов* (или многопортового варианта ретранслятора, называемого *концентратором*). Ретранслятор — это устройство, которое передает цифровые сигналы, аналогично тому, как усилитель передает аналоговые сигналы; ретрансляторы не понимают битов или кадров. Между любой парой хостов не могло быть размещено более четырех ретрансляторов, а это означало, что классический Ethernet имел общую длину всего 2500 м. Например, использование всего двух ретрансляторов между любой парой хостов поддерживает конфигурацию, подобную той, что показана на рис. 2.20; то есть сегмент, проходящий по центральной части здания, с сегментом на каждом этаже.

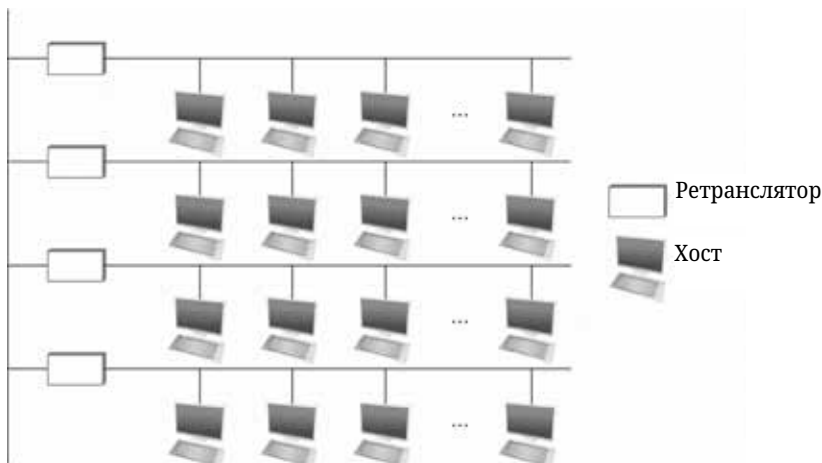


Рисунок 2.20. Ретранслятор Ethernet, соединяющий сегменты для формирования более крупного домена коллизий.

Любой сигнал, размещенный на Ethernet как хост, транслируется по всей сети; то есть сигнал распространяется в обоих направлениях, а ретрансляторы и концентраторы передают сигнал по всем исходящим сегментам. Терминаторы, прикрепленные к концу каждого сегмента, поглощают сигнал и предотвращают его отражение и вмешательство в последующие сигналы. Оригинальные спецификации Ethernet использовали манчестерскую схему кодирования, описанную ранее, тогда как схема кодирования 4В/5В (или аналогичная 8В/10В) используется сегодня на Ethernet с более высокой скоростью.

Важно понимать, что независимо от того, охватывает ли данный Ethernet один сегмент, линейную последовательность сегментов, соединенных ретрансляторами, или несколько сегментов, соединенных в звездообразную конфигурацию, данные, передаваемые любым хостом в этом Ethernet, достигают всех остальных хостов. Это хорошая новость. Плохая новость заключается в том, что все эти хосты конкурируют за доступ к одной и той же линии связи, и, как следствие, считается, что они находятся в одном и том же *домене коллизий*. Множественный доступ в Ethernet заключается в решении проблемы конкуренции за линию связи, которая возникает в зоне коллизий.

Глава 2.6.2. Протокол доступа

Теперь мы обратим внимание на алгоритм, который управляет доступом к общей сети Ethernet. Этот алгоритм обычно называется *контролем доступа к среде Ethernet* (media access control, MAC). Он, как правило, реализуется в аппаратном обеспечении на сетевом адаптере. Мы не будем описывать само аппаратное обеспечение, а сосредоточимся на алгоритме, который оно реализует. Однако сначала мы опишем формат кадра Ethernet и адреса.

Формат кадра

Каждый кадр Ethernet определяется форматом, показанным на рис. 2.21. 64-битная преамбула позволяет приемнику синхронизироваться с сигналом; это последовательность чередующихся 0 и 1. И источник, и получатель идентифицируются с помощью 48-битного адреса. Поле типа пакета служит в качестве ключа для демультиплексирования; оно указывает, какому из возможных многих протоколов верхнего уровня должен быть доставлен этот кадр. Каждый кадр содержит до 1500 байт данных. Ми-

нимально кадр должен содержать не менее 46 байт данных, даже если это означает, что хосту придется дополнить кадр перед передачей. Причина минимального размера кадра заключается в том, что кадр должен быть достаточно длинным, чтобы обнаружить коллизию; об этом мы поговорим ниже. Наконец, каждый кадр включает 32-битный CRC. Как и протокол HDLC, описанный в предыдущем разделе, Ethernet является протоколом кадрирования, ориентированным на биты. Обратите внимание, что с точки зрения хоста кадр Ethernet имеет 14-байтный заголовок: два 6-байтных адреса и 2-байтное поле типа. Передатчик добавляет преамбулу и CRC перед передачей, а приемник удаляет их.



Рисунок 2.21. Формат кадра Ethernet.

Адреса

Каждый хост в сети Ethernet — на самом деле, каждый хост Ethernet в мире — имеет уникальный Ethernet-адрес. Технически адрес принадлежит адаптеру, а не хосту; он обычно зашит в ПЗУ. Ethernet-адреса обычно печатаются в форме, удобной для чтения человеком, как последовательность из шести чисел, разделенных двоеточиями. Каждое число соответствует 1 байту 6-байтного адреса и представлено парой шестнадцатеричных цифр, по одной на каждую из 4-битных половин байта; начальные нули опускаются. Например, 8:0:2b:e4:b1:2 — это представление Ethernet-адреса в удобочитаемой форме:

```
00001000 00000000 00101011 11100100 10110001 00000010
```

Чтобы обеспечить уникальность адреса каждого адаптера, каждому производителю Ethernet-устройств выделяется различный префикс, который должен быть добавлен к адресу каждого создаваемого ими адаптера. Например, компании Advanced Micro Devices был присвоен 24-битный префикс 080020 (или 8:0:20). Каждый производитель затем обеспечивает уникальность производимых им суффиксов адресов.

Каждый кадр, передаваемый по Ethernet, принимается каждым адаптером, подключенным к этому Ethernet. Каждый адаптер распознает те кадры, которые адресованы его адресу, и передает только эти кадры хосту. (Адаптер также может быть запрограммирован на работу в режиме перехвата, в этом случае он передает все полученные кадры хосту, но это не является обычным режимом работы.)

В дополнение к этим *уникастным* адресам Ethernet-адрес, состоящий из всех единиц, рассматривается как *широковещательный* адрес; все адаптеры передают кадры, адресованные широковещательному адресу, хосту. Аналогично адрес, у которого первый бит установлен в 1, но который не является широковещательным, называется *многоадресным*. Определенный хост может запрограммировать свой адаптер на прием некоторых *мультикастных* адресов. Мультикастные адреса используются для отправки сообщений некоторой подгруппе хостов в Ethernet (например, всем файловым серверам). Подводя итог, отметим, что адаптер Ethernet принимает все кадры и обрабатывает:

- Кадры, адресованные его собственному адресу
- Кадры, адресованные широковещательному адресу

- Кадры, адресованные мультикастному адресу, если он был настроен на прием этих адресов
- Все кадры, если он настроен на режим перехвата

Он передает хосту только те кадры, которые принимает.

Алгоритм передатчика

Как мы только что узнали, сторона приемника протокола Ethernet проста; настоящие «мозги» реализованы на стороне отправителя. Алгоритм передатчика определяется следующим образом.

Когда адаптер имеет кадр для отправки и линия простаивает, он немедленно передает кадр; никакого согласования с другими адаптерами не требуется. Верхний предел в 1500 байт в сообщении означает, что адаптер может занимать линию только в течение фиксированного времени.

Когда адаптер имеет кадр для отправки и линия занята, он ждет, пока линия освободится, и затем немедленно передает кадр. (Если быть точнее, все адаптеры ждут 9,6 мкс после окончания передачи одного кадра, прежде чем начать передавать следующий кадр. Это верно как для отправителя первого кадра, так и для тех узлов, которые просматривают, чтобы линия стала свободной.) Ethernet называется *1-постоянным протоколом*, потому что адаптер с кадром для отправки передает с вероятностью 1 всякий раз, когда занятая линия становится свободной. В общем, *p*-постоянный алгоритм передает с вероятностью $0 \leq p \leq 1$ после того, как линия становится свободной, и откладывает передачу с вероятностью $q = 1 - p$. Обоснование выбора $p < 1$ заключается в том, что может быть несколько адаптеров, ожидающих, когда занятая линия станет свободной, и мы не хотим, чтобы все они начали передавать одновременно. Если каждый адаптер передает немедленно с вероятностью, скажем, 33 %, то от одного до трех адаптеров ожидают передачи, и велика вероятность, что только один из них начнет передавать, когда линия станет свободной. Несмотря на это обоснование, адаптер Ethernet всегда начинает передавать немедленно после обнаружения, что сеть стала свободной, и этот способ оказался очень эффективным.

Чтобы завершить рассказ о *p*-постоянных протоколах, приведем пример: в случае, когда $p < 1$, вы можете задать вопрос, как долго отправитель, которому не везет и он проигрывает в подбрасывании монеты (то есть решает отложить передачу), должен ждать перед тем, как он сможет начать передачу. Ответ для сети Aloha, которая изначально разработала такой стиль протокола, заключался в том, чтобы разделить время на дискретные слоты, каждый из которых соответствует времени, необходимому для передачи полного кадра. Всякий раз, когда узел имеет кадр для отправки и находит пустой (свободный) слот, он передает с вероятностью p и откладывает передачу до следующего слота с вероятностью $q = 1 - p$. Если следующий слот также пуст, узел снова решает, передавать или откладывать передачу, с вероятностями p и q соответственно. Если следующий слот не пуст — т. е. какой-то другой узел решил передавать, — тогда узел просто ждет следующего свободного слота, и алгоритм повторяется.

Как мы уже обсуждали, так как в Ethernet нет централизованного управления, возможно, что два (или более) адаптера начнут передачу одновременно, либо потому что оба обнаружили, что линия свободна, либо потому что оба ждали, пока занятая линия станет свободной. Когда это происходит, говорят, что два (или более) кадра сталкиваются в сети. Каждый отправитель, благодаря поддержке Ethernet обнаружения коллизий, может определить, что коллизия происходит. В момент, когда адаптер обнаруживает, что его кадр сталкивается с другим, он сначала убеждается, что передает 32-битную последовательность заглушки, а затем прекращает передачу. Таким обра-

зом, передатчик минимально отправит 96 бит в случае коллизии: 64-битную преамбулу плюс 32-битную последовательность прекращения.

Один из способов, при котором адаптер отправит только 96 бит (что иногда называется ошибочным кадром), — если два хоста находятся близко друг к другу. Если бы два хоста находились дальше друг от друга, им пришлось бы передавать дольше и, следовательно, отправить больше бит до обнаружения коллизии. На самом деле худший сценарий происходит, когда два хоста находятся на противоположных концах Ethernet. Чтобы быть уверенным, что кадр, который он только что отправил, не столкнулся с другим кадром, передатчику может понадобиться отправить до 512 бит. Не случайно каждый кадр Ethernet должен быть не менее 512 бит (64 байта) в длину: 14 байт заголовка плюс 46 байт данных плюс 4 байта CRC.

Почему 512 бит? Ответ связан с другим вопросом, который вы можете задать об Ethernet: почему его длина ограничена только 2500 м? Почему не 10 или 1000 км? Ответ на оба вопроса связан с тем, что чем дальше друг от друга находятся два узла, тем дольше требуется кадру, отправленному одним, чтобы достичь другого, и сеть уязвима для коллизий в течение этого времени.

Рис. 2.22 иллюстрирует худший сценарий, когда хосты А и В находятся на противоположных концах сети. Предположим, что хост А начинает передавать кадр в момент времени t , как показано на рисунке (а). Кадру требуется одна задержка на канале (обозначим ее d), чтобы достичь хоста В. Таким образом, первый бит кадра А достигает В в момент времени $t + d$, как показано на рисунке (б). Предположим, за мгновение до прибытия кадра А (то есть В все еще видит линию свободной) хост В начинает передавать свой собственный кадр. Кадр В немедленно столкнется с кадром А, и эта коллизия будет обнаружена хостом В (с). Хост В отправит 32-битную последовательность прекращения, как описано выше. (Кадр В будет ошибочным.) К сожалению, хост А не узнает, что произошла коллизия, пока кадр В не достигнет его, что произойдет через одну задержку на канале, в момент времени $t + 2 \times d$, как показано на рисунке (д). Хост А должен продолжать передавать до этого времени, чтобы обнаружить коллизия. Другими словами, хост А должен передавать в течение $2 \times d$, чтобы быть уверенным, что он обнаружит все возможные коллизии. Учитывая, что максимальная конфигурация Ethernet составляет 2500 м, и что может быть до четырех повторителей между двумя хостами, задержка в два раза была определена как 51,2 мкс, что на Ethernet со скоростью 10 Мбит/с соответствует 512 битам. Другой способ взглянуть на эту ситуацию заключается в том, что мы должны ограничить максимальную задержку Ethernet до достаточно малого значения (например, 51,2 мкс), чтобы алгоритм доступа работал; следовательно, максимальная длина Ethernet должна быть порядка 2500 м.

После того как адаптер обнаружил коллизия и прекратил передачу, он ждет определенное время и пытается снова. Каждый раз, когда он пытается передать, но терпит неудачу, адаптер удваивает время ожидания перед следующей попыткой. Эта стратегия удвоения интервала задержки между каждой попыткой повторной передачи называется общей техникой *экспоненциального отката*. Точнее говоря, адаптер сначала задерживается на 0 или 51,2 мкс, выбирая это значение случайным образом. Если эта попытка не удалась, он затем ждет 0, 51,2, 102,4 или 153,6 мкс (выбирая случайным образом) перед следующей попыткой; это $k \times 51,2$ для $k = 0..3$. После третьей коллизии он ждет $k \times 51,2$ для $k = 0, 2^3 - 1$, опять же выбирая случайным образом. В общем, алгоритм случайным образом выбирает k между 0 и $2^3 - 1$ и ждет $k \times 51,2$ мкс, где n — количество испытанных коллизий. Адаптер сдается после определенного числа попыток и сообщает об ошибке передачи хосту. Адаптеры обычно повторяют попытки до 16 раз, хотя алгоритм отката ограничивает n в вышеупомянутой формуле.

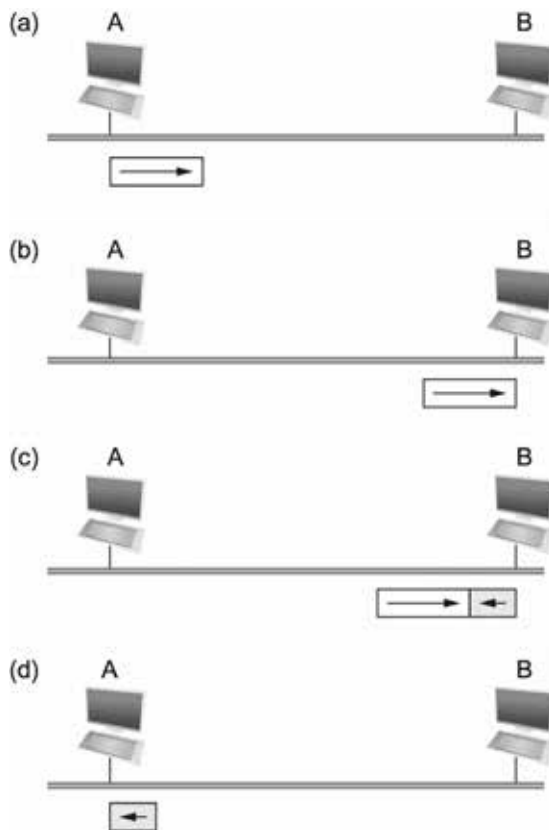


Рисунок 2.22. Наихудший сценарий: (a) A посылает кадр в момент времени t ;
 (b) кадр A поступает в B в момент времени $t+d$;
 (c) B начинает передачу в момент времени $t+d$ и сталкивается с кадром A;
 (d) раунт-кадр B (32-битный) поступает в A в момент времени $t+2 \times d$.

Глава 2.6.3. Долговечность Ethernet

Ethernet остается доминирующей технологией локальных сетей более 30 лет. Сегодня его обычно используют в конфигурации «точка-точка», а не подключают к коаксиальному кабелю. Часто он работает на скоростях 1 или 10 Гбит/с, а не 10 Мбит/с, и поддерживает огромные пакеты (джамбо-пакеты) с данными до 9000 байт вместо 1500 байт. Но при этом он остается совместимым с оригинальным стандартом. Эти данные наводят нас на обсуждение, почему Ethernet оказался таким успешным, чтобы мы могли понять, какие свойства должны быть у любой технологии, которая стремится его заменить.

Во-первых, Ethernet чрезвычайно прост в администрировании и обслуживании: нет необходимости в маршрутизации или обновлении таблиц конфигурации, и легко добавить новый хост в сеть. Трудно представить более простую сеть для администрирования. Во-вторых, он недорогой: кабель/оптоволокно относительно дешевы, и единственные дополнительные затраты — это сетевой адаптер на каждом хосте. Ethernet стал глубоко укоренившейся технологией по этим причинам, и любая подходящая для его замены коммутируемая технология требует дополнительных инвестиций в инфраструктуру (коммутаторы), помимо стоимости каждого адаптера. Коммутируемый вариант Ethernet в конечном итоге действительно заменил многоадресный Ethernet, но это произошло в основном потому, что он мог внедряться постепенно — с некоторыми хостами, подклю-

ченными по схеме «точка-точка» к коммутаторам, в то время как другие оставались подключенными к коаксиальному кабелю и подключенными к ретрансляторам или концентраторам, при этом сохраняя простоту управления сетью.

Глава 2.7. Беспроводные сети

Беспроводные технологии отличаются от проводных связей по некоторым важным аспектам, но при этом имеют много общих свойств. Как и в проводных связях, вопросы ошибок битов имеют большое значение, обычно даже больше из-за непредсказуемого шума в большинстве беспроводных соединений. Кадрирование и надежность также необходимо учитывать. В отличие от проводных связей, для беспроводных сетей важен вопрос питания, особенно потому, что беспроводные соединения часто используются малыми мобильными устройствами (такими как телефоны и датчики), которые имеют ограниченный доступ к источникам энергии (например, небольшой аккумулятор). Кроме того, нельзя передавать радиосигналы с произвольно высокой мощностью — существуют опасения по поводу помех другим устройствам, и обычно существуют нормативы по уровню мощности, которую устройство может излучать на любой данной частоте.

Беспроводные среды также по своей природе являются многоадресными; сложно направить радиосигнал только одному получателю или избежать приема радиосигналов от любого передатчика с достаточной мощностью в вашем окружении. Следовательно, управление доступом к среде является центральным вопросом для беспроводных соединений. И поскольку сложно контролировать, кто получает ваш сигнал, когда вы передаете его по воздуху, также могут возникнуть вопросы прослушивания.

Существует множество различных беспроводных технологий, каждая из которых допускает различные компромиссы в различных аспектах. Один из простых способов классифицировать различные технологии — по скоростям передачи данных и расстоянию между узлами, которые могут связываться. Другие важные различия включают, какую часть электромагнитного спектра они используют (включая необходимость лицензии) и сколько энергии они потребляют. В этом разделе мы обсудим две известные беспроводные технологии: Wi-Fi (более формально известный как 802.11) и Bluetooth. Далее обсуждаются сотовые сети в контексте услуг доступа ISP. Таблица 2.3 дает обзор этих технологий и их сравнительную характеристику.

Таблица 2.3.
Обзор ведущих беспроводных технологий.

	Bluetooth (802.15.1)	Wi-Fi (802.11)	Сотовая связь 4G
Типичная длина канала связи	10 м	100 м	Десятки километров
Типичная скорость передачи данных	2 Мбит/с (общий доступ)	150–450 Мбит/с	1–5 Мбит/с
Типичное использование	Подключение периферийного устройства к компьютеру	Подключение компьютера к проводной базе	Подключение телефона к проводной вышке
Аналогия проводной технологии	USB	Ethernet	PON

Вы можете помнить, что под пропускной способностью иногда понимается ширина полосы частот в герцах, а иногда — скорость передачи данных по каналу. Поскольку оба

этих понятия встречаются в обсуждениях беспроводных сетей, здесь мы будем использовать термин «пропускная способность» в его строгом смысле — ширина полосы частот, а термин «*скорость передачи данных*» будем использовать для описания количества битов в секунду, которые могут быть отправлены по каналу, как в табл. 2.3.

Глава 2.7.1. Основные проблемы

Поскольку все беспроводные каналы связи используют одно и то же средство передачи, задача состоит в том, чтобы эффективно разделить это средство, не создавая чрезмерных помех друг для друга. Большая часть этого разделения осуществляется путем разделения по частоте и пространству. Исключительное использование определенной частоты в определенной географической области может быть предоставлено отдельной организации, такой как корпорация. Возможно ограничить область покрытия электромагнитного сигнала, поскольку такие сигналы *ослабевают* с увеличением расстояния от их источника. Чтобы уменьшить область покрытия сигнала, уменьшают мощность передатчика.

Эти распределения обычно определяются государственными органами, такими как Федеральная комиссия по связи (Federal Communication Commission, FCC) в США. Конкретные полосы частот (диапазоны) выделяются для определенных применений. Некоторые полосы зарезервированы для использования государством. Другие полосы зарезервированы для таких целей, как АМ-радио, FM-радио, телевидение, спутниковая связь и сотовые телефоны. Конкретные частоты в этих диапазонах затем лицензируются отдельным организациям для использования в определенных географических областях. Наконец, несколько диапазонов частот выделяются для использования без лицензии — диапазоны, в которых лицензия не требуется.

Устройства, использующие частоты без лицензии, все равно подлежат определенным ограничениям, чтобы обеспечить эффективное разделение спектра. Самое важное из этих ограничений — это ограничение мощности передачи. Это ограничивает диапазон сигнала, делая его менее вероятным для создания помех другому сигналу. Например, беспроводной (мобильный) телефон (распространенное нелицензированное устройство) может иметь радиус действия около 30 метров.

Одной из идей, часто встречающихся при разделении спектра между множеством устройств и приложений, является метод расширенного спектра. Идея расширенного спектра заключается в распространении сигнала на более широкий диапазон частот, чтобы минимизировать влияние помех от других устройств. (Расширенный спектр изначально был разработан для военного использования, потому что «другие устройства» часто пытались заглушить сигнал.) Например, *скачкообразная перестройка частоты* — это метод расширенного спектра, который предполагает передачу сигнала на случайной последовательности частот; то есть сначала на одной частоте, затем на второй, затем на третьей и так далее. Последовательность частот не является по-настоящему случайной, а вычисляется алгоритмически с помощью генератора псевдослучайных чисел. Приемник использует тот же алгоритм, что и передатчик, и инициализирует его тем же способом; таким образом, он может менять частоты синхронно с передатчиком, чтобы правильно принимать кадр. Эта схема уменьшает помехи, делая маловероятным использование двумя сигналами одной и той же частоты более чем для редких отдельных битов.

Вторая практика расширенного спектра, называемая *прямой последовательностью*, добавляет избыточность для большей устойчивости к помехам. Каждый бит данных представляется несколькими битами в передаваемом сигнале, так что если некоторые из переданных битов повреждаются из-за помех, избыточность позволяет восстановить исходный бит. Для каждого бита, который отправитель хочет передать, он на самом деле отправляет результат операции исключающего ИЛИ (XOR) этого бита и n случайных битов. Как и в случае с частотным скачкообразным спектром, после-

довательность случайных битов генерируется псевдослучайным генератором чисел, известным как отправителю, так и приемнику. Передаваемые значения, известные как *n*-битный код чипирования, распространяют сигнал по полосе частот, которая в *n* раз шире, чем полоса, требуемая для обычного кадра. На рис. 2.23 приведен пример 4-битной последовательности чипирования.

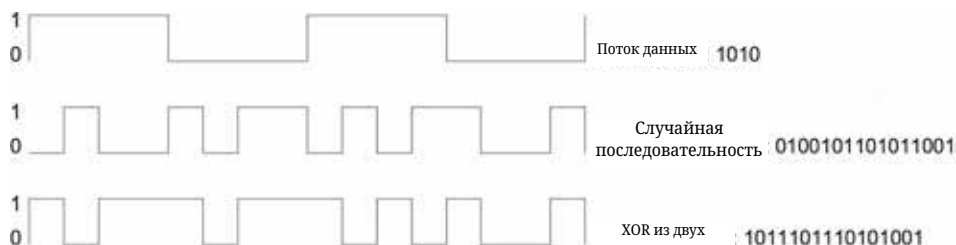


Рисунок 2.23. Пример 4-битной последовательности чипирования.

Разные части электромагнитного спектра обладают разными свойствами, что делает одни из них более подходящими для связи, а другие менее. Например, некоторые из них могут проникать сквозь здания, а некоторые — нет. Правительства регулируют только основную часть спектра, используемую для связи: диапазоны радиоволн и микроволн. По мере роста спроса на основные диапазоны спектра возникает большой интерес к спектру, который становится доступным по мере того, как аналоговое телевидение уступает место цифровому.

На многих беспроводных сетях сегодня мы наблюдаем два различных класса конечных точек. Одна конечная точка, иногда описываемая как *базовая станция*, обычно не имеет мобильности, но имеет проводное (или, по крайней мере, высокоскоростное) соединение с Интернетом или другими сетями, как показано на рис. 2.24. Узел на другом конце канала — показанный здесь как клиентский узел — часто мобильный и полагается на свое соединение с базовой станцией для всей своей связи с другими узлами.

Обратите внимание, что на рис. 2.24 мы использовали волнистую пару линий для обозначения беспроводного «соединения», существующего между двумя устройствами (например, между базовой станцией и одним из ее клиентских узлов). Одним из интересных аспектов беспроводной связи является то, что она естественным образом поддерживает связь «точка-мультиточка», потому что радиоволны, посылаемые одним устройством, могут одновременно приниматься многими устройствами. Однако часто полезно создать абстракцию соединения «точка-точка» для протоколов более высоких уровней, и мы увидим примеры того, как это работает, далее.

Заметьте, что на рис. 2.24 связь между узлами, которые не находятся в базе (не клиентскими), осуществляется через базовую станцию. Это несмотря на то, что радиоволны, излучаемые одним клиентским узлом, могут быть приняты другими клиентскими узлами — общая модель базовой станции не позволяет прямую связь между клиентскими узлами.

Такая топология подразумевает три качественно разных уровня мобильности. Первый уровень — отсутствие мобильности, например, когда приемник должен находиться в фиксированном месте, чтобы получить направленную передачу от базовой станции. Второй уровень — мобильность в пределах диапазона базовой станции, как в случае с Bluetooth. Третий уровень — мобильность между базовыми станциями, как в случае с сотовыми телефонами и Wi-Fi.

Альтернативная топология, которая вызывает все больший интерес, — это *сеть mesh* или *ad hoc*. В беспроводной сети mesh узлы являются равноправными; то есть нет специального узла базовой станции. Сообщения могут передаваться через цепочку равноправных узлов, пока каждый узел находится в пределах досягаемости предыдущего узла. Это показано на рис. 2.25. Это позволяет беспроводной части сети распространяться за пределы ограниченного диапазона одного радиопередатчика. С точки зрения конкуренции между технологиями это позволяет технологии с меньшим радиусом действия расширить свой диапазон

и потенциально конкурировать с технологией с большим радиусом действия. Mesh-сети также предлагают отказоустойчивость, обеспечивая множество маршрутов для передачи сообщения от точки А к точке В. Mesh-сеть можно расширять постепенно, с поэтапными затратами. С другой стороны, mesh-сеть требует, чтобы не клиентские узлы имели определенный уровень сложности в аппаратном и программном обеспечении, что потенциально увеличивает стоимость на единицу и потребление энергии, что является критическим фактором для устройств с батарейным питанием. Беспроводные mesh-сети представляют значительный исследовательский интерес, но они все еще находятся в начальном состоянии по сравнению с сетями с базовыми станциями. Беспроводные сенсорные сети, другая актуальная развивающаяся технология, часто образуют беспроводные mesh-сети.

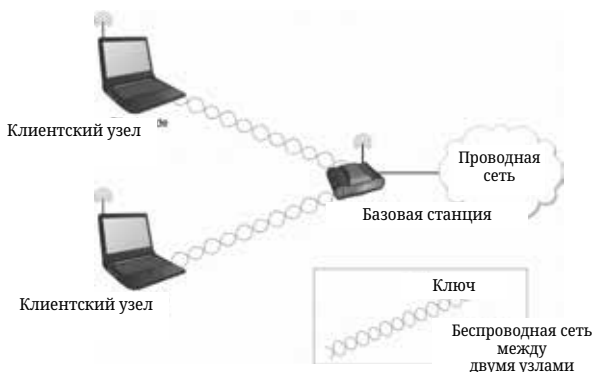


Рисунок 2.24. Беспроводная сеть с использованием базовой станции.

Теперь, когда мы рассмотрели некоторые общие вопросы, связанные с беспроводными сетями, давайте рассмотрим детали двух распространенных беспроводных технологий.

Глава 2.7.2. 802.11/Wi-Fi

Большинство читателей, вероятно, использовали беспроводную сеть, основанную на стандартах IEEE 802.11, часто называемую Wi-Fi. Wi-Fi технически является торговой маркой, принадлежащей торговой группе под названием Wi-Fi Alliance, которая сертифицирует соответствие продукции стандарту 802.11. Как и Ethernet, 802.11 предназначен для использования в ограниченной географической области (дома, офисные здания, кампусы), и его основная задача — управлять доступом к общему средству связи — в данном случае сигналам, распространяющимся в пространстве.

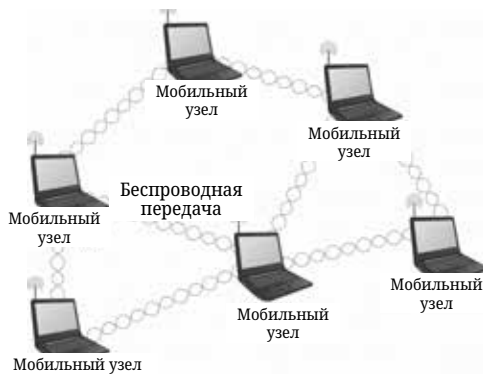


Рисунок 2.25. Беспроводная сеть ad hoc или mesh.

Физические свойства

802.11 определяет несколько различных физических уровней, которые работают в различных частотных диапазонах и обеспечивают различную скорость передачи данных.

Оригинальный стандарт 802.11 определял два радиобазированных стандарта физического уровня: один использовал частотное скачкообразное расширение спектра (frequency hopping) (по 79 частотным полосам шириной 1 МГц каждая), а другой — прямую последовательность расширения спектра (direct sequence spread spectrum) (с 11-битным чиповым кодом). Оба обеспечивали скорость передачи данных в диапазоне 2 Мбит/с. Впоследствии был добавлен стандарт физического уровня 802.11b, который, используя вариант прямой последовательности, поддерживал скорость до 11 Мбит/с. Эти три стандарта работали в лицензионно-свободном частотном диапазоне 2,4 ГГц электромагнитного спектра. Затем появился стандарт 802.11a, который обеспечивал скорость до 54 Мбит/с, используя вариант частотного разделения каналов, называемый ортогональным частотным разделением каналов (orthogonal frequency division multiplexing, OFDM). 802.11a работает в лицензионно-свободном диапазоне 5 ГГц. За ним последовал 802.11g, который также использует OFDM и обеспечивает скорость до 54 Мбит/с. 802.11g совместим с 802.11b и возвращается к диапазону 2,4 ГГц.

На момент написания этого текста многие устройства поддерживают 802.11n или 802.11ac, которые обычно достигают скорости передачи данных на устройство от 150 Мбит/с до 450 Мбит/с соответственно. Это улучшение частично связано с использованием нескольких антенн и увеличением ширины беспроводных каналов. Использование нескольких антенн часто называется MIMO (multiple-input, multiple-output). Последний развивающийся стандарт, 802.11ax, обещает еще одно значительное улучшение пропускной способности, частично за счет использования многих методов кодирования и модуляции, применяемых в сотовых сетях 4G/5G, которые мы рассмотрим в следующем разделе.

Обычно коммерческие продукты поддерживают более одного стандарта 802.11; многие базовые станции поддерживают все пять вариантов (a, b, g, n и ac). Это не только обеспечивает совместимость с любым устройством, поддерживающим любой из стандартов, но и позволяет двум таким продуктам выбирать самый высокий вариант пропускной способности для конкретной среды.

Стоит отметить, что хотя все стандарты 802.11 определяют максимальную поддерживаемую скорость передачи данных, они также поддерживают и более низкие скорости (например, 802.11a допускает скорости передачи данных 6, 9, 12, 18, 24, 36, 48 и 54 Мбит/с). При более низких скоростях легче декодировать передаваемые сигналы в условиях наличия шума. Различные схемы модуляции используются для достижения различных скоростей передачи данных. Кроме того, количество избыточной информации в виде кодов исправления ошибок варьируется. Больше избыточной информации означает большую устойчивость к битовым ошибкам за счет снижения эффективной скорости передачи данных (поскольку больше передаваемых битов являются избыточными).

Системы пытаются выбрать оптимальную скорость передачи данных в зависимости от окружающей среды шума; алгоритмы выбора скорости передачи данных могут быть довольно сложными. Интересно, что стандарты 802.11 не указывают конкретный подход, оставляя разработку алгоритмов различным производителям. Основной подход к выбору скорости передачи данных заключается в оценке уровня битовых ошибок, либо путем прямого измерения отношения сигнал/шум (signal-to-noise ratio, SNR) на физическом уровне, либо путем оценки SNR, измеряя, как часто пакеты успешно передаются и подтверждаются. В некоторых подходах отправитель время от времени проверяет более высокую скорость передачи данных, отправляя один или несколько пакетов на этой скорости, чтобы проверить, получится ли это.

Избежание коллизий

На первый взгляд может показаться, что беспроводной протокол должен следовать тому же алгоритму, что и Ethernet: ждать, пока канал освободится, перед началом передачи, и отступать, если произошла коллизия. И, по большому счету, именно так и ра-

ботает 802.11. Дополнительная сложность для беспроводной связи заключается в том, что, хотя узел в Ethernet получает передачи от всех других узлов и может одновременно передавать и принимать, ни одно из этих условий не выполняется для беспроводных узлов. Это делает обнаружение коллизий значительно более сложным. Причина, по которой беспроводные узлы обычно не могут одновременно передавать и принимать (на одной и той же частоте), заключается в том, что мощность, генерируемая передатчиком, намного выше любой принятой мощности и поэтому заглушает принимающую схему. Причина, по которой узел может не принимать передачи от другого узла, заключается в том, что этот узел может находиться слишком далеко или быть заблокированным препятствием. Эта ситуация немного сложнее, чем кажется на первый взгляд, как покажет следующее обсуждение.

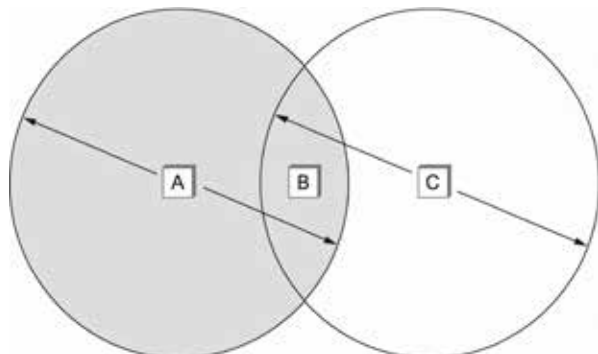


Рисунок 2.26. Проблема скрытого узла. Хотя узлы A и C скрыты друг от друга, их сигналы могут столкнуться в узле B. (Зона действия узла B не показана).

Рассмотрим ситуацию, изображенную на рис. 2.26, где A и C находятся в пределах досягаемости B, но не друг друга. Предположим, что и A, и C хотят общаться с B, и поэтому каждый из них отправляет ему кадр. A и C не знают о существовании друг друга, так как их сигналы не доходят настолько далеко. Эти два кадра сталкиваются друг с другом в B, но, в отличие от Ethernet, ни A, ни C не знают об этой коллизии. Говорят, что A и C являются скрытыми узлами по отношению друг к другу.

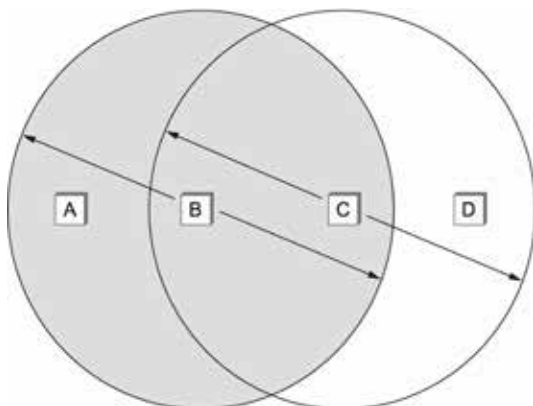


Рисунок 2.27. Проблема незащищенного узла. Хотя B и C подвержены воздействию сигналов друг друга, помех не будет, если B будет передавать сигнал A, а C — D. (Зоны действия A и D не показаны).

Проблема, называемая проблемой открытого узла, возникает в условиях, изображенных на рис. 2.27, где каждый из четырех узлов может отправлять и принимать сигналы, которые достигают только узлов, расположенных непосредственно слева и справа от них. Например, В может обмениваться кадрами с А и С, но не может достичь D, в то время как С может достичь В и D, но не А. Предположим, что В отправляет данные А. Узел С знает об этом, так как слышит передачу В. Однако ошибкой было бы заключить, что С не может передавать какому-либо узлу, просто потому что он слышит передачу В. Например, предположим, что С хочет передать данные узлу D. Это не проблема, так как передача С к D не будет мешать способности А принимать от В. (Это будет мешать А отправлять данные В, но в нашем примере В передает.)

802.11 решает эти проблемы с помощью CSMA/CA, где CA означает предотвращение коллизий, в отличие от обнаружения коллизий в CSMA/CD, используемого в Ethernet. Для этого нужно несколько шагов.

Часть Carrier Sense кажется достаточно простой: перед отправкой пакета передатчик проверяет, слышит ли он другие передачи; если нет, он отправляет пакет. Однако из-за проблемы скрытого узла просто ожидание отсутствия сигналов от других передатчиков не гарантирует, что коллизия не произойдет с точки зрения приемника. По этой причине одна часть CSMA/CA — это явное подтверждение (ACK) от приемника к отправителю. Если пакет был успешно декодирован и прошел проверку CRC у приемника, приемник отправляет ACK обратно отправителю.

Обратите внимание, что если коллизия все-таки произойдет, весь пакет будет испорчен. По этой причине 802.11 добавляет опциональный механизм под названием RTS-CTS (готов к отправке — чист для отправки (Ready to Send-Clear to Send)). Это частично решает проблему скрытого узла. Отправитель посылает RTS (короткий пакет) к предполагаемому приемнику, и если этот пакет успешно принят, приемник отвечает другим коротким пакетом, CTS. Даже если RTS не был услышан скрытым узлом, CTS, вероятно, будет услышан. Это эффективно говорит узлам в пределах досягаемости приемника, что они не должны ничего отправлять в течение некоторого времени — время предполагаемой передачи указано в пакетах RTS и CTS. После этого времени плюс небольшой интервал канал можно считать снова доступным, и другой узел может попытаться отправить данные.

Конечно, два узла могут обнаружить свободный канал и попытаться передать кадр RTS одновременно, что приведет к их коллизии. Отправители понимают, что произошла коллизия, когда они не получают кадр CTS после определенного времени, и в этом случае каждый из них ждет случайное время перед повторной попыткой. Время задержки для данного узла определяется экспоненциальным алгоритмом обратного отсчета, очень похожим на тот, что используется в Ethernet.

После успешного обмена RTS-CTS отправитель отправляет свой пакет данных и, если все идет хорошо, получает ACK для этого пакета. В случае отсутствия своевременного ACK отправитель снова пытается запросить использование канала, используя тот же процесс, описанный выше. К этому времени, конечно, другие узлы могут снова пытаться получить доступ к каналу.

Система распространения

Как уже говорилось, 802.11 подходит для сети с топологией mesh (ad hoc), и разработка стандарта 802.11s для mesh-сетей близится к завершению. В настоящее время, однако, почти все сети 802.11 используют топологию с базовыми станциями.

Вместо того чтобы все узлы были равны, некоторые узлы могут перемещаться (например, ваш ноутбук), а некоторые подключены к проводной сетевой инфраструктуре. 802.11 называет эти базовые станции *точками доступа* (access points, AP), и они подключены друг к другу через так называемую *систему распределения*. Рис. 2.28 иллюстрирует систему распределения, которая соединяет три точки доступа, каждая из которых обслуживает узлы в определенной области. Каждая точка доступа работает на некотором канале в соответствующем диапазоне частот, и каждая AP обычно будет на другом канале, отличном от соседних.

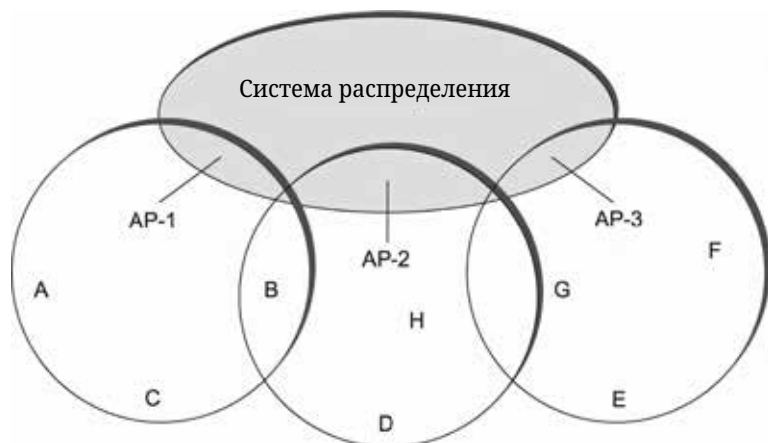


Рисунок 2.28. Точки доступа, подключенные к системе распределения.

Детали системы распределения не важны для данного обсуждения — это может быть, например, Ethernet. Единственно важный момент заключается в том, что распределительная сеть работает на канальном уровне, на том же уровне протокола, что и беспроводные соединения. Другими словами, она не зависит от каких-либо протоколов более высокого уровня (таких как сетевой уровень).

Хотя два узла могут напрямую общаться друг с другом, если они находятся в зоне досягаемости, идея этой конфигурации заключается в том, что каждый узел ассоциирует себя с одной точкой доступа. Чтобы узел А мог общаться с узлом Е, например, А сначала отправляет кадр своей точке доступа (AP-1), которая пересылает кадр через систему распределения к AP-3, которая, наконец, передает кадр узлу Е. Как AP-1 узнал, что нужно переслать сообщение на AP-3, выходит за рамки стандарта 802.11; возможно, был использован протокол мостов. Что определяет 802.11, так это то, как узлы выбирают свои точки доступа и, что более интересно, как работает этот алгоритм в условиях, когда узлы перемещаются из одной ячейки в другую.

Метод выбора точки доступа называется сканированием и включает следующие четыре шага:

1. Узел отправляет кадр запроса (Probe).
2. Все точки доступа в зоне досягаемости отвечают кадром ответа на запрос (Probe Response).
3. Узел выбирает одну из точек доступа и отправляет ей кадр запроса на ассоциацию (Association Request).
4. Точка доступа отвечает кадром ответа на ассоциацию (Association Response).

Узел использует этот протокол всякий раз, когда подключается к сети, а также когда он становится неудовлетворен своей текущей точкой доступа. Это может произойти, например, из-за ослабления сигнала от текущей точки доступа по мере удаления узла от нее. Всякий раз, когда узел подключается к новой точке доступа, новая точка доступа уведомляет старую точку доступа об изменении (это происходит на шаге 4) через систему распределения.

Рассмотрим ситуацию, показанную на рис. 2.29, где узел С перемещается из ячейки, обслуживаемой AP-1, в ячейку, обслуживаемую AP-2. По мере перемещения он отправляет кадры запроса (Probe), что в конечном итоге приводит к получению кадров ответа на запрос (Probe Response) от AP-2. В какой-то момент С предпочитает AP-2 вместо AP-1 и повторно ассоциирует себя с этой точкой доступа.

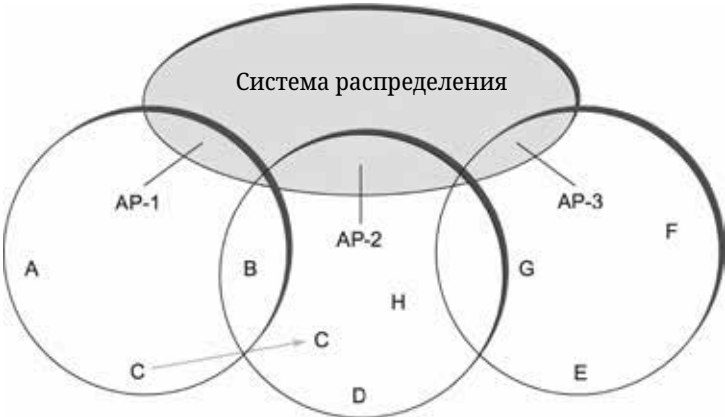


Рисунок 2.29. Мобильность узла.

Описанный механизм называется *активным сканированием*, так как узел активно ищет точку доступа. Точки доступа также периодически отправляют кадры Beacon (маяки), рекламирующие возможности точки доступа; они включают скорости передачи, поддерживаемые точкой доступа. Это называется *пассивным сканированием*, и узел может переключиться на эту точку доступа, основываясь на кадре маяка, просто отправив кадр Association Request обратно к точке доступа.

Формат кадра

Большая часть формата кадра 802.11, который изображен на рис. 2.30, соответствует нашим ожиданиям. Кадр содержит адреса источника и назначения, каждый из которых длиной 48 бит; до 2312 байт данных; и 32-битный CRC. Поле Control содержит три подчиненных поля, представляющих интерес (не показаны): 6-битное поле Type, которое указывает, содержит ли кадр данные, является кадром RTS или CTS или используется алгоритмом сканирования, и пару 1-битных полей — называемых ToDS и FromDS, которые описаны ниже.

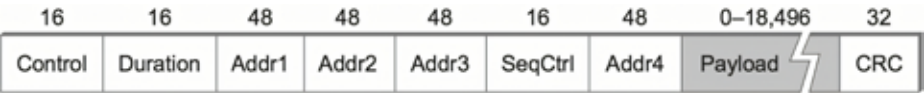


Рисунок 2.30. Формат кадра стандарта 802.11.

Особенность формата кадра 802.11 заключается в том, что он содержит четыре, а не два адреса. То, как эти адреса интерпретируются, зависит от настроек битов ToDS и FromDS в поле Control кадра. Это необходимо для учета возможности того, что кадр должен был быть передан через систему распределения, что означало бы, что оригинальный отправитель не обязательно совпадает с последним передающим узлом. Аналогичное рассуждение относится и к адресу назначения. В самом простом случае, когда один узел отправляет данные непосредственно другому, оба бита DS равны 0, Addr1 указывает на целевой узел, а Addr2 указывает на узел-источник. В самом сложном случае оба бита DS установлены в 1, указывая, что сообщение прошло от беспроводного узла в систему распределения, а затем от системы распределения к другому беспроводному узлу.

Если оба бита установлены, то Addr1 идентифицирует конечный пункт назначения, Addr2 — непосредственного отправителя (того, который переслал кадр из системы распределения в конечный пункт назначения), Addr3 — промежуточный пункт

назначения (тот, который принял кадр от беспроводного узла и переслал его через систему распределения), а Addr4 — первоначальный источник. В примере, приведенном на рис. 2.28, Addr1 соответствует E, Addr2 идентифицирует AP-3, Addr3 соответствует AP-1, а Addr4 идентифицирует A.

Безопасность беспроводных соединений

Одной из очевидных проблем беспроводных соединений по сравнению с проводами или оптоволоконным является то, что невозможно точно определить, куда ушли ваши данные. Вы, вероятно, сможете понять, были ли они получены предполагаемым получателем, но нельзя сказать, сколько других получателей также могли принять вашу передачу. Поэтому, если вы обеспокоены конфиденциальностью ваших данных, беспроводные сети представляют собой проблему.

Даже если вы не беспокоитесь о конфиденциальности данных — или, возможно, позаботились об этом каким-то другим образом, — вы можете быть обеспокоены несанкционированным пользователем, внедряющим данные в вашу сеть. В крайнем случае такой пользователь может потреблять ресурсы, которые вы предпочли бы использовать сами, например, ограниченную пропускную способность между вашим домом и вашим интернет-провайдером.

По этим причинам беспроводные сети обычно имеют какой-либо механизм для контроля доступа как к самому каналу связи, так и к передаваемым данным. Эти механизмы часто классифицируются как *безопасность беспроводных сетей*. Широко используемый стандарт WPA2 описан в разделе 8.

Глава 2.7.3. Bluetooth (802.15.1)

Bluetooth заполняет нишу очень короткодействующей связи между мобильными телефонами, КПК, ноутбуками и другими персональными или периферийными устройствами. Например, Bluetooth можно использовать для подключения мобильного телефона к гарнитуре или ноутбука к клавиатуре. Грубо говоря, Bluetooth является более удобной альтернативой соединению двух устройств с помощью провода. В таких случаях не требуется большой диапазон или высокая пропускная способность. Это означает, что Bluetooth-радиопередатчики могут использовать довольно низкую мощность передачи, поскольку мощность передачи является одним из основных факторов, влияющих на пропускную способность и дальность беспроводных соединений. Это соответствует целевым приложениям для устройств с поддержкой Bluetooth — большинство из них работают от батарей (например, повсеместно используемая телефонная гарнитура), поэтому важно, чтобы они не потребляли много энергии.

Bluetooth работает в лицензируемом диапазоне на частоте 2,45 ГГц. Bluetooth-соединения имеют типичную пропускную способность около 1–3 Мбит/с и дальность около 10 м. По этой причине и потому, что устройства, как правило, принадлежат одному человеку или группе, Bluetooth иногда классифицируется как *персональная сеть* (Personal Area Network, PAN).

Bluetooth определен отраслевым консорциумом под названием *Bluetooth Special Interest Group*. Он определяет целый набор протоколов, выходящих за рамки канального уровня для определения прикладных протоколов, которые он называет *профилями*, для различных приложений. Например, существует профиль для синхронизации КПК с персональным компьютером. Другой профиль предоставляет мобильному компьютеру доступ к проводной локальной сети наподобие 802.11, хотя это не было изначальной целью Bluetooth. Стандарт IEEE 802.15.1 основан на Bluetooth, но исключает прикладные протоколы.

Базовая конфигурация сети Bluetooth, называемая пико-сетью (piconet), состоит из главного устройства и до семи ведомых устройств, как показано на рис. 2.31. Любая связь происходит между главным устройством и ведомым; ведомые устройства не взаимодействуют друг с другом напрямую. Поскольку ведомые устройства выполняют более простую роль, их оборудование и программное обеспечение Bluetooth могут быть проще и дешевле.



Рисунок 2.31. Пикосеть Bluetooth.

Поскольку Bluetooth работает в лицензируемом диапазоне, он должен использовать метод спектрального рассеяния для борьбы с возможными помехами в диапазоне. Он использует скачкообразную перестройку частоты с 79 каналами (частотами), используя каждый по 625 мкс. Это обеспечивает естественный временной интервал, который Bluetooth использует для синхронного временного мультиплексирования. Кадр занимает 1, 3 или 5 последовательных временных интервалов. Только главный может начинать передачу в нечетных интервалах. Ведомое устройство может начинать передачу в четном интервале — но только в ответ на запрос от главного устройства во время предыдущего интервала, тем самым предотвращая любую конкуренцию между ведомыми устройствами.

Ведомое устройство (slave) может быть переведено в режим ожидания, то есть установлено в неактивное состояние с низким энергопотреблением. Устройство в режиме ожидания не может общаться в пико-сети; его может активировать только главное (ведущее) устройство. Пико-сеть может иметь до 255 устройств в режиме ожидания в дополнение к своим активным ведомым устройствам.

В области очень низкого энергопотребления и связи на коротких расстояниях есть несколько технологий помимо Bluetooth. Одна из них — ZigBee, разработанная альянсом ZigBee и стандартизированная как IEEE 802.15.4. Она предназначена для ситуаций, когда требования к пропускной способности невысоки, а энергопотребление должно быть очень низким для обеспечения долгого времени работы от батареи. Также она предназначена для того, чтобы быть проще и дешевле, чем Bluetooth, что делает ее доступной для использования в более дешевых устройствах, таких как датчики. Датчики становятся все более важным классом сетевых устройств, так как технологии развиваются до уровня, когда очень дешевые маленькие устройства могут быть развернуты в больших количествах для мониторинга таких параметров, как температура, влажность и энергопотребление в здании.

Глава 2.8. Сети доступа

Помимо Ethernet и Wi-Fi подключений, которые мы обычно используем для подключения к Интернету дома, на работе, в школе и во многих общественных местах, большинство из нас подключается к Интернету через услугу *доступа* или *широкополосный сервис*, который мы покупаем у интернет-провайдера (ISP). В этом разделе описаны две такие технологии: *пассивные оптические сети* (PON), обычно называемые «оптика до дома» (fiber-to-the-home), и *сотовые сети*, которые подключают наши мобильные устройства. В обоих случаях сети являются мультидоступными (как Ethernet и Wi-Fi), но, как мы увидим, их подход к управлению доступом значительно отличается.

Для создания немного большего контекста: интернет-провайдеры (например, телекоммуникационные или кабельные компании) часто управляют национальной магистральной сетью, к периферии которой подключены сотни или тысячи пограничных узлов, каждый из которых обслуживает город или район. Эти пограничные узлы обычно называются *центральными офисами* (Central Offices) в мире телекоммуникаций и *головными станциями* (Head Ends) в мире кабельного телевидения, но, несмотря на их названия, подразумевающие «централизованность» и «корень иерархии», эти узлы находятся на самом краю сети интернет-провайдера; они находятся на стороне интернет-провайдера, непосредственно подключенной к клиентам. PON и сотовые сети доступа привязаны к этим объектам.¹

Глава 2.8.1. Пассивная оптическая сеть (PON)

PON — это технология, которая чаще всего используется для предоставления широкополосного доступа на основе оптоволокна в домах и офисах. PON использует структуру «точка-мультиточка», это означает, что сеть структурирована как дерево, с одной точкой, начинающейся в сети интернет-провайдера, и затем разветвляющейся, чтобы охватить до 1024 домов. PON получила свое название из-за того, что разветвители являются пассивными: они передают оптические сигналы вниз и вверх по потоку без активного хранения и пересылки кадров. Таким образом, они являются оптическим вариантом повторителей, используемых в классическом Ethernet. Формирование кадров происходит на источнике в помещениях интернет-провайдера, в устройстве, называемом *оптическим линейным терминалом* (Optical Line Terminal, OLT), и на конечных точках в отдельных домах, в устройстве, называемом *оптическим сетевым узлом* (Optical Network Unit, ONU).

Рис. 2.32 показывает пример PON, упрощенный для изображения только одного ONU и одного OLT. На практике в центральном офисе будет несколько OLT, подключающих тысячи домов клиентов. Для полноты картины рис. 2.32 также включает две другие детали о том, как PON подключена к магистральной сети интернет-провайдера (и, следовательно, к остальной части Интернета). *Агрегационный коммутатор* (Agg Switch) агрегирует трафик от набора OLT, а BNG (шлюз широкополосной сети (Broadband Network Gateway)) является устройством телекоммуникационной компании, которое, среди прочего, измеряет интернет-трафик для целей вычисления счета за услуги. Как следует из его названия, BNG фактически является шлюзом между сетью доступа (все, что находится слева от BNG) и Интернетом (все, что находится справа от BNG).

Поскольку разветвители пассивные, PON должна реализовать какую-то форму протокола многократного доступа. Подход, который она использует, можно резюмировать следующим образом. Во-первых, исходящий и входящий трафик передается на двух разных оптических длинах волн, так что они полностью независимы друг от друга. Входящий трафик начинается на OLT, и сигнал распространяется по всем линиям

¹ DSL — это устаревший медный аналог PON. DSL-соединения также заканчиваются в центральных офисах телекоммуникационных компаний, эта технология не описывается, так как она выводится из эксплуатации.

в PON. В результате каждый кадр достигает каждого ONU. Это устройство затем смотрит на уникальный идентификатор в отдельных кадрах, передаваемых по длине волны, и либо сохраняет кадр (если идентификатор предназначен для него), либо отбрасывает его (если нет). Шифрование используется, чтобы предотвратить подслушивание трафика соседей ONU.

Входящий трафик затем мультиплексируется по времени на восходящей длине волны, при этом каждый ONU периодически получает очередь на передачу. Поскольку ONU распределены на довольно большой площади (измеряемой в километрах) и на разных расстояниях от OLT, для них не практично передавать данные, основываясь на синхронизированных часах, как в SONET. Вместо этого OLT передает *разрешения* отдельным ONU, предоставляя им временной интервал, в течение которого они могут передавать данные. Иными словами, один OLT централизованно реализует круговую передачу в общей PON. Это включает возможность того, что OLT может предоставить каждому ONU разное количество времени, фактически реализуя различные уровни обслуживания.

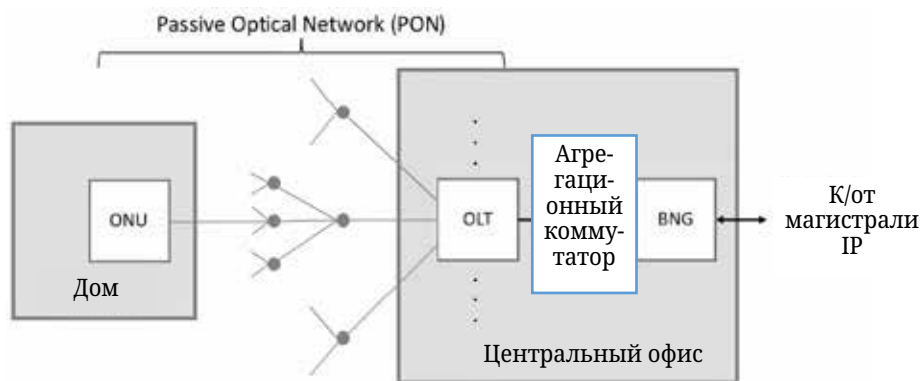


Рисунок 2.32. Пример PON, которая соединяет OLT в центральном офисе с ONU в домах и на предприятиях.

PON похожа на Ethernet в том смысле, что она определяет алгоритм совместного использования, который со временем развивался для поддержки все более высоких скоростей. G-PON (Gigabit-PON) является наиболее широко развернутой на сегодняшний день, поддерживая пропускную способность 2.25 Гбит/с. XGS-PON (10 Gigabit-PON) только начинает разворачиваться.

Глава 2.8.2. Сотовая сеть

Хотя технология сотовой связи берет свое начало в аналоговой голосовой связи, сейчас стандартами являются службы передачи данных на основе сотовых стандартов. Как и Wi-Fi, сотовые сети передают данные на определенных полосах частот в радиоспектре. В отличие от Wi-Fi, который позволяет любому использовать канал на частотах 2.4 или 5 ГГц (все, что нужно сделать, это настроить базовую станцию, как многие из нас делают дома), исключительное использование различных частотных полос было продано на аукционах и лицензировано поставщикам услуг, которые, в свою очередь, продают услуги мобильного доступа своим абонентам.

Полосы частот, используемые для сотовых сетей, различаются по всему миру и усложняются тем, что интернет-провайдеры часто одновременно поддерживают как старые/устаревшие технологии, так и новые/следующие поколения технологий, каждая из которых занимает свою частотную полосу. Если говорить кратко, традиционные сотовые тех-

нологии охватывают диапазон от 700 МГц до 2400 МГц, с новыми распределениями в среднем спектре, происходящими на частоте 6 ГГц, и распределениями миллиметровых волн (mmWave), открывающимися выше 24 ГГц.

Служба гражданского широкополосного радио (CBRS)

- Помимо лицензированных полос в Северной Америке также имеется нелицензированная полоса на частоте 3.5 ГГц, называемая *Службой гражданского широкополосного радио* (Citizens Broadband Radio Service, CBRS), которую может использовать любой обладатель сотового радио. Аналогичные нелицензированные полосы создаются и в других странах.
- Это открывает возможность для создания частных сотовых сетей, например, на территории университетского кампуса, предприятия или производственного предприятия.
- Для ясности, полоса CBRS позволяет трем категориям пользователей делить спектр: первоочередное право использования принадлежит первоначальным владельцам этого спектра — военно-морским радарам и наземным спутниковым станциям; за ними следуют приоритетные пользователи, получающие это право на полосу в 10 МГц на три года через региональные аукционы; и, наконец, остальное население, которое может получить доступ и использовать часть этой полосы при условии, что сначала они проверят центральную базу данных зарегистрированных пользователей.

Как и 802.11, сотовая технология полагается на использование базовых станций, подключенных к проводной сети. В случае сотовой сети базовые станции часто называются *широкополосными базовыми блоками* (Broadband Base Units, BBU), мобильные устройства, подключающиеся к ним, обычно называются *пользовательским оборудованием* (User Equipment, UE), и набор BBU привязан к *развитому пакетному ядру* (EPC), размещенному в центральном офисе. Беспроводная сеть, обслуживаемая EPC, часто называется *радиосетевой доступной сетью* (Radio Access Network, RAN).

BBU официально имеют другое название — Evolved NodeB, часто сокращаемые как eNodeB или eNB, где NodeB — название радиоблока в ранней версии сотовых сетей (с тех пор он эволюционировал). Поскольку мир сотовой связи продолжает быстро развиваться и eNB скоро будут обновлены до gNB, мы решили использовать более общее и менее запутанное обозначение BBU.

Рис. 2.33 показывает одну возможную конфигурацию от начала до конца, с несколькими дополнительными деталями. EPC имеет несколько субкомпонентов, включая MME (сущность управления мобильностью (Mobility Management Entity)), HSS (сервер домашних абонентов (Home Subscriber Server)) и пару S/PGW (шлюз сеанса/пакетный шлюз (Session/ Packet Gateway)); первый отслеживает и управляет перемещением UE по всей RAN, второй представляет собой базу данных, содержащую информацию, связанную с абонентами, а пара шлюзов обрабатывает и пересылает пакеты между RAN и Интернетом (это образует *пользовательскую плоскость* EPC). Мы говорим «одна возможная конфигурация», потому что стандарты сотовой связи допускают широкую вариативность в том, за сколько S/PGW отвечает данная MME, что позволяет одному MME управлять мобильностью на широкой географической территории, обслуживаемой несколькими центральными офисами. Наконец, хотя это явно не указано на рис. 2.33, иногда сеть PON интернет-провайдера используется для подключения удаленных BBU к центральному офису.

Географическая зона, обслуживаемая антенной BBU, называется *ячейкой* (или *сотой*). BBU может обслуживать одну ячейку или использовать несколько направленных антенн для обслуживания нескольких ячеек. Ячейки не имеют четких границ и могут перекрываться. В местах перекрытия UE (пользовательское оборудование) может потенциально общаться с несколькими BBU. Однако в любой момент времени UE находится в связи и под управлением только одного BBU. Когда устройство начинает покидать ячейку, оно перемещается в зону перекрытия с одной или несколькими другими ячейками. Текущий BBU улавливает ослабевающий сигнал от телефона и передает управление устройством той базовой станции, которая принимает от него самый силь-

ный сигнал. Если устройство в данный момент участвует в звонке или другой сетевой сессии, сессия должна быть передана новой базовой станции в процессе, который называется передачей. Процесс принятия решений по передачам находится в ведении ММЕ (сущность управления мобильностью), что исторически являлось собственностью поставщиков оборудования для сотовой связи (хотя теперь начинают появляться и открытые реализации ММЕ).

Существует несколько поколений протоколов, реализующих сотовую сеть, которые обычно называются 1G, 2G, 3G и так далее. Первые два поколения поддерживали только голосовую связь, а 3G определяет переход к широкополосному доступу, поддерживая скорости передачи данных, измеряемые в сотнях килобит в секунду. В настоящее время отрасль находится на уровне 4G (поддерживающего скорости передачи данных, обычно измеряемые в нескольких мегабитах в секунду) и находится в процессе перехода на 5G (с обещанием десятикратного увеличения скоростей передачи данных).

Начиная с 3G обозначение поколений фактически соответствует стандарту, определенному проектом партнерства третьего поколения (3GPP). Несмотря на то, что в названии содержится «3G», 3GPP продолжает определять стандарты для 4G и 5G, каждое из которых соответствует выпуску стандарта. Релиз 15, который уже опубликован, считается точкой раздела между 4G и 5G. Другая терминология для этой последовательности выпусков и поколений называется LTE, что означает «долгосрочная эволюция» (Long-Term Evolution). Основная идея заключается в том, что, несмотря на публикацию стандартов в виде последовательности отдельных выпусков, вся отрасль в целом идет по довольно четко определенному эволюционному пути, известному как LTE. В этом разделе используется терминология LTE, но при необходимости подчеркиваются изменения, которые приходят с 5G.

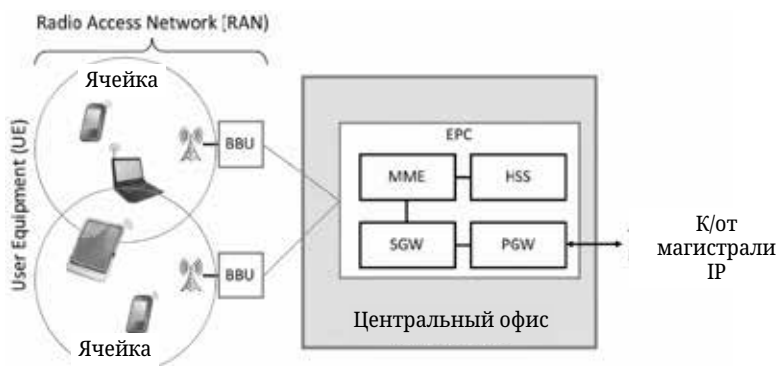


Рисунок 2.33. Сеть радиодоступа (RAN), соединяющая набор сотовых устройств (UE) с ядром Evolved Packet Core (EPC), расположенным в центральном офисе.

Главное нововведение в воздушном интерфейсе LTE заключается в том, как он распределяет доступный радиоспектр между UE. В отличие от Wi-Fi, который основан на конкуренции, LTE использует стратегию резервирования. Эта разница коренится в фундаментальном предположении каждой системы об использовании: Wi-Fi предполагает слабую загрузку сети (и поэтому оптимистично передает данные, когда беспроводное соединение свободно, и отступает при обнаружении конкуренции), в то время как сотовые сети предполагают высокую загрузку и стремятся к ней (и поэтому явно назначают различных пользователей на разные «доли» доступного радиоспектра).

Современный механизм доступа к среде для LTE называется *ортогональным частотным разделением каналов с множественным доступом* (Orthogonal Frequency-Division Multiple Access, OFDMA). Идея заключается в мультиплексировании данных на наборе

из 12 ортогональных поднесущих частот, каждая из которых модулируется независимо. Термин «множественный доступ» в OFDMA подразумевает, что данные могут одновременно передаваться для нескольких пользователей, каждый на разной поднесущей частоте и в течение разного времени. Поддиапазоны узкие (например, 15 кГц), но кодирование пользовательских данных в символы OFDMA спроектировано так, чтобы минимизировать риск потери данных из-за помех между соседними диапазонами.

Использование OFDMA естественным образом приводит к концептуализации радиоспектра как двумерного ресурса, как показано на рис. 2.34. Минимальная планируемая единица, называемая *ресурсным элементом* (Resource Element, RE), соответствует диапазону шириной 15 кГц вокруг одной поднесущей частоты и времени, необходимого для передачи одного символа OFDMA. Количество битов, которые могут быть закодированы в каждом символе, зависит от скорости модуляции, так, например, при использовании квадратурной амплитудной модуляции (QAM) 16-QAM дает 4 бита на символ, а 64-QAM дает 6 битов на символ.

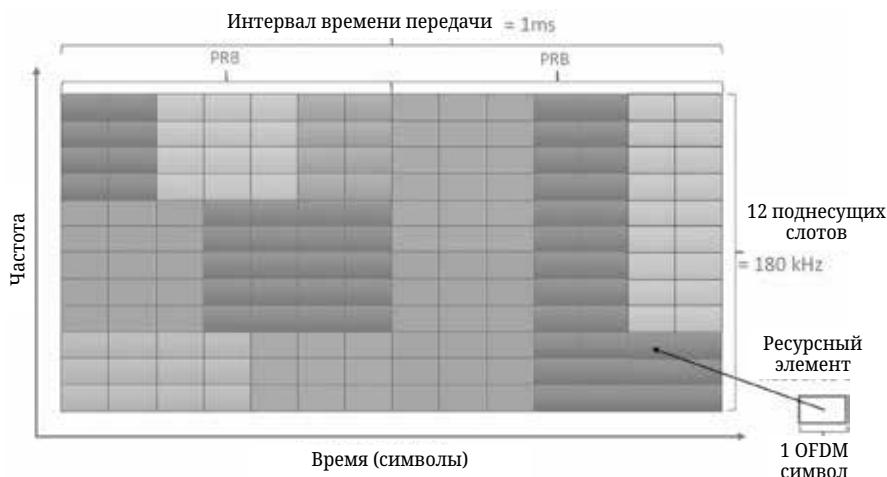


Рисунок 2.34. Доступный радиочастотный спектр, абстрактно представленный двухмерной сеткой планируемых элементов ресурсов.

Планировщик принимает решения о выделении на уровне блоков из $7 \times 12 = 84$ ресурсных элементов, которые называются *физическим ресурсным блоком* (Physical Resource Block, PRB). Рис. 2.34 показывает два последовательных PRB, где UE (пользовательские устройства) изображены разноцветными блоками. Конечно, время продолжает течь вдоль одной оси, и в зависимости от размера лицензированного частотного диапазона может быть доступно гораздо больше слотов поднесущих (и, следовательно, PRB) вдоль другой оси, поэтому планировщик, по сути, планирует последовательность PRB для передачи.

Интервал времени передачи (Transmission Time Interval, TTI) в 1 мс, показанный на рис. 2.34, соответствует временной рамке, в которой BBU получает обратную связь от UE о качестве сигнала, который они используют. Эта обратная связь, называемая *индикатором качества канала* (Channel Quality Indicator, CQI), по существу сообщает наблюдаемое отношение сигнал/шум, что влияет на способность UE восстанавливать биты данных. Базовая станция затем использует эту информацию для адаптации того, как она выделяет доступный радиоспектр UE, которые она обслуживает.

До этого момента описание того, как мы планируем радиоспектр, относится к 4G. Переход от 4G к 5G вводит дополнительные степени свободы в том, как радиоспектр планируется, что позволяет адаптировать сотовую сеть к более разнообразному набору устройств и областей применения.

Фундаментально 5G определяет семейство форматов сигналов — в отличие от 4G, который специфицировал только один формат сигнала — каждый из которых оптимизирован для разных диапазонов в радиоспектре¹. Диапазоны с несущими частотами ниже 1 ГГц предназначены для предоставления мобильного широкополосного доступа и массовых услуг Интернета вещей (IoT) с основным упором на дальность действия. Несущие частоты между 1 ГГц и 6 ГГц разработаны для обеспечения более широких полос пропускания, ориентируясь на мобильный широкополосный доступ и критически важные приложения. Несущие частоты выше 24 ГГц (миллиметровые волны) предназначены для обеспечения сверхшироких полос пропускания на коротких расстояниях в пределах прямой видимости.

Эти разные форматы сигналов влияют на планирование и интервалы поднесущих (то есть на «размер» ресурсных элементов, описанных ранее).

- Для диапазонов ниже 1 ГГц 5G допускаются максимальные полосы пропускания до 50 МГц. В этом случае существуют два формата сигналов: один с интервалом поднесущей 15 кГц и другой с интервалом 30 кГц. (В примере на рис. 2.34 использован интервал 15 кГц). Соответствующие интервалы планирования составляют 0,5 мс и 0,25 мс соответственно. (В примере на рис. 2.34 использован интервал 0,5 мс.)
- Для диапазонов 1 ГГц — 6 ГГц максимальные полосы пропускания увеличиваются до 100 МГц. Соответственно, существуют три формата сигналов с интервалами поднесущих 15 кГц, 30 кГц и 60 кГц, соответствующие интервалам планирования 0,5 мс, 0,25 мс и 0,125 мс соответственно.
- Для миллиметровых диапазонов полосы пропускания могут достигать 400 МГц. Существуют два формата сигналов с интервалами поднесущих 60 кГц и 120 кГц. Оба имеют интервалы планирования 0,125 мс.

Этот диапазон вариантов важен, потому что он добавляет еще одну степень свободы для планировщика. В дополнение к распределению ресурсных блоков между пользователями он имеет возможность динамически изменять размер ресурсных блоков, изменяя формат сигнала, используемый в диапазоне, за который он отвечает.

Будь то 4G или 5G, алгоритм планирования представляет собой сложную задачу оптимизации, целью которой является одновременно (а) максимизация использования доступного частотного диапазона и (б) обеспечение каждого UE (пользовательского устройства) необходимым уровнем обслуживания. Этот алгоритм не специфицирован 3GPP, а является интеллектуальной собственностью поставщиков BBU.

Перспектива: гонка на пределе

Когда мы начинаем исследовать, как софтверизация преобразует сеть, следует признать, что именно доступная сеть, которая соединяет дома, предприятия и мобильных пользователей с Интернетом, претерпевает самые радикальные изменения. Описанные в главе 2.8 сети «оптика до дома» и сотовые сети в настоящее время построены на сложных аппаратных устройствах (например, OLT, BNG, BBU, EPC). Эти устройства не только были закрытыми и проприетарными, но и поставщики, которые их продают, обычно объединяли в каждом устройстве широкую и разнообразную функциональность. В результате их стало дорого собирать, ими сложно управлять и долго модернизировать.

В ответ на это сетевые операторы активно переходят от этих специализированных устройств к открытому программному обеспечению, работающему на обычных серверах, коммутаторах и устройствах доступа. Эта инициатива часто называется *CORD*, что является акронимом для Central Office Re-architected as a Datacenter (центральный офис, переосмысленный как дата-центр), и, как следует из названия, идея заключается в построении центрального офиса телекоммуникационной компании (или кабельного узла, что при-

¹ Форма волны — это независимое от частоты, амплитуды и фазового сдвига свойство (форма) сигнала. Примером формы сигнала является синусоидальная волна.

водит к акрониму HERD) с использованием тех же технологий, что и в крупных дата-центрах, составляющих облако.

Мотивация операторов заключается отчасти в получении экономии за счет замены специализированных устройств на обычное оборудование, но главным образом это вызвано необходимостью ускорить темпы инноваций. Их цель — внедрение новых классов граничных сервисов, таких как общественная безопасность, автономные транспортные средства, автоматизированные фабрики, Интернет вещей (IoT), иммерсивные пользовательские интерфейсы, которые выигрывают от низкой задержки соединения с конечными пользователями, а более важно — с увеличивающимся количеством устройств, окружающих этих пользователей. Это приводит к многоуровневому облаку, подобному показанному на рис. 2.35.



Рисунок 2.35. Появляющиеся многоуровневые облака включают в себя публичные облака на базе IXP, распределенные облака на базе IXP и краевые облака на базе доступа, такие как CORD. В то время как в мире насчитывается порядка 150 облаков, размещенных на IXP, можно ожидать появления тысяч или даже десятков тысяч краевых облаков.

Все это часть растущей тенденции по перемещению функциональности из дата-центра ближе к краю сети, что ставит облачных провайдеров и сетевых операторов на курс столкновения. Облачные провайдеры, стремясь к приложениям с низкой задержкой и высокой пропускной способностью, выходят из дата-центра и движутся к краю сети в то время, когда сетевые операторы применяют лучшие практики и технологии облака к уже существующему краю, который реализует доступную сеть. Невозможно сказать, как это все развернется со временем; обе индустрии имеют свои определенные преимущества.

С одной стороны, облачные провайдеры считают, что, насыщая мегаполисы периферийными кластерами и абстрагируя сеть доступа, они смогут создать периферийное присутствие с достаточно низкой задержкой и достаточно высокой пропускной способностью для обслуживания следующего поколения периферийных приложений. В этом сценарии сеть доступа остается простой трубой для передачи данных, что позволяет облачным провайдерам отлично выполнять свою работу: запускать масштабируемые облачные сервисы на стандартном оборудовании.

С другой стороны, сетевые операторы считают, что, строя сеть доступа следующего поколения с использованием облачных технологий, они смогут размещать периферийные приложения непосредственно в сети доступа. Этот сценарий имеет встроенные преимущества: существующая и широко распределенная физическая инфраструктура, существующая операционная поддержка и нативная поддержка как мобильности, так и гарантированного сервиса.

Признавая обе эти возможности, стоит также рассмотреть третий исход, который не только заслуживает внимания, но и к нему стоит стремиться: *демократизация периферийной сети*. Идея заключается в том, чтобы сделать облако доступным для всех, а не только для действующих облачных провайдеров или сетевых операторов. Есть три причины, почему эта возможность будет оптимальна в будущем:

1. Аппаратное и программное обеспечение для сети доступа становится стандартным и открытым. Это ключевой фактор, о котором мы только что говорили. Если это помогает телекоммуникационным и кабельным компаниям быть гибкими, то это может предоставить те же преимущества для всех.
2. Есть спрос. Компании в автомобильной, фабричной и складской сферах все больше хотят развертывать частные сети 5G для различных случаев использования физической автоматизации (например, гараж, где удаленный парковщик паркует вашу машину, или фабричный цех, использующий автоматизированных роботов).
3. Спектр частот становится доступным. 5G открывается для использования в нелицензированной или слабо лицензированной модели в США и Германии, которые являются двумя основными примерами, и другие страны скоро последуют их примеру. Это означает, что для частного использования будет доступно около 100–200 МГц спектра частот для 5G.

Одним словом, сеть доступа исторически была прерогативой телекоммуникационных и кабельных компаний. Но софтверизация и виртуализация сети доступа открывает возможность для всех (от развитых городов до малообеспеченных сельских районов, от многоквартирных комплексов до производственных предприятий) создать облако на границе доступа и подключить его к публичному Интернету. Мы ожидаем, что сделать это будет так же просто, как сегодня установить WiFi-маршрутизатор. Это не только приведет к тому, что край доступа окажется в новых (более редких) средах, но и откроет сеть доступа для разработчиков, которые инстинктивно пойдут туда, где есть возможности для инноваций.

Раздел 3.

Межсетевое взаимодействие

Природа, похоже, достигает многих своих целей длинными извилистыми путями.

Рудольф Лотце

Проблема: не все сети имеют прямое соединение

Как мы уже видели, существует множество технологий, которые можно использовать для создания соединений «последней мили» или для подключения ограниченного количества узлов. Но как построить сети глобального масштаба? Одиночный Ethernet может объединить не более 1024 хостов, а «точка-точка» соединяет только два. Беспроводные сети ограничены радиусом действия своих радиопередатчиков. Чтобы построить глобальную сеть, нам нужен способ соединения различных типов соединений и сетей с множественным доступом. Концепция объединения разных типов сетей для создания большой глобальной сети является основной идеей Интернета и часто называется *межсетевым взаимодействием* (internetworking).

Мы можем разделить проблему меж сетевого взаимодействия на несколько подпроблем. Прежде всего нам нужен способ объединения соединений. Устройства, которые объединяют соединения одного типа, часто называются *коммутаторами* (switches) или иногда коммутаторами *уровня 2* (Layer 2 или L2 switches). Эти устройства являются первой темой данного раздела. Особенно важным классом L2-коммутаторов, используемых сегодня, являются те, которые применяются для объединения сегментов Ethernet. Эти коммутаторы также иногда называют *мостами* (bridges).

Основная задача коммутатора — принимать пакеты, которые поступают на вход, и *перенаправлять* (или *коммутировать*) их на правильный выход, чтобы они достигли своего места назначения. Существует множество способов, которыми коммутатор может определить «правильный» выход для пакета, и их можно грубо разделить на подходы без установления соединения и подходы с установлением соединения. Для этих подходов области применения были найдены много лет назад.

Учитывая огромное разнообразие типов сетей, нам также нужен способ соединения разнородных сетей и соединений (то есть работы с *гетерогенностью*). Устройства, выполняющие эту задачу, ранее назывались *шлюзами* (gateways), но теперь чаще всего называются *маршрутизаторами* (routers) или, альтернативно, коммутаторами *уровня 3* (Layer 3 или L3 switches). Протокол, который был изобретен для решения проблемы объединения разнородных типов сетей, — это Протокол Интернета (Internet Protocol, IP), который является темой второй главы в данном разделе.

После объединения большого количества соединений и сетей с помощью коммутаторов и маршрутизаторов вероятно возникновение множества различных возможных путей от одной точки к другой. Поиск подходящего пути или маршрута через сеть является одной из фундаментальных проблем сетевых технологий. Такие пути должны быть эффективными (например, не длиннее, чем необходимо), без циклов и способны реагировать на изменения в сети — узлы могут выходить из строя или перезагружаться, соединения могут обрываться, а новые узлы или соединения могут добавляться. Далее рассматриваются некоторые алгоритмы и протоколы, разработанные для решения этих вопросов.

После понимания проблем коммутирования и маршрутизации нам нужны устройства для выполнения этих функций. Раздел заканчивается обсуждением способов реализации коммутаторов и маршрутизаторов. Хотя многие пакетные коммутаторы и маршрутизаторы по своей структуре похожи на универсальные компьютеры, существуют ситуации, где используются более специализированные конструкции. Это особенно актуально для высокопроизводительных устройств, где существует постоянная потребность в большей пропускной способности для обработки все возрастающей нагрузки на ядро Интернета.

Глава 3.1. Основы коммутации

Говоря простым языком, коммутатор — это механизм, который позволяет нам соединять каналы для формирования более крупной сети. Коммутатор — это многовходовое, многовыходное устройство, которое передает пакеты от входа к одному или нескольким выходам. Таким образом, коммутатор добавляет звездную топологию (см. рис. 3.1) в набор возможных сетевых структур. Звездная топология обладает несколькими привлекательными свойствами:

- Несмотря на то, что коммутатор имеет фиксированное количество входов и выходов, что ограничивает количество хостов, которые можно подключить к одному коммутатору, крупные сети можно построить, объединяя несколько коммутаторов.
- Мы можем подключать коммутаторы друг к другу и к хостам, используя соединения «точка-точка», что обычно означает возможность создания сетей на обширной географической области.
- Добавление нового хоста к сети путем подключения его к коммутатору не всегда снижает производительность сети для других уже подключенных хостов.

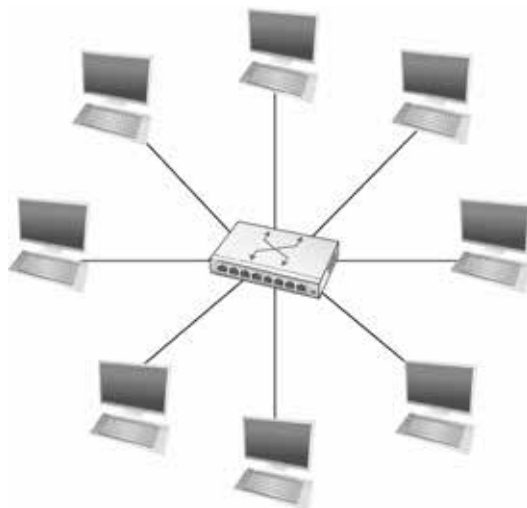


Рисунок 3.1. Коммутатор обеспечивает топологию «звезда».

Последнее утверждение не может быть однозначно для сетей с разделяемой средой, обсужденных ранее. Например, невозможно, чтобы два хоста на одном сегменте Ethernet с пропускной способностью 10 Мбит/с непрерывно передавали данные на скорости 10 Мбит/с, потому что они делят один и тот же передающий медиум. Каждый хост в коммутируемой сети имеет свое собственное соединение с коммутатором, поэтому вполне возможно, что многие хосты смогут передавать данные на полной скорости соединения (пропускная способность), при условии, что коммутатор спроектирован с достаточной совокупной пропускной способностью. Обеспечение высокой совокупной пропускной способности является одной из целей при проектировании коммутатора; мы вернемся к этой теме позже. В общем, коммутируемые сети считаются более *масштабируемыми* (то есть способными расти до большого числа узлов), чем сети с разделяемой средой, из-за их способности поддерживать множество хостов на полной скорости.

Плотное мультиплексирование с разделением по длине волны

- Наше внимание к пакетным коммутируемым сетям затмевает тот факт, что, особенно
- в глобальных сетях, подлежащая физической транспортной сети является полностью
- оптической: пакеты отсутствуют. На этом уровне коммерчески доступное оборудование

- DWDM (*плотное мультиплексирование с разделением по длине волны* (Dense Wavelength Division Multiplexing)) способно передавать большое количество оптических длин волн (colors) по одному волокну. Например, можно передавать данные на 100 и более разных длин волн, и каждая длина волны может нести до 100 Гбит/с данных.
- Соединение этих волокон осуществляется оптическим устройством, называемым ROADM (*перестраиваемый оптический мультиплексор ввода/вывода* (Reconfigurable Optical Add/Drop Multiplexers)). Коллекция ROADM (узлов) и волокон (каналов) формирует оптическую транспортную сеть, где каждый ROADM способен перенаправлять отдельные длины волн по многопереходному пути, создавая логическую сквозную цепь.
- С точки зрения пакетной коммутируемой сети, которая может быть построена на основе этой оптической транспортной сети, одна длина волны, даже если она проходит через несколько ROADM, представляется как одно соединение «точка-точка» между двумя коммутаторами, по которому можно передавать данные, используя протоколы кадрирования, такие как SONET или Ethernet, со скоростью 100 Гбит/с. Функция реконфигурации ROADM означает, что можно изменить эти подлежащие сквозные длины волн, эффективно создавая новую топологию на уровне пакетной коммутируемой сети.

Коммутатор подключен к набору каналов и для каждого из этих каналов использует соответствующий протокол канального уровня для связи с узлом на другом конце канала. Основная задача коммутатора — принимать входящие пакеты на одном из своих каналов и передавать их по какому-то другому каналу. Эта функция иногда называется либо *коммутацией*, либо *маршрутизацией*, и в терминах архитектуры Open Systems Interconnection (OSI) это основная функция сетевого уровня, также известного как *Уровень 2*.

Вопрос тогда заключается в том, как коммутатор решает, на какой выходной канал направить каждый пакет? Общий ответ состоит в том, что он смотрит на заголовок пакета, чтобы найти идентификатор, который он использует для принятия решения. Детали использования этого идентификатора различаются, но существуют два распространенных подхода. Первый — это *дейтаграммный* или *бесконтактный* подход. Второй — это *виртуальная цепь* или *контактный* подход. Третий подход, *маршрутизация по источнику*, менее распространен, чем первые два, но имеет свои полезные приложения.

Одна общая черта для всех сетей заключается в том, что нам нужно иметь способ идентификации конечных узлов. Такие идентификаторы обычно называются адресами. Мы уже видели примеры адресов, такие как 48-битный адрес, используемый для Ethernet. Единственное требование к адресам Ethernet заключается в том, чтобы ни у одного из узлов в сети не было одинаковых адресов. Это достигается за счет того, что все Ethernet-карты получают *глобально уникальный идентификатор*. В дальнейшем обсуждении мы предполагаем, что у каждого хоста есть глобально уникальный адрес. Позже мы рассмотрим другие полезные свойства, которыми может обладать адрес, но для начала глобальной уникальности достаточно.

Еще одно предположение, которое нам нужно сделать, заключается в том, что существует способ идентификации входных и выходных портов каждого коммутатора. Есть по крайней мере два разумных способа идентификации портов: один из них — это нумерация каждого порта, другой — это идентификация порта по имени узла (коммутатора или хоста), к которому он ведет. Сейчас мы будем использовать нумерацию портов.

Глава 3.1.1. Дейтаграммы

Идея, лежащая в основе дейтаграмм, невероятно проста: в каждый пакет включается достаточно информации, чтобы любой коммутатор мог решить, как доставить его до места назначения. То есть каждый пакет содержит полный адрес назначения. Рассмотрим пример сети, показанной на рис. 3.2, в которой хосты имеют адреса А, В, С и так далее. Чтобы решить, как переслать пакет, коммутатор обращается к *таблице маршрутизации* (иногда называемой *таблицей пересылки*), пример которой приведен

в табл. 3.1. Эта конкретная таблица показывает информацию, необходимую коммутатору 2 для пересылки дейтаграмм в сети. Довольно легко составить такую таблицу, имея полную карту простой сети, как показано здесь; можно представить, что оператор сети настраивает таблицы статически. Намного сложнее создать таблицы маршрутизации в больших, сложных сетях с динамически меняющейся топологией и множеством путей между пунктами назначения. Эта более сложная задача известна как *маршрутизация* и будет рассмотрена в следующей главе. Мы можем думать о маршрутизации как о процессе, который происходит в фоновом режиме, чтобы, когда появляется дата-пакет, у нас была правильная информация в таблице маршрутизации для его пересылки или коммутации.

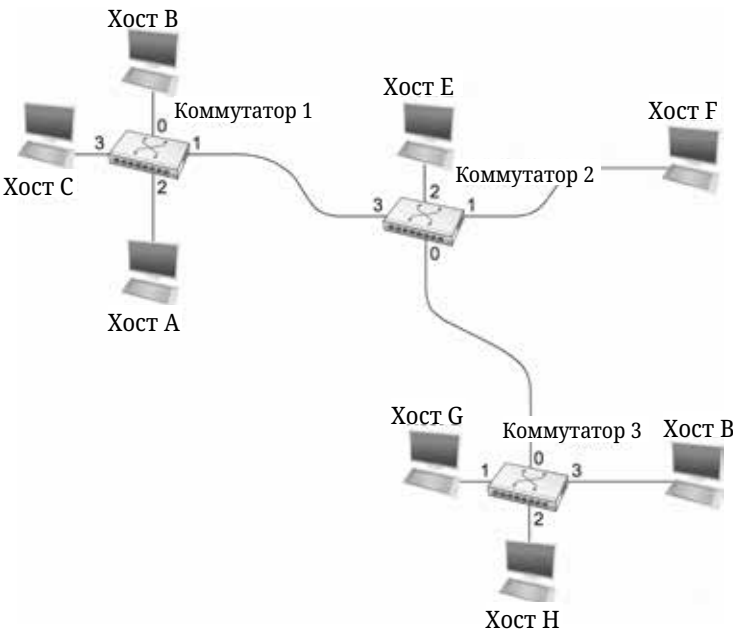


Рисунок 3.2. Пересылка дейтаграмм: пример сети.

Таблица 3.1.
Таблица переадресации для коммутатора 2.

Пункт назначения	Порт
A	3
B	0
C	3
D	3
E	2
F	1
G	0
H	0

Дейтаграммные сети имеют следующие характеристики:

- Хост может отправить пакет в любое время и в любое место, так как любой пакет, появившийся на коммутаторе, может быть немедленно переслан (при условии корректно заполненной таблицы маршрутизации). По этой причине дейтаграммные сети часто называются *бесконтактными*; это контрастирует с *контактными* сетями, описанными ниже, в которых необходимо установить некоторое *состояние соединения* перед отправкой первого дата-пакета.
- Когда хост отправляет пакет, он не знает, сможет ли сеть его доставить или работает ли хост назначения.
- Каждый пакет пересылается независимо от предыдущих пакетов, которые могли быть отправлены в тот же пункт назначения. Таким образом, два последовательных пакета от хоста А к хосту В могут следовать совершенно разными путями (возможно, из-за изменения таблицы маршрутизации на каком-то коммутаторе в сети).
- Сбой коммутатора или канала может не иметь серьезных последствий для связи, если можно найти альтернативный маршрут вокруг сбоя и обновить таблицу маршрутизации соответствующим образом.

Этот последний факт особенно важен для истории дейтаграммных сетей. Одной из важных целей проектирования Интернета является устойчивость к сбоям, и история показала, что он достаточно эффективно справляется с этой задачей. Поскольку сети на основе дейтаграмм являются доминирующей технологией, обсуждаемой в этой книге, мы отложим иллюстративные примеры на следующие главы и перейдем к двум основным альтернативам.

Глава 3.1.2. Коммутация виртуальных каналов

Вторая техника коммутации пакетов использует концепцию *виртуальной цепи* (virtual circuit, VC). Этот подход, также называемый *моделью, ориентированной на соединения*, требует настройки виртуального соединения от исходного хоста до конечного хоста перед отправкой любых данных. Чтобы понять, как это работает, рассмотрим рис. 3.3, где хост А снова хочет отправить пакеты хосту В. Это можно представить как двухэтапный процесс. Первый этап — это установка соединения. Второй — передача данных. Рассмотрим каждый этап поочередно.

На этапе установки соединения необходимо установить состояние соединения в каждом коммутаторе между исходным и конечным хостами. Состояние соединения для одного соединения состоит из записи в таблице VC в каждом коммутаторе, через который проходит соединение. Одна запись в таблице VC на одном коммутаторе содержит:

- *Идентификатор виртуальной цепи* (virtual circuit identifier, VCI), который уникально идентифицирует соединение в этом коммутаторе и который будет включен в заголовок пакетов, принадлежащих этому соединению.
- Входящий интерфейс, на котором пакеты для этого VC прибывают на коммутатор.
- Исходящий интерфейс, через который пакеты для этого VC покидают коммутатор.
- Потенциально другой VCI, который будет использоваться для исходящих пакетов.

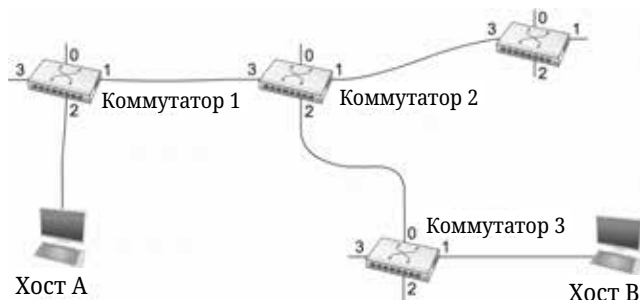


Рисунок 3.3. Пример виртуальной сети.

Семантика такой записи следующая: если пакет поступает на указанный входящий интерфейс и содержит указанное значение VCI в своем заголовке, то этот пакет должен быть отправлен через указанный исходящий интерфейс с указанным исходящим значением VCI, предварительно помещенным в заголовок.

Обратите внимание, что комбинация VCI-пакетов, когда они принимаются на коммутаторе, и интерфейса, на котором они принимаются, уникально идентифицирует виртуальное соединение. Конечно, в коммутаторе может быть установлено множество виртуальных соединений одновременно. Также мы наблюдаем, что входящие и исходящие значения VCI обычно не совпадают. Таким образом, VCI не является глобально значимым идентификатором для соединения; он имеет значение только на данном канале (то есть имеет локальную область действия канала).

Когда создается новое соединение, необходимо назначить новый VCI для этого соединения на каждом канале, который будет проходить соединение. Также необходимо убедиться, что в настоящее время выбранный VCI на данном канале не используется каким-либо существующим соединением.

Существует два основных подхода к установлению состояния соединения. Один из них — это конфигурация состояния сетевым администратором, в этом случае виртуальная цепь является постоянной. Конечно, она также может быть удалена администратором, поэтому постоянная виртуальная цепь (permanent virtual circuit, PVC) скорее всего лучше всего воспринимается как долго живущее или административно настроенное VC. Альтернативно хост может отправить сообщения в сеть, чтобы установить состояние. Это называется сигнализацией, и полученные виртуальные цепи называются коммутируемыми. Основная характеристика коммутируемой виртуальной цепи (SVC) заключается в том, что хост может динамически устанавливать и удалять такое VC без участия сетевого администратора. Обратите внимание, что SVC точнее следует называть *сигнализируемой* VC, так как именно использование сигнализации (а не коммутации) отличает SVC от PVC.

Допустим, сетевой администратор хочет вручную создать новое виртуальное соединение от хоста А к хосту В. Сначала администратору нужно определить путь через сеть от А до В. В примере сети на рис. 3.3 существует только один такой путь, но в общем случае это может быть не так. Затем администратор выбирает значение VCI, которое в данный момент не используется на каждом канале для соединения. В нашем примере предположим, что значение VCI 5 выбрано для канала от хоста А к коммутатору 1, а значение 11 выбрано для канала от коммутатора 1 к коммутатору 2. В этом случае коммутатор 1 должен иметь запись в своей таблице VC, настроенную так, как показано в табл. 3.2.

Таблица 3.2.
Пример записи в таблице виртуальных цепей для коммутатора 1.

Входящий интерфейс	Входящий VCI	Исходящий интерфейс	Исходящий VCI
2	5	1	11

Аналогично предположим, что значение VCI 7 выбрано для идентификации этого соединения на канале от коммутатора 2 к коммутатору 3, а значение VCI 4 выбрано для канала от коммутатора 3 к хосту В. В этом случае коммутаторы 2 и 3 должны быть настроены с записями в таблицах VC, как показано в таблице 3.3 и таблице 3.4 соответственно. Обратите внимание, что «исходящее» значение VCI на одном коммутаторе является «входящим» значением VCI на следующем коммутаторе.

Таблица 3.3.
Запись таблицы виртуальных цепей на коммутаторе 2.

Входящий интерфейс	Входящий VCI	Исходящий интерфейс	Исходящий VCI
3	11	2	7

Таблица 3.4.

Запись таблицы виртуальных цепей на коммутаторе 3.

Входящий интерфейс	Входящий VCI	Исходящий интерфейс	Исходящий VCI
0	7	1	4

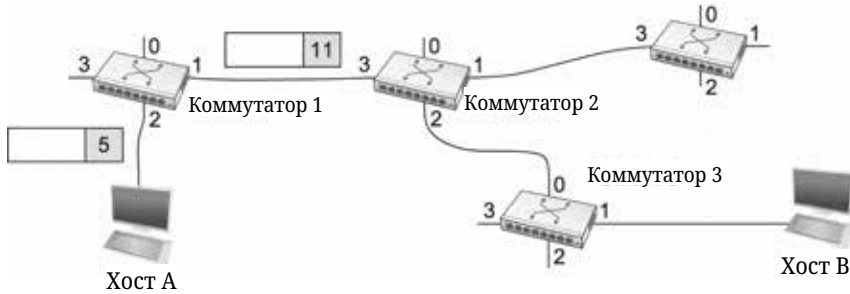


Рисунок 3.4. Пакет отправляется в виртуальную сеть.

После того как таблицы VC настроены, можно приступить к этапу передачи данных, как показано на рис. 3.4. Для любого пакета, который хост А хочет отправить хосту В, он помещает значение VCI 5 в заголовок пакета и отправляет его коммутатору 1. Коммутатор 1 принимает такой пакет на интерфейсе 2 и использует комбинацию интерфейса и значения VCI в заголовке пакета, чтобы найти соответствующую запись в таблице VC. Как показано в табл. 3.2, запись таблицы в данном случае указывает коммутатору 1 переслать пакет через интерфейс 1 и поместить значение VCI 11 в заголовок при отправке пакета. Таким образом, пакет поступит в коммутатор 2 на интерфейсе 3 со значением VCI 11. Коммутатор 2 ищет интерфейс 3 и значение VCI 11 в своей таблице VC (как показано в табл. 3.3) и отправляет пакет в коммутатор 3, обновив значение VCI в заголовке пакета соответственно, как показано на рис. 3.5. Этот процесс продолжается, пока пакет не придет к хосту В со значением VCI 4 в пакете. Для хоста В это значение идентифицирует пакет как пришедший от хоста А.

В реальных сетях разумного размера бремя корректной настройки таблиц VC в большом количестве коммутаторов быстро стало бы чрезмерным при использовании описанных выше процедур. Таким образом, либо инструмент управления сетью, либо какой-то вид сигнализации (или оба этих метода) почти всегда используется, даже при настройке «постоянных» VC. В случае PVC сигнализация инициируется сетевым администратором, тогда как SVC обычно настраиваются с использованием сигнализации одним из хостов. Теперь рассмотрим, как можно настроить тот же VC, описанный выше, с помощью сигнализации от хоста.

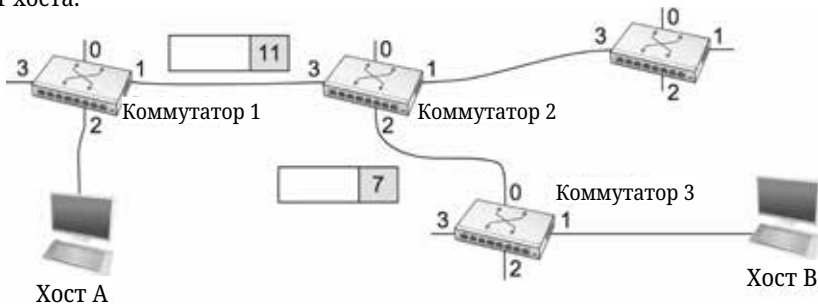


Рисунок 3.5. Пакет проходит путь через виртуальную сеть.

Чтобы начать процесс сигнализации, хост А отправляет сообщение о настройке в сеть — то есть на коммутатор 1. Сообщение о настройке содержит, среди прочего, полный адрес назначения хоста В. Сообщение о настройке должно дойти до хоста В, чтобы создать необходимое состояние соединения в каждом коммутаторе на пути. Мы можем увидеть, что доставка сообщения о настройке до хоста В похожа на доставку дейтаграммы до В, так как коммутаторы должны знать, какой выход использовать для отправки сообщения о настройке, чтобы оно в конечном итоге достигло В. Пока предположим, что коммутаторы достаточно осведомлены о топологии сети, чтобы понять, как это сделать, чтобы сообщение о настройке прошло через коммутаторы 2 и 3 и, наконец, достигло хоста В.

Когда коммутатор 1 получает запрос на соединение, помимо отправки его на коммутатор 2 он создает новую запись в своей таблице виртуальных цепей для этого нового соединения. Эта запись точно такая же, как показано ранее в табл. 3.2. Главное отличие заключается в том, что теперь задача назначения неиспользуемого значения VCI на интерфейсе выполняется коммутатором для этого порта. В этом примере коммутатор выбирает значение 5. Таблица виртуальных цепей теперь содержит следующую информацию: «Когда пакеты поступают на порт 2 с идентификатором 5, отправляйте их на порт 1». Другой вопрос заключается в том, что хост А каким-то образом должен узнать, что он должен вставлять значение VCI 5 в пакеты, которые он хочет отправить хосту В; далее мы увидим, как это происходит.

Когда коммутатор 2 получает сообщение о настройке, он выполняет аналогичный процесс; в этом примере он выбирает значение 11 в качестве входящего значения VCI. Аналогично коммутатор 3 выбирает значение 7 в качестве входящего VCI. Каждый коммутатор может выбирать любое число, какое он захочет, при условии, что это число в данный момент не используется для другого соединения на этом порту данного коммутатора. Как отмечалось выше, VCI имеют локальную значимость для канала; то есть они не имеют глобального значения.

Наконец, сообщение о настройке поступает к хосту В. Предполагая, что В работает исправно и готов принять соединение от хоста А, он также выделяет входящее значение VCI, в данном случае 4. Это значение VCI может быть использовано В для идентификации всех пакетов, поступающих от хоста А.

Теперь, чтобы завершить соединение, всем участникам нужно сообщить, какое значение VCI для этого соединения использует их следующий узел вниз по потоку. Хост В отправляет подтверждение настройки соединения коммутатору 3 и включает в это сообщение VCI, которое он выбрал (4). Теперь коммутатор 3 может завершить запись в таблице виртуальных цепей для этого соединения, так как он знает, что исходящее значение должно быть 4. Коммутатор 3 отправляет подтверждение дальше коммутатору 2, указывая VCI 7. Коммутатор 2 отправляет сообщение коммутатору 1, указывая VCI 11. Наконец, коммутатор 1 передаст подтверждение хосту А, указывая ему использовать VCI 5 для этого соединения.

На данном этапе все элементы имеют необходимую информацию для того, чтобы трафик мог проходить от хоста А к хосту В. Каждый коммутатор имеет полную запись в таблице виртуальных цепей для этого соединения. Более того, хост А имеет четкое подтверждение, что все готово до самого хоста В. В этот момент записи таблиц соединений установлены во всех трех коммутаторах так же, как в примере с административно настроенным соединением, но весь процесс произошел автоматически в ответ на сообщение о сигнализации, отправленное от А. Фаза передачи данных теперь может начаться и идентична используемой в случае PVC.

Когда хост А больше не хочет отправлять данные хосту В, он разрывает соединение, отправляя сообщение о разрыве коммутатору 1. Коммутатор удаляет соответствующую запись из своей таблицы и пересылает сообщение другим коммутаторам на пути, которые аналогично удаляют соответствующие записи из своих таблиц. В этот момент, если хост А отправит пакет с VCI 5 на коммутатор 1, он будет отброшен, как если бы соединение никогда не существовало.

Есть несколько вещей, которые нужно отметить относительно коммутации виртуальных каналов:

- Поскольку хосту А нужно дождаться, пока запрос на соединение достигнет другой стороны сети и вернется, прежде чем он сможет отправить свой первый пакет данных, существует задержка как минимум на время одного полного кругового пути (RTT) до отправки данных.
- В то время как запрос на соединение содержит полный адрес для хоста В (который может быть довольно большим, являясь глобальным идентификатором в сети), каждый пакет данных содержит только небольшой идентификатор, который уникален лишь на одном канале. Таким образом, накладные расходы на пакет, вызванные заголовком, сокращаются по сравнению с моделью дейтаграмм. Важно, что поиск происходит быстро, так как номер виртуальной цепи может рассматриваться как индекс в таблице, а не как ключ, который нужно искать.
- Если коммутатор или канал в соединении выходит из строя, соединение разрывается, и потребуются установить новое. Также старое соединение нужно разорвать, чтобы освободить место в таблице коммутаторов.
- Вопрос о том, как коммутатор решает, на какой канал пересылать запрос на соединение, был упущен. По сути, это та же проблема, что и создание таблицы маршрутизации для пересылки дейтаграмм, которая требует какого-то *алгоритма маршрутизации*. Маршрутизация описана далее, и алгоритмы, описанные там, обычно применимы как для настройки запросов на маршрутизацию, так и для дейтаграмм.

Одним из приятных аспектов виртуальных цепей является то, что к моменту, когда хост получает разрешение на отправку данных, он уже знает много о сети — например, что действительно существует маршрут к получателю и что получатель готов и способен принимать данные. Также возможно выделить ресурсы для виртуальной цепи в момент ее установки. Например, X.25 (ранняя и сейчас в значительной степени устаревшая технология сетей, основанная на виртуальных цепях) использовала следующую стратегию, состоящую из трех частей:

1. Буферы выделяются для каждой виртуальной цепи при ее инициализации.
2. Протокол «скользящего окна» запускается между каждой парой узлов вдоль виртуальной цепи, и этот протокол дополнен управлением потоком, чтобы отправляющий узел не перегружал буферы, выделенные на принимающем узле.
3. Цепь отклоняется данным узлом, если при обработке сообщения о запросе на соединение недостаточно буферов на этом узле.

Выполняя эти три действия, каждый узел обеспечивается необходимыми буферами для очереди пакетов, поступающих по этой цепи. Эта базовая стратегия обычно называется *управлением потоком по принципу «хоп за хопом»* (hop-by-hop).

Для сравнения, сеть дейтаграмм не имеет фазы установления соединения, и каждый коммутатор обрабатывает каждый пакет независимо, что делает менее очевидным, как сеть дейтаграмм могла бы выделять ресурсы значимым образом. Вместо этого каждый прибывающий пакет конкурирует со всеми другими пакетами за место в буфере. Если свободных буферов нет, входящий пакет должен быть отброшен. Мы отмечаем, однако, что даже в сети на основе дейтаграмм исходный хост часто отправляет последовательность пакетов тому же конечному хосту. Возможно, чтобы каждый коммутатор различал набор пакетов, которые он в настоящее время имеет в очереди, на основе пары источник/получатель, и таким образом коммутатор может обеспечить, чтобы пакеты, принадлежащие каждой паре источник/получатель, получали справедливую долю буферов коммутатора.

В модели виртуальной цепи мы можем представить, что каждая цепь получает разное *качество обслуживания* (quality of service, QoS). В этом контексте термин «*качество обслуживания*» обычно подразумевает, что сеть предоставляет пользователю ка-

кой-то вид гарантии, связанной с производительностью, что, в свою очередь, означает, что коммутаторы выделяют необходимые ресурсы для выполнения этой гарантии. Например, коммутаторы вдоль данной виртуальной цепи могут выделить процент пропускной способности каждого исходящего канала для этой цепи. Как другой пример, последовательность коммутаторов может обеспечить, чтобы пакеты, принадлежащие определенной цепи, не задерживались (не ставились в очередь) более чем на определенное время.

За эти годы было множество успешных примеров технологий виртуальных цепей, особенно X.25, Frame Relay и асинхронный режим передачи данных (Asynchronous Transfer Mode, ATM). Однако после успеха модели без соединений Интернета ни одна из них сегодня не пользуется большой популярностью. Одним из самых распространенных применений виртуальных цепей на протяжении многих лет было создание *виртуальных частных сетей* (virtual private networks, VPN), обсуждаемое далее. Даже ее применение сейчас в основном поддерживается с использованием интернет-технологий.

Асинхронный режим передачи (ATM)

Асинхронный режим передачи (ATM, Asynchronous Transfer Mode), вероятно, является самой известной сетевой технологией, основанной на виртуальных цепях, хотя сейчас она уже пережила свой пик в плане развертывания. ATM стала важной технологией в 1980-х и начале 1990-х годов по разным причинам, не в последнюю очередь потому, что она была принята телефонной индустрией, которая в то время была менее активна в компьютерных сетях (за исключением поставки каналов связи, на основе которых другие строили сети). ATM также оказалась в нужное время в нужном месте как технология высокоскоростной коммутации, появившаяся на сцене, когда такие разделяемые среды, как Ethernet и Token Ring, начали казаться слишком медленными для многих пользователей компьютерных сетей. В некотором смысле ATM конкурировала с Ethernet-коммутацией и воспринималась многими как конкурент IP.

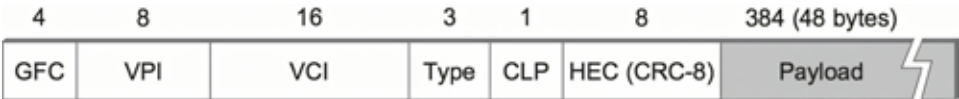


Рисунок 3.6. Формат ячейки ATM в UNI.

Подход ATM имеет некоторые интересные свойства, поэтому он будет более детально рассмотрен. Формат пакета ATM, более известного как ячейка ATM, показан на рис. 3.6 и иллюстрирует основные моменты. Мы пропустим биты общего управления потоком (GFC), которые никогда не использовались широко, и начнем с 24 битов, обозначенных как VPI (идентификатор виртуального пути — 8 битов) и VCI (идентификатор виртуальной цепи — 16 битов). Если рассматривать эти биты вместе как одно 24-битовое поле, они соответствуют идентификатору виртуальной цепи, введенному ранее. Причина разделения поля на две части заключается в обеспечении уровня иерархии: все цепи с одинаковым VPI могли в некоторых случаях рассматриваться как группа (виртуальный путь) и могли переключаться вместе, используя только VPI, что упрощало работу коммутатора, который мог игнорировать все биты VCI, и значительно уменьшало размер таблицы VC.

Перейдя к последнему байту заголовка, мы найдем 8-битную циклическую избыточную проверку (CRC), известную как *проверка ошибки заголовка* (header error check, HEC). Она использует CRC-8 и обеспечивает возможность обнаружения ошибок и коррекцию одноразрядных ошибок только в заголовке ячейки. Защита заголовка ячейки особенно важна, потому что ошибка в VCI приведет к неправильной доставке ячейки.

Вероятно, наиболее значительным аспектом ячейки ATM и причиной того, почему она называется ячейкой, а не пакетом, является ее фиксированный размер: 53 байта.

Почему это так? Одна из основных причин заключается в упрощении реализации аппаратных коммутаторов. Когда АТМ создавалась в середине и конце 1980-х годов, Ethernet со скоростью 10 Мбит/с был передовой технологией с точки зрения скорости. Для значительного увеличения скорости большинство людей думали в терминах аппаратного обеспечения. Кроме того, в телефонном мире масштабные коммутаторы являются привычным явлением — телефонные коммутаторы часто обслуживают десятки тысяч клиентов. Пакеты фиксированной длины оказываются очень полезными, если вы хотите строить быстрые, высокомасштабируемые коммутаторы. Существуют две основные причины для этого:

1. Проще создавать оборудование для выполнения простых задач, и задача обработки пакетов становится проще, когда вы заранее знаете длину каждого из них.
2. Если все пакеты имеют одинаковую длину, вы можете использовать множество коммутирующих элементов, выполняющих одну и ту же работу параллельно, при этом каждому из них потребуется одинаковое время для выполнения своей задачи.

Вторая причина, а именно возможность параллелизма, значительно улучшает масштабируемость проектирования коммутаторов. Было бы преувеличением утверждать, что быстрые параллельные аппаратные коммутаторы могут быть построены только с использованием ячеек фиксированной длины. Однако несомненно, что ячейки облегчают задачу построения такого оборудования, и в то время, когда стандарты АТМ были определены, существовало много знаний о том, как строить аппаратные коммутаторы для ячеек. Оказывается, этот же принцип до сих пор применяется во многих коммутаторах и маршрутизаторах, даже если они работают с пакетами переменной длины — такие пакеты разрезаются на ячейки для их пересылки от входного порта к выходному порту, но это все происходит внутри коммутатора.

Существует еще один аргумент в пользу маленьких ячеек АТМ, связанный с задержкой от конца до конца. АТМ была разработана для передачи как голосовых телефонных звонков (основное применение в то время), так и данных. Поскольку голос занимает малую полосу пропускания, но имеет строгие требования к задержке, последняя вещь, которую вы хотите, — это чтобы маленький голосовой пакет стоял в очереди за большим пакетом данных на коммутаторе. Если заставить все пакеты быть маленькими (то есть размером с ячейку), то большие пакеты данных все равно можно будет поддерживать путем повторной сборки набора ячеек в пакет, и вы получите преимущество в возможности чередовать пересылку голосовых ячеек и ячеек данных на каждом коммутаторе по пути от источника к получателю. Эта идея использования маленьких ячеек для уменьшения времени задержки от конца до конца сегодня активно применяется в сотовых сетях доступа.

После принятия решения об использовании маленьких пакетов фиксированной длины следующий вопрос заключался в том, какой длины эти пакеты должны быть. Если сделать их слишком короткими, то количество заголовочной информации, которая должна передаваться по отношению к количеству данных, помещающихся в одну ячейку, становится больше, и процент полосы пропускания канала, используемый для передачи данных, снижается. Еще более серьезной проблемой является то, что если вы строите устройство, обрабатывающее ячейки с максимальной скоростью определенного числа ячеек в секунду, то по мере сокращения длины ячеек общая скорость передачи данных падает пропорционально длине ячейки. Примером такого устройства может быть сетевой адаптер, который собирает ячейки в более крупные единицы перед передачей их на хост. Производительность такого устройства напрямую зависит от размера ячейки. С другой стороны, если сделать ячейки слишком большими, то возникает проблема неэффективного использования полосы пропускания из-за необходимости добавлять данные для заполнения полной ячейки. Если размер полезной нагрузки ячейки составляет 48 байт, а вам нужно передать 1 байт, то потре-

буется отправить 47 байт заполнителя. Если это происходит часто, то использование канала будет неэффективным. Сочетание относительно высокого соотношения заголовка к полезной нагрузке и частого отправления частично заполненных ячеек действительно привело к заметной неэффективности в ATM-сетях, которую некоторые критики называли «налогом на ячейки».

В итоге размер 48 байт был выбран для полезной нагрузки ячейки ATM как компромисс. Были веские аргументы в пользу как больших, так и маленьких ячеек, и размер 48 байт не устроил почти никого — степень двойки, безусловно, была бы лучше для обработки компьютерами.

Глава 3.1.3. Маршрутизация по источнику (Source Routing)

Третий подход к коммутации, не использующий ни виртуальных каналов, ни традиционных дейтаграмм, известен как *маршрутизация по источнику*. Название происходит от того, что вся информация о топологии сети, необходимая для передачи пакета через сеть, предоставляется исходным хостом.

Существует несколько способов реализации маршрутизации по источнику. Один из них заключается в присвоении номера каждому выходу каждого коммутатора и размещении этого номера в заголовке пакета. Функция коммутации тогда становится очень простой: для каждого пакета, прибывающего на вход, коммутатор читает номер порта в заголовке и передает пакет на этот выход. Однако, поскольку обычно на пути между отправляющим и принимающим хостами будет больше одного коммутатора, заголовков пакета должен содержать достаточно информации, чтобы каждый коммутатор на пути мог определить, на какой выход нужно направить пакет. Один из способов сделать это — поместить упорядоченный список портов коммутаторов в заголовок и вращать список так, чтобы следующий коммутатор в пути всегда был в начале списка. Рис. 3.7 иллюстрирует эту идею.

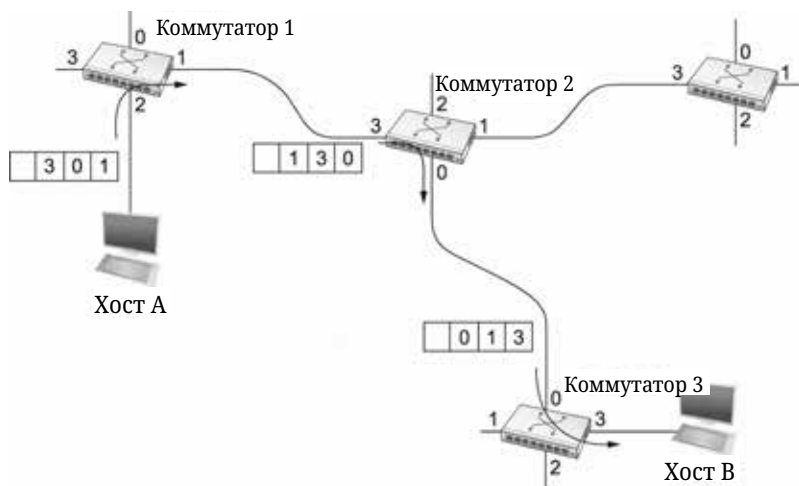


Рисунок 3.7. Маршрутизация источника в коммутируемой сети (где коммутатор считывает крайний правый номер).

В этом примере пакет должен пройти через три коммутатора, чтобы добраться от хоста А до хоста В. На коммутаторе 1 он должен выйти на порту 1, на следующем коммутаторе он должен выйти на порту 0, а на третьем коммутаторе — на порту 3. Таким образом, исходный заголовок, когда пакет покидает хост А, содержит список портов (3, 0, 1), где мы предполагаем, что каждый коммутатор читает самый правый

элемент списка. Чтобы убедиться, что следующий коммутатор получает соответствующую информацию, каждый коммутатор вращает список после того, как прочитал свою запись. Таким образом, заголовок пакета, когда он покидает коммутатор 1 по пути к коммутатору 2, теперь (1, 3, 0); коммутатор 2 выполняет еще одно вращение и отправляет пакет с заголовком (0, 1, 3). Хотя это и не показано, коммутатор 3 выполняет еще одно вращение, восстанавливая заголовок до того вида, который он имел, когда его отправлял хост А.

Есть несколько моментов, которые стоит отметить относительно этого подхода. Во-первых, предполагается, что хост А знает достаточно о топологии сети, чтобы сформировать заголовок, содержащий все правильные направления для каждого коммутатора на пути. Это отчасти аналогично задаче построения таблиц маршрутизации в дейтаграммной сети или определения, куда отправить пакет установки в сети с виртуальными каналами. На практике, однако, это первый коммутатор на входе в сеть (а не конечный хост, подключенный к этому коммутатору), который добавляет маршрут по источнику.

Во-вторых, обратите внимание, что мы не можем предсказать, насколько большим должен быть заголовок, поскольку он должен содержать одно слово информации для каждого коммутатора на пути. Это подразумевает, что заголовки, вероятно, имеют переменную длину без верхнего предела, если мы не можем с абсолютной уверенностью предсказать максимальное количество коммутаторов, через которые когда-либо придется пройти пакету.

В-третьих, существуют некоторые вариации этого подхода. Например, вместо того чтобы вращать заголовок, каждый коммутатор может просто удалять первый элемент по мере его использования. Вращение имеет преимущество перед удалением: хост В получает копию полного заголовка, что может помочь ему определить, как вернуться к хосту А. Еще одна альтернатива — заголовок может содержать указатель на текущий «следующий порт», так что каждый коммутатор просто обновляет указатель, а не вращает заголовок; это может быть более эффективно в реализации. Мы показываем эти три подхода на рис. 3.8. В каждом случае запись, которую должен прочитать этот коммутатор, — это А, а запись, которую должен прочитать следующий коммутатор, — это В.

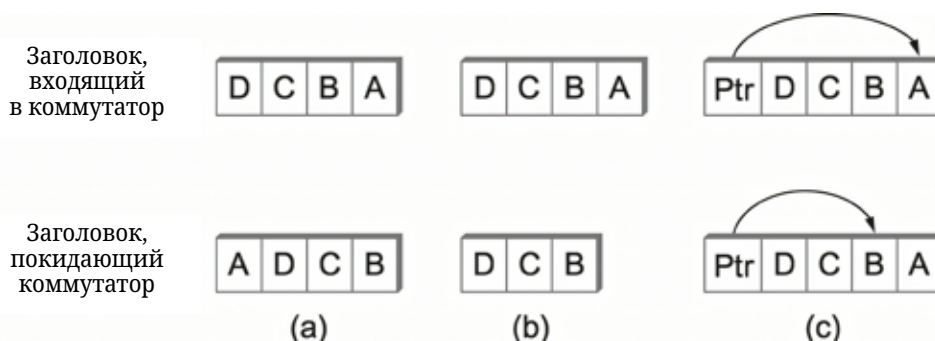


Рисунок 3.8. Три способа обработки заголовков для маршрутизации источника:

(a) поворот; (b) зачистка; (c) указатель. Метки читаются справа налево.

Маршрутизация по источнику может использоваться как в дейтаграммных сетях, так и в сетях с виртуальными каналами. Например, протокол Интернет (Internet Protocol), который является дейтаграммным протоколом, включает опцию маршрутизации по источнику, позволяющую отдельным пакетам маршрутизироваться по источнику, в то время как большинство пакетов передаются как обычные дейтаграммы. Маршрутизация по источнику также используется в некоторых сетях с виртуальными каналами как средство для передачи начального запроса на установление соединения по пути от источника к получателю.

Маршруты по источнику иногда делятся на *строгие* и *гибкие*. В строгом маршруте по источнику каждый узел на пути должен быть указан, тогда как гибкий маршрут по источнику указывает только набор узлов, через которые необходимо пройти, не указывая точно, как добраться от одного узла до другого. Гибкий маршрут по источнику можно рассматривать как набор путевых точек, а не полностью определенный маршрут. Опция гибкого маршрута может быть полезна для ограничения объема информации, которую должен получить источник для создания маршрута по источнику. В любой достаточно большой сети источнику, вероятно, будет трудно получить всю необходимую информацию о пути для создания корректного строгого маршрута по источнику к любому месту назначения. Однако оба типа маршрутов по источнику находят применение в определенных сценариях, как мы увидим далее.

Глава 3.2. Коммутируемый Ethernet

Рассмотрев основные идеи коммутации, мы теперь сосредоточимся более подробно на конкретной технологии коммутации: *коммутируемый Ethernet*. Коммутаторы, используемые для построения таких сетей, которые часто называют *коммутаторами уровня 2 (L2)*, широко используются в университетских и корпоративных сетях. Исторически их чаще называли *мостами*, поскольку они использовались для «*мостового соединения*» сегментов Ethernet с целью создания *расширенной локальной сети (LAN)*. Однако сегодня большинство сетей используют Ethernet в конфигурации «точка-точка», а эти связи соединены L2-коммутаторами для формирования коммутируемого Ethernet.

Далее мы начнем с исторической перспективы использования мостов для соединения набора сегментов Ethernet, а затем перейдем к перспективе, широко используемой сегодня (использование L2-коммутаторов для соединения набора связей «точка-точка»). Но независимо от того, называем ли мы устройство мостом или коммутатором — и сеть, которую вы строите, расширенной LAN или коммутируемым Ethernet, — оба работают одинаково.

Для начала предположим, что у вас есть пара сетей Ethernet, которые вы хотите соединить. Один из подходов, который вы можете попробовать, это поставить повторитель между ними. Однако это не будет рабочим решением, если при этом будут превышены физические ограничения Ethernet. (Напомним, что разрешено не более двух повторителей между любой парой хостов и не более 2500 м в длину.) Альтернативой было бы поставить узел с парой Ethernet-адаптеров между двумя сетями Ethernet и заставить узел пересылать кадры из одной сети Ethernet в другую. Этот узел будет отличаться от повторителя, который работает с битами, а не с кадрами, и просто копирует полученные биты с одного интерфейса на другой. Вместо этого узел полностью реализует протоколы обнаружения коллизий и доступа к среде Ethernet на каждом интерфейсе. Следовательно, ограничения на длину и количество хостов в Ethernet, которые касаются управления коллизиями, не будут применяться к соединенной паре сетей Ethernet, подключенных таким образом. Это устройство работает в *режиме перехвата (promiscuous mode)*, принимая все кадры, передаваемые на любом из Ethernet, и пересылая их в другой.

В своих простейших вариантах мосты просто принимают кадры LAN на своих входах и пересылают их на все остальные выходы. Эта простая стратегия использовалась ранними мостами, но имела серьезные ограничения, как мы увидим ниже. Со временем было добавлено несколько усовершенствований, чтобы сделать мосты эффективным механизмом для соединения набора LAN. Остальная часть этой главы раскрывает более интересные детали.

Глава 3.2.1. Обучающийся мост

Первое улучшение, которое мы можем сделать для моста, заключается в том, что ему не обязательно пересылать все кадры, которые он получает. Рассмотрим мост на рис. 3.9. Всякий раз, когда кадр от хоста А, адресованный хосту В, поступает на порт 1, нет необхо-

димости, чтобы мост пересылал этот кадр через порт 2. Вопрос в том, как мост может узнать, на каком порту находятся различные хосты.

Один из вариантов — это вручную загрузить таблицу в мост, аналогичную той, что представлена в табл. 3.5. Тогда всякий раз, когда мост получает кадр на порту 1, адресованный хосту А, он не будет пересылать кадр на порт 2; в этом нет необходимости, так как хост А уже получил бы кадр непосредственно через подключение к LAN на порту 1. Всякий раз, когда кадр, адресованный хосту А, поступает на порт 2, мост пересылает кадр на порт 1.

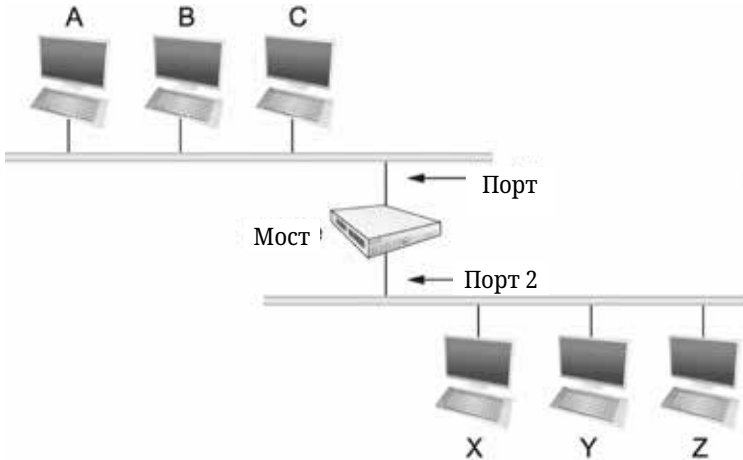


Рисунок 3.9. Иллюстрация обучающегося моста.

Таблица 3.5.

Таблица переадресации, поддерживаемая мостом.

Хост	Порт
A	1
B	1
C	1
X	2
Y	2
Z	2

Поддерживать эту таблицу вручную слишком трудно, и существует простой трюк, благодаря которому мост может самостоятельно обучаться. Идея состоит в том, чтобы каждый мост инспектировал *исходный адрес* во всех получаемых кадрах. Таким образом, когда хост А отправляет кадр хосту по любую сторону моста, мост получает этот кадр и фиксирует факт, что кадр от хоста А был только что получен на порту 1. Таким образом, мост может построить таблицу, аналогичную табл. 3.5.

Обратите внимание, что мост, использующий такую таблицу, реализует версию модели передачи дейтаграмм (или без соединения), описанную ранее. Каждый пакет несет глобальный адрес, и мост решает, на какой выходной порт отправить пакет, посмотрев этот адрес в таблице.

Когда мост впервые запускается, эта таблица пуста; записи добавляются со временем. Также с каждой записью связан тайм-аут, и мост удаляет запись после определенного периода времени. Это делается для защиты от ситуации, когда хост, а следовательно, и его LAN-адрес, перемещается с одной сети на другую. Таким образом, эта таблица не обязательно является полной. Если мост получает кадр, адресованный хосту, который в данный момент отсутствует в таблице, он пересылает кадр на все другие порты. Другими словами, эта таблица просто является оптимизацией, которая фильтрует некоторые кадры; она не требуется для корректной работы.

Глава 3.2.2. Реализация

Код, реализующий алгоритм обучающего моста, достаточно прост, и мы набросаем его здесь. Структура `BridgeEntry` определяет одну запись в таблице переадресации моста; эти записи хранятся в структуре `Map` (которая поддерживает операции `mapCreate`, `mapBind` и `mapResolve`), чтобы обеспечить эффективное нахождение записей при поступлении пакетов от источников, уже находящихся в таблице. Константа `MAX_TTL` указывает, как долго запись хранится в таблице до ее удаления.

```
#define BRIDGE_TAB_SIZE 1024 /* максимальный размер таблицы моста */
#define MAX_TTL 120 /* время (в секундах) до очистки записи */
typedef struct {
    MacAddr destination; /* MAC-адрес узла */
    int ifnumber; /* интерфейс для его достижения */
    u_short TTL; /* время жизни */
    Binding binding; /* привязка в Map */
} BridgeEntry;
```

```
int numEntries = 0;
Map bridgeMap = mapCreate(BRIDGE_TAB_SIZE, sizeof(BridgeEntry));
```

Функция, которая обновляет таблицу переадресации при поступлении нового пакета, называется `updateTable`. Передаваемые аргументы — это MAC-адрес (адрес управления доступом к среде), содержащийся в пакете, и номер интерфейса, на котором он был получен. Другая функция, не показанная здесь, вызывается через регулярные промежутки времени, сканирует записи в таблице переадресации и уменьшает поле `TTL` (время жизни (time to live)) каждой записи, удаляя любые записи, у которых `TTL` достиг 0. Обратите внимание, что `TTL` сбрасывается на `MAX_TTL` каждый раз, когда пакет поступает для обновления существующей записи таблицы, и что интерфейс, на котором может быть достигнут пункт назначения, обновляется, чтобы отразить последний полученный пакет.

```
void
updateTable (MacAddr src, int inif)
{
    BridgeEntry *b;
    if (mapResolve(bridgeMap, &src, (void **)&b) == FALSE)
    {
        /* этот адрес не в таблице, попробуем добавить его */
        if (numEntries < BRIDGE_TAB_SIZE)
        {
            b = NEW(BridgeEntry);
            b->binding = mapBind(bridgeMap, &src, b);
            /* используем адрес источника пакета как адрес назначения в таблице */
            b->destination = src;
            numEntries++;
        }
        else
```

```

    {
        /* сейчас не удастся поместить этот адрес в таблицу, поэтому откажемся */
        return;
    }
}
/* сбрасываем TTL и используем самый последний входной интерфейс */
b->TTL = MAX_TTL;
b->ifnumber = inif;
}

```

Обратите внимание, что эта реализация использует простую стратегию в случае, если таблица моста заполнена до предела — она просто не добавляет новый адрес. Напомним, что полнота таблицы моста не обязательна для корректной переадресации; это просто оптимизирует производительность. Если в таблице есть какая-то запись, которая в настоящее время не используется, она в конечном итоге истечет и будет удалена, создавая место для новой записи. Альтернативный подход заключался бы в вызове какого-либо алгоритма замены кеша при обнаружении, что таблица заполнена; например, мы могли бы найти и удалить запись с наименьшим TTL, чтобы освободить место для новой записи.

Глава 3.2.3. Алгоритм остовного дерева (spanning tree)

Предыдущая стратегия работает нормально, пока в сети нет циклов, в противном случае она терпит ужасный провал — кадры могут пересылаться бесконечно. Это легко понять на примере, показанном на рис. 3.10, где коммутаторы S1, S4 и S6 образуют петлю.

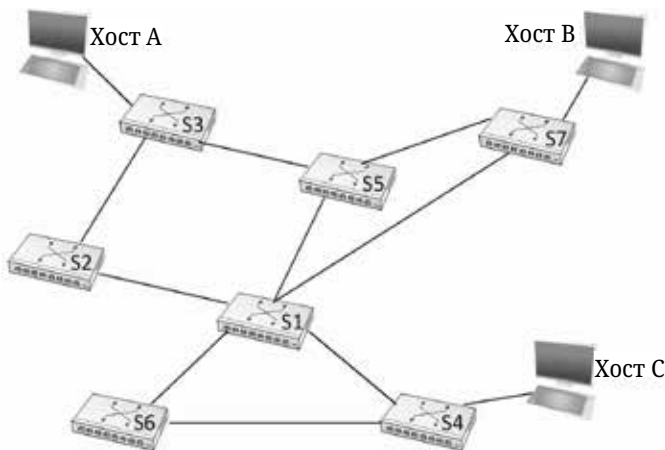


Рисунок 3.10. Коммутируемый Ethernet с петлями.

Заметим, что теперь мы переходим от названия каждого устройства пересылки «мост» (соединяющий сегменты, которые могут достигать множества других устройств) к названию «коммутатор уровня 2» (соединяющий каналы «точка-точка», которые достигают только одного другого устройства). Чтобы пример оставался управляемым, мы включаем только три хоста. На практике коммутаторы обычно имеют 16, 24 или 48 портов, а это означает, что они могут подключаться к такому же количеству хостов (и других коммутаторов).

В нашем примере коммутируемой сети предположим, что пакет входит в коммутатор S4 от хоста С, и что адрес назначения еще не находится в таблице пересылки ни одного

из коммутаторов: S4 отправляет копию пакета по двум другим своим портам: к коммутаторам S1 и S6. Коммутатор S6 пересылает пакет на S1 (а тем временем S1 пересылает пакет на S6), оба из которых в свою очередь пересылают свои пакеты обратно на S4. Коммутатор S4 все еще не имеет этого адреса назначения в своей таблице, поэтому он пересылает пакет по двум другим своим портам. Нет ничего, что могло бы уберечь этот цикл от бесконечного повторения, при котором пакеты будут перемещаться в обоих направлениях между S1, S4 и S6.

Почему в коммутируемой сети Ethernet (или расширенной LAN) может появиться цикл? Один из вариантов состоит в том, что сеть управляется более чем одним администратором, например, потому что она охватывает несколько отделов в организации. В такой ситуации возможно, что ни один человек не знает всю конфигурацию сети, а это означает, что коммутатор, замыкающий цикл, может быть добавлен без ведома кого-либо. Второй, более вероятный сценарий состоит в том, что циклы специально встроены в сеть — для обеспечения резервирования на случай отказа. В конце концов, сети без циклов требуется всего лишь один сбой соединения, чтобы она разделилась на две отдельные части.

Какова бы ни была причина, коммутаторы должны уметь правильно обрабатывать циклы. Эта проблема решается за счет выполнения коммутаторами распределенного *алгоритма остовного дерева*. Если представить сеть как граф, который может содержать циклы, то *spanning tree* — это подграф этого графа, который охватывает (покрывает) все вершины, но не содержит циклов. То есть *spanning tree* сохраняет все вершины исходного графа, но отбрасывает некоторые ребра. Например, на рис. 3.11 показан циклический граф слева и одно из возможных множеств остовного дерева справа.

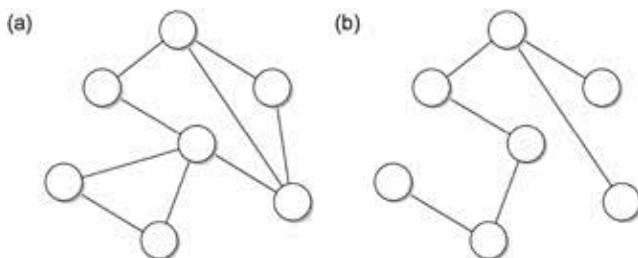


Рисунок 3.11. Пример (а) циклического графа; (б) соответствующего ему остовного дерева.

Идея остовного дерева достаточно проста: это подмножество фактической топологии сети, не имеющее циклов и охватывающее все устройства в сети. Трудность заключается в том, как все коммутаторы координируют свои решения, чтобы прийти к единому представлению об остовном дереве. Ведь одну топологию обычно можно покрыть несколькими остовными деревьями. Ответ заключается в протоколе остовного дерева, который мы сейчас опишем.

Алгоритм остовного дерева, разработанный Радией Перлман, тогда работавшей в Digital Equipment Corporation, представляет собой протокол, используемый набором коммутаторов для согласования остовного дерева для конкретной сети (спецификация IEEE 802.1 основана на этом алгоритме). На практике это означает, что каждый коммутатор решает, через какие порты он будет и не будет пересылать кадры. По сути, исключая порты из топологии, сеть сводится к ациклическому дереву. Возможно даже, что целый коммутатор не будет участвовать в пересылке кадров, что на первый взгляд может показаться странным. Однако алгоритм является динамическим, и это означает, что коммутаторы всегда готовы перестроиться в новое остовное дерево в случае сбоя какого-либо коммутатора, и поэтому неиспользуемые порты и коммутаторы обеспечивают резервную емкость, необходимую для восстановления после сбоев.

Основная идея остовного дерева заключается в том, чтобы коммутаторы выбирали порты, через которые они будут пересылать кадры. Алгоритм выбирает порты следующим образом. Каждый коммутатор имеет уникальный идентификатор; для наших целей мы используем метки S1, S2, S3 и так далее. Алгоритм сначала выбирает коммутатор с наименьшим идентификатором в качестве корня остовного дерева; как именно происходит это избрание, будет описано ниже. Корневой коммутатор всегда пересылает кадры через все свои порты. Далее каждый коммутатор вычисляет кратчайший путь до корня и отмечает, какой из его портов находится на этом пути. Этот порт также выбирается в качестве предпочтительного пути коммутатора к корню. Наконец, с учетом возможности того, что к его портам может быть подключен другой коммутатор, коммутатор выбирает один коммутатор, который будет отвечать за пересылку кадров к корню. Каждый назначенный коммутатор является ближайшим к корню. Если два или более коммутаторов равно удалены от корня, то для разрешения конфликта используются их идентификаторы, и выигрывает наименьший идентификатор. Конечно, каждый коммутатор может быть подключен к более чем одному другому коммутатору, поэтому он участвует в выборе назначенного коммутатора для каждого такого порта. По сути, это означает, что каждый коммутатор решает, является ли он назначенным коммутатором относительно каждого из своих портов. Коммутатор пересылает кадры через те порты, для которых он является назначенным коммутатором.

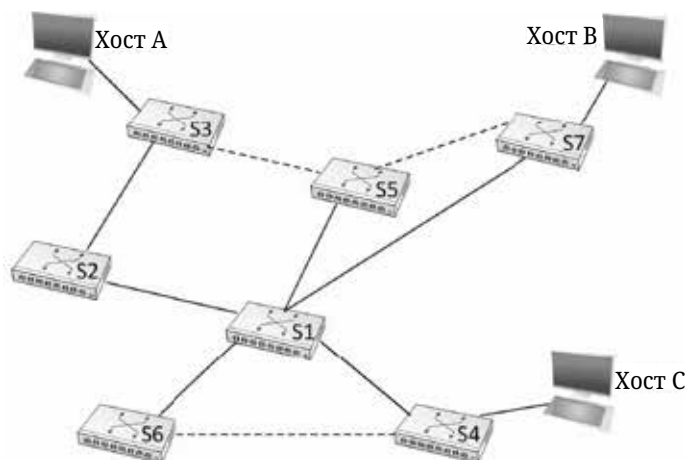


Рисунок 3.12. Остовное дерево с некоторыми не выбранными портами.

На рис. 3.12 показано остовное дерево, соответствующее сети, изображенной на рис. 3.10. В этом примере коммутатор S1 является корнем, так как у него наименьший идентификатор. Обратите внимание, что S3 и S5 подключены друг к другу, но S5 является назначенным коммутатором, так как он ближе к корню. Аналогично S5 и S7 подключены друг к другу, но в этом случае S5 является назначенным коммутатором, так как у него меньший идентификатор; оба находятся на одинаковом расстоянии от S1.

Хотя человек может взглянуть на сеть, показанную на рис. 3.10, и вычислить остовное дерево, показанное на рис. 3.12, согласно приведенным выше правилам, коммутаторы не имеют возможности видеть топологию всей сети, не говоря уже о том, чтобы заглянуть внутрь других коммутаторов и узнать их идентификаторы. Вместо этого им приходится обмениваться конфигурационными сообщениями друг с другом и затем решать, являются ли они корнем или назначенным коммутатором, на основе этих сообщений.

Конкретно конфигурационные сообщения содержат три части информации:

1. Идентификатор коммутатора, отправляющего сообщение.

2. Идентификатор коммутатора, который, по мнению отправляющего коммутатора, является корнем.
3. Расстояние, измеряемое в переходах, от отправляющего коммутатора до корневого коммутатора.

Каждый коммутатор записывает текущее *лучшее* конфигурационное сообщение, которое он видел на каждом из своих портов (определение «лучшего» будет описано ниже), включая как сообщения, полученные от других коммутаторов, так и сообщения, которые он сам отправлял.

Изначально каждый коммутатор считает себя корнем и отправляет конфигурационное сообщение на каждый из своих портов, указывая себя как корень и задавая расстояние до корня равным 0. При получении конфигурационного сообщения через определенный порт коммутатор проверяет, является ли новое сообщение лучше текущего лучшего конфигурационного сообщения, записанного для этого порта. Новое конфигурационное сообщение считается лучше текущей записи, если выполняется любое из следующих условий:

- Оно указывает корень с меньшим идентификатором.
- Оно указывает корень с равным идентификатором, но с меньшим расстоянием.
- Идентификатор корня и расстояние равны, но у отправляющего коммутатора меньший идентификатор.

Если новое сообщение лучше текущей записи, коммутатор отбрасывает старую информацию и сохраняет новую. Однако сначала он добавляет 1 к полю расстояния до корня, так как коммутатор находится на один переход дальше от корня, чем коммутатор, отправивший сообщение.

Когда коммутатор получает конфигурационное сообщение, указывающее, что он не является корнем — то есть сообщение от коммутатора с меньшим идентификатором, — коммутатор перестает генерировать собственные конфигурационные сообщения и вместо этого только пересылает конфигурационные сообщения от других коммутаторов, предварительно добавляя 1 к полю расстояния. Аналогично, когда коммутатор получает конфигурационное сообщение, указывающее, что он не является назначенным коммутатором для этого порта (то есть сообщение от коммутатора, который ближе к корню или находится на одинаковом расстоянии от корня, но с меньшим идентификатором), коммутатор перестает отправлять конфигурационные сообщения через этот порт. Таким образом, когда система стабилизируется, только корневой коммутатор продолжает генерировать конфигурационные сообщения, а другие коммутаторы пересылают эти сообщения только через порты, для которых они являются назначенными коммутаторами. На этом этапе остоное дерево построено, и все коммутаторы согласны с тем, какие порты используются для остоного дерева. Только эти порты могут использоваться для передачи данных.

Рассмотрим пример того, как это работает. Представим, что произошло на рис. 3.12, если на кампусе только что восстановилось питание, и все коммутаторы загружаются примерно в одно и то же время. Все коммутаторы начнут с утверждения, что они являются корнем. Мы обозначаем конфигурационное сообщение от узла X, в котором он утверждает, что находится на расстоянии d от корневого узла Y, как (Y, d, X). Например, на активности на S3 последовательность событий будет разворачиваться следующим образом:

1. S3 получает (S2, 0, S2).
2. Поскольку $2 < 3$, S3 принимает S2 в качестве корня.
3. S3 добавляет 1 к расстоянию, указанному S2 (0), и отправляет (S2, 1, S3) в сторону S5.
4. Тем временем S2 принимает S1 в качестве корня, потому что у него меньший идентификатор, и отправляет (S1, 1, S2) в сторону S3.
5. S5 принимает S1 в качестве корня и отправляет (S1, 1, S5) в сторону S3.

6. S3 принимает S1 в качестве корня и отмечает, что как S2, так и S5 ближе к корню, чем он сам, но у S2 меньший идентификатор, поэтому он остается на пути S3 к корню.

Это оставляет S3 с активными портами, как показано на рис. 3.12. Обратите внимание, что хосты А и В не могут общаться по кратчайшему пути (через S5), потому что кадры должны «протекать вверх по дереву и обратно вниз», но это цена, которую нужно заплатить, чтобы избежать петель.

Даже после стабилизации системы корневой коммутатор продолжает периодически отправлять конфигурационные сообщения, а другие коммутаторы продолжают пересылать эти сообщения, как описано выше. Если конкретный коммутатор выйдет из строя, коммутаторы, расположенные ниже по потоку, не будут получать эти конфигурационные сообщения, и после ожидания определенного времени они снова начнут утверждать, что они являются корнем, после чего алгоритм снова начнет работу для выбора нового корня и новых назначенных коммутаторов.

Важно заметить, что хотя алгоритм способен перенастраивать остовное дерево при отказе коммутатора, он не способен пересылать кадры по альтернативным путям с целью обхода перегруженного коммутатора.

Глава 3.2.4. Широковещательная и многоадресная рассылка

Предыдущие обсуждения сосредоточены на том, как коммутаторы пересылают одноадресные (unicast) кадры с одного порта на другой. Поскольку цель коммутатора — прозрачно расширить локальную сеть (LAN) через несколько сетей, и поскольку большинство локальных сетей поддерживают как широковещательную, так и многоадресную рассылку, коммутаторы также должны поддерживать эти две функции. Широковещательная рассылка проста: каждый коммутатор пересылает кадр с широковещательным адресом назначения на каждый активный (выбранный) порт, кроме того, на котором кадр был получен.

Многоадресную рассылку можно реализовать точно так же, с каждым узлом, решающим для себя, принимать ли сообщение. На практике это именно так и делается. Обратите внимание, что поскольку не все узлы являются членами любой конкретной группы многоадресной рассылки, можно сделать лучше. В частности, алгоритм остовного дерева может быть расширен для обрезки сетей, по которым кадры многоадресной рассылки не нужно пересылать. Рассмотрим кадр, отправленный узлом А в группу М на рис. 3.12. Если узел С не принадлежит к группе М, то нет необходимости, чтобы коммутатор S4 пересылал кадры по этой сети.

Как коммутатор узнает, нужно ли пересылать кадр многоадресной рассылки через определенный порт? Он узнает это точно так же, как коммутатор узнает, нужно ли пересылать одноадресный кадр через определенный порт — наблюдая за исходными адресами, которые он получает через этот порт. Конечно, группы обычно не являются источником кадров, поэтому нам нужно немного схитрить. В частности, каждый узел, являющийся членом группы М, должен периодически отправлять кадр с адресом группы М в поле исходного адреса заголовка кадра. Этот кадр будет иметь в качестве адреса назначения многоадресный адрес для коммутаторов.

Хотя описанное расширение для многоадресной рассылки когда-то предлагалось, оно не получило широкого распространения. Вместо этого многоадресная рассылка реализуется точно так же, как и широковещательная.

Глава 3.2.5. Виртуальные локальные сети (VLAN)

Одним из ограничений коммутаторов является их недостаточная масштабируемость. Нереально соединить более нескольких коммутаторов, где на практике «несколько» обычно означает «десятки». Одна из причин этого заключается в том, что алгоритм остовного дерева масштабируется линейно; то есть нет предусмотренных механизмов для наложе-

ния иерархии на набор коммутаторов. Вторая причина заключается в том, что коммутаторы пересылают все широковещательные кадры. В то время как для всех узлов в ограниченной среде (например, в отделе) разумно видеть широковещательные сообщения друг друга, маловероятно, что все узлы в более крупной среде (например, в большой компании или университете) захотят беспокоиться о широковещательных сообщениях друг друга. Иными словами, широковещательная рассылка не масштабируется, и, как следствие, сети на уровне L2 не масштабируются.

Одним из подходов к увеличению масштабируемости являются *виртуальные локальные сети* (VLAN). VLAN позволяют разделить одну расширенную локальную сеть на несколько кажущихся отдельными локальных сетей. Каждой виртуальной локальной сети присваивается идентификатор (иногда называемый *цветом* (*color*)), и пакеты могут перемещаться с одного сегмента на другой только в том случае, если оба сегмента имеют один и тот же идентификатор. Это ограничивает количество сегментов в расширенной локальной сети, которые будут получать любой данный широковещательный пакет.

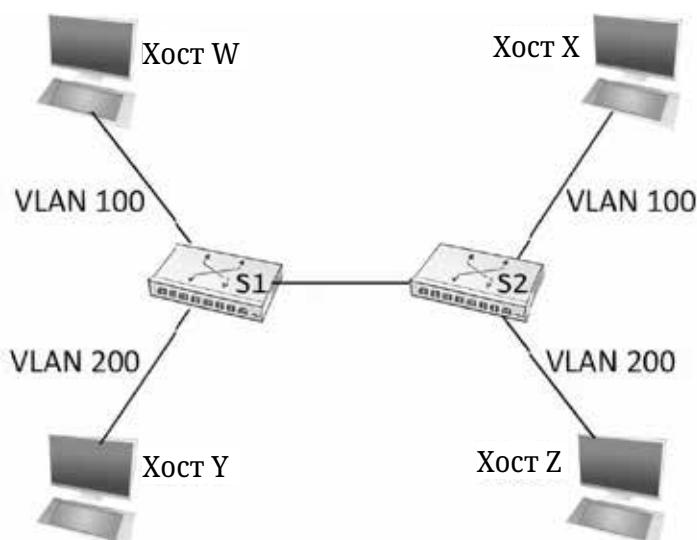


Рисунок 3.13. Две виртуальные локальные сети используют общую магистраль.

Мы можем понять, как работают виртуальные локальные сети (VLAN), на примере. На рис. 3.13 показаны четыре хоста и два коммутатора. При отсутствии VLAN любой широковещательный пакет от любого хоста достигнет всех остальных хостов. Теперь предположим, что мы определили сегменты, подключенные к хостам W и X, как находящиеся в одном VLAN, который мы назовем VLAN 100. Мы также определяем сегменты, подключенные к хостам Y и Z, как находящиеся в VLAN 200. Для этого нам нужно настроить идентификатор VLAN на каждом порту коммутаторов S1 и S2. Соединение между S1 и S2 считается частью обоих VLAN.

Когда пакет, отправленный хостом X, достигает коммутатора S2, коммутатор отмечает, что он пришел на порт, настроенный как принадлежащий к VLAN 100. Он вставляет заголовок VLAN между заголовком Ethernet и его полезной нагрузкой. Нужной частью заголовка VLAN является идентификатор VLAN; в данном случае этот идентификатор установлен на 100. Коммутатор теперь применяет свои обычные правила пересылки к пакету с дополнительным ограничением, что пакет не может быть отправлен на интерфейс, который не является частью VLAN 100. Таким образом, при любых обстоятельствах па-

кет (даже широковещательный пакет) не будет отправлен на интерфейс к хосту Z, который находится в VLAN 200. Однако пакет пересылается на коммутатор S1, который следует тем же правилам и, таким образом, может переслать пакет к хосту W, но не к хосту Y.

Привлекательной особенностью VLAN является возможность изменения логической топологии без перемещения проводов или изменения адресов. Например, если мы хотим сделать соединение, подключенное к хосту Z, частью VLAN 100 и таким образом позволить X, W и Z находиться в одной виртуальной локальной сети, нам нужно просто изменить одну конфигурацию на коммутаторе S2.

Поддержка VLAN требует довольно простого расширения исходной спецификации заголовка 802.1 путем вставки поля 12-битного идентификатора VLAN (VID) между полями SrcAddr и Type, как показано на рисунке 3.14. (Этот VID обычно называется *VLAN Tag*.) На самом деле в середину заголовка вставляются 32 бита, но первые 16 бит используются для сохранения обратной совместимости с исходной спецификацией (они используют Type = 0x8100, чтобы указать, что этот кадр включает расширение VLAN); другие четыре бита содержат управляющую информацию, используемую для определения приоритета кадров. Это означает, что можно сопоставить $2^{12} = 4096$ виртуальных сетей с одной физической локальной сетью.

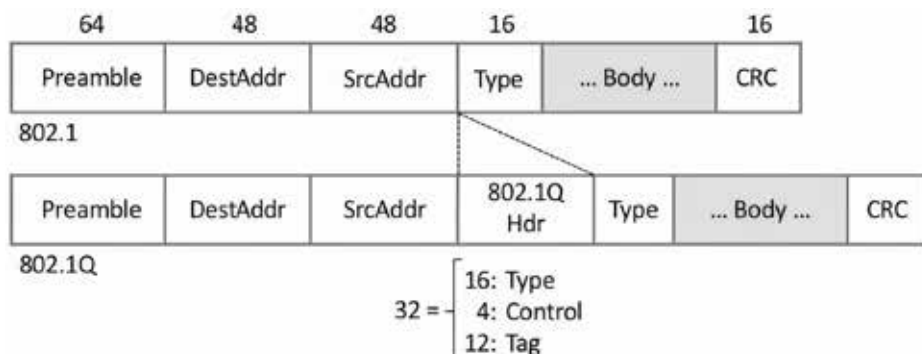


Рисунок 3.14. Тег 802.1Q VLAN, встроенный в заголовок Ethernet (802.1).

Мы завершаем это обсуждение, отмечая еще одно ограничение сетей, построенных путем соединения коммутаторов уровня 2: отсутствие поддержки гетерогенности. То есть коммутаторы ограничены в типах сетей, которые они могут соединять. В частности, коммутаторы используют заголовок кадра сети и, следовательно, могут поддерживать только сети, которые имеют точно такой же формат адресов. Например, коммутаторы можно использовать для соединения сетей Ethernet и 802.11, поскольку они имеют общий формат заголовка, но коммутаторы не могут легко работать с другими типами сетей с различными форматами адресов, такими как ATM, SONET, PON или сотовая сеть. В следующей главе объясняется, как обойти это ограничение, а также как масштабировать коммутируемые сети до еще больших размеров.

Глава 3.3. Интернет (IP)

В предыдущей главе мы узнали, что можно построить достаточно крупные локальные сети (LAN) с использованием мостов и коммутаторов LAN, однако такие подходы ограничены в своей способности к масштабированию и обработке гетерогенности. В этой главе мы рассмотрим некоторые способы преодоления ограничений сетей с мостами, что позволит нам создавать крупные, высокогетерогенные сети с относительно эффективной маршрутизацией. Такие сети мы называем интерсетями. Далее мы продолжим обсуждение того, как построить действительно глобальную интерсеть, но пока остановимся на основах. Начнем с более подробного рассмотрения того, что означает слово «интерсеть».

Глава 3.3.1. Что такое интернет?

Мы используем термин «*интерсеть*» (иногда просто «*интернет*» с маленькой буквы) для обозначения произвольной коллекции сетей, объединенных для предоставления какого-то вида услуг по доставке пакетов от хоста к хосту. Например, корпорация с многими филиалами может создать частную интерсеть, объединив LAN всех своих филиалов с помощью арендованных у телефонной компании линий. Когда мы говорим о широко используемой глобальной интерсети, к которой сейчас подключен большой процент сетей, мы называем ее *Интернетом* с большой буквы *И*. В соответствии с основными принципами этой книги, мы хотим, чтобы вы в первую очередь изучили принципы межсетевого взаимодействия с маленькой буквы «и», но иллюстрируем эти идеи реальными примерами из «*большого И*» — Интернета.

Другой термин, который может быть запутанным, — это разница между сетями, подсетями и интерсетями. Мы будем избегать термина «подсети» до более позднего этапа книги. Сейчас мы используем термин «сеть» для обозначения либо напрямую подключенной, либо коммутируемой сети, как описано в предыдущей главе и предыдущем разделе. Такая сеть использует одну технологию, такую как 802.11 или Ethernet. *Интерсеть* — это объединение таких сетей. Иногда, чтобы избежать неоднозначности, мы называем основные сети, которые мы объединяем, *физическими* сетями. Интернет в этом контексте — это логическая сеть, построенная из коллекции физических сетей. В этом контексте коллекция сегментов Ethernet, соединенных мостами или коммутаторами, все еще рассматривается как одна сеть.

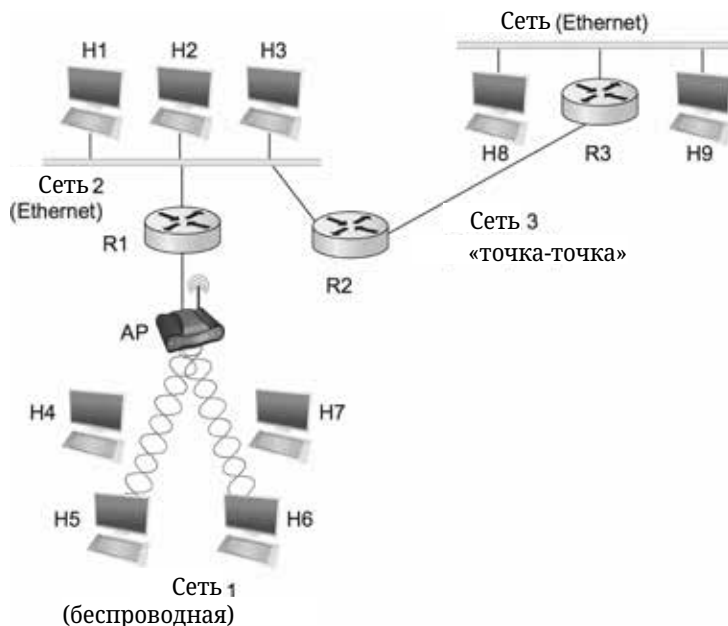


Рисунок 3.15. Простая интерсеть. Н обозначает хост, а R — маршрутизатор.

На рис. 3.15 показан пример интерсети. Интерсеть часто называют «сетью сетей», потому что она состоит из множества меньших сетей. На этом рисунке мы видим Ethernets, беспроводную сеть и соединение «точка-точка». Каждая из них — это однотехнологичная сеть. Узлы, которые соединяют сети, называются маршрутизаторами. Их также иногда на-

зывают *шлюзами*, но поскольку этот термин имеет несколько других значений, мы ограничиваем его использование термином «маршрутизатор».

Протокол Интернета (IP) — ключевой инструмент, используемый сегодня для создания масштабируемых гетерогенных интерсетей. Первоначально он был известен как протокол Кана — Серфа в честь его создателей. Один из способов представить IP заключается в том, что он работает на всех узлах (как на хостах, так и на маршрутизаторах) в коллекции сетей и определяет инфраструктуру, которая позволяет этим узлам и сетям функционировать как единая логическая интерсеть. Например, на рис. 3.16 показано, как hosts H5 и H8 логически соединены через интернет, изображенный на рис. 3.15, включая график протоколов, работающих на каждом узле. Обратите внимание, что протоколы более высокого уровня, такие как TCP и UDP, обычно работают поверх IP на хостах.

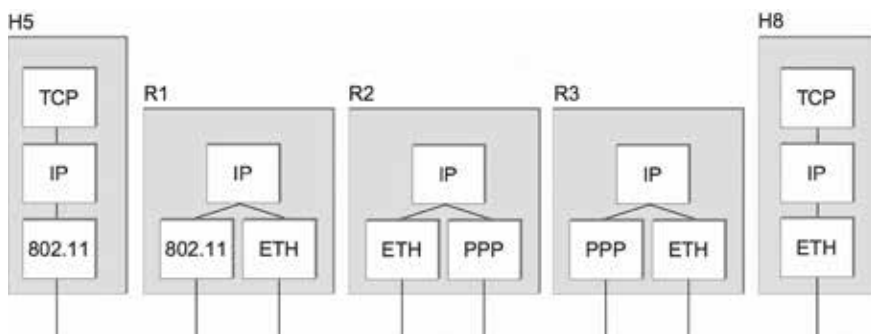


Рисунок 3.16. Простая сеть Интернета, показывающая уровни протоколов, используемых для соединения H5 и H8 на рисунке выше. ETH — это протокол, который работает через Ethernet.

Остальная часть этой и следующей главы посвящена различным аспектам IP. Хотя, безусловно, можно построить интерсеть, которая не использует IP, и в действительности в первые дни Интернета существовали альтернативные решения, IP является самым интересным случаем для изучения просто из-за размера Интернета. Иными словами, только Интернет на основе IP действительно столкнулся с проблемой масштабируемости. Таким образом, это предоставляет лучший пример для изучения масштабируемого протокола межсетевого взаимодействия.

Сети L2 и L3

- Как следует из предыдущей главы, Ethernet может рассматриваться как соединение «точка-точка», соединяющее пару коммутаторов, с сетью из взаимосвязанных коммутаторов, формирующих коммутируемый Ethernet. Эта конфигурация также известна как *сеть уровня 2 (L2)*.
- Но, как мы увидим в этой главе, Ethernet (даже когда он расположен в конфигурации «точка-точка», а не в общей сети CSMA/CD) может рассматриваться как *сеть*, соединяющая пару маршрутизаторов с сетью таких маршрутизаторов, формирующей Интернет. Эта конфигурация также известна как *сеть уровня 3 (L3)*.
- Запутанно, но это потому, что Ethernet «точка-точка» является одновременно и каналом, и сетью (хотя и тривиальной двухузловой сетью во втором случае), в зависимости от того, соединен ли он с парой коммутаторов уровня 2, работающих по алгоритму остовного дерева, или с парой маршрутизаторов уровня 3, работающих по IP (плюс протоколы маршрутизации, описанные позже в этой главе). Почему следует выбирать одну конфигурацию вместо другой? Частично это зависит от того, хотите ли вы, чтобы

- сеть была одним широковещательным доменом (если да, выбирайте L2), или вы хотите,
- чтобы узлы, подключенные к сети, находились в разных сетях (если да, выбирайте L3).
- Хорошая новость заключается в том, что когда вы полностью поймете последствия этой
- двойственности, вы преодолеете значительное препятствие в освоении современных
- пакетных сетей.

Глава 3.3.2. Модель обслуживания

Хорошее начало при создании интерсети — это определение ее *модели обслуживания*, то есть служб «хост-хост», которые вы хотите предоставить. Основная проблема при определении модели обслуживания для интерсети заключается в том, что мы можем предоставить службу «хост-хост» только в том случае, если эта служба каким-то образом может быть предоставлена через каждую из базовых физических сетей. Например, было бы бесполезно предполагать, что наша модель обслуживания интерсети будет обеспечивать гарантированную доставку каждого пакета за 1 мс или менее, если бы существовали базовые сетевые технологии, которые могли бы произвольно задерживать пакеты. Философия, использованная при определении модели обслуживания IP, заключалась в том, чтобы сделать ее достаточно непритязательной, чтобы почти любая сетевая технология, которая может появиться в интерсети, могла предоставить необходимую услугу.

Модель обслуживания IP можно рассматривать как модель, которая состоит из двух частей: схемы адресации, которая предоставляет способ идентификации всех узлов в интерсети, и модели доставки данных без установления соединения (дейтаграмм). Эту модель обслуживания иногда называют «*максимальным усилием*» (best effort), потому что, хотя IP делает все возможное для доставки дейтаграмм, он не дает никаких гарантий. Мы отложим обсуждение схемы адресации на потом и сначала рассмотрим модель доставки данных.

Доставка дейтаграмм

Дейтаграмма IP является основой протокола Интернета. Напомним из предыдущего раздела, что дейтаграмма — это тип пакета, который отправляется по сети без установления соединения. Каждая дейтаграмма несет в себе достаточно информации, чтобы сеть могла переслать пакет в нужное место назначения; нет необходимости в предварительном механизме настройки, чтобы сообщить сети, что делать, когда пакет прибывает. Вы просто отправляете его, и сеть делает все возможное, чтобы доставить его в нужное место назначения. Часть про «максимальные усилия» означает, что если что-то пойдет не так и пакет будет утерян, поврежден, доставлен не по назначению или каким-либо образом не достигнет места назначения, сеть ничего не делает — она сделала все возможное, и это все, что она обязана сделать. Она не пытается исправить ошибку. Это иногда называют *ненадежной* услугой.

Служба по принципу «максимальных усилий» без установления соединения — это самая простая услуга, которую можно потребовать от интерсети, и это является ее большим преимуществом. Например, если вы предоставляете услугу «максимальных усилий» через сеть, которая обеспечивает надежную услугу, это прекрасно — вы получаете услугу «максимальных усилий», которая просто всегда доставляет пакеты. С другой стороны, если бы у вас была модель надежной службы через ненадежную сеть, вам пришлось бы добавлять множество дополнительных функций в маршрутизаторы, чтобы компенсировать недостатки базовой сети. Одной из первоначальных целей проектирования IP было сделать маршрутизаторы как можно проще.

Способность IP «работать на чем угодно» часто называют одной из его самых важных характеристик. Примечательно, что многие технологии, по которым работает IP сегодня, не существовали, когда IP был изобретен. До сих пор не была изобретена ни одна сетевая

технология, которая оказалась бы слишком сложной для IP; в принципе, IP может работать даже в сети, которая передает сообщения с помощью почтовых голубей.

Доставка по принципу «максимальных усилий» означает не только то, что пакеты могут быть утеряны. Иногда они могут быть доставлены в неправильном порядке, а иногда один и тот же пакет может быть доставлен более одного раза. Протоколы более высокого уровня или приложения, работающие поверх IP, должны учитывать все эти возможные режимы отказа.

Формат пакета

Очевидно, что ключевая часть модели обслуживания IP — это типы пакетов, которые могут быть переданы. Дейтаграмма IP, как и большинство пакетов, состоит из заголовка, за которым следует некоторое количество байтов данных. Формат заголовка показан на рис. 3.17. Обратите внимание, что мы приняли другой стиль представления пакетов, чем тот, который использовался в предыдущих главах. Это связано с тем, что форматы пакетов на уровне интересети и выше, где мы будем сосредоточивать наше внимание в следующих главах, почти всегда спроектированы таким образом, чтобы выравниваться по границам в 32 бита, чтобы упростить их обработку в программном обеспечении. Таким образом, общепринятый способ их представления (используемый, например, в запросах комментариев Интернета) заключается в том, чтобы рисовать их как последовательность 32-битных слов. Верхнее слово передается первым, и самый левый байт каждого слова передается первым. В таком представлении вы легко можете распознать поля, длина которых кратна 8 битам. В редких случаях, когда длина полей не кратна 8 битам, вы можете определить длину полей, посмотрев на позиции битов, отмеченные в верхней части пакета.

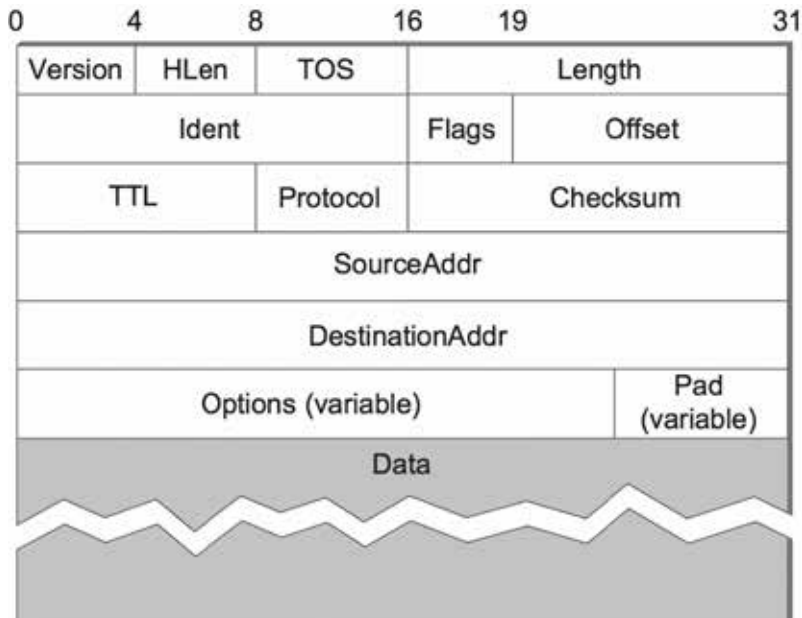


Рисунок 3.17. Заголовок пакета IPv4.

Рассматривая каждое поле в заголовке IP, мы видим, что «простая» модель доставки дейтаграмм по принципу «максимальных усилий» все же имеет некоторые тонкости. Поле Version (Версия) указывает версию IP. По-прежнему предполагается, что версия

IP — это 4, которая обычно называется IPv4. Заметьте, что размещение этого поля в самом начале дейтаграммы облегчает изменение всего остального в формате пакета в последующих версиях; программное обеспечение обработки заголовков начинает с просмотра версии и затем ветвится для обработки остальной части пакета в соответствии с определенным форматом. Следующее поле, HLen, указывает длину заголовка в 32-битных словах. Когда нет опций, что бывает в большинстве случаев, заголовок имеет длину 5 слов (20 байт). 8-битное поле TOS (тип обслуживания) имело несколько различных определений на протяжении многих лет, но его основная функция — позволить обрабатывать пакеты по-разному в зависимости от потребностей приложения. Например, значение TOS может определять, следует ли помещать пакет в специальную очередь, которая получает минимальную задержку.

Следующие 16 бит заголовка содержат длину дейтаграммы, включая заголовок. В отличие от поля HLen, поле Length (длина) учитывает байты, а не слова. Таким образом, максимальный размер дейтаграммы IP составляет 65 535 байт. Однако физическая сеть, по которой работает IP, может не поддерживать такие длинные пакеты. По этой причине IP поддерживает процесс фрагментации и повторной сборки. Второе слово заголовка содержит информацию о фрагментации, и детали его использования представлены в следующей главе, озаглавленной «Фрагментация и повторная сборка».

Переходя к третьему слову заголовка, следующий байт — это поле TTL (time to live, время жизни). Его название отражает его историческое значение, а не то, как оно обычно используется сегодня. Назначение поля заключается в том, чтобы отлавливать пакеты, которые застряли в петлях маршрутизации, и отбрасывать их, а не позволять им бесконечно потреблять ресурсы. Изначально TTL устанавливался на определенное количество секунд, в течение которых пакету было позволено существовать, и маршрутизаторы по пути уменьшали это поле до тех пор, пока оно не достигало 0. Однако, поскольку редко бывало, чтобы пакет задерживался в маршрутизаторе дольше одной секунды, и маршрутизаторы не имели общего доступа к часам, большинство маршрутизаторов просто уменьшали TTL на 1 при передаче пакета. Таким образом, это поле стало больше похоже на счетчик прыжков, чем на таймер, что все равно является хорошим способом отлавливать пакеты, застрявшие в петлях маршрутизации. Одной из тонкостей является начальная установка этого поля отправляющим хостом: если установить его слишком высоко, пакеты могут долго циркулировать перед сбросом; если установить слишком низко, они могут не достичь своей цели. Значение 64 является текущим значением по умолчанию.

Поле Protocol (протокол) является просто ключом демультимплексирования, идентифицирующим протокол более высокого уровня, которому должен быть передан этот IP-пакет. Существуют значения, определенные для TCP (Transmission Control Protocol, Протокол управления передачей 6), UDP (User Datagram Protocol, Протокол дейтаграмм пользователя 17) и многих других протоколов, которые могут находиться над IP в протокольной графе.

Контрольная сумма (Checksum) вычисляется путем рассмотрения всего заголовка IP как последовательности 16-битных слов, их суммирования с использованием арифметики с дополнением до единицы и взятия дополнения до единицы от результата. Таким образом, если любой бит в заголовке поврежден при передаче, контрольная сумма не будет содержать правильного значения при получении пакета. Поскольку поврежденный заголовок может содержать ошибку в адресе назначения и, как результат, может быть неверно доставлен, имеет смысл отбрасывать любой пакет, который не проходит проверку контрольной суммы. Следует отметить, что этот тип контрольной суммы не обладает такими же сильными свойствами обнаружения ошибок, как циклический избыточный код (CRC), но его намного проще вычислить в программном обеспечении.

Последние два обязательных поля в заголовке — это `SourceAddr` (адрес отправителя) и `DestinationAddr` (адрес получателя) для пакета. Последний является ключом к доставке дейтаграммы: каждый пакет содержит полный адрес своего предполагаемого назначения, чтобы на каждом маршрутизаторе могли быть приняты решения о пересылке. Адрес отправителя необходим, чтобы получатели могли решить, хотят ли они принять пакет, и чтобы они могли ответить. IP-адреса обсуждаются в следующем разделе — на данный момент важно знать, что IP определяет собственное глобальное адресное пространство, независимо от того, по каким физическим сетям он работает. Как мы увидим, это один из ключевых факторов поддержки гетерогенности.

Наконец, в конце заголовка может быть несколько опций. Наличие или отсутствие опций может быть определено путем изучения поля длины заголовка (`HLen`). Хотя опции используются довольно редко, полная реализация IP должна обрабатывать их все.

Фрагментация и повторная сборка

Одной из проблем предоставления единой модели обслуживания от хоста к хосту через гетерогенную коллекцию сетей является то, что каждая сетевая технология, как правило, имеет свое представление о том, какого размера может быть пакет. Например, классический Ethernet может принимать пакеты длиной до 1500 байт, но современные варианты могут передавать более крупные (джамбо) пакеты, которые содержат до 9000 байт полезной нагрузки. Это оставляет два варианта для модели обслуживания IP: убедиться, что все IP-дейтаграммы достаточно малы, чтобы поместиться в один пакет в любой сетевой технологии, или обеспечить возможность фрагментации и повторной сборки пакетов, когда они слишком большие для передачи через данную сетевую технологию. Последний вариант оказывается хорошим выбором, особенно если учесть тот факт, что постоянно появляются новые сетевые технологии, и IP должен работать поверх любой из них; это затруднило бы выбор подходящего малого предела размера дейтаграммы. Это также означает, что хост не будет отправлять неоправданно маленькие пакеты, что тратит полосу пропускания и потребляет ресурсы обработки, требуя больше заголовков на каждый байт переданных данных.

Основная идея здесь заключается в том, что каждый тип сети имеет *максимальный блок передачи данных* (*maximum transmission unit*, MTU), который является наибольшей IP-дейтаграммой, которую она может передать в кадре.¹ Обратите внимание, что это значение меньше наибольшего размера пакета в этой сети, потому что IP-дейтаграмма должна поместиться в *полезную нагрузку* канального кадра.

Таким образом, когда хост отправляет IP-дейтаграмму, он может выбрать любой размер, который ему нужен. Разумным выбором является MTU сети, к которой хост непосредственно подключен. Тогда фрагментация будет необходима только в том случае, если путь к назначению включает сеть с меньшим MTU. Если транспортный протокол, который работает поверх IP, передает IP-пакет большего размера, чем локальный MTU, то исходный хост должен его фрагментировать.

Фрагментация обычно происходит в маршрутизаторе, когда он получает дейтаграмму, которую он хочет переслать через сеть с меньшим MTU, чем у полученной дейтаграммы. Чтобы эти фрагменты могли быть собраны на принимающем хосте, они все несут один и тот же идентификатор в поле `Ident`. Этот идентификатор выбирается отправляющим хостом и предназначен для уникальности среди всех дейтаграмм, которые могут прибыть в пункт назначения от этого источника в течение некоторого разумного периода времени. Поскольку все фрагменты исходной дейтаграммы содержат этот идентификатор, собирающий хост сможет распознать те фрагменты, которые идут вместе. Если не все фрагменты прибывают на принимающий хост, хост прекращает процесс сборки и отбрасывает те фрагменты, которые прибыли. IP не пытается восстановить недостающие фрагменты.

¹ В ATM-сетях MTU, к счастью, значительно больше, чем одна ячейка, поскольку ATM имеет собственный механизм фрагментации и повторной сборки. Канальный кадр в ATM называется блоком данных протокола подуровня конвергенции (CS-PDU).

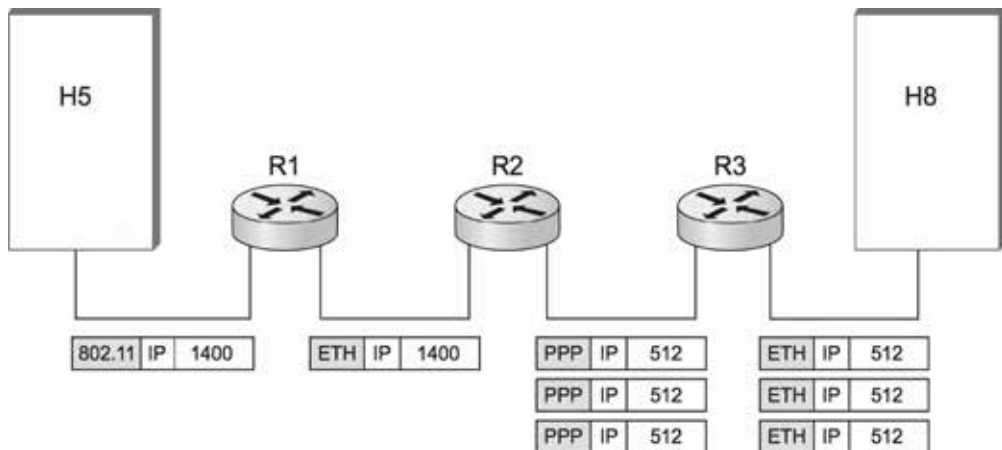


Рисунок 3.18. IP-дейтаграммы, проходящие через последовательность физических сетей, изображенных на предыдущем рисунке.

Чтобы понять, что это все означает, рассмотрим, что происходит, когда хост H5 отправляет дейтаграмму на хост H8 в примере интернет-сети, показанном на рис. 3.15. Предположим, что MTU составляет 1500 байт для двух Ethernet и сети 802.11, и 532 байта для сети «точка-точка», тогда 1420-байтовая дейтаграмма (20-байтовый IP-заголовок плюс 1400 байт данных), отправленная с H5, проходит через сеть 802.11 и первый Ethernet без фрагментации, но должна быть фрагментирована на три дейтаграммы в маршрутизаторе R2. Эти три фрагмента затем пересылаются маршрутизатором R3 через второй Ethernet на конечный хост.

Эта ситуация иллюстрируется на рис. 3.18. Данный рисунок также подчеркивает два важных момента:

1. Каждый фрагмент сам по себе является самостоятельной IP-дейтаграммой, которая передается по последовательности физических сетей, независимо от других фрагментов.
2. Каждая IP-дейтаграмма повторно инкапсулируется для каждой физической сети, по которой она передается.

Процесс фрагментации можно понять более детально, рассматривая поля заголовка каждой дейтаграммы, как показано на рис. 3.19. Нефрагментированный пакет, показанный вверху, содержит 1400 байт данных и 20-байтовый IP-заголовок. Когда пакет прибывает в маршрутизатор R2, у которого MTU составляет 532 байта, он должен быть фрагментирован. MTU в 532 байта оставляет 512 байт для данных после 20-байтового IP-заголовка, поэтому первый фрагмент содержит 512 байт данных. Маршрутизатор устанавливает бит M в поле Flags (см. рис. 3.17), а это означает, что будут следовать еще фрагменты, и устанавливает Offset на 0, так как этот фрагмент содержит первую часть оригинальной дейтаграммы.

Данные, содержащиеся во втором фрагменте, начинаются с 513-го байта оригинальных данных, поэтому поле Offset в этом заголовке установлено на 64, что равно $512/8$. Почему деление на 8? Потому что разработчики IP решили, что фрагментация всегда должна происходить на границах 8 байт, а это означает, что поле Offset учитывает 8-байтовые блоки, а не байты. (Мы оставляем вам в качестве упражнения выяснить, почему было принято такое решение.)

Третий фрагмент содержит последние 376 байт данных, и смещение теперь равно $2 \times 512/8 = 128$. Поскольку это последний фрагмент, бит M не установлен.



Рисунок 3.19. Поля заголовка, используемые при фрагментации IP:
(a) нефрагментированный пакет; (b) фрагментированный пакет.

Обратите внимание, что процесс фрагментации выполнен таким образом, что он может быть повторен, если фрагмент прибывает в другую сеть с еще меньшим MTU. Фрагментация создает более мелкие, действительные IP-дейтаграммы, которые могут быть легко собраны в оригинальную дейтаграмму при получении, независимо от порядка их прибытия. Повторная сборка осуществляется на принимающем хосте, а не в каждом маршрутизаторе.

Повторная сборка IP — далеко не простой процесс. Например, если один фрагмент потерян, приемник все равно попытается собрать дейтаграмму и в конечном итоге прекратит попытки, освободив ресурсы, использованные для неудавшейся сборки. Привлечение хоста к ненужному использованию ресурсов может быть основой для атаки типа «отказ в обслуживании».

По этой причине, среди прочих, фрагментация IP обычно считается хорошей вещью, которой следует избегать. Хосты сейчас настоятельно рекомендуют выполнять «обнаружение пути MTU», процесс, при котором фрагментация избегается путем отправки пакетов, которые достаточно малы, чтобы пройти через соединение с наименьшим MTU на пути от отправителя к получателю.

Глава 3.3.3. Глобальные адреса

В вышеописанном обсуждении модели обслуживания IP мы упомянули, что одна из ее функций — это схема адресации. В конце концов, если вы хотите иметь возможность отправлять данные на любой хост в любой сети, необходимо найти способ идентификации всех хостов. Таким образом, нам нужна глобальная схема адресации — такая, при которой у двух хостов не будет одинакового адреса. Глобальная уникальность — это первое свойство, которое должно быть обеспечено в схеме адресации.

Ethernet-адреса являются глобально уникальными, но этого недостаточно для схемы адресации в большой интерсети. Ethernet-адреса также являются *плоскими*, и это означает, что они не имеют структуры и предоставляют очень мало информации для маршрутизирующих протоколов. (На самом деле Ethernet-адреса имеют структуру для целей *обозначения* — первые 24 бита идентифицируют производителя, — но это не дает полезной информации для маршрутизирующих протоколов, так как эта структура не имеет отношения к топологии сети.) В отличие от них, IP-адреса являются *иерархическими*, это означает, что они состоят из нескольких частей, соответствующих определенной иерархии в интер-

сети. Если говорить конкретно, IP-адреса состоят из двух частей, обычно называемых *сетевой* частью и *хостовой* частью. Это довольно логичная структура для интерсети, которая состоит из множества взаимосвязанных сетей. Сетевая часть IP-адреса идентифицирует сеть, к которой подключен хост; все хосты, подключенные к одной сети, имеют одну и ту же сетевую часть в своем IP-адресе. Хостовая часть затем уникально идентифицирует каждый хост в этой сети. Таким образом, в простой интерсети, показанной на рис. 3.15, адреса хостов в сети 1, например, будут иметь одну и ту же сетевую часть и разные хостовые части.

Обратите внимание, что маршрутизаторы на рис. 3.15 подключены к двум сетям. Им необходимо иметь адрес в каждой сети, один для каждого интерфейса. Например, маршрутизатор R1, который находится между беспроводной сетью и Ethernet, имеет IP-адрес на интерфейсе к беспроводной сети, сетевая часть которого совпадает с частью всех хостов в этой сети. Он также имеет IP-адрес на интерфейсе к Ethernet, сетевая часть которого совпадает с частью хостов в этой сети. Таким образом, учитывая, что маршрутизатор может быть реализован как хост с двумя сетевыми интерфейсами, более точно считать, что IP-адреса принадлежат интерфейсам, а не хостам.

Как выглядят эти иерархические адреса? В отличие от некоторых других форм иерархических адресов, размеры двух частей не одинаковы для всех адресов. Изначально IP-адреса были разделены на три разных класса, как показано на рис. 3.20, каждый из которых определяет разные размеры сетевой и хостовой частей. (Существуют также адреса класса D, которые указывают на группу multicast (мультикаст), и адреса класса E, которые в настоящее время не используются.) В любом случае адрес имеет длину 32 бита.

Класс IP-адреса определяется по нескольким наиболее старшим битам. Если первый бит равен 0, это адрес класса А. Если первый бит равен 1, а второй — 0, это адрес класса В. Если первые два бита равны 1, а третий — 0, это адрес класса С. Таким образом, из примерно 4 миллиардов возможных IP-адресов половина относится к классу А, четверть — к классу В, и одна восьмая — к классу С. Каждый класс выделяет определенное количество битов для сетевой части адреса, а остальное — для хостовой части. Сети класса А имеют 7 бит для сетевой части и 24 бита для хостовой части, и это означает, что может существовать только 126 сетей класса А (значения 0 и 127 зарезервированы), но каждая из них может вмещать до $2^{24} - 2$ (примерно 16 миллионов) хостов (снова два значения зарезервированы). Адреса класса В выделяют 14 бит для сети и 16 бит для хоста, это означает, что каждая сеть класса В имеет место для 65 534 хостов. Наконец, адреса класса С имеют только 8 бит для хостовой части и 21 бит для сетевой части. Следовательно, сеть класса С может иметь только 256 уникальных идентификаторов хостов, что означает только 254 подключенных хоста (один идентификатор хоста, 255, зарезервирован для широковещательной рассылки, а 0 не является допустимым номером хоста). Однако схема адресации поддерживает 221 сеть класса С.



Рисунок 3.20. IP-адреса: (a) класс А; (b) класс В; (c) класс С.

На первый взгляд эта схема адресации обладает большой гибкостью, позволяя достаточно эффективно обслуживать сети различных размеров. Изначально предполагалось, что Интернет будет состоять из небольшого количества широкомаштабных сетей (это будут сети класса А), умеренного количества сетей уровня сайта (университетские сети) (это будут сети класса В) и большого количества локальных сетей (LAN) (это будут сети класса С). Однако оказалось, что такая схема недостаточно гибкая, как мы увидим дальше. Сегодня IP-адреса обычно являются «бесклассовыми»; подробности этого объясняются ниже.

Прежде чем мы рассмотрим, как используются IP-адреса, полезно ознакомиться с некоторыми практическими аспектами, например, как их записывать. По общепринятому соглашению IP-адреса записываются как четыре десятичных числа, разделенных точками. Каждое число представляет собой *десятичное* значение, содержащееся в 1 байте адреса, начиная с наиболее значимого. Например, адрес компьютера, на котором был напечатан этот текст, 171.69.210.245.

Важно не путать IP-адреса с доменными именами Интернета, которые также являются иерархическими. Доменные имена обычно являются строками ASCII, разделенными точками, такими как `cs.princeton.edu`. Важное значение имеют именно IP-адреса, поскольку они содержатся в заголовках IP-пакетов, и именно эти адреса используются IP-маршрутизаторами для принятия решений о пересылке.

Глава 3.3.4. Переадресация дейтаграмм в IP

Теперь мы готовы рассмотреть основной механизм, с помощью которого IP-маршрутизаторы пересылают дейтаграммы в интернет. Напомним из предыдущей главы, что *пересылка* — это процесс получения пакета на входе и отправки его на соответствующий выход, тогда как маршрутизация — это процесс построения таблиц, позволяющих определить правильный выход для пакета. Обсуждение здесь сосредоточено на пересылке; *маршрутизацию* мы рассмотрим далее.

Основные моменты, которые следует учитывать при обсуждении пересылки IP-дейтаграмм, следующие:

- Каждая IP-дейтаграмма содержит IP-адрес целевого узла.
- Сетевой сегмент IP-адреса уникально идентифицирует одну физическую сеть, которая является частью большего Интернета.
- Все узлы и маршрутизаторы, которые имеют одинаковую сетевую часть адреса, подключены к одной и той же физической сети и, следовательно, могут обмениваться данными, отправляя кадры по этой сети.
- Каждая физическая сеть, которая является частью Интернета, имеет по крайней мере один маршрутизатор, который по определению также подключен как минимум к одной другой физической сети; этот маршрутизатор может обмениваться пакетами с узлами или маршрутизаторами в обеих сетях.

Таким образом, пересылка IP-дейтаграмм может осуществляться следующим образом. Дейтаграмма отправляется от исходного узла к целевому узлу, возможно, проходя через несколько маршрутизаторов по пути. Любой узел, будь то хост или маршрутизатор, сначала пытается установить, подключен ли он к той же физической сети, что и целевой узел. Для этого он сравнивает сетевую часть адреса назначения с сетевой частью адреса каждого из своих сетевых интерфейсов. (Хосты обычно имеют только один интерфейс, тогда как маршрутизаторы обычно имеют два или более, так как они обычно подключены к двум или более сетям.) Если происходит совпадение, это означает, что целевой узел находится в той же физической сети, что и интерфейс, и пакет может быть доставлен непосредственно по этой сети. В одной из следующих глав будут объяснены некоторые детали этого процесса.

Если узел не подключен к той же физической сети, что и целевой узел, то ему нужно отправить дейтаграмму маршрутизатору. Как правило, у каждого узла есть выбор из не-

скольких маршрутизаторов, поэтому ему нужно выбрать лучший из них или хотя бы тот, который имеет разумный шанс приблизить дейтаграмму к ее цели. Маршрутизатор, который он выбирает, называется маршрутизатором *следующего перехода*. Маршрутизатор находит правильный следующий переход, консультируясь со своей таблицей пересылки. Таблица пересылки концептуально представляет собой просто список пар (NetworkNum, NextHop). (Как мы увидим ниже, таблицы пересылки на практике часто содержат некоторую дополнительную информацию, связанную со следующим переходом.) Обычно также существует маршрутизатор по умолчанию, который используется, если ни одна из записей в таблице не соответствует сетевому номеру адресата. Для хоста может быть вполне допустимо иметь маршрутизатор по умолчанию и ничего больше — это означает, что все дейтаграммы, предназначенные для узлов, не находящихся в физической сети, к которой подключен отправляющий хост, будут отправлены через маршрутизатор по умолчанию.

Мы можем описать алгоритм пересылки дейтаграмм следующим образом:

```

if (NetworkNum назначения = NetworkNum одного из моих интерфейсов), то
    доставить пакет к месту назначения через этот интерфейс
иначе
    if (NetworkNum назначения есть в моей таблице маршрутизации), то
        доставить пакет к маршрутизатору NextHop
    else
        доставить пакет к маршрутизатору по умолчанию

```

Для хоста, имеющего только один интерфейс и только маршрутизатор по умолчанию в таблице пересылки, это упрощается до

```

if (NetworkNum назначения = мой NetworkNum), то
    доставить пакет к месту назначения напрямую
else
    доставить пакет к маршрутизатору по умолчанию

```

Посмотрим, как это работает, на примере интерсети на рис. 3.15. Сначала предположим, что H1 хочет отправить дейтаграмму H2. Поскольку они находятся в одной физической сети, у H1 и H2 одинаковый сетевой номер в их IP-адресе. Таким образом, H1 заключает, что он может доставить дейтаграмму непосредственно H2 по Ethernet. Единственная проблема, которую нужно решить, это как H1 узнает правильный Ethernet-адрес для H2 — механизм разрешения, описанный в следующей главе, решает эту проблему.

Теперь предположим, что H5 хочет отправить дейтаграмму H8. Поскольку эти узлы находятся в разных физических сетях, у них разные сетевые номера, поэтому H5 заключает, что ему нужно отправить дейтаграмму маршрутизатору. R1 — единственный выбор, маршрутизатор по умолчанию, поэтому H5 отправляет дейтаграмму по беспроводной сети к R1. Аналогично R1 знает, что он не может доставить дейтаграмму непосредственно H8, так как ни один из его интерфейсов не находится в той же сети, что и H8. Предположим, что маршрутизатор по умолчанию для R1 — это R2; тогда R1 отправляет дейтаграмму R2 по Ethernet. Предположим, что у R2 есть таблица маршрутизации, показанная в табл. 3.6, он ищет сетевой номер H8 (сеть 4) и пересылает дейтаграмму по сети «точка-точка» к R3. Наконец, R3, находясь в той же сети, что и H8, пересылает дейтаграмму непосредственно H8.

Таблица 3.6.

Таблица переадресации для маршрутизатора R2.

NetworkNum	NextHop
1	R1
4	R3

Обратите внимание, что можно включить информацию о непосредственно подключенных сетях в таблицу маршрутизации. Например, мы могли бы обозначить сетевые интерфейсы маршрутизатора R2 как интерфейс 0 для связи «точка-точка» (сеть 3) и интерфейс 1 для Ethernet (сеть 2). Тогда у R2 была бы таблица маршрутизации, показанная в табл. 3.7.

Таблица 3.7.
Полная таблица переадресации для маршрутизатора R2.

NetworkNum	NextHop
1	R1
2	Интерфейс 1
3	Интерфейс 0
4	R3

Таким образом, для любого сетевого номера, который встречает R2 в пакете, он знает, что делать. Либо эта сеть непосредственно подключена к R2, и в этом случае пакет может быть доставлен к своей цели по этой сети, либо сеть достижима через какой-то следующий маршрутизатор, который R2 может достичь по сети, к которой он подключен. В любом случае R2 будет использовать ARP, описанный ниже, чтобы найти MAC-адрес узла, к которому следует отправить пакет.

Таблица маршрутизации, используемая R2, достаточно проста, чтобы ее можно было настроить вручную. Однако обычно эти таблицы более сложные и создаются путем запуска протокола маршрутизации, такого как один из описанных в следующей главе. Также обратите внимание, что на практике сетевые номера обычно длиннее (например, 128.96).

Теперь мы можем увидеть, как иерархическая адресация — разделение адреса на части сети и узла — улучшила масштабируемость большой сети. Теперь маршрутизаторы содержат таблицы маршрутизации, которые перечисляют только набор сетевых номеров, а не всех узлов в сети. В нашем простом примере это означало, что R2 мог хранить информацию, необходимую для достижения всех узлов в сети (в которой было восемь узлов), в таблице с четырьмя записями. Даже если бы в каждой физической сети было по 100 узлов, R2 все равно потребовались бы те же четыре записи. Это хороший первый шаг (хотя и не последний) к достижению масштабируемости.

Основные выводы

Это иллюстрирует один из самых важных принципов построения масштабируемых сетей: для достижения масштабируемости необходимо уменьшить количество информации, хранящейся в каждом узле и обмениваемой между узлами. Самый распространенный способ сделать это — *иерархическая агрегация*. IP вводит двухуровневую иерархию, с сетями на верхнем уровне и узлами на нижнем уровне. Мы агрегировали информацию, позволяя маршрутизаторам работать только с достижением правильной сети; информация, необходимая маршрутизатору для доставки дейтаграммы любому узлу в данной сети, представлена одним агрегированным элементом информации.

Глава 3.3.5. Подсети и бесклассовая адресация

Первоначальное намерение IP-адресов заключалось в том, чтобы сетевая часть уникально идентифицировала ровно одну физическую сеть. Оказалось, что у этого подхода есть несколько недостатков. Представьте себе большой кампус, который имеет множе-

ство внутренних сетей и решает подключиться к Интернету. Для каждой сети, независимо от ее размера, сайту нужен как минимум адрес сети класса С. Еще хуже, если в сети более 255 узлов, тогда ей нужен адрес класса В. Это может не казаться большой проблемой, и действительно, когда Интернет только зарождался, это не было так важно, но существует лишь конечное количество сетевых номеров, и адресов класса В значительно меньше, чем класса С. Адреса класса В пользуются особенно высоким спросом, потому что никто не знает, расширится ли сеть за пределы 255 узлов, поэтому проще использовать адрес класса В с самого начала, чем перенумеровать каждый узел, когда места в сети класса С не хватит. Проблема, которую мы здесь наблюдаем, это неэффективное распределение адресов: сеть с двумя узлами использует весь сетевой адрес класса С, тем самым тратя впустую 253 вполне пригодных адреса; сеть класса В с чуть более чем 255 узлами тратит более 64 000 адресов.

Присвоение одного сетевого номера на физическую сеть, таким образом, расходует пространство IP-адресов потенциально гораздо быстрее, чем нам хотелось бы. Хотя нам нужно подключить более 4 миллиардов узлов, чтобы исчерпать все действительные адреса, нам нужно подключить всего 2^{14} (около 16 000) сетей класса В, прежде чем эта часть адресного пространства закончится. Поэтому мы хотели бы найти способ более эффективного использования сетевых номеров.

Присвоение большого количества сетевых номеров имеет еще один недостаток, который становится очевидным, когда задумываешься о маршрутизации. Напомним, что объем состояния, хранящегося в узле, участвующем в протоколе маршрутизации, пропорционален количеству других узлов, и что маршрутизация в интерсети заключается в построении таблиц маршрутизации, которые сообщают маршрутизатору, как добраться до разных сетей. Таким образом, чем больше сетевых номеров используется, тем больше становятся таблицы маршрутизации. Большие таблицы маршрутизации увеличивают затраты на маршрутизаторы, и они потенциально медленнее обрабатываются для данной технологии, что снижает производительность маршрутизаторов. Это дает еще одну мотивацию для тщательного назначения сетевых номеров.

Подсеть является первым шагом к сокращению общего количества присвоенных номеров сетей. Идея заключается в том, чтобы взять один IP-номер сети и распределить IP-адреса с этим номером сети по нескольким физическим сетям, которые теперь называются *подсетями*. Для того чтобы это работало, нужно сделать несколько вещей. Во-первых, подсети должны быть расположены близко друг к другу. В результате из удаленной точки в Интернете все они будут выглядеть как одна сеть, имеющая только один номер сети. Это означает, что маршрутизатор сможет выбрать только один маршрут для достижения любой из подсетей, поэтому лучше, чтобы они все находились в одном общем направлении. Идеальная ситуация для использования подсетей — это большой университет или корпорация, у которых есть много физических сетей. Из-за пределов университета все, что вам нужно знать для доступа к любой подсети внутри университета, — это где университет подключается к остальной части Интернета. Это часто происходит в одной точке, поэтому одной записи в вашей таблице маршрутизации будет достаточно. Даже если есть несколько точек, в которых университет подключается к остальной части Интернета, знание того, как добраться до одной точки в сети кампуса, все равно является хорошим началом.

Механизм, с помощью которого один номер сети может быть разделен между несколькими сетями, включает конфигурирование всех узлов на каждой подсети с *маской подсети*. С простыми IP-адресами все узлы в одной сети должны иметь один и тот же номер сети. Маска подсети позволяет нам ввести номер подсети; все узлы в одной физической сети будут иметь один и тот же *номер подсети*, что означает, что узлы могут находиться в разных физических сетях, но иметь один и тот же номер сети. Эта концепция иллюстрируется на рис. 3.21.

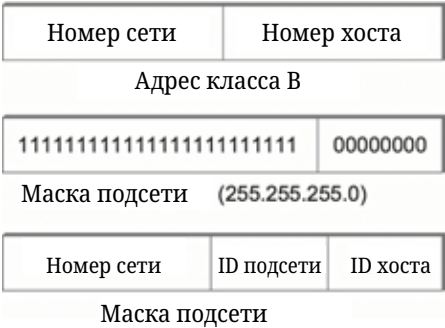


Рисунок 3.21. Адресация подсети.

Подсеть означает для хоста, что он теперь имеет IP-адрес и маску подсети для подсети, к которой он подключен. Например, узел H1 на рис. 3.22 настроен с адресом 128.96.34.15 и маской подсети 255.255.255.128. (Все узлы на данной подсети настроены с одной и той же маской; то есть для каждой подсети существует ровно одна маска подсети.) Побитовая операция AND (И) между этими двумя числами определяет номер подсети узла и всех других узлов в той же подсети. В данном случае 128.96.34.15 AND 255.255.255.128 равно 128.96.34.0, так что это номер подсети для верхней подсети на рисунке.

Когда узел хочет отправить пакет на определенный IP-адрес, первое, что он делает, это выполняет побитовую операцию AND (И) между своей собственной маской подсети и IP-адресом назначения. Если результат равен номеру подсети отправляющего узла, то он знает, что узел назначения находится в той же подсети, и пакет может быть доставлен непосредственно по подсети. Если результаты не равны, пакет необходимо отправить маршрутизатору для пересылки в другую подсеть. Например, если H1 отправляет пакет H2, тогда H1 выполняет операцию AND своей маски подсети (255.255.255.128) с адресом H2 (128.96.34.139), чтобы получить 128.96.34.128. Это не совпадает с номером подсети для H1 (128.96.34.0), поэтому H1 знает, что H2 находится в другой подсети. Поскольку H1 не может доставить пакет H2 напрямую через подсеть, он отправляет пакет своему маршрутизатору по умолчанию R1.

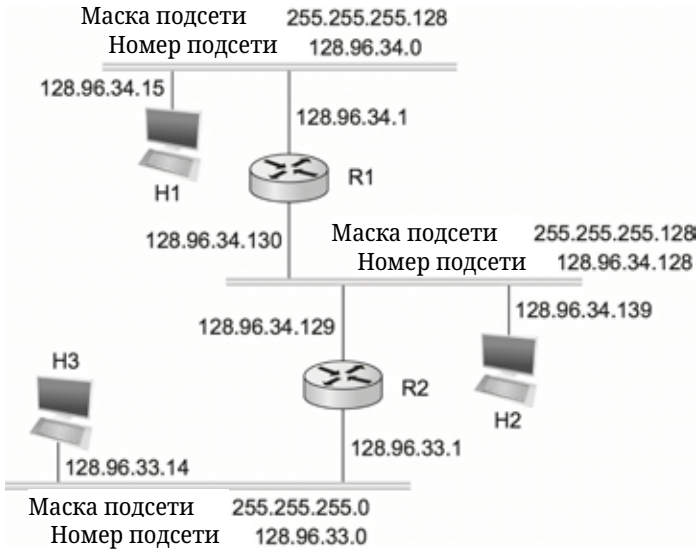


Рисунок 3.22. Пример подсети.

Таблица маршрутизации маршрутизатора также слегка меняется при введении подсети. Напомним, что ранее у нас была таблица маршрутизации, состоящая из записей вида (NetworkNum, NextHop). Для поддержки подсети теперь таблица должна содержать записи вида (SubnetNumber, SubnetMask, NextHop). Чтобы найти правильную запись в таблице, маршрутизатор выполняет операцию AND между адресом назначения пакета и SubnetMask для каждой записи поочередно; если результат совпадает с SubnetNumber записи, то это правильная запись для использования, и он пересылает пакет следующему маршрутизатору, указанному в записи. В примерной сети на рис. 3.22 маршрутизатор R1 будет иметь записи, показанные в табл. 3.8.

Таблица 3.8.
Пример таблицы переадресации с подсетью.

SubnetNumber	SubnetMask	NextHop
128.96.34.0	255.255.255.128	Интерфейс 0
128.96.34.128	255.255.255.128	Интерфейс 1
128.96.33.0	255.255.255.0	R2

Продолжая с примером отправки дейтаграммы от H1 к H2, R1 выполняет операцию AND между адресом H2 (128.96.34.139) и маской подсети первой записи (255.255.255.128) и сравнивает результат (128.96.34.128) с сетевым номером для этой записи (128.96.34.0). Поскольку они не совпадают, он переходит к следующей записи. На этот раз совпадение происходит, поэтому R1 доставляет дейтаграмму H2, используя интерфейс 1, который подключен к той же сети, что и H2.

Теперь мы можем описать алгоритм пересылки дейтаграмм следующим образом:

```

D = IP-адрес назначения
для каждой записи в таблице маршрутизации (SubnetNumber, SubnetMask, NextHop)
    D1 = SubnetMask & D
    if D1 = SubnetNumber
        if NextHop является интерфейсом
            доставить дейтаграмму напрямую к месту назначения
        else
            доставить дейтаграмму к NextHop (маршрутизатор)

```

Хотя это не показано в данном примере, обычно в таблицу включается маршрут по умолчанию, который используется, если явные совпадения не найдены. Обратите внимание, что наивная реализация этого алгоритма — включающая повторное выполнение операции AND с адресом назначения и маской подсети, которая может не отличаться каждый раз, и линейный поиск в таблице — была бы неэффективной.

Важное свойство подсети заключается в том, что разные части интернета видят мир по-разному. Снаружи нашего гипотетического университета маршрутизаторы видят одну сеть. В приведенном выше примере маршрутизаторы вне университета видят совокупность сетей на рис. 3.22 как сеть 128.96 и хранят одну запись в своих таблицах маршрутизации, чтобы знать, как до нее добраться. Однако маршрутизаторы внутри университета должны иметь возможность маршрутизировать пакеты в правильную подсеть. Таким образом, не все части интернета видят одну и ту же информацию о маршрутизации. Это пример *агрегации* информации о маршрутизации, которая является фундаментальной для масштабирования системы маршрутизации. В следующем разделе показано, как агрегацию можно вывести на другой уровень.

Бесклассовая адресация

Подсеть имеет аналог, иногда называемый *суперсетью*, но чаще называемый *бесклассовой междоменной маршрутизацией* или CIDR (Classless Interdomain Routing), произносится как «сидр». CIDR доводит идею подсети до логического завершения, фактически упраздняя классы адресов. Почему одной подсети недостаточно? По сути, подсеть позволяет нам только разделить адрес класса на несколько подсетей, в то время как CIDR позволяет объединить несколько адресов класса в одну «суперсеть». Это дополнительно решает проблему неэффективности адресного пространства, отмеченную выше, и делает это таким образом, чтобы система маршрутизации не была перегружена.

Чтобы понять, как связаны проблемы эффективности адресного пространства и масштабируемости системы маршрутизации, рассмотрим гипотетический случай компании, в сети которой есть 256 хостов. Это немного больше, чем для адреса класса С, поэтому логичным кажется присвоение класса В. Однако использование части адресного пространства, которая может адресовать 65535 хостов для адресации 256 хостов, имеет эффективность всего $256/65,535 = 0.39\%$. Даже если подсеть может помочь нам тщательно назначать адреса, она не решает проблему того, что любая организация с более чем 255 хостами или ожиданием, что их будет столько, хочет адрес класса В.

Первый способ, которым вы могли бы решить эту проблему, — отказать в присвоении адреса класса В любой организации, запрашивающей его, если она не сможет показать необходимость в чем-то близком к 64К адресов, и вместо этого дать ей соответствующее количество адресов класса С для покрытия ожидаемого количества хостов. Поскольку мы теперь будем выделять адресное пространство порциями по 256 адресов за раз, мы сможем точнее сопоставить объем потребляемого адресного пространства с размером организации. Для любой организации с как минимум 256 хостами мы можем гарантировать использование адресов не менее чем на 50 %, и обычно намного больше. (К сожалению, даже если вы можете обосновать запрос на номер сети класса В, не утруждайте себя, потому что все они давно были разобраны.)

Это решение, однако, создает проблему, которая не менее серьезна: чрезмерные требования к памяти маршрутизаторов. Если одному сайту назначено, скажем, 16 номеров сети класса С, это означает, что каждому магистральному маршрутизатору Интернета требуется 16 записей в таблицах маршрутизации для направления пакетов к этому сайту. Это верно, даже если путь ко всем этим сетям одинаков. Если бы мы присвоили сайту адрес класса В, та же информация о маршрутизации могла бы храниться в одной записи таблицы. Однако эффективность использования адресов была бы тогда только $6 \times 255 / 65,536 = 6.2\%$.

Следовательно, CIDR пытается сбалансировать желание минимизировать количество маршрутов, которые нужно знать маршрутизатору, с необходимостью эффективно раздавать адреса. Для этого CIDR помогает нам *агрегировать* маршруты. То есть он позволяет нам использовать одну запись в таблице маршрутизации, чтобы указать, как добраться до множества разных сетей. Как отмечено выше, он делает это, разрывая жесткие границы между классами адресов. Чтобы понять, как это работает, рассмотрим нашу гипотетическую организацию с 16 номерами сети класса С. Вместо того чтобы раздавать 16 адресов случайным образом, мы можем раздать блок непрерывных адресов класса С. Предположим, мы присваиваем номера сети класса С от 192.4.16 до 192.4.31. Обратите внимание, что старшие 20 бит всех адресов в этом диапазоне одинаковы (11000000 00000100 0001). Таким образом, мы фактически создали 20-битный номер сети — что-то среднее между номером сети класса В и номером сети класса С по количеству поддерживаемых хостов. Другими словами, мы получаем как высокую эффективность адресации, раздавая адреса порциями меньше, чем сеть класса В, так и один префикс сети, который можно использовать в таблицах маршрутизации. Обратите внимание, что для работы этой схемы мы должны раздавать блоки адресов класса С, которые имеют общий префикс, а это означает, что каждый блок должен содержать количество сетей класса С, равное степени двойки.

CIDR требует нового типа нотации для представления сетевых номеров, или *префиксов*, как их называют, потому что префиксы могут быть любой длины. Принято ставить /X после префикса, где X — это длина префикса в битах. Таким образом, для приведенного выше примера 20-битный префикс для всех сетей от 192.4.16 до 192.4.31 обозначается как 192.4.16/20. Напротив, если бы мы захотели представить один номер сети класса C, который имеет длину 24 бита, мы бы написали его как 192.4.16/24. Сегодня, когда CIDR является нормой, чаще можно услышать, как люди больше говорят о префиксах «слеш 24», чем о сетях класса C. Обратите внимание, что представление сетевого адреса таким образом похоже на подход (маска, значение), используемый в подсетировании, если маски состоят из непрерывных битов, начиная с самого значимого бита (что на практике почти всегда так).

Возможность агрегировать маршруты на границе сети, как мы только что видели, является лишь первым шагом. Представьте себе сеть интернет-провайдера, основная задача которой — предоставление интернет-подключения большому количеству корпораций и университетов (клиентов). Если мы присваиваем префиксы клиентам таким образом, что многие разные клиентские сети, подключенные к сети провайдера, разделяют общий, более короткий адресный префикс, тогда мы можем добиться еще большей агрегации маршрутов. Рассмотрим пример на рис. 3.23. Предположим, что восемь клиентов, обслуживаемых сетью провайдера, получили смежные 24-битные сетевые префиксы. Эти префиксы начинаются с одних и тех же 21 бита. Поскольку все клиенты доступны через одну и ту же сеть провайдера, он может рекламировать единый маршрут к ним, просто рекламируя общий 21-битный префикс, который они разделяют. И он может это сделать, даже если не все 24-битные префиксы были выданы, пока у провайдера есть право выдавать эти префиксы клиентам. Один из способов достичь этого — назначить часть адресного пространства провайдеру заранее, а затем позволить провайдеру назначать адреса из этого пространства своим клиентам по мере необходимости. Обратите внимание, что, в отличие от этого простого примера, не нужно, чтобы все клиентские префиксы были одинаковой длины.

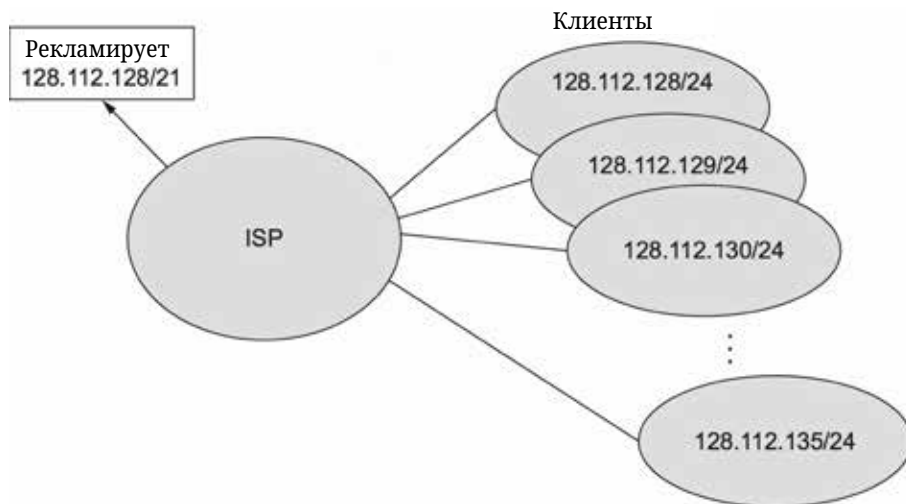


Рисунок 3.23. Агрегация маршрутов с помощью CIDR.

Переадресация IP-адресов

Во всех наших обсуждениях пересылки IP до сих пор мы предполагали, что можем найти номер сети в пакете, а затем найти этот номер в таблице пересылки. Однако теперь, когда мы ввели CIDR, нам нужно пересмотреть это предположение. CIDR означа-

ет, что префиксы могут быть любой длины, от 2 до 32 бит. Более того, иногда возможно, что префиксы в таблице пересылки «перекрываются» в том смысле, что некоторые адреса могут соответствовать более чем одному префиксу. Например, мы можем найти как 171.69 (16-битный префикс), так и 171.69.10 (24-битный префикс) в таблице пересылки одного маршрутизатора. В этом случае пакет, направленный, например, на 171.69.10.5, явно соответствует обоим префиксам. Правило в этом случае основано на принципе «самого длинного совпадения»; то есть пакет соответствует самому длинному префиксу, который в данном примере будет 171.69.10. С другой стороны, пакет, направленный на 171.69.20.5, будет соответствовать 171.69, а не 171.69.10, и при отсутствии другой соответствующей записи в таблице маршрутизации 171.69 будет самым длинным совпадением.

Задача эффективного нахождения самого длинного совпадения между IP-адресом и префиксами переменной длины в таблице пересылки была плодотворной областью исследований в течение многих лет. Наиболее распространенный алгоритм использует подход, известный как *дерево PATRICIA*, который был разработан задолго до появления CIDR.

Глава 3.3.6. Преобразование адресов (ARP)

В предыдущей главе мы говорили о том, как доставлять IP-дейтаграммы в нужную физическую сеть, но упустили вопрос о том, как доставить дейтаграмму к конкретному хосту или маршрутизатору в этой сети. Основная проблема заключается в том, что IP-дейтаграммы содержат IP-адреса, но физическое сетевое оборудование хоста или маршрутизатора, которому вы хотите отправить дейтаграмму, понимает только схему адресации этой конкретной сети. Таким образом, нам нужно перевести IP-адрес на адрес канального уровня, который имеет смысл в этой сети (например, 48-битный Ethernet-адрес). Мы можем затем инкапсулировать IP-дейтаграмму в кадр, содержащий этот адрес канального уровня, и отправить его либо на конечный пункт назначения, либо на маршрутизатор, который обещает переслать дейтаграмму к конечному пункту назначения.

Один простой способ сопоставить IP-адрес с физическим сетевым адресом — закодировать физический адрес хоста в части его IP-адреса. Например, хост с физическим адресом 00100001 01001001 (который имеет десятичное значение 33 в старшем байте и 81 в младшем байте) может получить IP-адрес 128.96.33.81. Хотя это решение использовалось в некоторых сетях, оно ограничено тем, что физические адреса сети в этом примере не могут быть длиннее 16 бит; они могут быть только 8 бит длиной в сети класса C. Это явно не подходит для 48-битных Ethernet-адресов.

Более общее решение заключается в том, чтобы каждый хост вел таблицу пар адресов; то есть таблица будет сопоставлять IP-адреса с физическими адресами. Хотя этой таблицей мог бы централизованно управлять системный администратор и затем копировать на каждый хост в сети, лучший подход заключается в том, чтобы каждый хост динамически заполнял таблицу с использованием сети. Это можно осуществить с помощью *протокола разрешения адресов* (Address Resolution Protocol, ARP). Цель ARP — позволить каждому хосту в сети создавать таблицу сопоставлений между IP-адресами и адресами канального уровня. Поскольку эти сопоставления могут меняться со временем (например, из-за того, что Ethernet-карта в хосте ломается и заменяется новой с новым адресом), записи периодически устаревают и удаляются. Это происходит примерно каждые 15 минут. Набор сопоставлений, хранящихся в данный момент на хосте, известен как ARP-кеш или ARP-таблица.

ARP использует тот факт, что многие сетевые технологии канального уровня, такие как Ethernet, поддерживают широковещательную рассылку. Если хост хочет отправить IP-дейтаграмму хосту (или маршрутизатору), который, как он знает, находится в той же сети (то есть отправляющие и принимающие узлы имеют один и тот же IP-номер сети), он сначала проверяет наличие сопоставления в кеше. Если сопоставление не найдено, он должен задействовать протокол разрешения адресов в сети. Он делает это, отправляя широковещательный ARP-запрос в сеть. Этот запрос содержит интересующий IP-адрес

(целевой IP-адрес). Каждый хост получает запрос и проверяет, совпадает ли он с его IP-адресом. Если совпадение обнаружено, хост отправляет ответное сообщение, содержащее его адрес канального уровня, обратно инициатору запроса. Инициатор добавляет информацию, содержащуюся в этом ответе, в свою ARP-таблицу.

Сообщение запроса также включает IP-адрес и адрес канального уровня отправляющего узла. Таким образом, когда узел передает запросное сообщение в широковещательном режиме, каждый узел в сети может узнать канальный и IP-адреса отправителя и поместить эту информацию в свою таблицу ARP. Однако не каждый узел добавляет эту информацию в свою таблицу ARP. Если в таблице уже есть запись для этого узла, она «обновляется»; то есть сбрасывается время до удаления записи. Если этот узел является целью запроса, то он добавляет информацию об отправителе в свою таблицу, даже если у него уже не было записи для этого узла. Это происходит потому, что есть высокая вероятность того, что исходный узел собирается отправить ему сообщение уровня приложения, и ему, возможно, придется отправить ответ или ACK обратно исходному узлу; для этого ему понадобится физический адрес отправителя. Если узел не является целью запроса и у него нет записи для источника в таблице ARP, он не добавляет запись для источника. Это происходит потому, что нет причин полагать, что этому узлу когда-либо понадобится канальный адрес источника; нет необходимости засорять свою таблицу ARP этой информацией.

Рис. 3.24 показывает формат пакета ARP для сопоставления адресов IP и Ethernet. На самом деле ARP может использоваться для множества других видов сопоставлений — основные различия заключаются в размерах адресов. В дополнение к IP и адресам канального уровня как отправителя, так и цели пакет содержит:

- Поле `HardwareType`, которое указывает тип физической сети (например, Ethernet).
- Поле `ProtocolType`, которое указывает протокол верхнего уровня (например, IP).
- Поля `HLen` (длина аппаратного адреса) и `PLen` (длина протокольного адреса), которые указывают длину адреса канального уровня и адреса протокола верхнего уровня соответственно.
- Поле `Operation`, которое указывает, является ли это запросом или ответом.
- Аппаратные (Ethernet) и протокольные (IP) адреса источника и цели.

0		8		16		31	
Hardware type = 1				ProtocolType = 0x0800			
HLen = 48		PLen = 32		Operation			
SourceHardwareAddr (bytes 0–3)							
SourceHardwareAddr (bytes 4–5)				SourceProtocolAddr (bytes 0–1)			
SourceProtocolAddr (bytes 2–3)				TargetHardwareAddr (bytes 0–1)			
TargetHardwareAddr (bytes 2–5)							
TargetProtocolAddr (bytes 0–3)							

Рисунок 3.24. Формат ARP-пакета для сопоставления IP-адресов с адресами Ethernet.

Отметим, что результаты процесса ARP могут быть добавлены в качестве дополнительного столбца в таблицу маршрутизации, подобную той, что представлена в табл. 3.6. Таким образом, например, когда R2 нужно переслать пакет в сеть 2, он не только определяет, что следующий узел — это R1, но также находит MAC-адрес, который необходимо указать в пакете для отправки его к R1.

Основные выводы

Теперь мы рассмотрели основные механизмы, которые IP предоставляет для работы как с гетерогенностью, так и с масштабированием. В вопросе гетерогенности IP начинается с определения модели сервиса с максимально возможными усилиями, которая делает минимальные предположения об основных сетях; наиболее заметно, что эта модель сервиса основана на ненадежных дейтаграммах. Затем IP вносит два важных дополнения к этой отправной точке: (1) общий формат пакета (фрагментация/сборка — это механизм, который делает данный формат работающим в сетях с различными MTU) и (2) глобальное адресное пространство для идентификации всех узлов (ARP — это механизм, который делает это глобальное адресное пространство работающим в сетях с различными физическими схемами адресации). В вопросе масштабирования IP использует иерархическую агрегацию для уменьшения объема информации, необходимой для пересылки пакетов. В частности, IP-адреса разделяются на сетевые и хост-компоненты, при этом пакеты сначала направляются к сети назначения, а затем доставляются к правильному узлу в этой сети.

Конфигурация хоста (DHCP)

Ethernet-адреса настраиваются в сетевых адаптерах производителем, и этот процесс управляется таким образом, чтобы гарантировать глобальную уникальность этих адресов. Это явно достаточное условие, чтобы гарантировать, что любая группа узлов, подключенных к одному Ethernet (включая расширенную LAN), будет иметь уникальные адреса. Более того, уникальность — это все, что требуется от Ethernet-адресов.

IP-адреса, напротив, должны быть не только уникальными в данной междетской сети, но и отражать структуру междетской сети. Как упомянуто выше, они содержат сетевую часть и хост-часть, и сетевая часть должна быть одинаковой для всех узлов в одной сети. Таким образом, невозможно настроить IP-адрес один раз на этапе производства узла, так как это означало бы, что производитель знает, какие узлы окажутся в каких сетях, и это означало бы, что узел, подключенный к одной сети, никогда не сможет переместиться в другую. По этой причине IP-адреса должны быть перенастраиваемыми.

Кроме IP-адреса есть еще несколько частей информации, которые узел должен иметь, прежде чем он сможет начать отправлять пакеты. Наиболее заметным из них является адрес маршрутизатора по умолчанию — место, куда он может отправлять пакеты, чьи адреса назначения не находятся в той же сети, что и отправляющий узел.

Большинство операционных систем хостов предоставляют способ для системного администратора или даже пользователя вручную настраивать IP-информацию, необходимую для узла; однако есть некоторые очевидные недостатки такой ручной конфигурации. Один из них заключается в том, что это просто много работы — настраивать все узлы в большой сети напрямую, особенно учитывая, что такие узлы недоступны по сети, пока они не настроены. Еще более важно, что процесс конфигурации подвержен ошибкам, так как необходимо обеспечить, чтобы каждый узел получил правильный сетевой номер, и чтобы у двух узлов не было одного и того же IP-адреса. По этим причинам требуются автоматизированные методы конфигурации. Основным методом является протокол, известный как *протокол динамической конфигурации хостов* (DHCP, Dynamic Host Configuration Protocol).

DHCP опирается на существование DHCP-сервера, который отвечает за предоставление конфигурационной информации узлам. В административном домене есть как минимум один DHCP-сервер. На самом простом уровне DHCP-сервер может функционировать как централизованное хранилище конфигурационной информации хоста. Рассмотрим, например, проблему управления адресами в междетской сети большой компании. DHCP освобождает сетевых администраторов от необходимости обходить каждый узел в компании с листом адресов и картой сети в руках и настраивать каждый узел вручную. Вместо этого конфигурационная информация для каждого узла может храниться на DHCP-сервере и автоматически извлекаться каждым узлом при его загрузке или подключении к сети. Однако администратор все

же будет выбирать адрес, который должен получить каждый узел; он просто будет хранить эту информацию на сервере. В этой модели конфигурационная информация для каждого узла хранится в таблице, индексируемой по какому-либо уникальному идентификатору клиента, обычно по аппаратному адресу (например, Ethernet-адресу его сетевого адаптера).

Использование DHCP освобождает сетевого администратора даже от обязанности назначать адреса отдельным узлам. В этой модели DHCP-сервер поддерживает пул доступных адресов, которые он выдает узлам по запросу. Это значительно снижает объем конфигурации, которую должен выполнять администратор, так как теперь необходимо только выделить диапазон IP-адресов (все с одним сетевым номером) для каждой сети.

Поскольку цель DHCP — минимизировать объем ручной настройки, необходимой для работы узла, это будет противоречить цели, если каждый узел должен был бы быть настроен с адресом DHCP-сервера. Таким образом, первой проблемой, с которой сталкивается DHCP, является проблема обнаружения сервера.

Процесс передачи сообщения от узла к удаленному DHCP-серверу показан на рисунке 3.25.

Для связи с DHCP-сервером вновь загруженный или подключенный узел отправляет сообщение DHCPDISCOVER на специальный IP-адрес (255.255.255.255), который является IP-адресом широковещательной рассылки. Это означает, что оно будет получено всеми узлами и маршрутизаторами в этой сети. (Маршрутизаторы не пересылают такие пакеты на другие сети, предотвращая широковещательную рассылку по всему Интернету.) В простейшем случае один из этих узлов является DHCP-сервером для сети. Сервер ответит узлу, который сгенерировал сообщение обнаружения (все остальные узлы его проигнорируют). Однако нежелательно требовать наличия одного DHCP-сервера в каждой сети, так как это все равно создаст потенциально большое количество серверов, которые нужно будет правильно и согласованно настраивать. Таким образом, DHCP использует концепцию *ретранслятора*. В каждой сети есть как минимум один ретранслятор, и он настроен только с одной информацией: IP-адресом DHCP-сервера. Когда ретранслятор получает сообщение DHCPDISCOVER, он отправляет его на DHCP-сервер и ждет ответа, который затем отправит обратно запрашивающему клиенту. Процесс ретрансляции сообщения от узла к удаленному DHCP-серверу показан на рис. 3.25.

Рис. 3.26 ниже показывает формат сообщения DHCP. Сообщение на самом деле отправляется с использованием протокола, называемого *протоколом дейтаграмм пользователя* (User Datagram Protocol, UDP), который работает поверх IP. UDP подробно рассматривается далее, но единственное, что он делает в этом контексте, это предоставляет ключ демultipлексирования, который говорит: «Это DHCP-пакет».

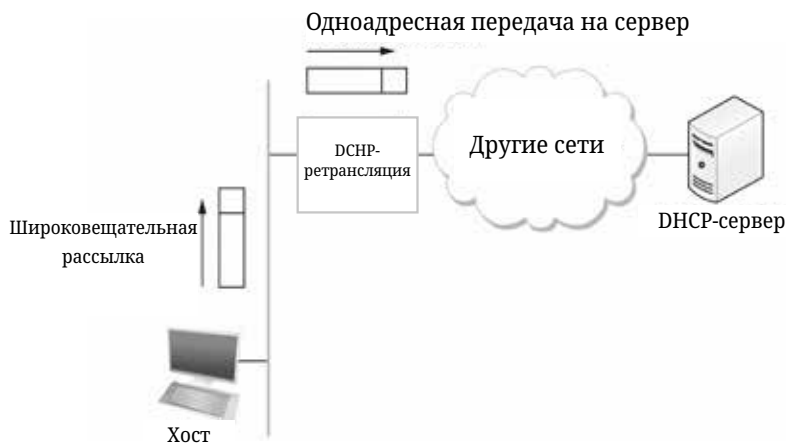


Рисунок 3.25. Агент ретрансляции DHCP получает широковещательное сообщение DHCPDISCOVER от хоста и отправляет одноадресное сообщение DHCPDISCOVER на DHCP-сервер.

DHCP был порожден от более раннего протокола под названием BOOTP, и поэтому некоторые поля пакета не строго относятся к конфигурации узла. При попытке получить конфигурационную информацию клиент помещает свой аппаратный адрес (например, свой Ethernet-адрес) в поле `chaddr`. DHCP-сервер отвечает, заполняя поле `yiaddr` (ваш IP-адрес) и отправляя его клиенту. Другая информация, такая как маршрутизатор по умолчанию, который должен использоваться этим клиентом, может быть включена в поле `options`.

В случае когда DHCP динамически назначает IP-адреса узлам понятно, что узлы не могут сохранять адреса бесконечно, так как это в конечном итоге приведет к исчерпанию пула адресов у сервера. В то же время нельзя рассчитывать на то, что узел вернет свой адрес, так как он мог выйти из строя, быть отключен от сети или выключен. Таким образом, DHCP позволяет арендовать адреса на определенный период времени. После истечения срока аренды сервер может вернуть этот адрес в свой пул. Узел с арендованным адресом, очевидно, должен периодически обновлять аренду, если он все еще подключен к сети и функционирует правильно.

Основные выводы

DHCP иллюстрирует важный аспект масштабирования: масштабирование управления сетью. Хотя обсуждения масштабирования часто сосредоточены на предотвращении быстрого роста состояния в сетевых устройствах, важно обращать внимание на рост сложности управления сетью. Позволяя сетевым администраторам настраивать диапазон IP-адресов для каждой сети, а не один IP-адрес для каждого узла, DHCP улучшает управляемость сети.

Следует отметить, что DHCP может также внести некоторую дополнительную сложность в управление сетью, так как делает привязку между физическими узлами и IP-адресами гораздо более динамичной. Это может усложнить работу сетевого администратора, если, например, потребуется найти неисправный узел.

Operation	HType	HLen	Hops
Xid			
Secs	Flags		
ciaddr			
yiaddr			
siaddr			
giaddr			
chaddr (16 bytes)			
sname (64 bytes)			
file (128 bytes)			
options			

Рисунок 3.26. Формат DHCP-пакета.

Глава 3.3.8. Сообщение об ошибках (ICMP)

Следующий вопрос — как Интернет обрабатывает ошибки. Хотя IP совершенно спокойно сбрасывает дейтаграммы, когда становится трудно, например, когда маршрутизатор не знает, как переслать дейтаграмму, или когда один фрагмент дейтаграммы не до-

стигает места назначения, он не обязательно делает это «молча». IP всегда настраивается с протоколом-компаньоном, известным как *протокол управляющих сообщений Интернета* (Internet Control Message Protocol, ICMP), который определяет набор сообщений об ошибках, отправляемых обратно на исходный узел, когда маршрутизатор или узел не могут успешно обработать IP-дейтаграмму. Например, ICMP определяет сообщения об ошибках, указывающие на то, что узел назначения недоступен (возможно, из-за сбоя соединения), что процесс сборки не удался, что TTL достиг нуля, что контрольная сумма заголовка IP не прошла проверку и так далее.

ICMP также определяет несколько управляющих сообщений, которые маршрутизатор может отправить обратно на исходный узел. Одно из наиболее полезных управляющих сообщений, называемое *ICMP-Redirect*, сообщает исходному узлу, что существует лучший маршрут к месту назначения. ICMP-Redirect используется в следующей ситуации. Предположим, что узел подключен к сети, к которой присоединены два маршрутизатора, назовем их R1 и R2, где узел использует R1 в качестве маршрутизатора по умолчанию. Если R1 когда-либо получит дейтаграмму от узла, где на основе своей таблицы маршрутизации он знает, что R2 был бы лучшим выбором для определенного адреса назначения, он отправляет ICMP-Redirect обратно на узел, инструктируя его использовать R2 для всех будущих дейтаграмм, адресованных к этому месту назначения. Узел затем добавляет этот новый маршрут в свою таблицу маршрутизации.

ICMP также предоставляет основу для двух широко используемых инструментов отладки: `ping` и `tracroute`. `ping` использует ICMP-эхо-сообщения для определения, доступен ли узел и функционирует ли он. `tracroute` использует слегка неочевидную технику для определения набора маршрутизаторов на пути к месту назначения, что является темой одного из упражнений в конце этой главы.

Глава 3.3.9. Виртуальные сети и туннели

Мы завершаем наше введение в IP, рассмотрев вопрос, который вы могли не предвидеть, но который становится все более важным. Наше обсуждение до этого момента было сосредоточено на том, чтобы сделать возможной связь между узлами в разных сетях без ограничений. Обычно это и есть цель в Интернете — все хотят иметь возможность отправлять электронную почту всем, а создатель нового веб-сайта хочет достичь максимально широкой аудитории. Однако есть много ситуаций, когда требуется более контролируемая связь. Важным примером такой ситуации является *виртуальная частная сеть* (virtual private network, VPN).

Термин *VPN* часто используется слишком широко, и определения могут различаться, но интуитивно мы можем определить VPN, сначала рассмотрев идею частной сети. Корпорации с множеством офисов часто строят частные сети, арендуя каналы у телефонных компаний и используя эти линии для соединения офисов. В такой сети связь ограничена только между офисами этой корпорации, что часто желательно по соображениям безопасности. Чтобы сделать частную сеть виртуальной, арендованные линии передачи, которые не используются другими корпорациями, будут заменены какой-либо общей сетью. *Виртуальная цепь* (virtual circuit, VC) является очень разумной заменой арендованной линии, потому что она все равно обеспечивает логическое соединение «точка-точка» между офисами корпорации. Например, если корпорация X имеет VC от офиса A до офиса B, то она явно может отправлять пакеты между офисами A и B. Но корпорация Y не сможет доставить свои пакеты в офис B без заранее установленной своей собственной виртуальной цепи к офису B, и установка такой VC может быть административно предотвращена, тем самым предотвращая нежелательную связь между корпорацией X и корпорацией Y.

Рис. 3.27 (а) показывает две частные сети для двух отдельных корпораций. На рис. 3.27 (b) они обе мигрируют в сеть виртуальных цепей. Ограниченная связь реальной частной сети сохраняется, но поскольку частные сети теперь используют одни и те же средства передачи и коммутаторы, мы говорим, что были созданы две виртуальные частные сети.

На рис. 3.27 используется сеть виртуальных цепей (например, с использованием АТМ), чтобы обеспечить контролируемую связь между офисами. Также можно предоставить аналогичную функцию, используя IP-сеть для обеспечения связи. Однако мы не можем просто подключить различные офисы корпораций к одной интерсети, потому что это обеспечит связь между корпорацией X и корпорацией Y, чего мы хотим избежать. Чтобы решить эту проблему, нам нужно ввести новое понятие, *IP-туннель*.

Мы можем представить IP-туннель как виртуальное соединение «точка-точка» между парой узлов, которые на самом деле разделены произвольным количеством сетей. Виртуальное соединение создается в маршрутизаторе на входе в туннель путем предоставления ему IP-адреса маршрутизатора на дальнем конце туннеля. Каждый раз, когда маршрутизатор на входе в туннель хочет отправить пакет через это виртуальное соединение, он инкапсулирует пакет внутри IP-дейтаграммы. Адрес назначения в IP-заголовке — это адрес маршрутизатора на дальнем конце туннеля, в то время как адрес источника — это адрес маршрутизатора, выполняющего инкапсуляцию.

В таблице маршрутизации маршрутизатора на входе в туннель это виртуальное соединение выглядит как обычное соединение. Рассмотрим, например, сеть на рис. 3.28. Туннель был настроен от R1 до R2 и назначен виртуальному интерфейсу номер 0. Таблица маршрутизации в R1 может выглядеть как табл. 3.9.

Таблица 3.9.
Таблица переадресации для маршрутизатора R1.

NetworkNum	NextHop
1	Интерфейс 0
2	Виртуальный интерфейс 0
По умолчанию	Интерфейс 1

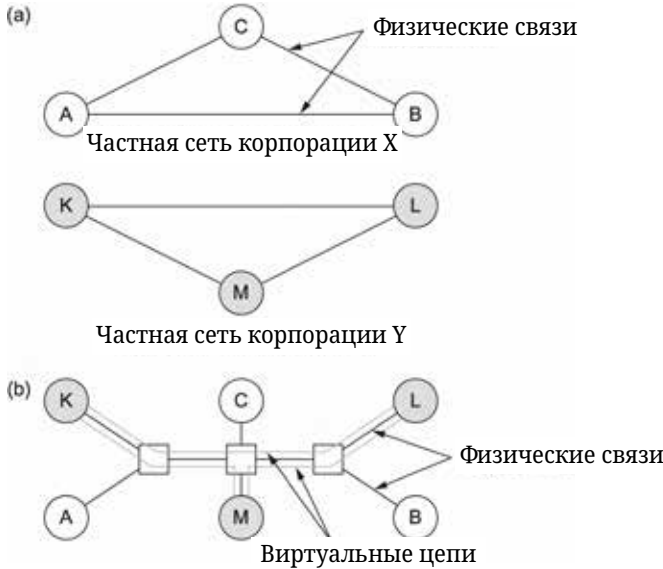


Рисунок 3.27. Пример виртуальных частных сетей: (а) две отдельные частные сети; (б) две виртуальные частные сети с общими коммутаторами.



Рисунок 3.28. Туннель через интернет. 18.5.0.1 — это адрес R2, к которому можно обратиться из R1 через интернет-сеть.

У R1 есть два физических интерфейса. Интерфейс 0 подключен к сети 1, а интерфейс 1 подключен к большой интерсети и поэтому используется по умолчанию для всего трафика, который не соответствует более конкретным записям в таблице маршрутизации. Кроме того, у R1 есть виртуальный интерфейс, который является интерфейсом туннеля. Предположим, что R1 получает пакет из сети 1, содержащий адрес в сети 2. Таблица маршрутизации указывает, что этот пакет должен быть отправлен через виртуальный интерфейс 0. Чтобы отправить пакет через этот интерфейс, маршрутизатор берет пакет, добавляет IP-заголовок, адресованный R2, и затем продолжает пересылать пакет, как если бы он только что был получен. Адрес R2 — 18.5.0.1; так как номер сети этого адреса — 18, а не 1 или 2, пакет, предназначенный для R2, будет отправлен через интерфейс по умолчанию в интернет.

Когда пакет покидает R1, он выглядит для остального мира как обычный IP-пакет, предназначенный для R2, и маршрутизируется соответствующим образом. Все маршрутизаторы в межсети пересылают его обычными способами, пока он не достигнет R2. Когда R2 получает пакет, он обнаруживает, что пакет содержит его собственный адрес, удаляет IP-заголовок и смотрит на полезную нагрузку пакета. Что он находит, так это внутренний IP-пакет, адрес назначения которого находится в сети 2. R2 теперь обрабатывает этот пакет как любой другой полученный IP-пакет. Поскольку R2 напрямую подключен к сети 2, он пересылает пакет в эту сеть. Рис. 3.28 показывает изменения инкапсуляции пакета по мере его перемещения через сеть.

Хотя R2 действует как конечная точка туннеля, это не мешает ему выполнять обычные функции маршрутизатора. Например, он может получать некоторые пакеты, которые не проходят через туннель, но адресованы к сетям, которых он может достичь, и он пересылает их обычным образом.

Вы можете задаться вопросом, зачем кому-то создавать туннель и изменять инкапсуляцию пакета по мере его прохождения через интернет. Одна из причин — безопасность. С дополнением шифрования туннель может стать приватным видом связи через общедоступную сеть. Другая причина может заключаться в том, что у R1 и R2 есть некоторые возможности, которые недоступны в промежуточных сетях, такие как маршрутизация мультимедиа. Соединив эти маршрутизаторы с помощью туннеля, мы можем создать виртуальную сеть, в которой все маршрутизаторы с этой возможностью будут казаться напрямую подключенными. Третья причина для создания туннелей — перенос пакетов протоколов, отличных от IP, через IP-сеть. Пока маршрутизаторы на обоих концах туннеля знают, как обрабатывать эти другие протоколы, IP-туннель выглядит для них как соединение «точка-точка», по которому они могут отправлять не-IP пакеты. Туннели также предоставляют механизм, с помощью которого мы можем за-

ставить пакет быть доставленным в определенное место, даже если его оригинальный заголовок — тот, который инкапсулируется внутри заголовка туннеля — может указывать на то, что он должен идти в другое место. Таким образом, мы видим, что туннелирование является мощной и довольно общей техникой для создания виртуальных соединений через интерсети. Настолько общей, что эта техника может рекурсивно использоваться, при этом наиболее распространенным случаем использования является туннелирование IP через IP.

Туннелирование имеет свои недостатки. Один из них заключается в увеличении длины пакетов; это может представлять значительную потерю пропускной способности для коротких пакетов. Длинные пакеты могут быть подвержены фрагментации, что имеет свои собственные недостатки. Также могут быть проблемы с производительностью для маршрутизаторов на обоих концах туннеля, поскольку они должны выполнять больше работы, чем при обычной пересылке, добавляя и удаляя заголовок туннеля. Наконец, существует стоимость управления для административной единицы, отвечающей за настройку туннелей и обеспечение их правильной обработки протоколами маршрутизации.

Глава 3.4. Маршрутизация

До сих пор в этом разделе мы предполагали, что коммутаторы и маршрутизаторы обладают достаточным пониманием топологии сети, чтобы они могли выбирать правильный порт для вывода каждого пакета. В случае виртуальных каналов маршрутизация является проблемой только для пакета запроса на установление соединения; все последующие пакеты следуют по тому же пути, что и запрос. В дейтаграммных сетях, включая IP-сети, маршрутизация является проблемой для каждого пакета. В любом случае коммутатор или маршрутизатор должен иметь возможность посмотреть на адрес назначения и затем определить, какой из выходных портов является лучшим выбором для доставки пакета по этому адресу. Как мы видели в предыдущей главе, коммутатор принимает это решение, консультируясь с таблицей маршрутизации. Основная проблема маршрутизации заключается в том, как коммутаторы и маршрутизаторы получают информацию в своих таблицах маршрутизации.

Основные выводы

Мы напоминаем вам про важное различие, которое часто игнорируется, между пересылкой и маршрутизацией. Пересылка заключается в приеме пакета, поиске его адреса назначения в таблице и отправке пакета в направлении, определенном этой таблицей. Мы видели несколько примеров пересылки ранее. Это простой и хорошо определенный процесс, выполняемый локально на каждом узле, и часто его называют «плоскостью данных» сети. Маршрутизация — это процесс, с помощью которого строятся таблицы пересылки. Он зависит от сложных распределенных алгоритмов и часто называется «плоскостью управления» сети.

Хотя термины «*таблица пересылки*» и «*таблица маршрутизации*» иногда используются как взаимозаменяемые, мы сделаем здесь различие между ними. Таблица пересылки используется, когда пакет пересылается, и поэтому должна содержать достаточно информации для выполнения функции пересылки. Это означает, что строка в таблице пересылки содержит отображение от сетевого префикса к исходящему интерфейсу и некоторую информацию о MAC-адресе, такую как Ethernet-адрес следующего перехода. С другой стороны, таблица маршрутизации — это таблица, которая строится алгоритмами маршрутизации как предварительный этап построения таблицы пересылки. Она обычно содержит отображения от сетевых префиксов к следующим переходам. Она также может содержать сведения о том, как была получена эта информация, чтобы маршрутизатор мог решить, когда следует удалить какую-либо информацию.

Являются ли таблица маршрутизации и таблица пересылки фактически отдельными структурами данных — это вопрос реализации, но существуют многочисленные причины для их разделения. Например, таблица пересылки должна быть структурирована для оптимизации процесса поиска адреса при пересылке пакета, в то время как таблица маршрутизации должна быть оптимизирована для целей вычисления изменений в топологии. Во многих случаях таблица пересылки, может быть, даже реализована в специализированном оборудовании, тогда как это редко, если вообще когда-либо, делается для таблицы маршрутизации.

Табл. 3.10 приводит пример строки из таблицы маршрутизации, которая указывает, что сетевой префикс 18/8 должен быть достигнут через маршрутизатор следующего перехода с IP-адресом 171.69.245.10.

Таблица 3.10.
Пример строки из таблицы маршрутизации.

Префикс/Длина	Следующий скачок
18/8	171.69.245.10

Табл. 3.11 дает пример строки из таблицы пересылки, которая содержит информацию о том, как именно пересылать пакет к следующему переходу: отправить его через интерфейс номер 0 с MAC-адресом 8:0:2b:e4:b:1:2. Обратите внимание, что последняя часть информации предоставляется протоколом разрешения адресов (ARP).

Таблица 3.11.
Пример строки из таблицы переадресации

Префикс/Длина	Интерфейс	MAC-адрес
18/8	if0	8:0:2b:e4:b:1:2

Перед тем как углубляться в детали маршрутизации, нам нужно напомнить себе о ключевом вопросе, который мы должны задавать каждый раз, когда пытаемся создать механизм для Интернета: «Масштабируется ли это решение?» Ответ для алгоритмов и протоколов, описанных в этом разделе, будет «не очень». Они разработаны для сетей довольно скромного размера — на практике до нескольких сотен узлов. Однако описанные решения служат строительными блоками для иерархической инфраструктуры маршрутизации, которая используется в Интернете сегодня. В частности, протоколы, описанные в этом разделе, в совокупности известны как *внутридоменные* протоколы маршрутизации, или *протоколы внутреннего шлюза* (interior gateway protocols, IGP). Чтобы понять эти термины, нам нужно определить домен маршрутизации. Хорошим рабочим определением будет интернет, в которой все маршрутизаторы находятся под одним административным контролем (например, на территории одного университетского кампуса или в сети одного интернет-провайдера). Значимость этого определения станет очевидной в следующем разделе, когда мы рассмотрим *междоменные* протоколы маршрутизации. На данный момент важно помнить, что мы рассматриваем проблему маршрутизации в контексте малых и средних сетей, а не для сети размером с Интернет.

Глава 3.4.1. Сеть как граф

Маршрутизация, по сути, является задачей теории графов. На рис. 3.29 показан граф, представляющий сеть. Узлы графа, обозначенные буквами от A до F, могут быть хостами, коммутаторами, маршрутизаторами или сетями. В начальном обсуждении мы сосредото-

точимся на случае, когда узлы являются маршрутизаторами. Ребра графа соответствуют сетевым ссылкам. Каждое ребро имеет связанную с ним *стоимость*, что дает некоторое представление о желательности отправки трафика по этой ссылке. Обсуждение того, как назначаются стоимости ребер, приведено позже.

Обратите внимание, что в примерах сетей (графов), используемых на протяжении всей этой главы, ребра ненаправленные, и им присваивается одна стоимость. Это фактически небольшое упрощение. Более точным будет сделать ребра направленными, что обычно означает наличие пары ребер между каждым узлом — одного в каждом направлении, и каждое с собственной стоимостью.

Основная проблема маршрутизации заключается в нахождении пути с наименьшей стоимостью между любыми двумя узлами, где стоимость пути равна сумме стоимостей всех ребер, составляющих этот путь. Для простой сети, такой как та, что на рис. 3.29, можно представить себе расчет всех кратчайших путей и загрузку их в некоторое энергонезависимое хранилище на каждом узле. Такой статический подход имеет несколько недостатков:

- Он не учитывает добавление новых узлов или ссылок.
- Он предполагает, что стоимости ребер не могут изменяться, хотя нам может потребоваться изменять стоимости ссылок со временем (например, присвоить высокую стоимость ссылке, которая сильно загружена).

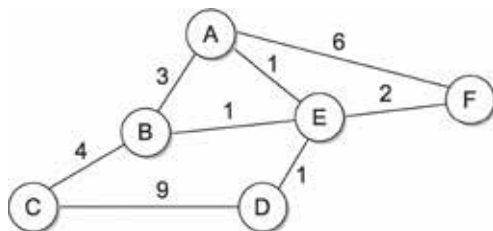


Рисунок 3.29. Сеть, представленная в виде графа.

По этим причинам маршрутизация в большинстве практических сетей осуществляется с помощью запуска протоколов маршрутизации между узлами. Эти протоколы предоставляют распределенный, динамический способ решения задачи нахождения пути с наименьшей стоимостью в условиях отказов ссылок и узлов, а также изменяющихся стоимостей ребер. Обратите внимание на слово «*распределенный*» в предыдущем предложении; централизованные решения сложно масштабировать, поэтому все широко используемые протоколы маршрутизации используют распределенные алгоритмы.

Распределенная природа алгоритмов маршрутизации является одной из основных причин, почему это поле было таким распространенным для исследований и разработок — существует множество вызовов в том, чтобы заставить распределенные алгоритмы работать хорошо. Например, распределенные алгоритмы повышают вероятность того, что два маршрутизатора в какой-то момент времени будут иметь разные представления о кратчайшем пути к некоторому пункту назначения. Фактически каждый из них может считать, что другой ближе к пункту назначения, и решит отправлять пакеты другому. Очевидно, такие пакеты будут застревать в цикле, пока разногласия между двумя маршрутизаторами не будут устранены, и хорошо бы разрешить их как можно скорее. Это всего лишь один пример типа проблемы, с которой должны справляться протоколы маршрутизации.

Чтобы начать наш анализ, предположим, что стоимости ребер в сети известны. Мы рассмотрим два основных класса протоколов маршрутизации: протоколы *дистанционно-векторной* (distance vector) маршрутизации и протоколы *состояния ссылок* (link state). В следующем разделе мы вернемся к проблеме вычисления стоимостей ребер значимым образом.

Глава 3.4.2. Алгоритм Distance-Vector (RIP)

Идея алгоритма Distance-Vector (дистанционно-векторного) следует из его названия. (Другое распространенное название для этого класса алгоритмов — алгоритм Беллмана – Форда, по именам его изобретателей.) Каждый узел создает одномерный массив (вектор), содержащий «расстояния» (стоимости) до всех других узлов, и распределяет этот вектор своим непосредственным соседям. Начальное предположение для маршрутизации с использованием дистанционно-векторного алгоритма заключается в том, что каждый узел знает стоимость ссылки до каждого из своих непосредственно подключенных соседей. Эти стоимости могут быть предоставлены, когда маршрутизатор настраивается сетевым администратором. Ссылке, которая не работает, присваивается бесконечная стоимость.

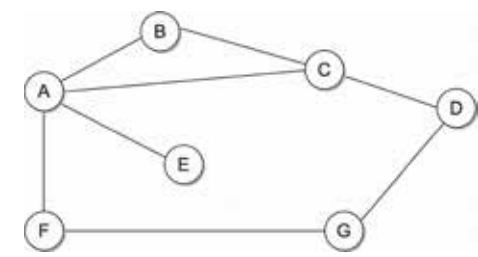


Рисунок 3.30. Дистанционно-векторная маршрутизация: пример сети.

Таблица 3.12.
Начальные расстояния, сохраненные в каждом узле (глобальный вид).

	A	B	C	D	E	F	G
A	0	1	1	∞	1	1	∞
B	1	0	1	∞	∞	∞	∞
C	1	1	0	1	∞	∞	∞
D	∞	∞	1	0	∞	∞	1
E	1	∞	∞	∞	0	∞	∞
F	1	∞	∞	∞	∞	0	1
G	∞	∞	∞	1	∞	1	0

Чтобы понять, как работает дистанционно-векторный алгоритм маршрутизации, проще всего рассмотреть пример, который показан на рисунке 3.30. В этом примере стоимость каждой ссылки равна 1, так что путь с наименьшей стоимостью просто является путем с наименьшим количеством переходов. (Поскольку все ребра имеют одинаковую стоимость, мы не показываем стоимости на графе.) Мы можем представить знания каждого узла о расстояниях до всех других узлов в виде таблицы, подобной табл. 3.12. Обратите внимание, что каждый узел знает только информацию в одной строке таблицы (той, что носит его имя в левой колонке). Глобальный вид, представленный здесь, недоступен в какой-либо одной точке сети.

Мы можем рассматривать каждую строку в табл. 3.12 как список расстояний от одного узла до всех других узлов, представляющий текущие убеждения этого узла. Изначально каждый узел устанавливает стоимость 1 до своих непосредственно подключенных соседей и бесконечность до всех других узлов. Таким образом, А изначально считает, что он может достигнуть В за один переход, а D недостижим.

Таблица маршрутизации, хранящаяся в А, отражает этот набор убеждений и включает имя следующего перехода, который А будет использовать для достижения любого достижимого узла. Изначально таблица маршрутизации А будет выглядеть как табл. 3.13.

Таблица 3.13.
Начальная таблица маршрутизации на узле А.

Пункт назначения	Стоимость	Следующий скачок
В	1	В
С	1	С
Д	∞	—
Е	1	Е
F	1	F
G	∞	—

Следующим шагом в дистанционно-векторной маршрутизации является то, что каждый узел отправляет сообщение своим непосредственно подключенным соседям, содержащее его личный список расстояний. Например, узел F сообщает узлу А, что он может достичь узла G за стоимость 1; А также знает, что он может достичь F за стоимость 1, поэтому он складывает эти стоимости, чтобы получить стоимость достижения G через F. Эта общая стоимость 2 меньше текущей стоимости бесконечности, поэтому А записывает, что он может достичь G за стоимость 2 через F. Аналогично А узнает от С, что D можно достичь от С за стоимость 1; он добавляет это к стоимости достижения С (1) и решает, что D можно достичь через С за стоимость 2, что лучше, чем старая стоимость, которая равна бесконечности. В то же время А узнает от С, что В можно достичь от С за стоимость 1, поэтому он заключает, что стоимость достижения В через С составляет 2. Поскольку это хуже текущей стоимости достижения В (1), эта новая информация игнорируется. На данном этапе А может обновить свою таблицу маршрутизации с новыми стоимостями и следующими переходами для всех узлов в сети. Результат показан в табл. 3.14.

Таблица 3.14.
Итоговая таблица маршрутизации в узле А.

Пункт назначения	Стоимость	Следующий скачок
В	1	В
С	1	С
Д	2	С
Е	1	Е
F	1	F
G	2	F

При отсутствии каких-либо изменений в топологии требуется всего несколько обменов информацией между соседями, чтобы каждый узел получил полную таблицу маршрутизации. Процесс получения согласованной информации о маршрутизации для всех узлов называется *сходимостью* (convergence). Табл. 3.15 показывает конечный набор стоимостей от каждого узла до всех других узлов, когда маршрутизация сошлась. Мы должны подчеркнуть, что в сети нет ни одного узла, который обладал бы всей информацией из этой таблицы — каждый узел знает только содержимое своей собственной таблицы маршрутизации. Преимущество такого распределенного алгоритма заключается в том, что он позволяет всем узлам достичь согласованного представления о сети без участия какой-либо центральной власти.

Таблица 3.15.
Итоговые расстояния, сохраненные в каждом узле (глобальный вид).

	A	B	C	D	E	F	G
A	0	1	1	2	1	1	2
B	1	0	1	2	2	2	3
C	1	1	0	1	2	2	2
D	2	2	1	0	3	2	1
E	1	2	2	3	0	2	1
F	1	2	2	2	2	0	1
G	2	3	2	1	3	1	0

Есть несколько деталей, которые нужно уточнить перед завершением нашего обсуждения дистанционно-векторной маршрутизации. Во-первых, отметим, что существуют две различные ситуации, при которых узел решает отправить обновление маршрутизации своим соседям. Одна из этих ситуаций — это *периодическое обновление*. В этом случае каждый узел автоматически отправляет сообщение об обновлении через определенные промежутки времени, даже если ничего не изменилось. Это служит для того, чтобы другие узлы знали, что этот узел все еще работает. Это также гарантирует, что они продолжают получать информацию, которая может понадобиться им, если их текущие маршруты станут непригодными. Частота этих периодических обновлений варьируется от протокола к протоколу, но обычно составляет от нескольких секунд до нескольких минут. Второй механизм, иногда называемый *триггерным обновлением*, происходит всякий раз, когда узел замечает сбой связи или получает обновление от одного из своих соседей, которое вызывает изменение одного из маршрутов в его таблице маршрутизации. Всякий раз, когда таблица маршрутизации узла изменяется, он отправляет обновление своим соседям, что может привести к изменению их таблиц, заставляя их отправлять обновление своим соседям.

Теперь рассмотрим, что происходит, когда связь или узел выходит из строя. Узлы, которые замечают это первыми, отправляют новые списки расстояний своим соседям, и система обычно довольно быстро приходит к новому состоянию. На вопрос о том, как узел обнаруживает сбой, есть несколько ответов. В одном подходе узел постоянно тестирует связь с другим узлом, отправляя управляющий пакет и проверяя, получает ли он подтверждение. В другом подходе узел определяет, что связь (или узел на другом конце связи) вышла из строя, если он не получает ожидаемое периодическое обновление маршрутизации в течение последних нескольких циклов обновлений.

Чтобы понять, что происходит, когда узел обнаруживает сбой связи, рассмотрим, что происходит, когда F обнаруживает, что его связь с G вышла из строя. Во-первых, F устанавливает новое расстояние до G равным бесконечности и передает эту информацию A. Поскольку A знает, что его двухпереходный путь к G проходит через F, A также устанавливает

расстояние до G равным бесконечности. Однако при следующем обновлении от С А узнает, что С имеет двухпереходный путь к G. Таким образом, А будет знать, что он может достичь G за 3 перехода через С, что меньше, чем бесконечность, и поэтому А обновит свою таблицу соответственно. Когда он сообщит это F, узел F узнает, что он может достигнуть G за 4 перехода через А, что меньше бесконечности, и система снова станет стабильной.

К сожалению, другие обстоятельства могут помешать стабилизации сети. Предположим, например, что связь от А к Е выходит из строя. В следующем раунде обновлений А объявляет о расстоянии до Е, равном бесконечности, но В и С объявляют о расстоянии до Е, равном 2. В зависимости от точного времени событий может произойти следующее: узел В, узнав, что Е можно достичь за 2 перехода от С, заключает, что он может достичь Е за 3 перехода, и объявляет об этом А; узел А заключает, что он может достигнуть Е за 4 перехода, и объявляет об этом С; узел С заключает, что он может достигнуть Е за 5 переходов, и так далее. Этот цикл прекращается только тогда, когда расстояния достигают некоторого числа, которое достаточно велико, чтобы считаться бесконечностью. В это время ни один из узлов фактически не знает, что Е недостижим, и таблицы маршрутизации сети не стабилизируются. Эта ситуация известна как *проблема счета до бесконечности* (count to infinity problem).

Существует несколько решений этой проблемы. Первое заключается в использовании относительно небольшого числа в качестве приближенного значения бесконечности. Например, мы можем решить, что максимальное количество переходов через определенную сеть никогда не превысит 16, и поэтому мы можем выбрать 16 в качестве значения, представляющего бесконечность. Это, по крайней мере, ограничивает время, необходимое для счета до бесконечности. Конечно, это также может вызвать проблему, если наша сеть разрастется до такой степени, что некоторые узлы будут отделены более чем 16 переходами.

Один из методов, улучшающих время стабилизации маршрутизации, называется *split horizon* (разделенный горизонт). Идея заключается в том, что, когда узел отправляет обновление маршрутизации своим соседям, он не отправляет те маршруты, которые он узнал от каждого соседа, обратно этому соседу. Например, если у В есть маршрут (Е, 2, А) в его таблице, то он знает, что этот маршрут был получен от А, и поэтому, когда В отправляет обновление маршрутизации А, он не включает маршрут (Е, 2) в это обновление. В более сильной вариации *split horizon*, называемой *split horizon with poison reverse* (разделенный горизонт с ядовитым реверсом), В фактически отправляет этот маршрут обратно А, но включает отрицательную информацию в маршрут, чтобы гарантировать, что А в конечном итоге не будет использовать В для достижения Е. Например, В отправляет маршрут (Е, ∞) А. Проблема обоих этих методов заключается в том, что они работают только для маршрутов, включающих два узла. Для более крупных маршрутов требуются более радикальные меры. Возвращаясь к предыдущему примеру, если бы В и С подождали некоторое время после получения информации о сбое связи от А перед тем, как объявить маршруты к Е, они бы обнаружили, что ни у одного из них на самом деле нет маршрута к Е. К сожалению, этот подход замедляет сходимость протокола; скорость сходимости является одним из ключевых преимуществ конкурирующего метода — маршрутизации состояния связи (link-state routing), который будет рассмотрен далее.

Реализация

Код, реализующий этот алгоритм, очень прост; здесь приведены только основные моменты. Структура Route определяет каждую запись в таблице маршрутизации, а константа MAX_TTL указывает, как долго запись хранится в таблице, прежде чем будет удалена.

```
#define MAX_ROUTES 128          /* максимальный размер таблицы маршрутизации */
#define MAX_TTL 120             /* время (в секундах) до истечения срока действия
маршрута */
```



```
typedef struct {
    NodeAddr Destination; /* адрес назначения */
    NodeAddr NextHop; /* адрес следующего узла */
    int Cost; /* метрика расстояния */
    u_short TTL; /* время жизни */
} Route;
int numRoutes = 0;
Route routingTable[MAX_ROUTES];
```

Процедура, обновляющая таблицу маршрутизации локального узла на основе нового маршрута, представлена командой `mergeRoute`. Хотя это не показано, функция таймера периодически сканирует список маршрутов в таблице маршрутизации узла, уменьшает поле TTL (время жизни) каждого маршрута и отбрасывает все маршруты, время жизни которых равно 0. Заметьте, однако, что поле TTL сбрасывается до `MAX_TTL` каждый раз, когда маршрут подтверждается сообщением обновления от соседнего узла.

```
void mergeRoute(Route *new) {
    int i;
    for (i = 0; i < numRoutes; ++i) {
        if (new->Destination == routingTable[i].Destination) {
            if (new->Cost + 1 < routingTable[i].Cost) {
                /* найден лучший маршрут: */
                break;
            } else if (new->NextHop == routingTable[i].NextHop) {
                /* метрика для текущего следующего узла могла измениться: */
                break;
            } else {
                /* маршрут неинтересен – просто игнорируем его */
                return;
            }
        }
    }

    if (i == numRoutes) {
        /* это совершенно новый маршрут; есть ли место для него? */
        if (numRoutes < MAX_ROUTES) {
            ++numRoutes;
        } else {
            /* не удастся добавить этот маршрут в таблицу, поэтому отказываемся */
            return;
        }
    }

    routingTable[i] = *new;
    /* сбросить TTL */
    routingTable[i].TTL = MAX_TTL;
    /* учитывать переход до следующего узла */
    ++routingTable[i].Cost;
}
```

Наконец, процедура `updateRoutingTable` является основной рутинной, вызывающей `mergeRoute` для включения всех маршрутов, содержащихся в обновлении маршрутизации, полученном от соседнего узла.

```

void updateRoutingTable(Route *newRoute, int numNewRoutes) {
    int i;
    for (i = 0; i < numNewRoutes; ++i) {
        mergeRoute(&newRoute[i]);
    }
}

```

Протокол маршрутизации информации (RIP)

Один из наиболее широко используемых протоколов маршрутизации в IP-сетях — это *протокол маршрутизации информации* (Routing Information Protocol, RIP). Его широкое использование в ранние дни IP было обусловлено в немалой степени тем, что он распространялся вместе с популярной версией Unix от Berkeley Software Distribution (BSD), из которой были получены многие коммерческие версии Unix. Он также является чрезвычайно простым. RIP является каноническим примером протокола маршрутизации, построенного на основе описанного выше алгоритма вектора расстояний.

Протоколы маршрутизации в межсетевых соединениях немного отличаются от идеализированной модели графа, описанной выше. В межсетевых соединениях задача маршрутизаторов заключается в том, чтобы научиться пересылать пакеты в различные *сети*. Таким образом, вместо того чтобы рекламировать стоимость достижения других маршрутизаторов, маршрутизаторы рекламируют стоимость достижения сетей. Например, на рис. 3.31 маршрутизатор С будет рекламировать маршрутизатору А факт, что он может достичь сетей 2 и 3 (к которым он напрямую подключен) с нулевой стоимостью, сетей 5 и 6 со стоимостью 1 и сети 4 со стоимостью 2.

Мы можем увидеть доказательства этого в формате пакета RIP (версия 2) на рис. 3.32. Большая часть пакета занята тройками информации (адрес, маска, расстояние). Однако принципы работы алгоритма маршрутизации остаются прежними. Например, если маршрутизатор А узнает от маршрутизатора В, что сеть Х может быть достигнута с меньшей стоимостью через В, чем через существующий следующий скачок в таблице маршрутизации, А обновляет информацию о стоимости и следующем скачке для номера сети соответствующим образом.

RIP на самом деле является довольно простой реализацией маршрутизации вектора расстояний. Маршрутизаторы, работающие с RIP, отправляют свои объявления каждые 30 секунд; маршрутизатор также отправляет сообщение об обновлении всякий раз, когда обновление от другого маршрутизатора заставляет его изменить свою таблицу маршрутизации. Одним из интересных моментов является то, что он поддерживает несколько семейств адресов, а не только IP — это причина, по которой в объявлениях присутствует часть Family. RIP версии 2 (RIPv2) также ввел подмаски, описанные в предыдущем разделе, тогда как RIP версии 1 работал со старыми классами адресов IP.

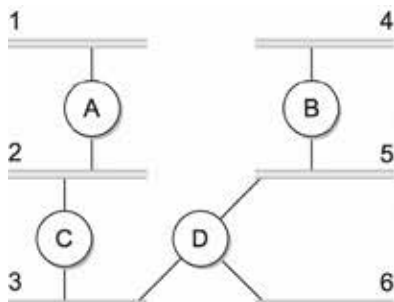


Рисунок 3.31. Пример сети, в которой работает RIP.

0	8	16	31
Команда	Версия	Должна быть нулевой	
Семейство сети 1		Теги маршрута	
Префикс адреса 1			
Маска сети 1			
Расстояние до сети 1			
Семейство сети 2		Теги маршрута	
Префикс адреса 2			
Маска сети 2			
Расстояние до сети 2			

Рисунок 3.32. Формат пакета RIPv2.

Как мы увидим ниже, можно использовать различные метрики или стоимости для ссылок в протоколе маршрутизации. RIPv2 принимает самый простой подход, при котором все стоимости ссылок равны 1, как в нашем примере выше. Таким образом, он всегда старается найти маршрут с минимальным количеством переходов. Допустимые расстояния варьируются от 1 до 15, где 16 представляет бесконечность. Это также ограничивает RIPv2 использованием в довольно маленьких сетях — тех, где нет путей длиннее 15 переходов.

Глава 3.4.3. Маршрутизация по состоянию канала (OSPF)

Маршрутизация по состоянию канала — это второй основной класс внутридоменных протоколов маршрутизации. Начальные предположения для маршрутизации по состоянию канала довольно похожи на предположения для маршрутизации по вектору расстояний. Предполагается, что каждый узел способен определить состояние канала к своим соседям (включен или выключен) и стоимость каждого канала. Снова мы хотим предоставить каждому узлу достаточно информации, чтобы он мог найти путь с наименьшей стоимостью к любой цели. Основная идея протоколов состояния канала очень проста: каждый узел знает, как добраться до своих напрямую подключенных соседей, и если мы убедимся, что совокупность этих знаний будет передана каждому узлу, то каждый узел будет иметь достаточно информации о сети, чтобы построить полную карту сети. Это является достаточным условием (хотя и не необходимым) для нахождения кратчайшего пути к любой точке в сети. Таким образом, протоколы маршрутизации по состоянию канала полагаются на два механизма: надежное распространение информации о состоянии канала и вычисление маршрутов на основе суммы всех накопленных знаний о состоянии канала.

Надежное широковещение

Надежное широковещение — это процесс, который обеспечивает, чтобы все узлы, участвующие в протоколе маршрутизации, получили копию информации о состоянии канала от всех других узлов. Как следует из термина «широковещение», основная идея заключается в том, что узел отправляет информацию о состоянии канала по всем своим

напрямую подключенным каналам; каждый узел, получивший эту информацию, затем пересылает ее по всем своим каналам. Этот процесс продолжается до тех пор, пока информация не достигнет всех узлов в сети.

Если говорить более точно, каждый узел создает пакет обновления, также называемый *пакетом состояния канала* (LSP, link-state packet), который содержит следующую информацию:

- Идентификатор узла, создавшего LSP
- Список напрямую подключенных соседей этого узла с указанием стоимости канала к каждому из них
- Порядковый номер
- Время жизни этого пакета

Первые два пункта необходимы для расчета маршрута; последние два используются для того, чтобы процесс широковещания пакета до всех узлов был надежным. Надежность включает в себя обеспечение наличия самой последней копии информации, поскольку в сети могут передаваться несколько противоречивых LSP от одного узла. Обеспечение надежного широковещания оказалось довольно сложной задачей. (Например, ранняя версия маршрутизации по состоянию канала, использовавшаяся в ARPANET, вызвала сбой этой сети в 1981 году.)

Широковещание работает следующим образом. Во-первых, передача LSP между смежными маршрутизаторами осуществляется надежно с использованием подтверждений и повторных передач, как в надежном протоколе канального уровня. Однако для надежного широковещания LSP на всех узлах в сети необходимо выполнить несколько дополнительных шагов.

Рассмотрим узел X, который получает копию LSP, созданную другим узлом Y. Обратите внимание, что Y может быть любым другим маршрутизатором в той же области маршрутизации, что и X. X проверяет, сохранил ли он уже копию LSP от Y. Если нет, он сохраняет LSP. Если у него уже есть копия, он сравнивает порядковые номера; если новый LSP имеет больший порядковый номер, предполагается, что он является более новым, и этот LSP сохраняется, заменяя старый. Меньший (или равный) порядковый номер подразумевает, что LSP старше (или не новее) сохраненного, поэтому он отбрасывается и дальнейших действий не требуется. Если полученный LSP был более новым, X затем отправляет копию этого LSP всем своим соседям, кроме соседа, от которого этот LSP только что был получен. То, что LSP не отправляется обратно узлу, от которого он был получен, помогает прекратить широковещание LSP. Поскольку X передает LSP всем своим соседям, которые затем делают то же самое, самая последняя копия LSP в конечном итоге достигает всех узлов.

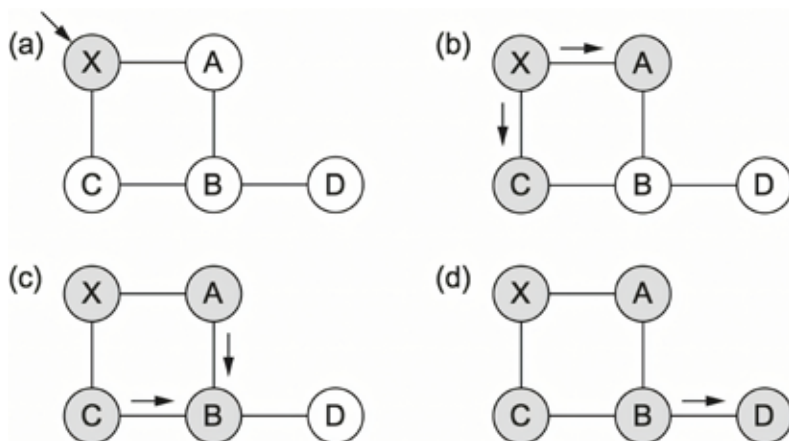


Рисунок 3.33. Рассылка пакетов с состоянием связей: (a) LSP прибывает в узел X; (b) X передает LSP в A и C; (c) A и C передают LSP в B (но не в X); (d) передача завершена.

На рис. 3.33 показано, как LSP распространяется в небольшой сети. Каждый узел закрашивается, когда сохраняет новый LSP. На рис. 3.33 (а) LSP поступает на узел X, который отправляет его соседям A и C на рис. 3.33 (б). A и C не отправляют его обратно на X, но передают его на B. Так как B получает две одинаковые копии LSP, он примет первую из них и проигнорирует вторую как дубликат. Затем он передает LSP на D, у которого нет соседей для дальнейшего распространения, и процесс завершается.

Как и в RIP, каждый узел генерирует LSP в двух случаях. Либо по истечении периодического таймера, либо при изменении топологии узел может сгенерировать новый LSP. Однако единственной причиной, связанной с топологией, для генерации LSP является сбой одного из напрямую подключенных каналов или ближайших соседей. Отказ канала в некоторых случаях может быть обнаружен протоколом канального уровня. Потеря соседа или потеря связи с ним могут быть обнаружены с помощью периодических пакетов «hello». Каждый узел отправляет эти пакеты своим ближайшим соседям через определенные интервалы. Если в течение достаточно долгого времени не поступает «hello» от соседа, канал к этому соседу считается неработоспособным, и генерируется новый LSP, чтобы отразить этот факт.

Одной из важных целей разработки механизма широковещания протокола состояния канала является обеспечение того, чтобы новейшая информация была как можно быстрее распространена по всем узлам, а старая информация была удалена из сети и не продолжала циркулировать. Кроме того, желательно минимизировать общий объем трафика маршрутизации, отправляемого по сети; в конце концов, это всего лишь накладные расходы с точки зрения пользователей, которые действительно используют сеть для своих приложений. В следующих нескольких абзацах описаны некоторые способы достижения этих целей.

Один из простых способов уменьшить накладные расходы — это избегать генерации LSP, если это не абсолютно необходимо. Это можно сделать, используя очень длинные таймеры (часто на уровне часов) для периодической генерации LSP. Учитывая, что протокол широковещания действительно надежен при изменении топологии, можно предположить, что сообщения о том, что «ничего не изменилось», не нужно отправлять очень часто.

Чтобы гарантировать, что старая информация заменяется новой, LSP имеют порядковые номера. Каждый раз, когда узел генерирует новый LSP, он увеличивает порядковый номер на 1. В отличие от большинства порядковых номеров, используемых в протоколах, эти порядковые номера не должны сбрасываться, поэтому поле должно быть довольно большим (например, 64 бита). Если узел выходит из строя и затем снова включается, он начинает с порядкового номера 0. Если узел был отключен долгое время, все старые LSP этого узла истекут (как описано ниже); в противном случае этот узел в конечном итоге получит копию своего собственного LSP с более высоким порядковым номером, который он сможет увеличить и использовать в качестве своего собственного порядкового номера. Это обеспечивает замену нового LSP на все его старые LSP, оставшиеся до отключения узла.

LSP также имеют время жизни (TTL). Это используется для гарантии того, что старая информация о состоянии канала со временем удаляется из сети. Узел всегда уменьшает TTL вновь полученного LSP перед его широковещанием своим соседям. Он также «старит» LSP, пока он хранится в узле. Когда TTL достигает 0, узел повторно отправляет LSP с TTL, равным 0, что интерпретируется всеми узлами в сети как сигнал для удаления этого LSP.

Расчет маршрута

Как только узел получает копию LSP (Link-State Packet) от каждого другого узла, он может построить полную карту топологии сети и на основе этой карты выбрать наилучший маршрут к каждому пункту назначения. Вопрос заключается в том, как именно рассчитываются маршруты на основе этой информации. Решение основано на известном алгоритме из теории графов — алгоритме поиска кратчайшего пути Дейкстры.

Сначала определим алгоритм Дейкстры в терминах теории графов. Представим, что узел берет все полученные LSP и строит графическое представление сети, где N обозначает множество узлов в графе, $l(i, j)$ обозначает неотрицательную стоимость (вес), связанную с ребром между узлами i и j в N , и $l(i, j) = \infty$, если узлы i и j не соединены. В следующем описании мы обозначим s в N как этот узел, то есть узел, выполняющий алгоритм

для поиска кратчайшего пути до всех остальных узлов в N . Также алгоритм поддерживает следующие две переменные: M обозначает множество узлов, включенных на данный момент алгоритмом, и $C(n)$ обозначает стоимость пути от s до каждого узла n . Учитывая эти определения, алгоритм определяется следующим образом:

```

M = {s}
для каждого узла n из N - {s}
    C(n) = l(s, n)
пока (N != M)
    M = M + {w}, такой что C(w) минимально для всех w из (N-M)
    для каждого узла n из (N-M)
        C(n) = MIN(C(n), C(w) + l(w, n))

```

Основная идея алгоритма такова. Мы начинаем с M , содержащего узел s , и затем инициализируем таблицу стоимостей (массив $C(n)$) для других узлов, используя известные стоимости до напрямую подключенных узлов. Затем мы ищем узел, достижимый по наименьшей стоимости (w), и добавляем его в M . Наконец, мы обновляем таблицу стоимостей, учитывая стоимость достижения узлов через w . В последней строке алгоритма мы выбираем новый маршрут к узлу n , который проходит через узел w , если общая стоимость от источника до w , а затем следуя от w до n , меньше старого маршрута до n . Эта процедура повторяется до тех пор, пока все узлы не будут включены в M .

На практике каждый коммутатор вычисляет свою таблицу маршрутизации непосредственно из собранных LSP, используя реализацию алгоритма Дейкстры, называемую алгоритмом *прямого поиска*. В частности, каждый коммутатор поддерживает два списка, известных как Tentative (предварительный) и Confirmed (подтвержденный). Каждый из этих списков содержит набор записей вида (Destination, Cost, NextHop). Алгоритм работает следующим образом:

1. Инициализировать список Confirmed записью для себя; эта запись имеет стоимость 0.
2. Для узла, только что добавленного в список Confirmed на предыдущем шаге, назвать его узел Next и выбрать его LSP.
3. Для каждого соседа (Neighbor) Next вычислить стоимость (Cost) для достижения этого Neighbor как сумму стоимости от себя до Next и от Next до Neighbor.
 1. Если Neighbor в данный момент не находится ни в списке Confirmed, ни в списке Tentative, то добавить (Neighbor, Cost, NextHop) в список Tentative, где NextHop — это направление, в котором нужно идти, чтобы достичь Next.
 2. Если Neighbor в данный момент находится в списке Tentative, и Cost меньше текущей стоимости для Neighbor, то заменить текущую запись на (Neighbor, Cost, NextHop), где NextHop — это направление, в котором нужно идти, чтобы достичь Next.
 3. Если список Tentative пуст, остановиться. В противном случае выбрать запись из списка Tentative с наименьшей стоимостью, переместить ее в список Confirmed и вернуться к шагу 2.

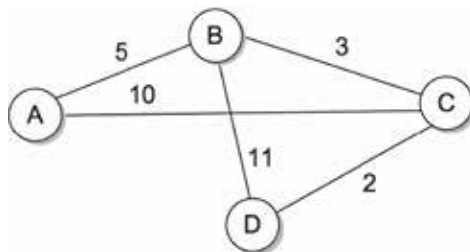


Рисунок 3.34. Маршрутизация с состоянием связей: пример сети.

Это станет гораздо проще понять, когда мы рассмотрим пример. Рассмотрим сеть, изображенную на рис. 3.34. Обратите внимание, что, в отличие от нашего предыдущего примера, эта сеть имеет диапазон различных стоимостей ребер. Табл. 3.16 прослежива-

ет шаги по построению таблицы маршрутизации для узла D. Мы обозначаем два выхода из D, используя имена узлов, к которым они подключены, В и С. Обратите внимание, как алгоритм, казалось бы, идет по ложным путям (например, путь к узлу В с затратами 11 единиц, который был первым добавлением в список Tentative), но в конечном итоге находит пути с наименьшей стоимостью ко всем узлам.

Таблица 3.16.
Шаги построения таблицы маршрутизации для узла D.

Шаг	Confirmed	Tentative	Комментарии
1	(D,0,-)		Поскольку D — единственный новый член подтвержденного списка, посмотрите на его LSP.
2	(D,0,-)	(B,11,B) (C,2,C)	LSP D говорит, что мы можем добраться до В через В по цене 11, что лучше, чем что-либо другое в обоих списках, поэтому включите его в список Tentative; то же самое для С.
3	(D,0,-) (C,2,C)	(B,11,B)	Поместите самого дешевого участника из Tentative(C) в список Confirmed. Затем изучите LSP нового подтвержденного участника (C).
4	(D,0,-) (C,2,C)	(B,5,C) (A,12,C)	Стоимость достижения В через С равна 5, поэтому заменим (B,11,B). LSP С сообщает нам, что мы можем достичь А с затратами 12.
5	(D,0,-) (C,2,C) (B,5,C)	(A,12,C)	Переместите член Tentative (B) с наименьшими затратами в Confirmed, затем посмотрите на его LSP.
6	(D,0,-) (C,2,C) (B,5,C)	(A,10,C)	Поскольку мы можем достичь А с затратами 5 через В, замените запись Tentative.
7	(D,0,-) (C,2,C) (B,5,C) (A,10,C)		Переместите член Tentative (A) с наименьшими затратами в Confirmed, и все готово.

Алгоритм маршрутизации состояния канала имеет много хороших свойств: было доказано, что он быстро стабилизируется, не генерирует много трафика и быстро реагирует на изменения топологии или отказ узла. С другой стороны, объем информации, хранящейся на каждом узле (один LSP для каждого другого узла в сети), может быть довольно большим. Это одна из основных проблем маршрутизации и пример более общей проблемы масштабируемости. Некоторые решения как для конкретной проблемы (потенциально требуемый объем памяти на каждом узле), так и для общей проблемы (масштабируемость) будут обсуждены далее.

Основные выводы

Алгоритмы маршрутизации с использованием векторного расстояния и состояния канала являются распределенными, но используют разные стратегии. В алгоритме векторного расстояния каждый узел общается только с непосредственно подключенными соседями, но сообщает им все, что узнал (т. е. расстояние до всех узлов). В алгоритме состояния канала каждый узел общается со всеми другими узлами, но сообщает им только то, что он точно знает (т. е. только состояние своих непосредственно подключенных каналов). В отличие от этих двух алгоритмов, мы рассмотрим более централизованный подход к маршрутизации позже, когда будем обсуждать программно-определяемые сети (SDN).

Протокол Open Shortest Path First (OSPF)

Одним из самых широко используемых протоколов маршрутизации на основе состояния связей является OSPF. Первое слово, «Open», указывает на то, что это открытый, непатентованный стандарт, созданный под эгидой рабочей группы Internet Engineering Task Force (IETF). Часть «SPF» происходит от альтернативного названия для маршрутизации на основе состояния связей. OSPF добавляет довольно много функций к базовому алгоритму состояния связей, описанному выше, включая следующие:

- **Аутентификация сообщений маршрутизации** — одной из особенностей распределенных алгоритмов маршрутизации является то, что они распространяют информацию от одного узла ко многим другим узлам, и вся сеть может быть затронута неверной информацией от одного узла. По этой причине полезно быть уверенным, что всем узлам, участвующим в протоколе, можно доверять. Аутентификация сообщений маршрутизации помогает достичь этого. Ранние версии OSPF использовали простой 8-байтовый пароль для аутентификации. Это недостаточно сильная форма аутентификации, чтобы предотвратить атаки со стороны целенаправленных злоумышленников, но она смягчает некоторые проблемы, вызванные неправильной настройкой или случайными атаками. (Подобная форма аутентификации была добавлена в RIP во второй версии.) Позднее была добавлена криптографическая аутентификация.
- **Дополнительная иерархия** — иерархия является одним из фундаментальных инструментов, используемых для повышения масштабируемости систем. OSPF вводит еще один уровень иерархии в маршрутизацию, позволяя разделить домен на *области*. Это означает, что маршрутизатор в пределах домена не обязательно должен знать, как достичь каждой сети в этом домене — достаточно знать, как попасть в нужную область. Таким образом, уменьшается объем информации, которую нужно передавать и хранить в каждом узле.
- **Балансировка нагрузки** — OSPF позволяет назначать одинаковую стоимость нескольким маршрутам до одного и того же места и будет распределять трафик равномерно по этим маршрутам, что позволяет лучше использовать доступную емкость сети.

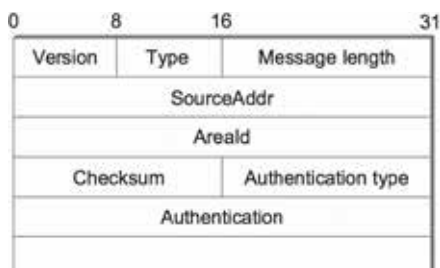


Рисунок 3.35. Формат заголовка OSPF.

Существует несколько различных типов сообщений OSPF, но все они начинаются с одного и того же заголовка, как показано на рис. 3.35. Поле Version в настоящее время установлено на 2, и поле Type может принимать значения от 1 до 5. SourceAddr идентифицирует отправителя сообщения, а AreaId является 32-битным идентификатором области, в которой находится узел. Весь пакет, за исключением данных аутентификации, защищен 16-битной контрольной суммой с использованием того же алгоритма, что и заголовок IP. Тип аутентификации равен 0, если аутентификация не используется; в противном случае он может быть равен 1, что подразумевает использование простого пароля, или 2, что указывает на использование криптографической контрольной суммы аутентификации. В последних случаях поле Authentication содержит пароль или криптографическую контрольную сумму.

Из пяти типов сообщений OSPF тип 1 — это сообщение «hello», которое маршрутизатор отправляет своим коллегам (пирам), чтобы уведомить их о том, что он все еще жив и подключен, как описано выше. Оставшиеся типы используются для запроса, отправки и подтверждения получения сообщений состояния связей. Основным строительным блоком сообщений состояния связей в OSPF является объявление состояния связи (LSA). Одно сообщение может содержать много LSA. Мы предоставим несколько деталей о LSA здесь.

Как и любой протокол маршрутизации интерсети, OSPF должен предоставлять информацию о том, как достичь сетей. Таким образом, OSPF должен предоставлять немного больше информации, чем простой протокол на основе графов, описанный выше. В частности, маршрутизатор, работающий под управлением OSPF, может генерировать пакеты состояния связи, которые рекламируют одну или несколько сетей, непосредственно подключенных к этому маршрутизатору. Кроме того, маршрутизатор, который подключен к другому маршрутизатору по какому-то каналу, должен рекламировать стоимость достижения этого маршрутизатора по данному каналу. Эти два типа объявлений необходимы, чтобы позволить всем маршрутизаторам в домене определить стоимость достижения всех сетей в этом домене и соответствующий следующий переход для каждой сети.

LS Age		Options		Type = 1
Link-state ID				
Advertising router				
LS sequence number				
LS checksum		Length		
0	Flags	0	Number of links	
Link ID				
Link data				
Link type	Num_TOS		Metric	
Optional TOS information				
More links				

Рисунок 3.36. Объявление состояния связей OSPF.

Рис. 3.36 показывает формат пакета для объявления состояния связи типа 1. LSA типа 1 рекламируют стоимость каналов между маршрутизаторами. LSA типа 2 используются для рекламы сетей, к которым подключен рекламирующий маршрутизатор, в то время как другие типы используются для поддержки дополнительной иерархии, как описано в следующей главе. Многие поля в LSA должны быть знакомы из предыдущего обсуждения. Поле LS Age эквивалентно времени жизни, за исключением того, что оно отсчитывается вверх и LSA истекает, когда возраст достигает определенного максимального значения. Поле Type сообщает нам, что это LSA типа 1.

В LSA типа 1 поле идентификатора состояния канала (Link state ID) и поле рекламирующего маршрутизатора (Advertising router) идентичны. Каждое из них содержит 32-битный идентификатор маршрутизатора, создавшего этот LSA. Хотя для назначения этого идентификатора может использоваться несколько стратегий, важно, чтобы он был уникальным в домене маршрутизации, и чтобы данный маршрутизатор всегда использовал один и тот же идентификатор маршрутизатора. Один из способов выбрать идентификатор маршрутизатора, который соответствует этим требованиям, — выбрать наименьший IP-адрес среди всех IP-адресов, назначенных этому маршрутизатору. (Напомним, что у маршрутизатора может быть разный IP-адрес на каждом из его интерфейсов.)

Порядковый номер LS (LS sequence number) используется точно так же, как описано выше, для обнаружения старых или дублирующих LSA. Контрольная сумма LS (LS checksum) аналогична тем, которые мы видели в других протоколах; она, конечно же, используется для проверки того, что данные не были повреждены. Она охватывает все поля в пакете, за исключением возраста LSA (LS age), так что нет необходимости пересчитывать контрольную сумму каждый раз при увеличении возраста LSA. Длина (Length) — это длина в байтах полного LSA.

Теперь мы переходим к фактической информации о состоянии соединения. Это немного усложняется наличием информации TOS (type of service). Не обращая внимания на это, каждое соединение в LSA представлено идентификатором соединения, некоторыми данными соединения и метрикой. Первые два из этих полей идентифицируют соединение.

Для этого можно использовать идентификатор маршрутизатора на дальнем конце соединения в качестве идентификатора соединения, а затем использовать данные соединения для определения нескольких параллельных соединений, если это необходимо. Метрика — это, конечно, стоимость соединения. Тип говорит нам что-то о канале — например, является ли он каналом «точка-точка».

Информация о TOS существует, чтобы позволить OSPF выбирать разные маршруты для IP-пакетов в зависимости от значения в их поле TOS. Вместо назначения одной метрики связи возможно назначить разные метрики в зависимости от значения TOS данных. Например, если у нас есть связь в сети, которая очень хороша для трафика, чувствительного к задержкам, мы могли бы назначить ей низкую метрику для значения TOS, представляющего низкую задержку, и высокую метрику для всего остального. OSPF тогда выберет другой кратчайший путь для тех пакетов, у которых поле TOS установлено на это значение. Стоит отметить, что на момент написания этой функции она не была широко внедрена.

Глава 3.4.4. Метрики

Предыдущее обсуждение предполагает, что стоимости связей, или метрики, известны при выполнении алгоритма маршрутизации. В этом разделе мы рассмотрим некоторые способы вычисления стоимости связей, которые оказались эффективными на практике. Один пример, который мы уже видели, вполне разумный и очень простой, заключается в назначении стоимости 1 всем связям — наименьшая стоимость маршрута тогда будет маршрутом с наименьшим количеством переходов. Однако такой подход имеет несколько недостатков. Во-первых, он не различает связи на основе задержки. Таким образом, спутниковая связь с задержкой 250 мс выглядит для протокола маршрутизации так же привлекательно, как и наземная связь с задержкой 1 мс. Во-вторых, он не различает маршруты на основе пропускной способности, делая 1-Мбит/с связь такой же хорошей, как и 10-Гбит/с связь. Наконец, он не различает связи на основе их текущей загрузки, что делает невозможным обход перегруженных связей. Оказывается, эта последняя проблема самая сложная, потому что вы пытаетесь отразить сложные и динамические характеристики связи в единой скалярной стоимости.

ARPANET был испытательной площадкой для ряда различных подходов к вычислению стоимости связи. (Это также было место, где была продемонстрирована превосходная стабильность маршрутизации на основе состояния связи по сравнению с маршрутизацией на основе векторного расстояния; оригинальный механизм использовал вектор расстояния, тогда как более поздняя версия использовала состояние связи.) Следующее обсуждение прослеживает эволюцию метрики маршрутизации ARPANET и, делая это, исследует тонкие аспекты проблемы.

Первоначальная метрика маршрутизации ARPANET измеряла количество пакетов, стоящих в очереди на передачу по каждой связи, а это означало, что связи с 10 пакетами в очереди на передачу назначалась большая стоимость, чем связи с 5 пакетами в очереди на передачу. Однако использование длины очереди в качестве метрики маршрутизации не работало хорошо, так как длина очереди является искусственной мерой нагрузки — она перемещает пакеты к самой короткой очереди, а не к пункту назначения, что слиш-

ком хорошо знакомо тем, кто перескакивает из одной очереди в супермаркете в другую. Если говорить более точно, оригинальный механизм маршрутизации ARPANET страдал от того, что он не учитывал ни пропускную способность, ни задержку связи.

Вторая версия алгоритма маршрутизации ARPANET учитывала как пропускную способность, так и задержку связи и использовала задержку, а не просто длину очереди, как меру нагрузки. Это делалось следующим образом. Во-первых, каждый входящий пакет помечался временем его прибытия на маршрутизатор (ArrivalTime); время его отправления с маршрутизатора (DepartTime) также фиксировалось. Во-вторых, когда квитанция уровня канала (АСК) была получена с другой стороны, узел вычислял задержку для этого пакета как:

$$\text{Delay} = (\text{DepartTime} - \text{ArrivalTime}) + \text{TransmissionTime} + \text{Latency}$$

...где TransmissionTime (время передачи) и Latency (задержка) были статически определены для связи и отражали пропускную способность и задержку связи соответственно. Обратите внимание, что в этом случае разница DepartTime - ArrivalTime представляет собой время, на которое пакет был задержан (стоял в очереди) в узле из-за нагрузки. Если АСК не был получен, и пакет, наоборот, истек по тайм-ауту, то время DepartTime сбрасывалось на время, когда пакет был передан повторно. В этом случае разница DepartTime - ArrivalTime отражает надежность связи — чем чаще происходит повторная передача пакетов, тем менее надежна связь и тем больше мы хотим ее избежать. Наконец, вес, присваиваемый каждой связи, определялся на основе средней задержки, испытываемой пакетами, недавно переданными по этой связи.

Хотя это улучшало работу сети по сравнению с оригинальным механизмом, такой подход также имел много проблем. При низкой нагрузке он работал довольно хорошо, так как два статических фактора задержки доминировали над стоимостью. Однако при высокой нагрузке перегруженная связь начинала рекламировать очень высокую стоимость. Это приводило к тому, что весь трафик уходил с этой связи, оставляя ее простаивающей, и тогда она начинала рекламировать низкую стоимость, привлекая обратно весь трафик, и так далее. Эффектом этой нестабильности было то, что при высокой нагрузке многие связи фактически проводили много времени в простое, а это последнее, чего хочется при высокой нагрузке.

Еще одной проблемой было то, что диапазон значений связей был слишком большим. Например, сильно загруженная связь на 9,6 кбит/с могла выглядеть в 127 раз более дорогой, чем слабо загруженная связь на 56 кбит/с. (Помните, мы говорим об ARPANET образца 1975 года.) Это означает, что алгоритм маршрутизации выбрал бы маршрут с 126 переходами по слабо загруженным связям на 56 кбит/с вместо 1-переходного маршрута на 9,6 кбит/с. Хотя отвлечение части трафика с перегруженной линии — это хорошая идея, делать ее настолько непривлекательной, чтобы она теряла весь свой трафик, — это слишком. Использование 126 переходов, когда можно обойтись одним, в общем, является плохим использованием ресурсов сети. Также спутниковые связи были неоправданно урезаны, так что простая 56-кбит/с спутниковая связь выглядела значительно более дорогой, чем простая 9,6-кбит/с наземная связь, даже если первая обеспечивала бы лучшую производительность для приложений с высокой пропускной способностью.

Третий подход решал эти проблемы. Основные изменения заключались в значительном сжатии динамического диапазона метрики, учете типа связи и сглаживании изменений метрики со временем.

Сглаживание достигалось несколькими механизмами. Во-первых, измерение задержки преобразовывалось в использование связи, и это число усреднялось с последним отчетом об использовании для подавления резких изменений. Во-вторых, существовал жесткий лимит на то, насколько метрика могла измениться от одного цикла измерения к следующему. Сглаживая изменения стоимости, такой метод значительно снижал вероятность того, что все узлы сразу откажутся от маршрута.

Сжатие динамического диапазона было достигнуто путем подачи измеренной нагрузки, типа связи и скорости связи, в функцию, которая графически показана на рис. 3.37 ниже. Обратите внимание на следующее:

- Сильно загруженная связь никогда не показывает стоимость, более чем в три раза превышающую стоимость в состоянии простоя.
- Самая дорогая связь стоит всего в семь раз больше, чем самая дешевая.
- Высокоскоростная спутниковая связь более привлекательна, чем низкоскоростная наземная связь.
- Стоимость является функцией загрузки связи только при средней и высокой нагрузке.

Все эти факторы означают, что связь гораздо менее вероятно будет повсеместно отвергнута, так как троекратное увеличение стоимости, вероятно, сделает связь непривлекательной для некоторых маршрутов, позволяя ей оставаться лучшим выбором для других. Наклоны, смещения и точки перегиба для кривых на рис. 3.37 были получены в результате большого количества проб и ошибок и были тщательно настроены для обеспечения хорошей производительности.

Несмотря на все эти улучшения, оказалось, что в большинстве реальных сетевых развертываний метрики изменяются редко, если вообще изменяются, и только под контролем сетевого администратора, а не автоматически, как описано выше. Причина этого отчасти заключается в том, что общепринятое мнение теперь гласит, что динамически изменяющиеся метрики слишком нестабильны, хотя это, вероятно, не всегда правда. Возможно, более значимо то, что во многих современных сетях нет большой разницы в скоростях и задержках соединений, которая была в ARPANET. Таким образом, статические метрики являются нормой. Один из распространенных подходов к установке метрик — использование константы, умноженной на (1/пропускная способность связи).

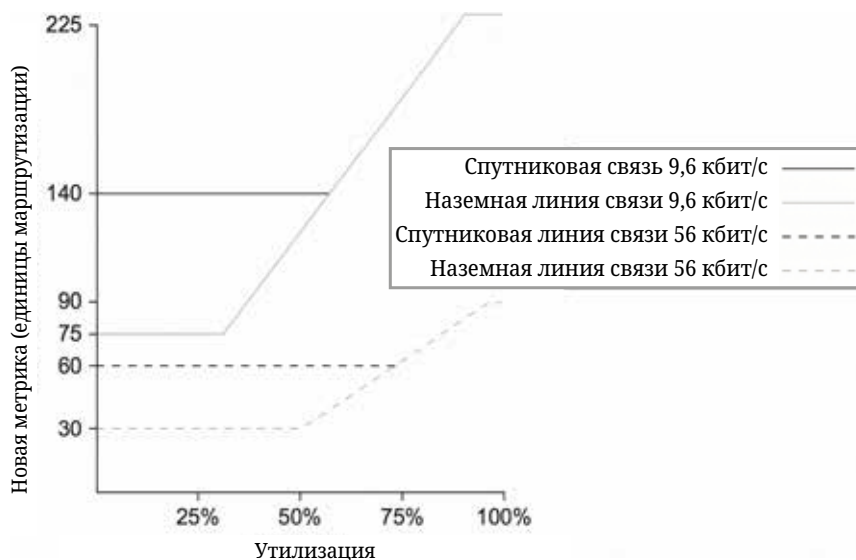


Рисунок 3.37. Пересмотренная метрика маршрутизации ARPANET в зависимости от загрузки канала.

Основные выводы

Почему мы до сих пор рассказывали историю об алгоритме, который больше не используется? Потому что он прекрасно иллюстрирует два ценных урока. Первый заключается в том, что компьютерные системы часто разрабатываются *итеративно на основе опыта*. Мы редко делаем все правильно с первого раза, поэтому важно внедрить простое решение как можно скорее, ожидая улучшений со временем.

Застревание в фазе проектирования на неопределенный срок обычно не является хорошим планом. Второй урок — это широко известный принцип KISS: «*Keep it Simple, Stupid*» («Делай проще, глупец»). При создании сложной системы простое зачастую означает лучшее. Возможностей для создания сложных оптимизаций предостаточно, и это привлекательная перспектива. Хотя такие оптимизации иногда имеют краткосрочную ценность, поразительно, как часто простой подход оказывается лучшим со временем. Это происходит потому, что когда система имеет много движущихся частей, как это, несомненно, происходит в Интернете, сохранение каждой части как можно более простой обычно является наилучшим подходом.

Глава 3.5. Реализация

До сих пор мы говорили о том, что должны делать коммутаторы и маршрутизаторы, не описывая, как они это делают. Существует простой способ построить коммутатор или маршрутизатор: купить универсальный процессор и оснастить его несколькими сетевыми интерфейсами. Такое устройство, работающее с подходящим программным обеспечением, может принимать пакеты на одном из своих интерфейсов, выполнять любые функции коммутации или пересылки, описанные в этой главе, и отправлять пакеты через другой интерфейс. Такой так называемый *программный коммутатор* не слишком отличается от архитектуры многих коммерческих сетевых устройств среднего и низкого класса.¹ Реализации, обеспечивающие высокую производительность, обычно используют дополнительное аппаратное ускорение. Мы называем их *аппаратными коммутаторами*, хотя оба подхода очевидно включают сочетание аппаратного и программного обеспечения.

Этот раздел дает обзор как программно-ориентированных, так и аппаратно-ориентированных дизайнов, но стоит отметить, что в вопросе коммутаторов и маршрутизаторов различие не так уж важно. Оказывается, что реализация коммутаторов и маршрутизаторов имеет столько общего, что сетевой администратор обычно покупает один блок пересылки и затем настраивает его как L2 коммутатор, L3 маршрутизатор или их комбинацию. Поскольку их внутренние дизайны настолько похожи, мы будем использовать слово «*коммутатор*» для обозначения обеих вариаций на протяжении всего этого раздела, избегая утомительного повторения «коммутатор или маршрутизатор». Мы будем указывать различия между ними, когда это необходимо.

Глава 3.5.1. Программный коммутатор

На рис. 3.38 показан программный коммутатор, построенный с использованием универсального процессора с четырьмя сетевыми интерфейсными картами (NIC). Путь для типичного пакета, который приходит, скажем, на NIC 1 и пересылается через NIC 2, прост: когда NIC 1 принимает пакет, он копирует его байты напрямую в основную память через шину ввода-вывода (в данном примере PCIe) с использованием техники, называемой *прямым доступом к памяти* (direct memory access, DMA). После того как пакет находится в памяти, процессор анализирует его заголовок, чтобы определить, через какой интерфейс пакет должен быть отправлен, и инструктирует NIC 2 передать пакет, снова напрямую из основной памяти, используя DMA. Важный вывод состоит в том, что пакет буферизуется в основной памяти (это «хранение» в концепции хранения и пересылки (store-and-forward)), при этом процессор читает только необходимые поля заголовка во внутренние регистры для обработки.

¹ Это также то, как первые интернет-маршрутизаторы, часто называемые шлюзами в то время, были реализованы в ранние дни Интернета.

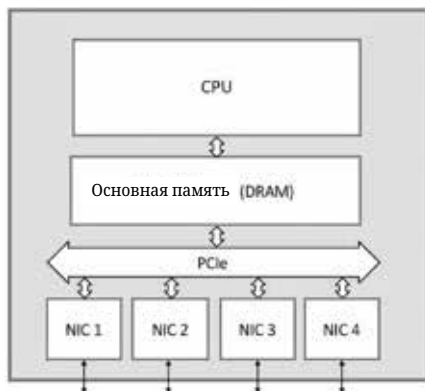


Рисунок 3.38. Процессор общего назначения, используемый в качестве программного коммутатора.

Существует два потенциальных «узких» места при таком подходе, одно из которых или оба могут ограничить совокупную пропускную способность программного коммутатора.

Первая проблема заключается в том, что производительность ограничена тем, что все пакеты должны проходить в основную память и выходить из нее. Производительность будет зависеть от того, сколько вы готовы потратить на оборудование, но, например, машина, ограниченная шиной памяти с частотой 1333 МГц и шириной 64 бита, может передавать данные с пиковой скоростью чуть более 100 Гбит/с — достаточно, чтобы построить коммутатор с несколькими портами Ethernet на 10 Гбит/с, но явно недостаточно для высокопроизводительного маршрутизатора в ядре Интернета.

Более того, этот верхний предел предполагает, что перемещение данных — единственная проблема. Это справедливо для длинных пакетов, но неверно для коротких пакетов, что является худшим сценарием, который должны учитывать разработчики коммутаторов. При минимальном размере пакетов стоимость обработки каждого пакета (анализ заголовка и принятие решения о том, через какой выходной канал передавать его) может доминировать и потенциально стать узким местом. Предположим, например, что процессор может выполнить всю необходимую обработку для пересылки 40 миллионов пакетов каждую секунду. Это иногда называется скоростью обработки пакетов (pps). Если средний пакет имеет размер 64 байта, это будет означать

$$\begin{aligned} \text{Throughput (пропускная способность)} &= \text{pps} \times \text{BitsPerPacket (биты на пакет)} \\ &= 40 \times 10^6 \times 64 \times 8 \\ &= 2048 \times 10^7 \end{aligned}$$

...то есть пропускную способность около 20 Гбит/с — это быстро, но существенно ниже диапазона, который пользователи требуют от своих коммутаторов сегодня. Имейте в виду, что эти 20 Гбит/с будут разделены между всеми пользователями, подключенными к коммутатору, так же как пропускная способность одного (некоммутируемого) сегмента Ethernet разделяется между всеми пользователями, подключенными к общей среде. Таким образом, например, 16-портовый коммутатор с такой совокупной пропускной способностью сможет справляться со средней скоростью передачи данных около 1 Гбит/с на каждый порт.¹

¹ Эти примеры производительности не представляют абсолютную максимальную скорость пропускной способности, которую может достичь высоко оптимизированное программное обеспечение, работающее на высокопроизводительном сервере, но они указывают на пределы, с которыми в конечном итоге сталкиваются при использовании этого подхода.

Последняя важная часть, которую необходимо понять при оценке реализации коммутаторов. Нетривиальные алгоритмы, обсуждаемые в этой главе — алгоритм построения остовного дерева, используемый обучающими мостами, алгоритм дистанционно-векторной маршрутизации, используемый RIP, и алгоритм состояния канала, используемый OSPF, — *не являются* частью принятия решений о пересылке пакетов. Они работают периодически в фоновом режиме, но коммутаторы не должны выполнять, скажем, код OSPF для каждого пересылаемого пакета. Самая затратная процедура, которую процессор, вероятно, выполнит для каждого пакета, — это поиск в таблице, например, поиск номера VCI в таблице VC, IP-адреса в таблице маршрутизации L3 или Ethernet-адреса в таблице маршрутизации L2.

Основные выводы

Различие между этими двумя видами обработки настолько важно, что ему дали название: *плоскость управления*. Оно соответствует фоновому процессу, необходимому для «управления» сетью (например, выполнение OSPF, RIP или протокола BGP, описанного в следующем разделе), а *плоскость данных* соответствует обработке каждого пакета, необходимой для перемещения пакетов от входного порта к выходному порту. По историческим причинам это различие называется *плоскостью управления* и *пользовательской плоскостью* в сетях сотового доступа, но идея та же, и на самом деле стандарт 3GPP определяет CUPS (разделение плоскостей управления и пользователя (Control/User Plane Separation)) как архитектурный принцип. Эти два вида обработки легко спутать, когда оба работают на одном и том же процессоре, как это происходит в программном коммутаторе, изображенном на рис. 3.38, но производительность может быть значительно улучшена за счет оптимизации того, как реализована плоскость данных, и, соответственно, определения четко определенного интерфейса между плоскостью управления и плоскостью данных.

Глава 3.5.2. Аппаратный коммутатор

На протяжении большей части истории Интернета высокопроизводительные коммутаторы и маршрутизаторы были специализированными устройствами, построенными на базе интегральных схем специального назначения (Application-Specific Integrated Circuits, ASIC). Хотя было возможно создавать низкоуровневые маршрутизаторы и коммутаторы, используя стандартные серверы, работающие на С-программах, для достижения требуемых скоростей передачи данных были необходимы ASIC.

Проблема с ASIC заключается в том, что разработка и производство оборудования занимают много времени, а это означает, что задержка в добавлении новых функций в коммутатор обычно измеряется годами, а не днями или неделями, как привыкла сегодняшняя индустрия софта. В идеале мы хотели бы получить преимущества производительности ASIC и гибкости программного обеспечения.

К счастью, недавние достижения в области специализированных процессоров (и других стандартных компонентов) сделали это возможным. Не менее важно, что полные архитектурные спецификации для коммутаторов, использующих эти новые процессоры, теперь доступны онлайн — аппаратный эквивалент *программного обеспечения с открытым исходным кодом*. Это означает, что любой может создать высокопроизводительный коммутатор, загрузив чертеж из Интернета (см. примеры в проекте Open Compute Project, OCP), так же, как можно собрать свой собственный ПК. В обоих случаях вам все еще потребуется программное обеспечение для работы на этом оборудовании, но так же, как Linux доступен для работы на вашем собранном ПК, теперь есть открытые стеки L2 и L3, доступные на GitHub для работы на вашем самодельном коммутаторе. В качестве альтернативы вы можете просто купить готовый коммутатор у производителя стандартных коммутаторов и загрузить на него свое собственное программное обеспечение. Да-

лее описаны эти открытые коммутаторы типа *white-box* (белого ящика), называемые так в противовес закрытым устройствам типа «черный ящик», которые исторически доминировали в отрасли.

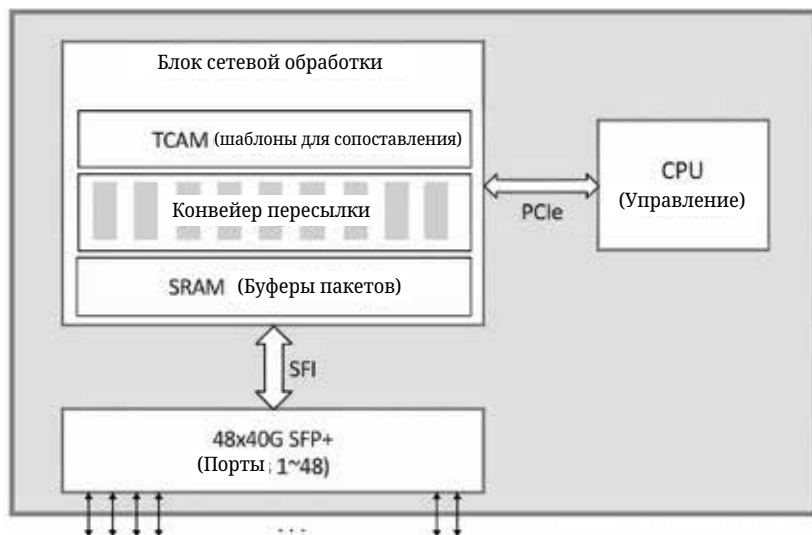


Рисунок 3.39. Коммутатор «белого ящика» с использованием блока сетевой обработки.

Рис. 3.39 — это упрощенное изображение коммутатора типа *white-box*. Ключевое отличие от более ранней реализации на универсальном процессоре заключается в добавлении сетевого процессора (Network Processor Unit, NPU), специализированного процессора с архитектурой и набором инструкций, оптимизированных для обработки заголовков пакетов (т.е. для реализации плоскости данных). NPU похожи по своей сути на GPU, архитектура которых оптимизирована для рендеринга компьютерной графики, но в данном случае NPU оптимизированы для разбора заголовков пакетов и принятия решений о пересылке. NPU способны обрабатывать пакеты (ввод, принятие решения о пересылке и вывод) со скоростью, измеряемой в терабитах в секунду (Тбит/с или Tbps), достаточно быстро, чтобы справляться с 32x100-Гбит/с портами или 48x40-Гбит/с портами, показанными на диаграмме.

Сетевые процессоры

- Наше использование термина NPU несколько нестандартно. Исторически NPU было названием более узко определенных сетевых процессорных чипов, используемых, например, для реализации интеллектуальных брандмауэров или глубокой инспекции пакетов. Они не были такими универсальными, как NPU, о которых мы говорим здесь;
- они также не были такими высокопроизводительными. Вероятно, текущий подход делает специально построенные сетевые процессоры устаревшими, но в любом случае мы предпочитаем терминологию NPU, потому что она соответствует тенденции создания программируемых специализированных процессоров, включая GPU для графики и TPU (тензорные процессоры, Tensor Processing Units) для ИИ.

Преимущество этой новой конструкции коммутатора заключается в том, что данный *white-box* теперь можно запрограммировать как коммутатор L2, маршрутизатор L3 или комбинацию обоих — все это вопрос программирования. Тот же самый стек программного

обеспечения плоскости управления, используемый в программном коммутаторе, все еще работает на управляющем процессоре, но, кроме того, программы плоскости данных загружаются на NPU, чтобы отразить решения о пересылке, принимаемые программным обеспечением плоскости управления. То, как именно «программируется» NPU, зависит от производителя чипа, которых в настоящее время несколько. В некоторых случаях конвейер пересылки фиксирован, и управляющий процессор просто загружает таблицу пересылки в NPU (под фиксированным мы подразумеваем, что NPU знает, как обрабатывать только определенные заголовки, такие как Ethernet и IP), но в других случаях конвейер пересылки сам по себе программируем. P4 — новый язык программирования, который можно использовать для программирования таких конвейеров пересылки на основе NPU. Среди прочего P4 пытается скрыть многие различия в наборах инструкций базового NPU.

Внутренне NPU использует три технологии. Во-первых, быстрая память на основе SRAM буферизует пакеты, пока они обрабатываются. SRAM (статическая оперативная память, Static Random Access Memory) примерно на порядок быстрее, чем DRAM (динамическая оперативная память, Dynamic Random Access Memory), которая используется для основной памяти. Во-вторых, память на основе TCAM хранит битовые шаблоны, которые сопоставляются с пакетами, находящимися в обработке. «CAM» в TCAM означает «память с адресацией по содержимому», что означает, что ключ, который вы хотите найти в таблице, может эффективно использоваться в качестве адреса в памяти, реализующей эту таблицу. «Т» означает «троичная», что является сложным способом сказать, что ключ, который вы хотите найти, может содержать подстановочные знаки (например, ключ 10*1 соответствует как 1001, так и 1011). Наконец, обработка, необходимая для пересылки каждого пакета, реализуется посредством конвейера пересылки. Этот конвейер реализован на базе ASIC, но при хорошей конструкции поведение пересылки конвейера можно изменить, изменив программу, которую он выполняет. На высоком уровне эта программа выражается в виде набора пар (*сопоставление* (Match), *действие* (Action)): если вы сопоставляете какое-то поле в заголовке, то выполняете такое-то действие.

Значимость того, что обработка пакетов реализована с помощью многоступенчатого конвейера, а не одноступенчатого процессора, заключается в том, что пересылка одного пакета, вероятно, включает в себя просмотр нескольких полей заголовка. Каждая ступень может быть запрограммирована на просмотр различной комбинации полей. Многоступенчатый конвейер добавляет немного сквозной задержки к каждому пакету (измеряемой в наносекундах), но также означает, что несколько пакетов могут обрабатываться одновременно. Например, Ступень 2 может выполнять второй поиск по пакету A, в то время как Ступень 1 выполняет начальный поиск по пакету B, и так далее. Это означает, что NPU в целом способен справляться с линейными скоростями. На момент написания этого текста передовое достижение скорости составляет 12,8 Тбит/с.

Наконец, рис. 3.39 включает другие стандартные компоненты, которые делают все это практичным. В частности, теперь можно приобрести модули подключаемых *трансиверов*, которые берут на себя все детали доступа к среде — будь то Gigabit Ethernet, 10-Gigabit Ethernet или SONET, а также оптика. Эти трансиверы соответствуют стандартным форм-факторам, таким как SFP+, которые, в свою очередь, могут быть подключены к другим компонентам через стандартную шину (например, SFI). Опять же ключевой вывод заключается в том, что сеть только сейчас входит в тот же мир стандартизации, которым вычислительная индустрия наслаждается последние два десятилетия.

Глава 3.5.3. Программно-определяемые сети

С коммутаторами, которые становятся все более стандартизированными, внимание справедливо смещается к программному обеспечению, которое их контролирует. Это помещает нас прямо в центр тенденции создания *программно-определяемых сетей* (Software Defined Networks, SDN), идея которых начала зарождаться около десяти лет назад. На самом деле это были ранние этапы SDN, которые подтолкнули сетевую индустрию к переходу на коммутаторы типа white-box.

Основная идея SDN — это та, о которой мы уже говорили: отделить плоскость управления сетью (т.е. где работают алгоритмы маршрутизации, такие как RIP, OSPF и BGP) от плоскости данных сети (т.е. где принимаются решения о пересылке пакетов), при этом первая перемещается в программное обеспечение, работающее на стандартных серверах, а вторая реализуется коммутаторами типа *white-box*. Ключевая идея, позволившая реализовать SDN, заключалась в том, чтобы сделать этот процесс разделения на шаг дальше и определить стандартный интерфейс между плоскостью управления и плоскостью данных. Это позволяет любой реализации плоскости управления взаимодействовать с любой реализацией плоскости данных; это разрушает зависимость от решений, связанных с каким-либо одним производителем. Оригинальный интерфейс называется *OpenFlow*, и эта идея разделения плоскостей управления и данных стала известна как дезагрегация. (Язык P4, упомянутый в предыдущей главе, представляет собой попытку второго поколения определить этот интерфейс, обобщив *OpenFlow*.)

Еще один важный аспект дезагрегации заключается в том, что логически централизованная плоскость управления может использоваться для управления распределенной плоскостью данных сети. Мы говорим «логически централизованная», потому что, хотя состояние, собираемое плоскостью управления, поддерживается в глобальной структуре данных, такой как сетевая карта, реализация этой структуры данных все же может быть распределена по нескольким серверам. Например, она может работать в облаке. Это важно как для масштабируемости, так и для доступности, где ключевым является то, что две плоскости настраиваются и масштабируются независимо друг от друга. Эта идея быстро прижилась в облаке, где современные облачные провайдеры используют решения на базе SDN как внутри своих центров обработки данных, так и через магистральные сети, которые соединяют их центры обработки данных.

Одним из последствий этой архитектуры, которое не сразу очевидно, является то, что логически централизованная плоскость управления управляет не только сетью физических (аппаратных) коммутаторов, которые соединяют физические серверы, но и сетью виртуальных (программных) коммутаторов, которые соединяют виртуальные серверы (например, виртуальные машины и контейнеры). Если вы считаете «порты коммутатора» (хороший показатель всех устройств, подключенных к вашей сети), то количество виртуальных портов в Интернете превысило количество физических портов в 2012 году.

Одним из ключевых факторов успеха SDN, как показано на рис. 3.40, является *сетевая операционная система* (Network Operating System, NOS). Как операционная система сервера (например, Linux, iOS, Android, Windows), которая предоставляет набор высокоуровневых абстракций, упрощающих реализацию приложений (например, вы можете читать и записывать файлы вместо прямого доступа к дисковым накопителям), NOS упрощает реализацию функций управления сетью, известных как *приложения управления*. Хорошая NOS абстрагирует детали сетевых коммутаторов и предоставляет разработчику приложений абстракцию *сетевой карты* (Network map).

NOS обнаруживает изменения в подлежащей сети (например, коммутаторы, порты и соединения, которые активируются и деактивируются), и приложение управления просто реализует желаемое поведение на этом абстрактном графе. Это означает, что NOS берет на себя задачу сбора сетевого состояния (сложная часть распределенных алгоритмов, таких как алгоритмы состояния соединений (Link-State) и алгоритмы дистанционно-векторной маршрутизации (Distance-Vector)), а приложение освобождается для простой реализации алгоритма кратчайшего пути и загрузки правил пересылки в подлежащие коммутаторы. Централизуя эту логику, отметим, что цель состоит в том, чтобы получить глобально оптимизированное решение. Опубликованные данные от облачных провайдеров, которые приняли этот подход, подтверждают это преимущество.

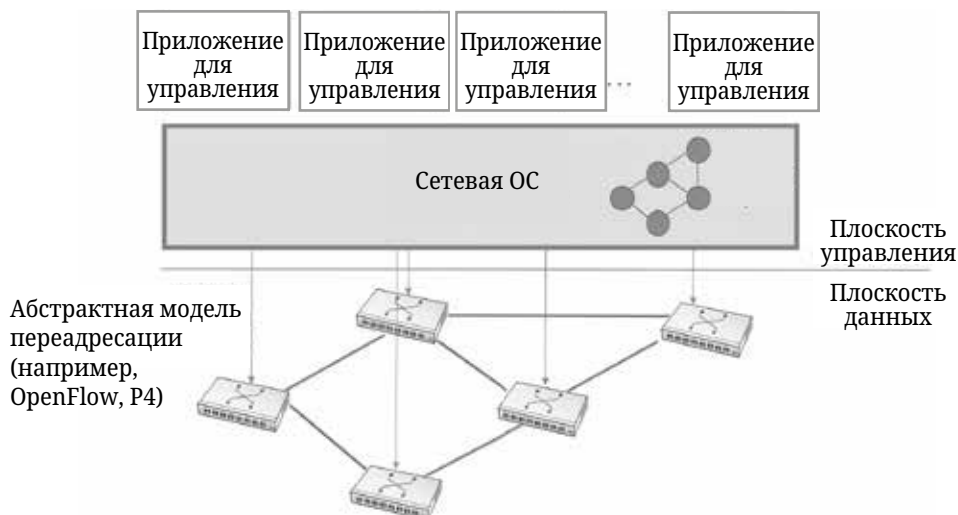


Рисунок 3.40. Сетевая операционная система (NOS), содержащая набор управляющих приложений и обеспечивающая логически централизованную точку управления плоскостью данных базовой сети.

Основные выводы

Важно понимать, что SDN — это стратегия реализации. Она не решает магическим образом фундаментальные проблемы, такие как необходимость вычисления таблицы пересылки. Вместо того чтобы нагружать коммутаторы необходимостью обмениваться сообщениями друг с другом в рамках распределенного алгоритма маршрутизации, логически централизованный SDN-контроллер отвечает за сбор информации о состоянии соединений и портов от отдельных коммутаторов, создание глобального представления графа сети и предоставление этого графа приложениям управления. С точки зрения приложения управления вся необходимая информация для вычисления таблицы пересылки доступна локально. Учитывая, что SDN-контроллер логически централизован, но физически реплицирован на нескольких серверах (для обеспечения масштабируемой производительности и высокой доступности), все еще остается спорным вопросом, какой подход лучше: централизованный или распределенный.

Хотя облачные провайдеры смогли получить значительные преимущества от SDN, его внедрение в корпоративных сетях и телекоммуникационных компаниях идет гораздо медленнее. Это частично связано со способностью различных рынков управлять своими сетями. Такие компании, как Google, Microsoft и Amazon, имеют инженеров и навыки DevOps, необходимые для использования этой технологии, в то время как другие все еще предпочитают готовые и интегрированные решения, поддерживающие знакомые им интерфейсы управления и командной строки.

Перспектива: виртуальные сети во всех слоях

Практически с самого начала существования сетей с коммутацией пакетов существовали идеи о том, как их виртуализировать, начиная с виртуальных каналов. Но что же на самом деле означает виртуализировать сеть?

Пример виртуальной памяти может быть полезен. Виртуальная память создает абстракцию большого и приватного пула памяти, даже если подлежащая физическая память может быть разделена между многими приложениями и значительно меньше видимого пула виртуальной памяти. Эта абстракция позволяет программистам работать

под иллюзией, что имеется достаточно памяти и никто другой ее не использует, в то время как система управления памятью занимается такими задачами, как сопоставление виртуальной памяти с физическими ресурсами и предотвращение конфликтов между пользователями.

Аналогично виртуализация серверов представляет абстракцию виртуальной машины (VM), которая обладает всеми характеристиками физической машины. Опять же, множество виртуальных машин может быть поддержано на одном физическом сервере, и операционная система и пользователи на виртуальной машине не осознают, что VM отображается на физические ресурсы.

Ключевой момент заключается в том, что виртуализация вычислительных ресурсов сохраняет абстракции и интерфейсы, которые существовали до их виртуализации. Это важно, потому что означает, что пользователям этих абстракций не нужно ничего менять — они видят точное воспроизведение виртуализируемого ресурса. Виртуализация также означает, что различные пользователи (иногда называемые *арендаторами*) не могут вмешиваться в дела друг друга. Так что же происходит, когда мы пытаемся виртуализировать сеть?

VPN, описанные в главе 3.3, были одним из первых успешных примеров виртуальных сетей. Они позволили операторам представить корпоративным клиентам иллюзию, что у них есть собственная частная сеть, даже если на самом деле они делили подлежащие соединения и коммутаторы с другими пользователями. Однако VPN виртуализировала только несколько ресурсов, в частности адресные и таблицы маршрутизации. Виртуализация сети, как она обычно понимается сегодня, идет дальше, виртуализируя все аспекты сетевой инфраструктуры. Это означает, что виртуальная сеть должна поддерживать все базовые абстракции физической сети. В этом смысле они аналогичны виртуальной машине, поддерживающей все ресурсы сервера: ЦПУ, хранилище, ввод-вывод и так далее.

Для этой цели VLAN, описанные в главе 3.2, являются способом виртуализации сети уровня L2. VLAN оказались весьма полезными для предприятий, которые хотели изолировать различные внутренние группы (например, отделы, лаборатории), предоставляя каждой из них видимость собственной частной локальной сети. VLAN также рассматривались как перспективный способ виртуализации сетей уровня L2 в облачных центрах обработки данных, позволяя предоставлять каждому арендатору свою собственную сеть L2, чтобы изолировать их трафик от трафика всех других арендаторов. Но возникла проблема: 4096 возможных VLAN недостаточно, чтобы учесть всех арендаторов, которых может разместить облако, и, чтобы усложнить ситуацию, в облаке сеть должна соединять виртуальные машины, а не физические машины, на которых эти виртуальные машины работают.

Для решения этой проблемы был введен другой стандарт под названием *Virtual Extensible LAN* (VXLAN). В отличие от первоначального подхода, который фактически инкапсулировал виртуализированный Ethernet-кадр в другой Ethernet-кадр, VXLAN инкапсулирует виртуальный Ethernet-фрейм внутри UDP-пакета. Это означает, что виртуальная сеть на основе VXLAN (которая часто называется наложенной сетью) работает поверх сети на основе IP, которая, в свою очередь, работает на подлежащем Ethernet (или, возможно, в одной VLAN подлежащего Ethernet). VXLAN также позволяет одному арендатору облака иметь несколько VLAN, что позволяет им разделить свой внутренний трафик. Это означает, что в конечном итоге можно иметь VLAN, инкапсулированный в VXLAN-оверлей, инкапсулированный в VLAN.

Мощь виртуализации заключается в том, что, когда все сделано правильно, должно быть возможно вложить один виртуализированный ресурс в другой виртуализированный ресурс, поскольку, в конце концов, виртуальный ресурс должен вести себя так же, как физический ресурс, и мы знаем, как виртуализировать физические ресурсы. Другими словами, способность виртуализировать виртуальный ресурс является лучшим доказательством того, что вы хорошо справились с виртуализацией исходного физического

ресурса. Переосмысливая мифологию о Мире на Черепахе: это виртуальные сети на всех уровнях.

Заголовок VXLAN на самом деле прост, как показано на рис. 3.41. Он включает 24-битный *идентификатор виртуальной сети* (Virtual Network Id, VNI), а также некоторые флаги и зарезервированные биты. Он также подразумевает определенные настройки полей UDP исходного и назначения портов (см. раздел 5.1), причем порт назначения 4789 официально зарезервирован для VXLAN. Определение уникальных идентификаторов виртуальных LAN (теги VLAN) и виртуальных сетей (VXLAN VIDs) — это легкая часть. Это связано с тем, что инкапсуляция является фундаментальным краеугольным камнем виртуализации; все, что вам нужно добавить, это идентификатор, который указывает, какому из множества возможных пользователей принадлежит этот инкапсулированный пакет.

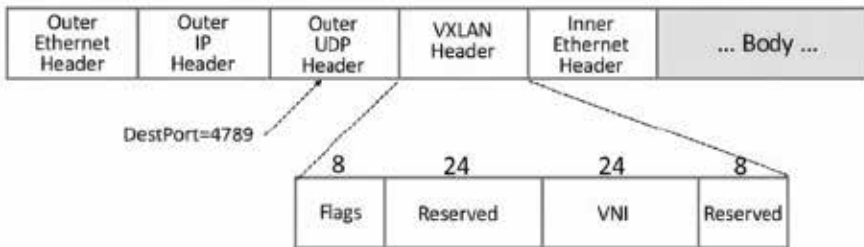


Рисунок 3.41. Заголовок VXLAN, инкапсулированный в пакет UDP/IP-заголовок.

Трудная часть — это справиться с идеей о том, что виртуальные сети вложены (инкапсулированы) внутри виртуальных сетей, что является версией рекурсии в сетях. Другая задача заключается в понимании того, как автоматизировать создание, управление, миграцию и удаление виртуальных сетей, и в этом направлении еще есть много места для улучшений. Овладение этой задачей будет в центре сетевых технологий в следующем десятилетии, и хотя часть этой работы, безусловно, будет происходить в проприетарных условиях, есть открытые платформы для виртуализации сетей (например, проект *Tungsten Fabric* от Linux Foundation), которые ведут нас в будущее.

Раздел 4.

Продвинутое множество взаимодействий

Каждое кажущееся равенство скрывает иерархию.

Мейсон Кули

Проблема: масштабирование до миллиардов

Теперь мы узнали, как построить интернет, состоящую из нескольких сетей разных типов, то есть мы рассмотрели проблему *гетерогенности*. Вторая критическая проблема в межсетевом взаимодействии (пожалуй, фундаментальная проблема всех сетей) — это *масштабирование*. Чтобы понять проблему масштабирования сети, стоит учесть рост Интернета, который примерно удваивался в размере каждый год на протяжении 30 лет. Этот рост заставляет нас сталкиваться с рядом проблем.

Главная из них — как создать систему маршрутизации, которая сможет справиться с сотнями тысяч сетей и миллиардами конечных узлов? Как мы увидим в этом разделе, большинство подходов к решению проблемы масштабируемости маршрутизации зависят от введения иерархии. Мы можем ввести иерархию в виде областей внутри домена; мы также используем иерархию для масштабирования системы маршрутизации между доменами. Протокол междоменной маршрутизации, который позволил Интернету масштабироваться до его текущего размера, — это BGP. Мы рассмотрим, как работает BGP, и обсудим проблемы, с которыми сталкивается BGP по мере роста Интернета.

Тесно связана с масштабируемостью маршрутизации проблема адресации. Еще два десятилетия назад стало ясно, что 32-битная схема адресации IPv4 не будет работать вечно. Это привело к определению новой версии IP — версии 6, поскольку версия 5 использовалась в более раннем эксперименте. IPv6 в первую очередь расширяет адресное пространство, но также добавляет ряд новых функций, некоторые из которых были внедрены и в IPv4.

Хотя Интернет продолжает расти в размерах, ему также нужно развивать свою функциональность. Последние главы этого раздела охватывают некоторые значительные улучшения возможностей Интернета. Первое, мультикаст, является улучшением базовой модели услуги. Мы покажем, как мультикаст (возможность эффективной доставки одних и тех же пакетов группе получателей) может быть внедрен в интернет, и опишем несколько протоколов маршрутизации, разработанных для поддержки мультикаста. Второе улучшение, *многопротокольная коммутация по меткам* (Multiprotocol Label Switching, MPLS), модифицирует механизм пересылки IP-сетей. Эта модификация позволила внести изменения в способ выполнения маршрутизации IP и в услуги, предлагаемые IP-сетями. Наконец, мы рассмотрим влияние мобильности на маршрутизацию и опишем некоторые улучшения IP для поддержки мобильных узлов и маршрутизаторов. Для каждого из этих улучшений вопросы масштабируемости продолжают оставаться важными.

Глава 4.1. Глобальный интернет

На данный момент мы видели, как подключить гетерогенную коллекцию сетей для создания интернета и как использовать простую иерархию IP-адреса для того, чтобы сделать маршрутизацию в интернете несколько масштабируемой. Мы говорим «несколько» масштабируемой, потому что, хотя каждому маршрутизатору и не нужно знать обо всех хостах, подключенных к интернету, ему все же в модели, описанной до сих пор, нужно знать обо всех сетях, подключенных к интернету. Сегодняшний Интернет имеет сотни тысяч подключенных сетей (или больше, в зависимости от способа подсчета). Протоколы маршрутизации, такие как те, которые мы только что обсу-

дили, не масштабируются до таких объемов. В этой главе рассматриваются различные методы, которые значительно улучшили масштабируемость и позволили Интернету вырасти до его текущего размера.

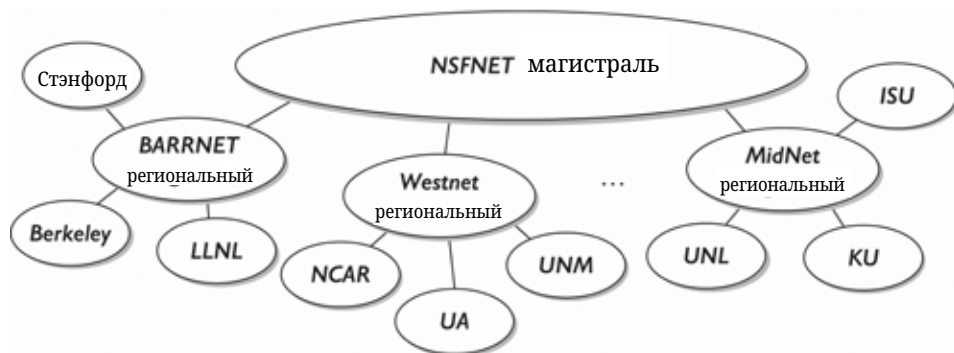


Рисунок 4.1. Древовидная структура Интернета в 1990 году.

Прежде чем перейти к этим методам, нам нужно получить общее представление о том, как выглядит глобальный Интернет. Это не просто случайное соединение Ethernet, а структура, которая отражает тот факт, что он соединяет множество различных организаций. Рис. 4.1 дает простое изображение состояния Интернета в 1990 году. С тех пор топология Интернета стала намного сложнее, чем предполагает этот рисунок (мы представили немного более точное изображение текущего Интернета далее), но это изображение подойдет для начала.

Одна из заметных особенностей этой топологии заключается в том, что она состоит из сайтов конечных пользователей (например, Стэнфордский университет), которые подключены к сетям поставщиков услуг (например, BARRNET был сетью поставщика, обслуживающей сайты в районе залива Сан-Франциско). В 1990 году многие поставщики обслуживали ограниченный географический регион и поэтому были известны как *региональные сети*. Региональные сети, в свою очередь, были соединены общенациональной магистралью. В 1990 году эта магистраль финансировалась Национальным научным фондом (NSF) и поэтому называлась магистралью *NSFNET*.

NSFNET уступила место Internet2, которая до сих пор управляет магистралью от имени научных и образовательных учреждений в США (в других странах есть аналогичные научные и образовательные сети), но, конечно, большинство людей получают подключение к Интернету от коммерческих поставщиков. Хотя на рисунке это не показано, сегодня крупнейшие сети поставщиков (называемые tier-1) обычно состоят из десятков высокопроизводительных маршрутизаторов, расположенных в крупных мегаполисах (разговорно называемых «города NFL»), соединенных соединениями «точка-точка» (часто с пропускной способностью 100 Гбит/с). Аналогично каждый сайт конечного пользователя обычно не является одной сетью, а состоит из нескольких физических сетей, соединенных коммутаторами и маршрутизаторами.

Обратите внимание, что каждый поставщик и конечный пользователь, вероятно, являются административно независимыми единицами. Это имеет некоторые значительные последствия для маршрутизации. Например, вполне вероятно, что у разных поставщиков будут разные представления о том, какой протокол маршрутизации лучше использовать в их сетях и как должны назначаться метрики для соединений в их сети. Из-за этой независимости сеть каждого поставщика обычно является одной *автономной системой* (AS). Мы определим этот термин более точно в следующей главе, но пока достаточно думать об AS как о сети, которая администрируется независимо от других AS.

Тот факт, что у Интернета есть различимая структура, можно использовать в нашу пользу, решая проблему масштабируемости. На самом деле нам нужно справиться с двумя связанными проблемами масштабируемости. Первая — это масштабируемость маршрутизации. Нам нужно найти способы минимизировать количество номеров сетей, которые передаются в протоколах маршрутизации и хранятся в таблицах маршрутизации маршрутизаторов. Вторая — это использование адресного пространства, то есть обеспечение того, чтобы пространство IP-адресов не исчерпывалось слишком быстро.

На протяжении всей этой книги мы видим принцип иерархии, который снова и снова используется для улучшения масштабируемости. В предыдущем разделе мы видели, как иерархическая структура IP-адресов, особенно с гибкостью, предоставляемой бесклассовой междоменной маршрутизацией (CIDR) и подсетями, может улучшить масштабируемость маршрутизации. В следующих двух главах мы увидим дальнейшее использование иерархии (и ее партнера — агрегации) для обеспечения большей масштабируемости, сначала в одном домене, а затем между доменами. В последней подглаве будет рассмотрен IP версии 6, изобретение которого в значительной степени стало результатом проблем с масштабируемостью.

Глава 4.1.1. Области маршрутизации

В качестве первого примера использования иерархии для масштабирования системы маршрутизации мы рассмотрим, как протоколы маршрутизации состояния соединения (такие как OSPF и IS-IS) могут использоваться для разделения домена маршрутизации на поддомены, называемые *областями*. (Терминология несколько различается между протоколами — здесь мы используем терминологию OSPF.) Добавив этот дополнительный уровень иерархии, мы позволяем одному домену становиться больше, не перегружая протоколы маршрутизации и не прибегая к более сложным междоменным протоколам маршрутизации, описанным далее.

Область — это набор маршрутизаторов, которые административно настроены для обмена информацией о состоянии соединения друг с другом. Существует одна специальная область — магистральная область, также известная как область 0. Пример домена маршрутизации, разделенного на области, показан на рис. 4.2. Маршрутизаторы R1, R2 и R3 являются членами магистральной области. Они также являются членами по крайней мере одной немагистральной области; R1 фактически является членом как области 1, так и области 2. Маршрутизатор, который является членом как магистральной области, так и немагистральной области, называется граничным маршрутизатором области (area border router, ABR). Обратите внимание, что эти маршрутизаторы отличаются от маршрутизаторов, находящихся на краю AS, которые для ясности называются граничными маршрутизаторами AS.

Маршрутизация внутри одной области происходит точно так же, как описано в предыдущем разделе. Все маршрутизаторы в области отправляют друг другу объявления о состоянии соединения и таким образом развивают полную, согласованную карту области. Однако объявления о состоянии соединения маршрутизаторов, которые не являются граничными маршрутизаторами области, не покидают область, в которой они возникли. Это позволяет процессам распространения и вычисления маршрутов быть значительно более масштабируемыми. Например, маршрутизатор R4 в области 3 никогда не увидит объявление о состоянии соединения от маршрутизатора R8 в области 1. В результате он не будет знать ничего о детальной топологии областей, кроме своей собственной.

Как тогда маршрутизатор в одной области определяет правильный следующий шаг для пакета, предназначенного для сети в другой области? Ответ на этот вопрос становится ясным, если представить путь пакета, который должен пройти от одной немагистральной области до другой, разделенным на три части. Сначала он путешествует от своей исходной сети до магистральной области, затем пересекает магистраль, после чего путе-

шествует от магистральной до целевой сети. Чтобы это работало, граничные маршрутизаторы области суммируют информацию о маршрутизации, которую они получили из одной области, и делают ее доступной в своих объявлениях для других областей. Например, R1 получает объявления о состоянии соединения от всех маршрутизаторов в области 1 и таким образом может определить стоимость достижения любой сети в области 1. Когда R1 отправляет объявления о состоянии соединения в область 0, он рекламирует стоимость достижения сетей в области 1 так, как будто все эти сети напрямую подключены к R1. Это позволяет всем маршрутизаторам области 0 узнать стоимость достижения всех сетей в области 1. Граничные маршрутизаторы области затем суммируют эту информацию и рекламируют ее в немагистральные области. Таким образом, все маршрутизаторы узнают, как достичь всех сетей в домене.

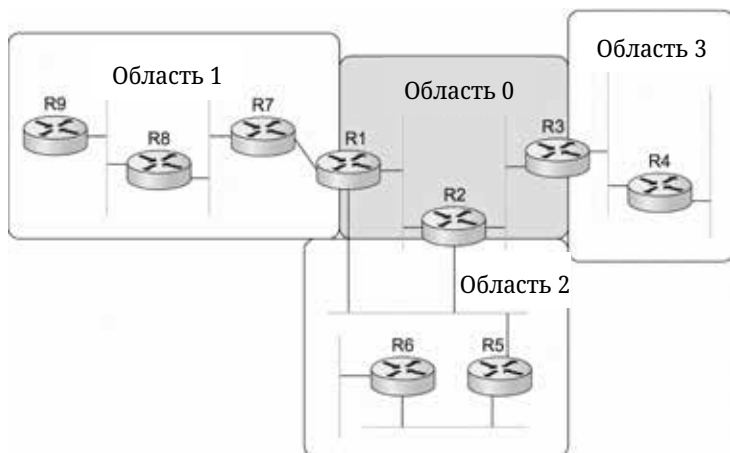


Рисунок 4.2. Домен, разделенный на области.

Заметьте, что в случае области 2 есть два ABR, и маршрутизаторы в области 2, таким образом, должны сделать выбор, какой из них использовать для достижения магистральной. Это достаточно просто, так как и R1, и R2 будут рекламировать стоимости до различных сетей, так что станет ясно, какой вариант лучше, когда маршрутизаторы в области 2 запустят свой алгоритм кратчайшего пути. Например, вполне очевидно, что R1 будет лучшим выбором, чем R2, для назначения в области 1.

При разделении домена на области сетевой администратор старается достичь компромисса между масштабируемостью и оптимальностью маршрутизации. Использование областей заставляет все пакеты, путешествующие из одной области в другую, проходить через магистральную область, даже если доступен более короткий путь. Например, даже если R4 и R5 напрямую соединены, пакеты не будут течь между ними, потому что они находятся в разных немагистральных областях. Оказывается, что потребность в масштабируемости часто важнее, чем необходимость использовать абсолютно кратчайший путь.

Основные выводы

Это обсуждение иллюстрирует важный принцип в проектировании сетей. Часто возникает компромисс между масштабируемостью и некоторым видом оптимальности. Когда вводится иерархия, информация скрывается от некоторых узлов в сети, что мешает им принимать решения. Однако сокрытие информации является важным для масштабирования решения, так как это избавляет все узлы от необходимости иметь глобальные знания. В больших сетях всегда справедливо, что масштабируемость является более насущной целью проектирования, чем выбор оптимального маршрута.

Наконец, отметим, что существует трюк, с помощью которого сетевые администраторы могут более гибко решать, какие маршрутизаторы входят в область 0. Этот трюк использует идею *виртуальной связи* между маршрутизаторами. Такая виртуальная связь создается путем настройки маршрутизатора, который не подключен напрямую к области 0, на обмен информацией о маршрутизации магистральной с маршрутизатором, который подключен. Например, виртуальная связь может быть настроена между R8 и R1, таким образом делая R8 частью магистральной. Теперь R8 будет участвовать в распространении объявлений о состоянии соединения с другими маршрутизаторами в области 0. Стоимость виртуальной связи от R8 до R1 определяется обменом информацией о маршрутизации, который происходит в области 1. Этот метод может помочь улучшить оптимальность маршрутизации.

Глава 4.1.2. Междоменная маршрутизация (BGP)

В начале этого раздела мы узнали, что Интернет организован в виде автономных систем (autonomous system, AS), каждая из которых находится под контролем одной административной структуры. Сложная внутренняя сеть корпорации может быть одной автономной системой, как и национальная сеть любого отдельного интернет-провайдера (ISP). Рис. 4.3 показывает простую сеть с двумя автономными системами.

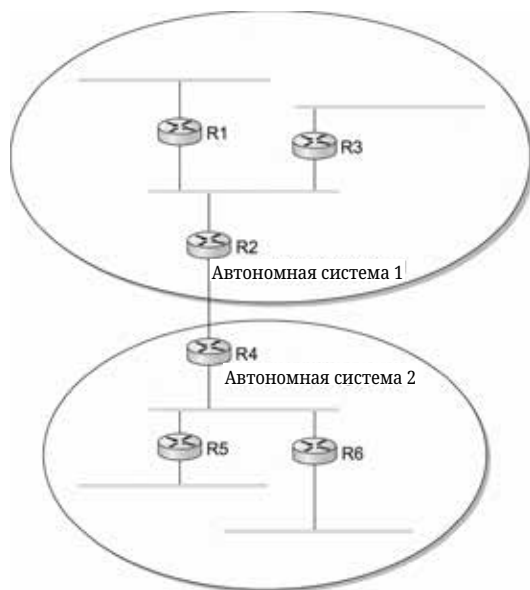


Рисунок 4.3. Сеть с двумя автономными системами.

Основная идея автономных систем заключается в предоставлении дополнительного способа иерархической агрегации информации о маршрутизации в большом интернете, таким образом улучшая масштабируемость. Теперь мы разделяем задачу маршрутизации на две части: маршрутизацию внутри одной автономной системы и маршрутизацию между автономными системами. Так как еще одно название для автономных систем в Интернете — это *домены* маршрутизации, мы называем две части проблемы маршрутизации междоменной маршрутизацией и внутримоменной маршрутизацией. Помимо улучшения масштабируемости модель AS отделяет внутримоменную маршрутизацию, происходящую в одной AS, от той, что происходит в другой AS. Таким образом, каждая AS может использовать любые протоколы внутримоменной маршрутизации, которые она выберет. Она может даже использовать статические

маршруты или несколько протоколов, если пожелает. Проблема междоменной маршрутизации заключается в том, чтобы разные AS обменивались информацией о достижимости — описаниями набора IP-адресов, которые могут быть достигнуты через данную AS, — друг с другом.

Проблемы междоменной маршрутизации

Возможно, самой важной проблемой междоменной маршрутизации сегодня является необходимость для каждой AS определять свою собственную политику маршрутизации. Простой пример политики маршрутизации, реализуемой в конкретной AS, может выглядеть так: «Когда это возможно, я предпочитаю отправлять трафик через AS X, а не через AS Y, но буду использовать AS Y, если это единственный путь, и я никогда не хочу переносить трафик из AS X в AS Y или наоборот». Такая политика была бы типична, если бы я заплатил деньги как AS X, так и AS Y, чтобы подключить мою AS к остальной части Интернета, и AS X является моим предпочтительным провайдером подключения, с AS Y в качестве запасного варианта. Поскольку я рассматриваю как AS X, так и AS Y в качестве провайдеров (и, вероятно, я заплатил им за эту роль), я не ожидаю, что буду помогать им, перенося трафик между ними через свою сеть (это называется *транзитным трафиком*). Чем больше автономных систем я подключаю, тем более сложной может становиться моя политика, особенно если учитывать магистральных провайдеров, которые могут подключаться к десяткам других провайдеров и сотням клиентов и иметь различные экономические соглашения (которые влияют на политику маршрутизации) с каждым из них.

Ключевой целью проектирования междоменной маршрутизации является то, что политики, подобные приведенному выше примеру, и гораздо более сложные, должны поддерживаться системой междоменной маршрутизации. Для усложнения задачи мне нужно быть способным реализовать такую политику без какой-либо помощи от других автономных систем и в условиях возможных ошибок настройки или вредоносного поведения других автономных систем. Кроме того, часто существует желание сохранить политику в тайне, потому что субъекты, управляющие автономными системами (в основном ISPs), часто конкурируют друг с другом и не хотят, чтобы их экономические соглашения становились публичными.

В истории Интернета было два основных протокола междоменной маршрутизации. Первым был *протокол внешних шлюзов* (Exterior Gateway Protocol, EGP), который имел ряд ограничений, возможно, самое серьезное из которых заключалось в значительном ограничении топологии Интернета. EGP был разработан, когда Интернет имел древовидную топологию, как показано на рис. 4.1, и не позволял топологии стать более общей. В этой простой древовидной структуре существует единственный магистральный канал, и автономные системы соединяются только как «родители» и «дети», а не как равноправные узлы.

Заменой EGP стал *протокол пограничного шлюза* (Border Gateway Protocol, BGP), который прошел через четыре версии (BGP-4). BGP часто считается одной из самых сложных частей Интернета. Сейчас мы рассмотрим некоторые его ключевые моменты.

В отличие от своего предшественника EGP, BGP практически не делает предположений о том, как автономные системы взаимосвязаны — они образуют произвольный граф. Эта модель достаточно общая, чтобы учитывать не древовидные структуры интернета, как упрощенная картинка многопровайдерского Интернета, показанная на рис. 4.4. (Оказывается, у Интернета все же есть некоторая структура, как мы увидим далее, но она все не такая простая, как дерево, и BGP не делает предположений о такой структуре.)

В отличие от простой древовидной структуры Интернета, показанной на рисунке 4.1, или даже довольно простой картинки на рисунке 4.4, сегодняшний Интернет состоит из широко взаимосвязанных сетей, в основном управляемых частными компаниями (ISP), а не государствами. Многие интернет-провайдеры (ISP) существуют главным образом для предоставления услуг «потребителям» (т.е. людям с компьютерами дома), в то время как другие предлагают нечто большее, чем старая магистральная служба, связывая

других провайдеров и иногда крупные корпорации. Часто многие провайдеры договариваются о взаимосвязи друг с другом в одной точке обмена *трафиком*.

Чтобы лучше понять, как мы можем управлять маршрутизацией среди этого сложно-го взаимосвязанного набора автономных систем, начнем с определений нескольких терминов. Мы определяем *локальный трафик* как трафик, который начинается или заканчивается на узлах внутри AS, и *транзитный трафик* как трафик, проходящий через AS. Мы можем классифицировать автономные системы на три широких типа:

- Stub AS — AS, которая имеет только одно соединение с другой AS; такая AS будет обрабатывать только локальный трафик. Маленькая корпорация на рис. 4.4 является примером stub AS.
- Multihomed AS — AS, которая имеет соединения с более чем одной другой AS, но отказывается обрабатывать транзитный трафик, как большая корпорация в верхней части рис. 4.4.
- Transit AS — AS, которая имеет соединения с более чем одной другой AS и предназначена для обработки как транзитного, так и локального трафика, как магистральные провайдеры на рис. 4.4.

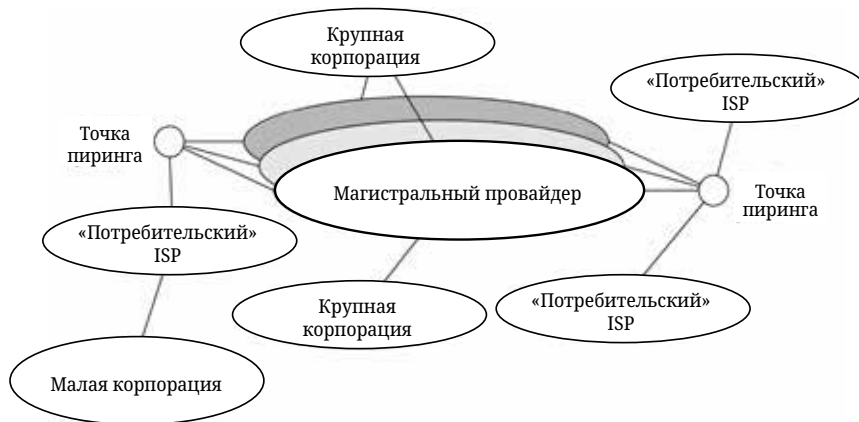


Рисунок 4.4. Простой многопровайдерский Интернет.

В то время как обсуждение маршрутизации в предыдущем разделе фокусировалось на нахождении оптимальных путей на основе минимизации какого-либо показателя канала, цели междоменной маршрутизации гораздо более сложные. Во-первых, необходимо найти *некоторый* путь к предполагаемому пункту назначения, который не образует циклов. Во-вторых, пути должны соответствовать политикам различных автономных систем на пути, и, как мы уже видели, эти политики могут быть почти произвольно сложными. Таким образом, в то время как внутридоменная маршрутизация фокусируется на хорошо определенной задаче оптимизации скалярной стоимости пути, междоменная маршрутизация фокусируется на нахождении нециклического пути, соответствующего политикам, что представляет собой гораздо более сложную задачу оптимизации.

Существуют дополнительные факторы, которые усложняют междоменную маршрутизацию. Первый — это просто масштаб. Магистральный маршрутизатор Интернета должен быть способен пересылать любой пакет, предназначенный для любого места в Интернете. Это означает, что в его таблице маршрутизации должно быть соответствие для любого действительного IP-адреса. Хотя CIDR помог контролировать количество различных префиксов, которые передаются в магистральной маршрутизации Интернета, неизбежно, что информации о маршрутах будет много — примерно 700 000 префиксов на середину 2018 года.

Дополнительная трудность междоменной маршрутизации возникает из-за автономного характера доменов. Следует отметить, что каждый домен может использовать свои соб-

ственные внутренние протоколы маршрутизации и любые схемы для назначения метрик путям. Это означает, что невозможно вычислить значимые затраты пути для маршрута, пересекающего несколько автономных систем. Стоимость 1000 у одного провайдера может означать отличный путь, а у другого провайдера — неприемлемо плохой. В результате междоменная маршрутизация рекламирует только *достижимость*. Концепция достижимости — это по сути заявление о том, что «вы можете достичь этой сети через эту AS». Это означает, что выбрать оптимальный путь для междоменной маршрутизации практически невозможно.

Автономный характер междоменной маршрутизации поднимает вопрос доверия. Провайдер А может не захотеть верить определенным объявлениям от провайдера В, опасаясь, что провайдер В будет распространять неверную информацию о маршрутах. Например, доверие провайдеру В, когда он объявляет отличный маршрут в любую точку Интернета, может оказаться катастрофическим, если провайдер В ошибочно настроил свои маршрутизаторы или не имеет достаточной пропускной способности для передачи трафика.

Вопрос доверия также связан с необходимостью поддержки сложных политик, как отмечено выше. Например, я могу быть готов доверять конкретному провайдеру только тогда, когда он объявляет достижимость до определенных префиксов, и поэтому у меня будет политика, которая гласит: «Используй AS X для достижения только префиксов *p* и *q*, если и только если AS X объявляет достижимость до этих префиксов».

Основы BGP

Каждая AS имеет один или несколько *пограничных маршрутизаторов*, через которые пакеты входят и выходят из AS. В нашем простом примере на рис. 4.3 маршрутизаторы R2 и R4 будут пограничными маршрутизаторами. (За эти годы маршрутизаторы иногда называли *шлюзами*, отсюда и названия протоколов BGP и EGP). Пограничный маршрутизатор — это просто IP-маршрутизатор, который отвечает за пересылку пакетов между автономными системами.

Каждая AS, участвующая в BGP, также должна иметь как минимум один BGP-спикер — маршрутизатор, который «говорит» на языке BGP с другими BGP-спикерами в других автономных системах. Обычно пограничные маршрутизаторы также являются BGP-спикерами, но это не обязательно.

BGP не принадлежит ни к одному из двух основных классов протоколов маршрутизации — протоколам дистанционно-векторной или линейно-сетевой маршрутизации. В отличие от этих протоколов, BGP рекламирует *полные пути* как перечисленный список автономных систем для достижения конкретной сети. По этой причине его иногда называют протоколом векторного пути. Реклама полных путей необходима для принятия решений о политике в соответствии с пожеланиями конкретной AS. Это также позволяет легко обнаруживать петли маршрутизации.

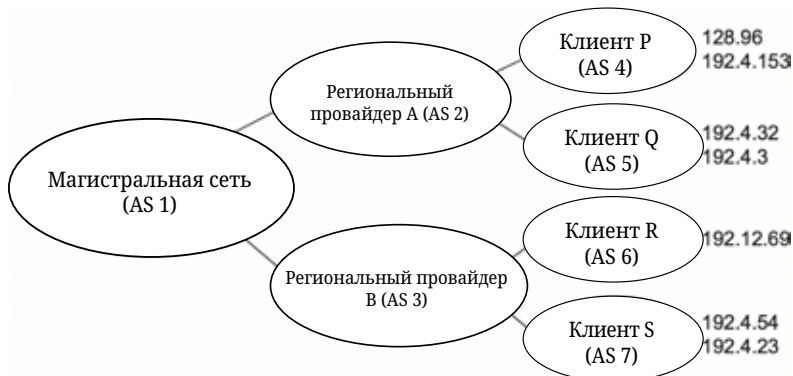


Рисунок 4.5. Пример сети, в которой работает BGP.

Чтобы понять, как это работает, рассмотрим очень простой пример сети на рис. 4.5. Предположим, что провайдеры являются транзитными сетями, а сети клиентов — конечными. BGP-спикер для AS провайдера А (AS 2) сможет рекламировать информацию о достижимости для каждого из сетевых номеров, присвоенных клиентам Р и Q. Таким образом, он скажет, по сути, «сети 128.96, 192.4.153, 192.4.32 и 192.4.3 могут быть достигнуты напрямую из AS 2». Основная сеть, получив это объявление, может рекламировать: «сети 128.96, 192.4.153, 192.4.32 и 192.4.3 могут быть достигнуты по пути (AS 1, AS 2)». Аналогично она могла бы рекламировать: «сети 192.12.69, 192.4.54 и 192.4.23 могут быть достигнуты по пути (AS 1, AS 3)».

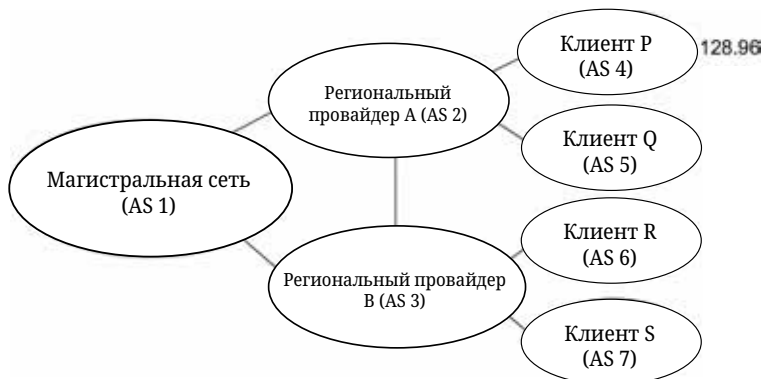


Рисунок 4.6. Пример петли между автономными системами.

Важная задача BGP — предотвратить создание петлевых путей. Например, рассмотрим сеть, изображенную на рис. 4.6. Она отличается от рис. 4.5 только добавлением дополнительной связи между AS 2 и AS 3, но теперь в графе автономных систем есть петля. Предположим, AS 1 узнает, что может достичь сети 128.96 через AS 2, поэтому она рекламирует этот факт AS 3, которая, в свою очередь, рекламирует его обратно AS 2. При отсутствии какого-либо механизма предотвращения петель AS 2 могла бы теперь решить, что AS 3 является предпочтительным маршрутом для пакетов, адресованных 128.96. Если AS 2 начнет отправлять пакеты, адресованные 128.96, в AS 3, AS 3 отправит их в AS 1; AS 1 отправит их обратно в AS 2; и они будут находиться в петле бесконечно. Это предотвращается за счет передачи полного пути AS в маршрутных сообщениях. В данном случае объявление о пути к 128.96, полученное AS 2 от AS 3, будет содержать путь AS (AS 3, AS 1, AS 2, AS 4). AS 2 видит себя в этом пути и, следовательно, заключает, что этот путь для нее не полезен.

Чтобы эта техника предотвращения петель работала, номера AS, передаваемые в BGP, должны быть уникальными. Например, AS 2 может распознать себя в пути AS в приведенном выше примере, только если никакая другая AS не идентифицирует себя таким же образом. Номера AS теперь 32-битные, и они назначаются центральным органом для обеспечения уникальности.

Определенная автономная система (AS) будет рекламировать только те маршруты, которые она считает достаточно хорошими для себя. То есть если BGP-спикер имеет выбор из нескольких различных маршрутов к месту назначения, он выберет лучший по своим внутренним политикам и именно этот маршрут будет рекламировать. Более того, BGP-спикер не обязан рекламировать какой-либо маршрут к месту назначения, даже если у него такой есть. Таким образом, AS может реализовать политику отказа от транзита, отказываясь рекламировать маршруты к префиксам, которые не содержатся в этой AS, даже если она знает, как до них добраться.

Поскольку ссылки могут отказать, а политики измениться, BGP-спикеры должны иметь возможность отменять ранее рекламируемые пути. Это делается с помощью формы отрицательной рекламы, известной как «*отозванный маршрут*». Как положительная, так и отрицательная информация о достижимости передается в BGP-сообщении обновления, формат которого показан на рисунке 4.7. (Обратите внимание, что поля на этом рисунке кратны 16 битам, в отличие от других форматов пакетов в этой главе.)

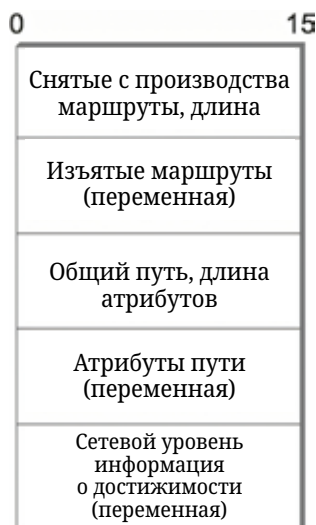


Рисунок 4.7. Формат пакета обновления BGP-4.

В отличие от описанных в предыдущей главе протоколов маршрутизации, BGP определяется как работающий поверх TCP, надежного транспортного протокола. Поскольку BGP-спикеры могут рассчитывать на надежность TCP, это означает, что любая информация, отправленная от одного спикера к другому, не нуждается в повторной отправке. Таким образом, пока ничего не изменилось, BGP-спикер может просто отправлять периодическое *keepalive*-сообщение, которое говорит, по сути: «Я все еще здесь и ничего не изменилось». Если этот маршрутизатор выйдет из строя или потеряет связь с парой, он прекратит отправку *keepalive*-сообщений, и другие маршрутизаторы, которые узнали маршруты от него, предположат, что эти маршруты больше не действительны.

Общие отношения и политики AS (автономной системы)

Хотя было сказано, что политики могут быть произвольно сложными, существует несколько общих, отражающих распространенные отношения между автономными системами. На рис. 4.8 иллюстрируются три распространенные отношения и соответствующие им политики:

- **Провайдер-Клиент:** Провайдеры занимаются подключением своих клиентов к остальной части Интернета. Клиентом может быть корпорация или меньший ISP (который может иметь своих клиентов). Обычная политика заключается в том, чтобы рекламировать все известные маршруты своему клиенту и рекламировать маршруты, полученные от своего клиента, всем остальным.
- **Клиент-Провайдер:** В другом направлении клиент хочет, чтобы его провайдер направлял ему трафик (и его клиентам, если они у него есть) и чтобы он мог отправлять трафик в остальную часть Интернета через своего провайдера. Обычная политика в этом случае заключается в том, чтобы рекламировать свои собственные префиксы и маршруты, полученные от своих клиентов, своему провайдеру,

рекламировать маршруты, полученные от своего провайдера, своим клиентам, но не рекламировать маршруты, полученные от одного провайдера, другому провайдеру. Последняя часть важна, чтобы клиент не оказался в положении, когда ему придется передавать трафик от одного провайдера другому, что ему не выгодно, если он платит провайдерам за передачу трафика для него.

- *Пир (партнер)*: Третий вариант — симметричное пиринговое соединение между автономными системами. Два провайдера, которые считают себя равными, обычно обмениваются пирингом, чтобы получить доступ к клиентам друг друга без необходимости платить другому провайдеру. Типичная политика здесь заключается в том, чтобы рекламировать маршруты, полученные от своих клиентов, своему пиру, рекламировать маршруты, полученные от своего пира, своим клиентам, но не рекламировать маршруты от своего пира любому провайдеру или наоборот.

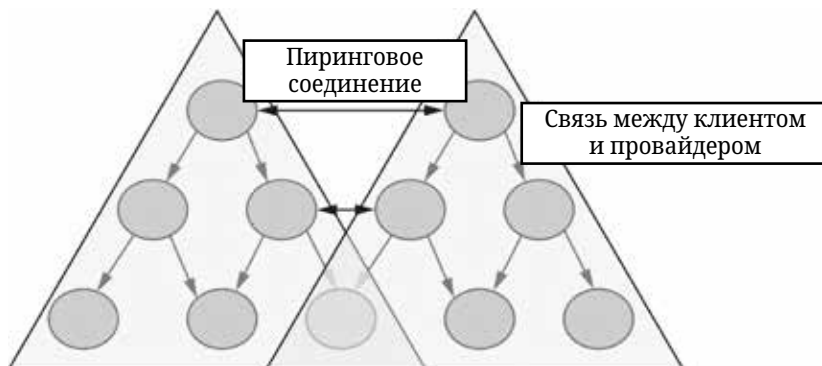


Рисунок 4.8. Общие отношения AS.

Одна из вещей, которую стоит отметить на этом рисунке, это то, как он привнес некоторую структуру в кажущийся неструктурированным Интернет. В нижней части иерархии находятся заглушечные сети, которые являются клиентами одного или нескольких провайдеров, и по мере продвижения вверх по иерархии мы видим провайдеров, у которых другие провайдеры являются клиентами. На вершине находятся провайдеры, у которых есть клиенты и равноправные партнеры (пиры), но которые сами не являются чьи-либо клиентами. Эти провайдеры известны как провайдеры *первого уровня* (Tier-1).

Основные выводы

Вернемся к основному вопросу: как все это помогает нам строить масштабируемые сети? Во-первых, количество узлов, участвующих в BGP, соответствует количеству автономных систем, что значительно меньше, чем количество сетей. Во-вторых, нахождение хорошего междоменного маршрута сводится к нахождению пути к нужному пограничному маршрутизатору, которых в каждой AS всего несколько. Таким образом, мы аккуратно подразделили проблему маршрутизации на управляемые части, снова используя новый уровень иерархии для повышения масштабируемости. Сложность междоменной маршрутизации теперь пропорциональна количеству автономных систем, а сложность внутридоменной маршрутизации пропорциональна количеству сетей в одной AS.

Интеграция междоменной и внутридоменной маршрутизации

Хотя предыдущее обсуждение иллюстрирует, как BGP-спикер узнает информацию о междоменной маршрутизации, остается вопрос о том, как все остальные маршрутизаторы в домене получают эту информацию. Существует несколько способов решения этой проблемы.

Начнем с очень простой ситуации, которая также является широко распространенной. В случае глухой AS, которая подключается к другим автономным системам только в одной точке, пограничный маршрутизатор явно является единственным выбором для всех маршрутов, находящихся за пределами AS. Такой маршрутизатор может внедрить маршрут *по умолчанию* во внутридоменный протокол маршрутизации. Фактически это заявление о том, что любая сеть, которая не была явно анонсирована во внутридоменном протоколе, доступна через пограничный маршрутизатор. Напомним из обсуждения пересылки IP ранее, что запись по умолчанию в таблице пересылки идет после всех более конкретных записей и совпадает с чем угодно, что не совпало с конкретной записью.

Следующий шаг по сложности — это когда пограничные маршрутизаторы внедряют конкретные маршруты, которые они узнали из-за пределов AS. Рассмотрим, например, пограничный маршрутизатор провайдера AS, который подключается к клиентской AS. Этот маршрутизатор может узнать, что сетевой префикс 192.4.54/24 находится внутри клиентской AS, либо через BGP, либо потому, что информация была сконфигурирована в пограничном маршрутизаторе. Он может внедрить маршрут к этому префиксу во внутридоменный протокол маршрутизации, работающий внутри провайдера AS. Это будет реклама вида «у меня есть связь с 192.4.54/24 с ценой X». Это заставит другие маршрутизаторы в провайдерской AS узнать, что этот пограничный маршрутизатор — место, куда следует отправлять пакеты, предназначенные для этого префикса.

Последний уровень сложности возникает в магистральных сетях, которые узнают так много информации о маршрутизации через BGP, что становится слишком затратно внедрять ее во внутридоменный протокол. Например, если пограничный маршрутизатор хочет внедрить 10,000 префиксов, о которых он узнал из другой AS, ему придется отправлять очень большие пакеты состояния связи другим маршрутизаторам в этой AS, и их вычисления кратчайших путей станут очень сложными. По этой причине маршрутизаторы в магистральной сети используют вариант BGP, называемый *внутренним* BGP (iBGP), чтобы эффективно перераспределять информацию, полученную BGP-спикерами на границах AS, всем другим маршрутизаторам в AS. (Другой вариант BGP, обсуждаемый выше, работает между автономными системами и называется внешним BGP, или eBGP). iBGP позволяет любому маршрутизатору в AS узнать лучший пограничный маршрутизатор для использования при отправке пакета на любой адрес. В то же время каждый маршрутизатор в AS отслеживает, как добраться до каждого пограничного маршрутизатора, используя обычный внутридоменный протокол без внедренной информации. Объединяя эти два набора информации, каждый маршрутизатор в AS может определить соответствующий следующий шаг для всех префиксов.

Для того чтобы понять, как это все работает, рассмотрим простую сеть, представляющую одну AS, изображенную на рис. 4.9. Три пограничных маршрутизатора, A, D и E, используют eBGP для взаимодействия с другими автономными системами и получают информацию о доступе к различным префиксам. Эти три пограничных маршрутизатора обмениваются информацией друг с другом и с внутренними маршрутизаторами B и C, создавая сетку из iBGP-сессий между всеми маршрутизаторами в AS. Теперь давайте сосредоточимся на том, как маршрутизатор B формирует свое полное представление о том, как пересылать пакеты к любому префиксу. Посмотрите на верхнюю левую часть рис. 4.10, где показана информация, которую маршрутизатор B получает из своих iBGP-сессий. Он узнает, что некоторые префиксы лучше всего достигаются через маршрутизатор A, некоторые через D, а некоторые через E. В то же время все маршрутизаторы в AS также используют внутридоменный протокол маршрутизации, такой как Routing Information Protocol (RIP) или Open Shortest Path First (OSPF). (Общий термин для внутридоменных протоколов — это внутренний шлюзовой протокол, или IGP.) Из этого совершенно отдельного протокола B узнает, как достигать других узлов *внутри* домена, как показано в таблице в верхней правой части. Например, чтобы достичь маршрутизатора E, B должен отправлять пакеты в направлении маршрутизатора C. Наконец, в ниж-

ней таблице В собирает всю информацию, комбинируя информацию о внешних префиксах, полученную из iBGP, с информацией о внутренних маршрутах к пограничным маршрутизаторам, полученной из IGP. Таким образом, если префикс, такой как 18.0/16, достижим через пограничный маршрутизатор Е, и лучший внутренний путь к Е лежит через С, то это означает, что любой пакет, предназначенный для 18.0/16, должен быть направлен к С. Таким образом, любой маршрутизатор в AS может построить полную таблицу маршрутизации для любого префикса, который достижим через какой-либо пограничный маршрутизатор AS.

Глава 4.2. IP-версия 6

Мотивация для определения новой версии IP проста: необходимость справиться с исчерпанием адресного пространства IP. CIDR значительно помог сдерживать скорость, с которой потреблялось адресное пространство Интернета, а также помог контролировать рост информации о маршрутизации, необходимой в маршрутизаторах Интернета. Однако эти методы больше не являются достаточными. В частности, практически невозможно достичь 100 % эффективности использования адресов, поэтому адресное пространство было исчерпано задолго до того, как к Интернету было подключено 4-миллиардное устройство. Даже если бы мы смогли использовать все 4 миллиарда адресов, сейчас ясно, что IP-адреса нужно назначать не только традиционным компьютерам, но и смартфонам, телевизорам, бытовым приборам и дронам. Оглядываясь назад с позиции сегодняшнего дня, 32-битное адресное пространство кажется достаточно маленьким.

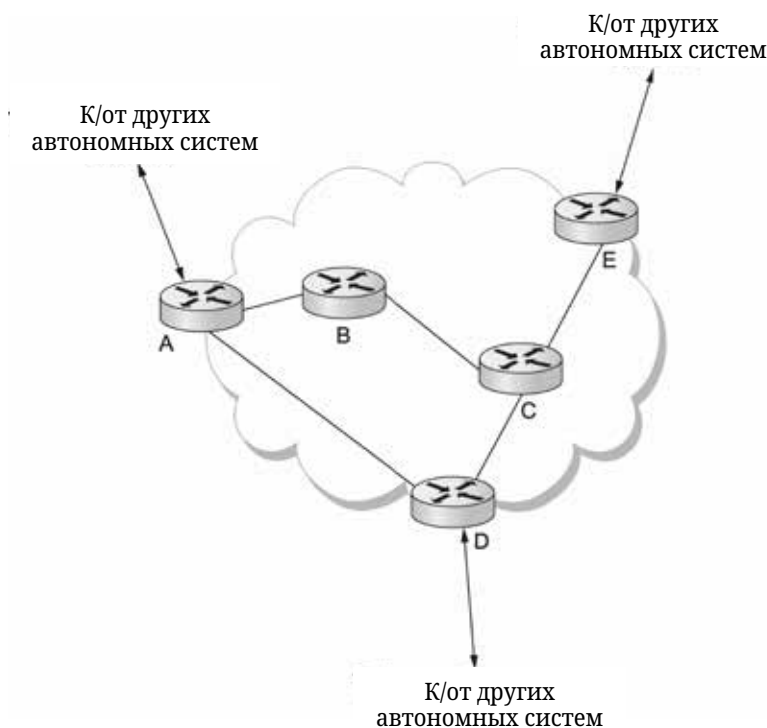


Рисунок 4.9. Пример междоменной и внутридоменной маршрутизации. На всех маршрутизаторах запущены iBGP и протокол внутридоменной маршрутизации. Пограничные маршрутизаторы А, D и Е также используют eBGP для связи с другими автономными системами.

Префикс	BGP Следующий скачок	Маршрутизатор	IGP Путь
18.0/16	E	A	A
12.5.5/24	A	C	C
128.34/16	D	D	C
128.69/16	A	E	C

BGP таблица для A

IGP таблица для B

Префикс	IGP Путь
18.0/16	C
12.5.5/24	A
128.34/16	C
128.69/16	A

Комбинированная таблица для маршрутизатора B

Рисунок 4.10. Таблица маршрутизации BGP, таблица маршрутизации IGP и комбинированная таблица на маршрутизаторе B.

Глава 4.2.1. Историческая перспектива

IETF начала рассматривать проблему расширения адресного пространства IP в 1991 году, и было предложено несколько альтернатив. Поскольку IP-адрес передается в заголовке каждого IP-пакета, увеличение размера адреса требует изменения заголовка пакета. Это означает новую версию Интернет-протокола и, как следствие, необходимость нового программного обеспечения для каждого хоста и маршрутизатора в Интернете. Очевидно, что это не тривиальное дело — это серьезное изменение, которое нужно тщательно обдумать.

Работа по определению новой версии IP изначально была известна как IP Next Generation, или IPng. По мере продвижения работы была присвоена официальная версия IP, так что IPng стал IPv6. Обратите внимание, что версия IP, обсуждаемая в данном разделе до сих пор, это версия 4 (IPv4). Очевидный разрыв в нумерации обусловлен тем, что номер версии 5 был использован для экспериментального протокола много лет назад.

Значимость перехода на новую версию IP вызвала эффект снежного кома. Общим мнением среди сетевых дизайнеров было то, что если вы собираетесь делать такое значительное изменение, то стоит одновременно исправить как можно больше других вещей в IP. В результате IETF запросил технические документы от всех желающих, прося внести предложения по функциональности, которая может быть желательна в новой версии IP. В дополнение к необходимости обеспечения масштабируемой маршрутизации и адресации некоторые из других пожеланий для IPng включали:

- Поддержку сервисов реального времени
- Поддержку безопасности
- Автоконфигурацию (то есть способность хостов автоматически настраиваться, получая такие данные, как свой IP-адрес и доменное имя)
- Расширенные возможности маршрутизации, включая поддержку мобильных хостов

Интересно отметить, что хотя многие из этих функций отсутствовали в IPv4 на момент разработки IPv6, поддержка их всех появилась в IPv4 в последние годы, часто с использованием схожих методов в обоих протоколах. Можно утверждать, что свобода мысли об IPv6 как о чистом листе облегчила разработку новых возможностей для IP, которые затем были адаптированы в IPv4.

В дополнение к списку желаемых функций абсолютно необходимой характеристикой IPv6 было наличие плана перехода с текущей версии IP (версии 4) на новую версию. Учитывая, что Интернет настолько велик и не имеет централизованного управления, совершенно невозможно было бы устроить «день смены флага», когда все отключили бы свои хосты и маршрутизаторы и установили новую версию IP. Архитекторы ожидали длительного переходного периода, в течение которого некоторые хосты и маршрутизаторы будут работать только с IPv4, некоторые — с IPv4 и IPv6, а некоторые — только с IPv6. Вряд ли они предполагали, что этот переходный период будет длиться уже 30 лет.

Глава 4.2.2. Адреса и маршрутизация

Прежде всего IPv6 предоставляет 128-битное адресное пространство, в отличие от 32-битного в версии 4. Таким образом, в то время как версия 4 потенциально может адресовать 4 миллиарда узлов при достижении 100 % эффективности назначения адресов, IPv6 может адресовать $3,4 \times 10^{38}$ узлов, также предполагая 100 % эффективность. Однако, как мы видели, 100 % эффективность назначения адресов маловероятна. Анализ других схем адресации, таких как французская и американская телефонные сети, а также IPv4, выявил некоторые эмпирические данные об эффективности назначения адресов. Основываясь на самых пессимистичных оценках эффективности, сделанных в этом исследовании, адресное пространство IPv6, по прогнозам, обеспечит более 1500 адресов на квадратный фут поверхности Земли, чего, несомненно, должно быть достаточно, даже если тостеры на Венере будут иметь собственные IP-адреса.

Выделение адресного пространства

Опираясь на эффективность CIDR в IPv4, адреса IPv6 также не имеют классов, но адресное пространство все же подразделяется различными способами на основе ведущих битов. Вместо указания различных классов адресов ведущие биты определяют различные способы использования адреса IPv6. Текущие назначения префиксов перечислены в табл. 4.1.

Таблица 4.1.
Назначения префиксов адресов для IPv6.

Префикс	Использование
00...0 (128 bits)	Неопределенно
00...1 (128 bits)	Шлейф
1111 1111	Адреса мультикаста
1111 1110 10	Локальный уникаст
Все остальное	Глобальный уникаст

Это выделение адресного пространства заслуживает некоторого обсуждения. Во-первых, вся функциональность трех основных классов адресов IPv4 (А, В и С) содержится внутри диапазона «всё остальное». Глобальные уникальные адреса, как мы вскоре увидим, очень похожи на бесклассовые адреса IPv4, только они гораздо длиннее. На данный момент они являются основными, с более чем 99% всего адресного пространства IPv6, доступного для этой важной формы адресов. (На момент написания книги уникальные адреса IPv6 выделяются из блока, начинающегося с 001, при этом оставшееся адресное пространство (около 87%) зарезервировано для будущего использования.)

Адресное пространство для многоадресной рассылки (multicast) предназначено для многоадресной рассылки, выполняя ту же роль, что и адреса класса D в IPv4. Обратите внимание, что адреса для многоадресной рассылки легко отличить — они начинаются с байта, заполненного единицами. Как используются эти адреса, мы рассмотрим в последующих главах.

Идея адресов для локального использования (link-local use) заключается в том, чтобы позволить хосту создать адрес, который будет работать в сети, к которой он подключен, не беспокоясь о глобальной уникальности адреса. Это может быть полезно для автоконфигурации, как мы увидим далее. Аналогично адреса для использования в пределах сайта (site-local use) предназначены для создания допустимых адресов в рамках сайта (например, частной корпоративной сети), который не подключен к большому Интернету; снова, глобальная уникальность не является проблемой.

В глобальном адресном пространстве уникальных адресов есть несколько важных типов специальных адресов. Узлу может быть назначен IPv6-адрес, совместимый с IPv4, путем дополнения 32-битного адреса IPv4 нулями до 128 бит. Узлу, который способен понимать только IPv4, может быть назначен IPv6-адрес с отображением IPv4 путем добавления перед 32-битным адресом IPv4 двух байтов, заполненных единицами, и затем дополнения нулями до 128 бит. Эти два типа специальных адресов полезны в процессе перехода с IPv4 на IPv6.

Нотация адресов

Как и в случае с IPv4, для записи адресов IPv6 существует специальная нотация. Стандартное представление — это `x:x:x:x:x:x:x:x`, где каждый `x` представляет собой шестнадцатеричное значение 16-битной части адреса. Примером может быть

```
47CD:1234:4422:AC02:0022:1234:A456:0124
```

Любой IPv6-адрес может быть записан с использованием этой нотации. Так как существует несколько специальных типов IPv6-адресов, разработаны также специальные нотации, которые могут быть полезны в определенных обстоятельствах. Например, адрес с большим количеством последовательных нулей может быть записан более компактно, с пропуском всех нулевых полей. Таким образом,

```
47CD:0000:0000:0000:0000:A456:0124
```

может быть записан как

```
47CD::A456:0124
```

Очевидно, что эта форма сокращения может использоваться только для одного набора последовательных нулей в адресе, чтобы избежать неоднозначности.

Два типа IPv6-адресов, содержащих встроенный IPv4-адрес, имеют свою специальную нотацию, которая облегчает извлечение адреса IPv4. Например, IPv6-адрес с отображением узла IPv4, чей IPv4-адрес был 128.96.33.81, может быть записан как

```
::FFFF:128.96.33.81
```

То есть последние 32 бита записываются в нотации IPv4, а не как пара шестнадцатеричных чисел, разделенных двоеточием. Обратите внимание, что двойное двоеточие в начале указывает на ведущие нули.

Глобальные адреса одноадресной рассылки (unicast)

На сегодняшний день наиболее важным видом адресации, который должен обеспечивать IPv6, является обычная одноадресная рассылка (unicast addressing). Она должна поддерживать быстрый темп добавления новых хостов в Интернет и обеспечивать масштабируемую маршрутизацию по мере роста числа физических сетей в Интернете. Таким образом, в основе IPv6 лежит план распределения одноадресных рассылок, который опре-

деляет, как одноадресные рассылки будут назначаться поставщикам услуг, автономным системам, сетям, хостам и маршрутизаторам.

Фактически предложенный план распределения адресов для одноадресных рассылок IPv6 чрезвычайно схож с тем, который используется в IPv4 с применением CIDR. Чтобы понять, как он работает и как обеспечивает масштабируемость, полезно ввести некоторые новые термины. Мы можем рассматривать нетранзитную автономную систему (например, тупиковую или мультидомашнюю автономную систему) как абонента, а транзитную автономную систему — как поставщика. Более того, мы можем подразделить поставщиков на прямых и непрямых. Первые напрямую подключены к абонентам. Вторые в основном подключают других поставщиков, не подключены напрямую к абонентам и часто называются *магистральными сетями*.

С этим набором определений мы можем видеть, что Интернет — это не просто произвольно соединенный набор автономных систем; у него есть некоторая внутренняя иерархия. Сложность заключается в использовании этой иерархии без создания механизмов, которые перестают работать при строгом несоблюдении иерархии, как это произошло с EGP. Например, различие между прямыми и непрямыми поставщиками становится размытым, когда абонент подключается к магистральной или когда прямой поставщик начинает подключаться ко многим другим поставщикам.

Как и в случае с CIDR, цель плана распределения адресов IPv6 заключается в предоставлении агрегации маршрутизационной информации для снижения нагрузки на внутридомашние маршрутизаторы. Ключевая идея снова заключается в использовании префикса адреса (набора смежных битов на наиболее значащем конце адреса) для агрегации информации о доступности для большого числа сетей и даже для большого числа автономных систем. Основной способ достижения этого заключается в назначении префикса адреса прямому поставщику, который затем назначает более длинные префиксы, начинающиеся с этого префикса, своим абонентам. Таким образом, поставщик может рекламировать один префикс для всех своих абонентов.

Конечно, недостаток заключается в том, что если сайт решает сменить поставщика, ему придется получить новый префикс адреса и перенумеровать все узлы на сайте. Это может быть колоссальной задачей, достаточной, чтобы отговорить большинство людей от смены поставщика. По этой причине продолжаются исследования других схем адресации, таких как географическая адресация, в которой адрес сайта зависит от его местоположения, а не от поставщика, к которому он подключается. В настоящее время, однако, адресация на основе поставщика необходима для эффективной работы маршрутизации.

Стоит отметить, что хотя назначение адресов IPv6 по существу эквивалентно тому, как происходило назначение адресов в IPv4 с момента введения CIDR, у IPv6 есть значительное преимущество — отсутствие большого установленного пула назначенных адресов, который нужно было бы учитывать в его планах.

Один из вопросов заключается в том, имеет ли смысл иерархическая агрегация на других уровнях иерархии. Например, должны ли все поставщики получать свои префиксы адресов в пределах префикса, выделенного для магистральной, к которой они подключены? Учитывая, что большинство поставщиков подключены к нескольким магистральям, это, вероятно, не имеет смысла. Кроме того, поскольку число поставщиков значительно меньше числа сайтов, преимущества агрегации на этом уровне наименее.

Одно место, где агрегация может иметь смысл — это национальный или континентальный уровень. Континентальные границы образуют естественные разделения в топологии Интернета. Если бы все адреса в Европе, например, имели общий префикс, то можно было бы выполнить значительную агрегацию, и большинству маршрутизаторов на других континентах понадобилась бы только одна запись в таблице маршрутизации для всех сетей с европейским префиксом. Поставщики в Европе выбрали бы свои префиксы таким образом, чтобы они начинались с европейского префикса. Используя эту схему, IPv6-адрес может выглядеть как на рис. 4.11. RegistryID может быть идентификатором, присвоенным европейскому регистратору адресов, с различными идентификаторами,

присвоенными другим континентам или странам. Обратите внимание, что префиксы в этом сценарии будут иметь разную длину. Например, поставщик с небольшим числом клиентов может иметь более длинный префикс (и, следовательно, меньше доступного адресного пространства), чем поставщик с большим числом клиентов.



Рисунок 4.11. Одноадресный адрес IPv6 на основе провайдера.

Одна сложная ситуация может возникнуть, когда абонент подключен к более чем одному поставщику. Какой префикс должен использовать абонент для своего сайта? Нет идеального решения этой проблемы. Например, предположим, что абонент подключен к двум поставщикам, X и Y. Если абонент берет свой префикс у X, то Y придется рекламировать префикс, который не имеет отношения к другим его абонентам и, следовательно, не может быть агрегирован. Если абонент нумерует часть своей автономной системы префиксом X и часть префиксом Y, он рискует тем, что половина его сайта станет недоступной, если соединение с одним из поставщиков выйдет из строя. Одно решение, которое работает достаточно хорошо, если у X и Y много общих абонентов, заключается в том, чтобы у них было три префикса между ними: один для абонентов только X, один для абонентов только Y и один для сайтов, которые являются абонентами как X, так и Y.

Глава 4.2.3. Формат пакета

Несмотря на то что IPv6 расширяет IPv4 несколькими способами, его формат заголовка на самом деле проще. Эта простота обусловлена целенаправленными усилиями по удалению ненужной функциональности из протокола. На рис. 4.12 показан результат.

Как и в случае со многими заголовками, этот начинается с поля Version, которое для IPv6 установлено на 6. Поле Version находится в том же месте относительно начала заголовка, что и поле Version в IPv4, так что программное обеспечение обработки заголовков может сразу определить, какой формат заголовка следует искать. Поля TrafficClass и FlowLabel оба относятся к вопросам качества обслуживания.

Поле PayloadLen указывает длину пакета, исключая заголовков IPv6, измеренную в байтах. Поле NextHeader заменяет как опции IP, так и поле Protocol в IPv4. Если требуются опции, то они передаются в одном или нескольких специальных заголовках, следующих за IP-заголовком, и это указывается значением поля NextHeader. Если специальных заголовков нет, поле NextHeader служит демукс-ключом, идентифицирующим протокол более высокого уровня, работающий поверх IP (например, TCP или UDP); то есть оно выполняет ту же функцию, что и поле Protocol в IPv4. Также фрагментация теперь обрабатывается как опциональный заголовок, а это означает, что поля, связанные с фрагментацией в IPv4, не включены в заголовок IPv6. Поле HopLimit — это просто TTL из IPv4, переименованное для отражения его фактического использования.

Наконец, большая часть заголовка занята исходным и конечным адресами, каждый из которых составляет 16 байт (128 бит). Таким образом, заголовок IPv6 всегда имеет длину 40 байт. Учтявая, что адреса IPv6 в четыре раза длиннее, чем адреса IPv4, это вполне сопоставимо с заголовком IPv4, который имеет длину 20 байт при отсутствии опций.

Способ, которым IPv6 обрабатывает опции, является значительным улучшением по сравнению с IPv4. В IPv4, если присутствовали какие-либо опции, каждый маршрутизатор должен был анализировать все поле опций, чтобы выяснить, какие из них имеют отношение к нему. Это происходило потому, что все опции были помещены в конце заголовка IP как неупорядоченная коллекция кортежей «(тип, длина, значение)». В отличие от этого IPv6 рассматривает опции как *заголовки расширения*, которые должны, если присутствуют, появляться в определенном порядке. Это означает, что каждый маршрутиза-

тор может быстро определить, какие из опций имеют отношение к нему; в большинстве случаев они не будут иметь отношения. Обычно это можно определить, просто взглянув на поле `NextHeader`. В конечном итоге обработка опций в IPv6 гораздо более эффективна, что является важным фактором для производительности маршрутизаторов.

Кроме того, новый формат опций в виде заголовков расширения означает, что они могут иметь произвольную длину, тогда как в IPv4 они были ограничены максимум 44 байтами. Далее мы увидим, как некоторые из опций используются. Каждая опция имеет свой собственный тип заголовка расширения. Тип каждого заголовка расширения идентифицируется значением поля `NextHeader` в заголовке, который предшествует ему, и каждый заголовок расширения содержит поле `NextHeader` для идентификации заголовка, следующего за ним. Последний заголовок расширения будет сопровождаться заголовком транспортного уровня (например, TCP), и в этом случае значение поля `NextHeader` совпадает со значением поля `Protocol` в заголовке IPv4. Таким образом, поле `NextHeader` выполняет двойную функцию; оно может либо идентифицировать тип следующего заголовка расширения, либо, в последнем заголовке расширения, служить демукс-ключом для идентификации протокола верхнего уровня, работающего поверх IPv6.

Рассмотрим пример заголовка фрагментации, показанный на рисунке 4.13. Этот заголовок обеспечивает функциональность, аналогичную полям фрагментации в заголовке IPv4, но он присутствует только в том случае, если фрагментация необходима. Если предположить, что это единственный присутствующий заголовок расширения, то поле `NextHeader` заголовка IPv6 будет содержать значение 44 — значение, присвоенное для обозначения заголовка фрагментации. Поле `NextHeader` заголовка фрагментации само по себе содержит значение, описывающее следующий за ним заголовок. Опять же, если предположить, что нет других заголовков расширения, то следующим заголовком может быть заголовок TCP, в результате чего `NextHeader` будет содержать значение 6, как и поле `Protocol` в IPv4.

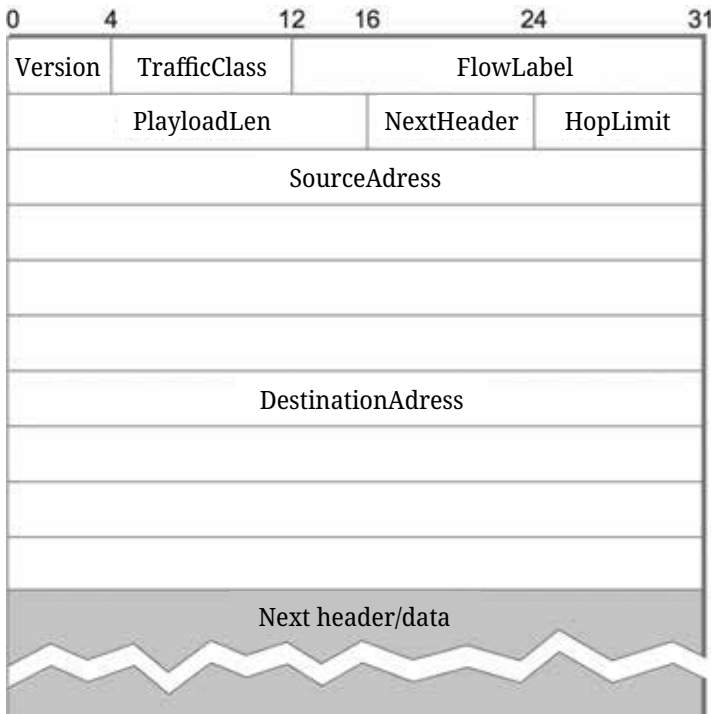


Рисунок 4.12. Заголовок пакета IPv6.

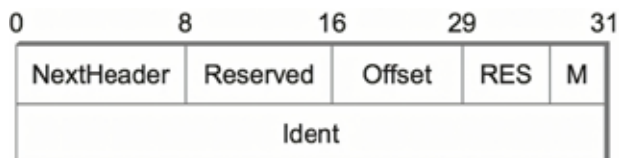


Рисунок 4.13. Заголовок расширения фрагментации IPv6.

Глава 4.2.4. Расширенные возможности

Как упоминалось в начале этого раздела, основным мотивом разработки IPv6 была поддержка Интернета, который продолжал расти. Однако как только заголовок IP пришлось изменить ради адресов, появилась возможность для широкого спектра других изменений, два из которых мы опишем ниже. IPv6 включает несколько дополнительных функций, большинство из которых освещены в других частях этой книги, например, мобильность, безопасность, качество обслуживания. Интересно отметить, что в большинстве этих областей возможности IPv4 и IPv6 стали практически неотличимыми, поэтому основной движущей силой для IPv6 остается потребность в больших адресах.

Автоконфигурация

Хотя рост Интернета впечатляет, одним из факторов, сдерживающих более быстрое принятие технологии, является тот факт, что подключение к Интернету обычно требовало значительных знаний в области системного администрирования. В частности, каждый хост, подключенный к Интернету, должен быть настроен с определенным минимальным объемом информации, такой как действительный IP-адрес, маска подсети для связи, к которой он подключается, и адрес сервера имен. Таким образом, невозможно было просто распаковать новый компьютер и подключить его к Интернету без предварительной настройки. Одной из целей IPv6 является обеспечение поддержки автоконфигурации, иногда называемой работой в режиме «*plug-and-play*».

Как мы видели в предыдущей главе, автоконфигурация возможна для IPv4, но она зависит от существования сервера, настроенного на выдачу адресов и другой конфигурационной информации клиентам по протоколу динамической конфигурации хоста (Dynamic Host Configuration Protocol, DHCP). Более длинный формат адреса в IPv6 помогает обеспечить полезную новую форму автоконфигурации, называемую *бесстатусной* автоконфигурацией, которая не требует сервера.

Помните, что уникальные адреса IPv6 иерархичны, и наименее значительная часть — это идентификатор интерфейса. Таким образом, мы можем разделить задачу автоконфигурации на две части:

1. Получить идентификатор интерфейса, уникальный для связи, к которой подключен хост.
2. Получить правильный префикс адреса для этой подсети.

Первая часть оказывается довольно простой, поскольку каждый хост на связи должен иметь уникальный адрес уровня связи. Например, все хосты в Ethernet имеют уникальный 48-битный Ethernet-адрес. Его можно превратить в действительный адрес локального использования канала связи, добавив соответствующий префикс из *numref* «Table %s <fig-v6tab> (1111 1110 10), за которым следует достаточное количество нулей, чтобы получилось 128 бит. Для некоторых устройств (например, принтеров или хостов в небольшой сети без маршрутизаторов, не подключающихся к другим сетям) этот адрес может быть вполне достаточным. Те устройства, которым нужен глобально действительный адрес, зависят от маршрутизатора на той же связи, который периодически объявляет соответствующий префикс для связи. Очевидно, что это требует настройки маршрутизатора с правильным префиксом адреса и выбора этого префикса таким обра-

зом, чтобы в конце оставалось достаточно места (например, 48 бит) для присоединения соответствующего адреса уровня связи.

Возможность встроить адреса уровня связи длиной до 48 бит в адреса IPv6 была одной из причин выбора такого большого размера адреса. 128 бит позволяют не только встроить эти адреса, но и оставляют достаточно места для многоуровневой иерархии адресации, которую мы обсуждали выше.

Маршрутизация, направляемая источником

Еще один из заголовков расширения IPv6 — это заголовок маршрутизации. При отсутствии этого заголовка маршрутизация для IPv6 мало чем отличается от маршрутизации IPv4 с использованием CIDR. Заголовок маршрутизации содержит список IPv6-адресов, представляющих узлы или топологические области, которые пакет должен посетить по пути к своему пункту назначения. Топологической областью может быть, например, сеть магистрального провайдера. Указание, что пакеты должны проходить через эту сеть, является способом реализации выбора провайдера на основе пакета. Таким образом, хост может указать, что он хочет, чтобы некоторые пакеты шли через дешевого провайдера, другие — через провайдера, обеспечивающего высокую надежность, а последние — через провайдера, которому хост доверяет в плане безопасности.

Чтобы обеспечить возможность указания топологических сущностей, а не отдельных узлов, IPv6 определяет адрес *anycast* (эникаст). Адрес *anycast* назначается набору интерфейсов, и пакеты, отправленные на этот адрес, будут доставлены к «ближайшему» из этих интерфейсов, при этом ближайший определяется маршрутизационными протоколами. Например, всем маршрутизаторам магистрального провайдера может быть присвоен единый адрес *anycast*, который будет использоваться в заголовке маршрутизации.

Глава 4.3. Многоадресная рассылка (multicast)

Сети с множественным доступом, такие как Ethernet, реализуют мультикаст на аппаратном уровне. Однако существуют приложения, которым требуется более широкая возможность мультикастинга, эффективная в масштабах Интернета. Например, когда радиостанция транслируется через Интернет, одни и те же данные должны быть отправлены на все хосты, где пользователь настроился на эту станцию. В данном примере коммуникация является одноадресной. Другие примеры одноадресных приложений включают передачу тех же новостей, текущих цен на акции, обновлений программного обеспечения или телевизионных каналов на несколько хостов. Последний пример обычно называется IPTV.

Существуют также приложения, коммуникация которых является многоадресной, такие как мультимедийные видеоконференции, онлайн-многопользовательские игры или распределенные симуляции. В таких случаях участники группы получают данные от нескольких отправителей, как правило, друг от друга. От любого конкретного отправителя они все получают одни и те же данные.

Обычная IP-коммуникация, при которой каждый пакет должен быть адресован и отправлен на один хост, не подходит для таких приложений. Если у приложения есть данные для отправки группе, ему придется отправить отдельный пакет с идентичными данными каждому члену группы. Эта избыточность потребляет больше пропускной способности, чем необходимо. Кроме того, избыточный трафик не распределяется равномерно, а концентрируется вокруг отправляющего хоста и может легко превысить пропускную способность отправляющего хоста и ближайших сетей и маршрутизаторов.

Чтобы лучше поддерживать многоадресную и одноадресную коммуникацию, IP предоставляет IP-уровень мультикастинга, аналогичный мультикасту канального уровня, предоставляемому сетями с множественным доступом, такими как Ethernet. Теперь, когда мы вводим понятие мультикаста для IP, нам также нужен термин для традиционного

одноадресного сервиса IP, который был описан до сих пор: этот сервис называется *одноадресной рассылкой* (unicast).

Базовая IP-модель многоадресной рассылки (мультикаста) — это модель многоадресного взаимодействия, основанная на *мультикаст-группах*, где каждая группа имеет свой собственный IP-мультикаст-адрес. Хосты, которые являются членами группы, получают копии любых пакетов, отправленных на мультикаст-адрес этой группы. Хост может состоять в нескольких группах и может свободно присоединяться и покидать группы, уведомляя свой локальный маршрутизатор с помощью протокола, который мы рассмотрим позже. Таким образом, если мы считаем, что уникаст-адреса связаны с узлом или интерфейсом, то мультикаст-адреса связаны с абстрактной группой, членство в которой динамически изменяется со временем. Более того, оригинальная модель мультикаста IP позволяет любому хосту отправлять мультикаст-трафик в группу; он не обязан быть членом группы, и может быть любое количество таких отправителей в данной группе.

Используя IP-мультикаст для отправки идентичного пакета каждому члену группы, хост отправляет единую копию пакета, адресованную на мультикаст-адрес группы. Отправляющему хосту не нужно знать индивидуальные одноадресные IP-адреса каждого члена группы, потому что, как мы увидим, эти знания распределены между маршрутизаторами в интернете. Точно так же отправляющему хосту не нужно отправлять несколько копий пакета, так как маршрутизаторы создадут копии всякий раз, когда им придется передать пакет по нескольким ссылкам. По сравнению с использованием одноадресного IP для доставки тех же пакетов многим получателям IP-мультикаст более масштабируем, так как он устраняет избыточный трафик (пакеты), которые были бы отправлены много раз по тем же самым ссылкам, особенно по тем, которые находятся рядом с отправляющим хостом.

Оригинальная модель мультикаста "от многих ко многим" в IP была дополнена поддержкой формы одноадресного мультикаста. В этой модели одноадресного мультикаста, называемой *Source-Specific Multicast* (SSM), принимающий хост указывает как мультикаст-группу, так и конкретный отправляющий хост. Принимающий хост будет получать мультикаст-адресацию только к указанной группе и только от указанного отправителя. Многие интернет-приложения для мультикаста (например, радиовещание) соответствуют модели SSM. В отличие от SSM, оригинальная модель мультикаста от многих ко многим в IP иногда называется *Any Source Multicast* (ASM).

Хост сигнализирует о своем желании присоединиться к мультикаст-группе или выйти из нее, общаясь со своим локальным маршрутизатором с помощью специального протокола для этой цели. В IPv4 этот протокол называется *Internet Group Management Protocol* (IGMP); в IPv6 — *Multicast Listener Discovery* (MLD). Маршрутизатор затем берет на себя ответственность за корректное поведение мультикаста по отношению к этому хосту. Поскольку хост может не выйти из мультикаст-группы, когда должен (например, после сбоя или другой ошибки), маршрутизатор периодически опрашивает сеть, чтобы определить, какие группы все еще интересны подключенным хостам.

Глава 4.3.1. Адреса многоадресной рассылки (мультикаста)

IP имеет поддиапазон своего адресного пространства, зарезервированный для мультикаст-адресов. В IPv4 эти адреса назначаются в адресном пространстве класса D, и IPv6 также имеет часть своего адресного пространства, зарезервированного для мультикаст-групп. Некоторые поддиапазоны мультикастовых диапазонов зарезервированы для внутридоменной многоадресной рассылки (мультикаста), чтобы их можно было повторно использовать независимо в различных доменах.

Таким образом, в IPv4 имеется 28 бит возможного мультикаст-адреса, если игнорировать префикс, общий для всех мультикаст-адресов. Это представляет собой проблему при попытке воспользоваться аппаратной многоадресной рассылкой в локальной сети

(LAN). Возьмем, к примеру, Ethernet. Ethernet-мультикаст-адреса имеют только 23 бита, если игнорировать их общий префикс. Другими словами, чтобы воспользоваться многоадресной рассылкой Ethernet, IP должен сопоставить 28-битные IP-мультикаст-адреса с 23-битными адресами многоадресной рассылки Ethernet. Это реализуется путем использования младших 23 бит любого IP-мультикаст-адреса в качестве его мультикаст-адреса Ethernet и игнорирования старших пяти бит. Таким образом, $32 (2^5)$ IP-адреса сопоставляются с каждым из Ethernet-адресов.

В этой главе мы используем Ethernet в качестве канонического примера сетевой технологии, поддерживающей мультикаст на аппаратном уровне, но то же самое верно и для PON (пассивные оптические сети), которые часто используются в качестве технологии доступа для предоставления оптоволокна в дом. Фактически IP-мультикаст поверх PON теперь является обычным способом доставки IPTV в дома.

Когда хост (или узел) в сети Ethernet присоединяется к группе IP многоадресной рассылки, он настраивает свой интерфейс Ethernet для приема любых пакетов с соответствующим многоадресным адресом Ethernet. К сожалению, это приводит к тому, что принимающий узел получает не только желаемый многоадресный трафик, но и трафик, отправленный на любую из другой 31 группы IP многоадресной рассылки, которые отображаются на тот же адрес Ethernet, если они маршрутизируются в эту сеть Ethernet. Поэтому IP на принимающем хосте должен проверить заголовок IP каждого многоадресного пакета, чтобы определить, действительно ли пакет принадлежит желаемой группе. В итоге несоответствие размеров многоадресных адресов означает, что многоадресный трафик может создавать нагрузку на хосты, которые даже не заинтересованы в группе, на которую был отправлен трафик. К счастью, в некоторых коммутируемых сетях (таких как коммутируемый Ethernet) эта проблема может быть смягчена с помощью схем, при которых коммутаторы распознают ненужные пакеты и отбрасывают их.

Одним из запутанных вопросов является то, как отправители и получатели узнают, какие многоадресные адреса использовать. Обычно это решается с помощью внешних средств, и существуют довольно сложные инструменты для обеспечения возможности рекламирования адресов групп в Интернете.

Глава 4.3.2. Многоадресная маршрутизация (DVMRP, PIM, MSDP)

Таблицы пересылки одноадресной рассылки маршрутизатора указывают для любого IP-адреса, какую ссылку использовать для пересылки одноадресного пакета. Для поддержки многоадресной рассылки маршрутизатор должен дополнительно иметь таблицы пересылки многоадресной рассылки, которые на основе многоадресного адреса указывают какие ссылки — возможно, более одной — использовать для пересылки многоадресного пакета (маршрутизатор дублирует пакет, если его нужно пересылать по нескольким ссылкам). Таким образом, там, где таблицы пересылки одноадресной рассылки коллективно указывают набор путей, таблицы пересылки многоадресной рассылки коллективно указывают набор деревьев: *деревья распределения многоадресной рассылки*. Более того, для поддержки Source-Specific Multicast (и, как оказывается, для некоторых типов Any Source Multicast) таблицы пересылки многоадресной рассылки должны указывать, какие ссылки использовать на основе комбинации многоадресного адреса и (одноадресного) IP-адреса источника, снова указывая набор деревьев.

Многоадресная маршрутизация — это процесс, с помощью которого определяют деревья распределения многоадресной рассылки или, если говорить более конкретно, процесс, с помощью которого строятся таблицы пересылки многоадресной рассылки. Как и в случае одноадресной маршрутизации, недостаточно, чтобы протокол многоадресной маршрутизации «работал»; он также должен достаточно хорошо масштабироваться по мере роста сети и учитывать автономию различных доменов маршрутизации.

DVMRP

Маршрутизация с вектором расстояния, используемая в одноадресной рассылке, может быть расширена для поддержки многоадресной рассылки. Получившийся протокол называется *протоколом маршрутизации многоадресной рассылки с вектором расстояния, или DVMRP* (Distance Vector Multicast Routing Protocol). DVMRP был первым протоколом многоадресной маршрутизации, который получил широкое распространение.

Напомним, что в алгоритме вектора расстояния каждый маршрутизатор поддерживает таблицу с кортежами Destination, Cost, NextHop и обменивается списком пар (Destination, Cost) с его непосредственно подключенными соседями. Расширение этого алгоритма для поддержки многоадресной рассылки — двухэтапный процесс. Сначала мы создаем механизм широковещания, который позволяет пересылать пакет на все сети в интернете. Во-вторых, нам нужно усовершенствовать этот механизм, чтобы он «подрезал» сети, в которых нет узлов, принадлежащих к многоадресной группе. Следовательно, DVMRP — это один из нескольких протоколов маршрутизации многоадресной рассылки, описываемых как протоколы типа «широковещание и подрезка» (flood-and-prune).

Имея таблицу маршрутизации одноадресной рассылки, каждый маршрутизатор знает, что текущий кратчайший путь к заданному пункту назначения проходит через NextHop. Таким образом, всякий раз, когда он получает многоадресный пакет от источника S, маршрутизатор пересылает пакет по всем исходящим ссылкам (за исключением той, по которой пакет прибыл) только в том случае, если пакет прибыл по ссылке, которая находится на кратчайшем пути к S (то есть пакет пришел от NextHop, связанного с S в таблице маршрутизации). Эта стратегия эффективно распространяет пакеты от S наружу, но не возвращает их обратно к S.

У этого подхода есть два основных недостатка. Первый заключается в том, что он действительно перенагружает сеть; у него нет механизма для избежания широковещания в локальных сетях (LAN), не имеющих участников в многоадресной группе. Мы рассмотрим эту проблему позже. Второе ограничение заключается в том, что данный пакет будет пересылаться по локальной сети каждым из маршрутизаторов, подключенных к данной сети. Это происходит из-за стратегии пересылки, при которой пакеты распространяются по всем ссылкам, кроме той, по которой пакет прибыл, независимо от того, являются ли эти ссылки частью дерева кратчайших путей, укорененного в источнике.

Решение второй проблемы заключается в устранении дублирующих широковещательных пакетов, которые генерируются, когда более одного маршрутизатора подключено к данной локальной сети. Один из способов сделать это — назначить один маршрутизатор в качестве *родительского* маршрутизатора для каждой ссылки по отношению к источнику, при этом только родительскому маршрутизатору разрешается пересылать многоадресные пакеты от этого источника по локальной сети. Маршрутизатор, имеющий кратчайший путь к источнику S, выбирается в качестве родительского; при равенстве двух маршрутизаторов выбирается маршрутизатор с наименьшим адресом. Данный маршрутизатор может узнать, является ли он родительским для локальной сети (снова по отношению к каждому возможному источнику), на основе сообщений с вектором расстояния, которыми он обменивается с соседями.

Обратите внимание, что это усовершенствование требует, чтобы каждый маршрутизатор сохранял для каждого источника бит для каждой из его инцидентных ссылок, указывающий, является ли он родительским для данной пары «источник/ссылка». Помните, что в межсетевом окружении источником является сеть, а не узел, так как межсетевой маршрутизатор заинтересован только в пересылке пакетов между сетями. Получившийся механизм иногда называют *обратным широковещанием по пути* (Reverse Path Broadcast, RPB) или *пересылкой по обратному пути* (Reverse Path Forwarding, RPF). Путь называется обратным, потому что мы рассматриваем кратчайший путь к *источнику* при принятии решений о пересылке, в отличие от одноадресной маршрутизации, которая ищет кратчайший путь к заданному пункту назначения.

Описанный механизм RPB реализует широковещание по кратчайшему пути. Теперь мы хотим сократить набор сетей, которые получают каждый пакет, адресованный группе G, исключив те, в которых нет узлов-участников группы G. Это можно сделать в два этапа. Во-первых, нам нужно определить, когда конечная сеть не имеет участников группы. Определить, что сеть является конечной, легко — если родительский маршрутизатор, как описано выше, является единственным маршрутизатором в сети, то сеть является конечной. Определить, есть ли участники группы в сети, можно, если каждый узел, являющийся участником группы G, периодически объявляет об этом по сети, как описано в нашем предыдущем описании многоадресной рассылки с отслеживанием состояния связи. Затем маршрутизатор использует эту информацию, чтобы решить, пересылать ли многоадресный пакет, адресованный группе G, по этой локальной сети.

Второй этап заключается в распространении информации «здесь нет участников группы G» вверх по дереву кратчайшего пути. Это делается так: маршрутизатор дополняет пары (Destination, Cost), которые он отправляет своим соседям, набором групп, для которых конечная сеть заинтересована в получении многоадресных пакетов. Эта информация затем может передаваться от маршрутизатора к маршрутизатору, так что для каждой из своих ссылок данный маршрутизатор знает, для каких групп он должен пересылать многоадресные пакеты.

Отметим, что включение всей этой информации в обновление маршрутизации является довольно дорогостоящей операцией, поэтому на практике эта информация обменивается только тогда, когда какой-то источник начинает отправлять пакеты этой группе. Другими словами, стратегия заключается в использовании RPB, который добавляет небольшое количество накладных расходов к базовому алгоритму вектора расстояния, до тех пор, пока конкретный многоадресный адрес не станет активным. В этот момент маршрутизаторы, которые не заинтересованы в получении пакетов, адресованных этой группе, заявляют об этом, и эта информация распространяется на другие маршрутизаторы.

PIM-SM

Протокол независимой многоадресной рассылки (Protocol Independent Multicast, PIM) был разработан в ответ на проблемы масштабируемости более ранних протоколов маршрутизации многоадресной рассылки. В частности, было признано, что существующие протоколы плохо масштабируются в средах, где относительно небольшая доля маршрутизаторов хочет получать трафик для определенной группы. Например, широковещание трафика всем маршрутизаторам до тех пор, пока они явно не попросят убрать их из распределения, не является хорошим решением, если большинство маршрутизаторов не хотят получать этот трафик в первую очередь. Эта ситуация достаточно распространена, поэтому PIM разделяет пространство проблемы на режимы *разреженной* и *плотной* рассылки, где разреженный и плотный режимы относятся к доле маршрутизаторов, которые будут желать получать многоадресную рассылку. Режим плотной рассылки PIM (PIM-DM) использует алгоритм широковещания и подрезки (flood-and-prune), подобный DVMRP, и страдает от той же проблемы масштабируемости. Режим разреженной рассылки PIM (PIM-SM) стал доминирующим протоколом маршрутизации многоадресной рассылки и является основным объектом нашего обсуждения в этой главе. Кстати, аспект «независимости протокола» в PIM означает, что, в отличие от более ранних протоколов, таких как DVMRP, PIM не зависит от какого-либо конкретного типа одноадресной маршрутизации — его можно использовать с любым протоколом одноадресной маршрутизации, как мы увидим ниже.

В PIM-SM маршрутизаторы явно присоединяются к дереву распределения многоадресной рассылки, используя сообщения протокола PIM, известные как сообщения Join. Обратите внимание на контраст с подходом DVMRP, который сначала создает де-

рево широковещания, а затем подрезает незаинтересованные маршрутизаторы. Возникает вопрос, куда отправлять эти сообщения Join, потому что, в конце концов, любой узел (и любое количество узлов) может отправлять сообщения в многоадресную группу. Для решения этой проблемы PIM-SM назначает для каждой группы специальный маршрутизатор, известный как *точка randevu* (rendezvous point, RP). В общем случае в домене настраивается несколько маршрутизаторов-кандидатов RP, и PIM-SM определяет набор процедур, с помощью которых все маршрутизаторы в домене могут согласовать, какой маршрутизатор использовать в качестве RP для данной группы. Эти процедуры довольно сложны, поскольку они должны учитывать широкий спектр сценариев, таких как отказ кандидата RP и разделение домена на две отдельные сети из-за ряда отказов ссылок или узлов. В оставшейся части этого обсуждения мы предполагаем, что все маршрутизаторы в домене знают одноадресный IP-адрес RP для данной группы.

Дерево многоадресной рассылки строится в результате отправки маршрутизаторами сообщений Join на RP. PIM-SM позволяет строить два типа деревьев: *общее дерево*, которое может использоваться всеми отправителями, и *дерево, специфичное для источника*, которое может использоваться только конкретным отправляющим узлом. Обычный режим работы заключается в создании сначала общего дерева, за которым следует одно или несколько деревьев, специфичных для источника, если для этого достаточно трафика. Поскольку построение деревьев устанавливает состояние в маршрутизаторах вдоль дерева, важно, чтобы по умолчанию было только одно дерево для группы, а не по одному для каждого отправителя в группу.

Когда маршрутизатор отправляет сообщение Join в сторону RP для группы G, оно отправляется с использованием обычной одноадресной IP-трансмиссии. Это показано на рис. 4.14(a), где маршрутизатор R4 отправляет сообщение Join к точке randevu для некоторой группы. Первоначальное сообщение Join «обобщенное» (wildcarded); то есть оно применяется ко всем отправителям. Сообщение Join, очевидно, должно пройти через определенную последовательность маршрутизаторов, прежде чем достигнет RP (например, R2). Каждый маршрутизатор на пути рассматривает сообщение Join и создает запись в таблице пересылки для общего дерева, называемую записью (*, G) (где * означает «все отправители»). Чтобы создать запись в таблице пересылки, он смотрит на интерфейс, по которому прибыло сообщение Join, и отмечает этот интерфейс как тот, по которому он должен пересылать данные пакеты для этой группы. Затем он определяет, какой интерфейс он будет использовать для пересылки сообщения Join в сторону RP. Это будет единственный допустимый интерфейс для входящих пакетов, отправленных в эту группу. Затем он пересылает сообщение Join в сторону RP. В конце концов сообщение достигает RP, завершая строительство ветви дерева. Общее дерево, построенное таким образом, показано сплошной линией от RP до R4 на рис. 4.14 (a).

По мере того как больше маршрутизаторов отправляют сообщения Join в сторону RP, они вызывают добавление новых ветвей к дереву, как показано на рис. 4.14(b). Обратите внимание, что в этом случае сообщение Join нужно отправить только до R2, который может добавить новую ветвь к дереву, просто добавив новый исходящий интерфейс к записи в таблице пересылки, созданной для этой группы. R2 не нужно пересылать сообщение Join дальше к RP. Также обратите внимание, что конечный результат этого процесса заключается в построении дерева, корнем которого является RP.

На этом этапе предположим, что узел хочет отправить сообщение группе. Для этого он создает пакет с соответствующим мультикаст-адресом группы в качестве пункта назначения и отправляет его маршрутизатору в своей локальной сети, известному как *назначенный маршрутизатор* (DR). Предположим, что DR — это R1 на рис. 4.14. В данный момент нет состояния для этой многоадресной группы между R1 и RP, поэтому вместо простой пересылки многоадресного пакета R1 *туннелирует* его к RP. То есть R1 инкапсулирует многоадресный пакет в сообщение PIM Register, которое он отправляет на одноадресный IP-адрес RP. Как и конечная точка IP-туннеля, RP

получает пакет, адресованный ему, просматривает содержимое сообщения Register и находит внутри IP-пакет, адресованный многоадресному адресу этой группы. RP, конечно же, знает, что делать с таким пакетом — он отправляет его по общему дереву, корнем которого является RP. В примере на рис. 4.14 это означает, что RP отправляет пакет на R2, который может пересылать его дальше на R4 и R5. Полная доставка пакета от R1 к R4 и R5 показана на рис. 4.15. Мы видим туннелированный пакет, который путешествует от R1 к RP с дополнительным IP-заголовком, содержащим одноадресный адрес RP, а затем многоадресный пакет, адресованный G, который движется по общему дереву к R4 и R5.

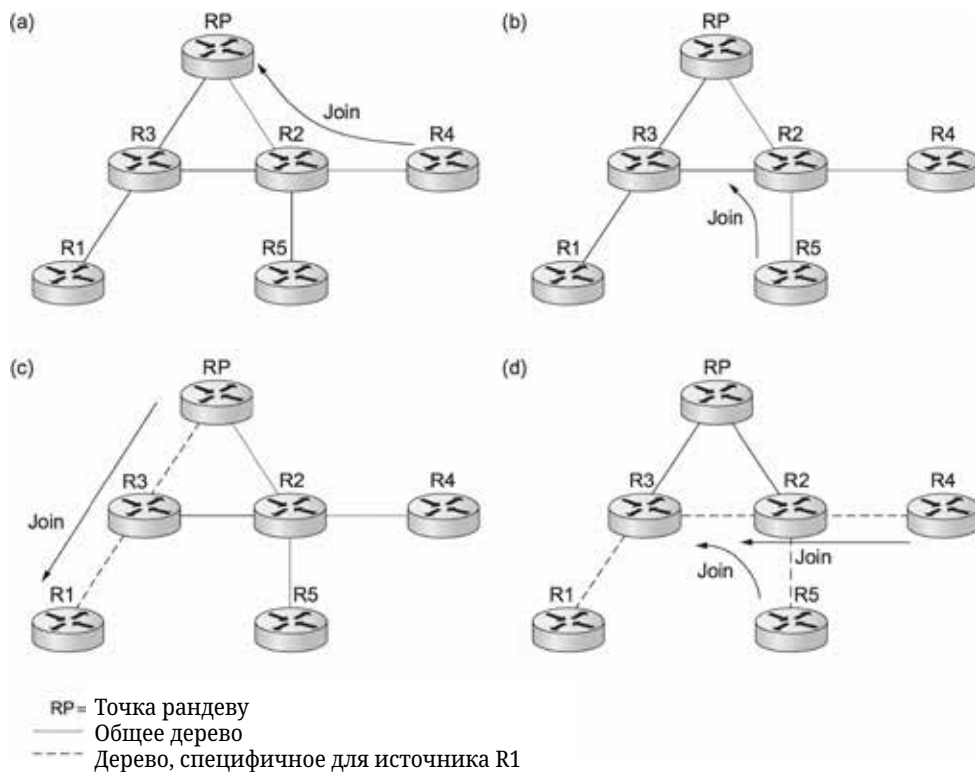


Рисунок 4.14. Работа PIM:

- (a) R4 посылает сообщение Join в RP и присоединяется к общему дереву;
- (b) R5 присоединяется к общему дереву;
- (c) RP строит дерево, специфичное для источника, для R1, посылая сообщение Join в R1;
- (d) R4 и R5 строят дерево, специфичное для источника, для R1, посылая сообщения Join в R1.

На этом этапе можно было бы предположить, что цель достигнута, так как все узлы могут отправлять пакеты всем получателям таким образом. Однако существуют некоторая неэффективность использования полосы пропускания и затраты на обработку при инкапсуляции и деккапсуляции пакетов по пути к RP, поэтому RP передает информацию об этой группе промежуточным маршрутизаторам, чтобы избежать туннелирования. Он отправляет сообщение Join в сторону отправляющего узла (рис. 4.14(c)). По мере того как это сообщение Join перемещается к узлу, оно заставляет маршрутизаторы на пути (R3) узнать о группе, так что DR сможет отправлять пакеты в группу в виде нативных (то есть не туннелированных) многоадресных пакетов.

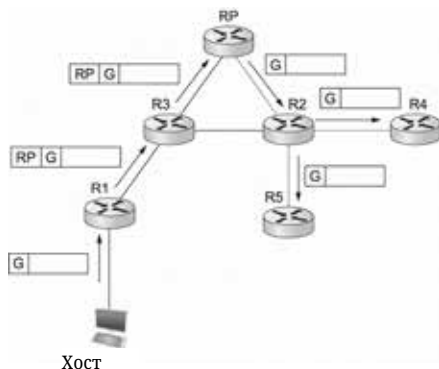


Рисунок 4.15. Доставка пакета по общему дереву. R1 туннелирует пакет к RP, который пересылает его по общему дереву к R4 и R5.

Важная деталь на данном этапе заключается в том, что сообщение Join, отправленное RP отправляющему узлу, специфично для этого отправителя, тогда как предыдущие, отправленные R4 и R5, применялись ко всем отправителям. Таким образом, эффект нового Join заключается в создании состояния, *специфичного для отправителя*, в маршрутизаторах между идентифицированным источником и RP. Это называется состоянием (S, G), так как оно применяется к одному отправителю и одной группе и контрастирует с состоянием (*, G), которое было установлено между получателями и RP и применяется ко всем отправителям. Таким образом, на рис. 4.14(с) мы видим маршрут, специфичный для источника, от R1 к RP (обозначен пунктирной линией), и дерево, действительное для всех отправителей от RP к получателям (обозначено сплошной линией).

Следующая возможная оптимизация заключается в замене всего общего дерева на дерево, специфичное для источника. Это желательно, потому что путь от отправителя к получателю через RP может быть значительно длиннее, чем кратчайший возможный путь. Опять же, вероятно, это будет вызвано высоким уровнем данных, наблюдаемым от некоторого отправителя. В этом случае маршрутизатор на конце дерева (скажем, R4 в нашем примере) отправляет сообщение Join, специфичное для источника, в сторону источника. Следуя кратчайшему пути к источнику, маршрутизаторы по пути создают состояние (S, G) для этого дерева, и результатом является дерево, корнем которого является источник, а не RP. Предположим, что и R4, и R5 переключились на дерево, специфичное для источника, тогда мы получим дерево, показанное на рис. 4.14(d). Обратите внимание, что это дерево больше не включает RP. Мы убрали общее дерево с этого рисунка для упрощения диаграммы, но на самом деле все маршрутизаторы с получателями для группы должны оставаться на общем дереве на случай появления новых отправителей.

Теперь мы можем понять, почему PIM является независимым от протоколов. Все его механизмы построения и поддержания деревьев используют одноадресную маршрутизацию, не завися при этом от какого-либо конкретного одноадресного маршрутизирующего протокола. Формирование деревьев полностью определяется маршрутами, по которым следуют сообщения Join, что определяется выбором кратчайших путей, сделанным одноадресной маршрутизацией. Таким образом, если быть точным, PIM является «одноадресным маршрутизирующим протокол-независимым», в отличие от DVMRP. Обратите внимание, что PIM очень сильно связан с Интернет-протоколом — он не является протокол-независимым в терминах сетевых протоколов.

Дизайн PIM-SM снова иллюстрирует трудности в построении масштабируемых сетей и то, как масштабируемость иногда противостоит некоторому виду оптимальности. Общее дерево, безусловно, более масштабируемо, чем дерево, специфичное для источника, в том смысле, что оно уменьшает общий объем состояния в маршрутизаторах до порядка числа групп, а не числа отправителей, умноженного на число групп. Однако дерево, специфичное для источника, вероятно, необходимо для достижения эффективной маршрутизации и эффективного использования полосы пропускания каналов.

Междоменная многоадресная рассылка (MSDP)

PIM-SM имеет некоторые значительные недостатки, когда дело доходит до междоменной многоадресной передачи. В частности, существование одного RP для группы противоречит принципу автономии доменов. Для данной многоадресной группы все участвующие домены зависели бы от домена, где расположен RP. Более того, если существуют определенная многоадресная группа, для которой отправитель и некоторые получатели находятся в одном домене, многоадресный трафик все равно должен был бы первоначально маршрутизироваться от отправителя к этим получателям через домен, в котором находится RP для этой многоадресной группы. Следовательно, протокол PIM-SM обычно не используется между доменами, а только внутри домена.

Для расширения многоадресной передачи через домены с использованием PIM-SM был разработан протокол обнаружения источников многоадресной рассылки (Multicast Source Discovery Protocol, MSDP). MSDP используется для подключения различных доменов — каждый из которых работает на PIM-SM внутри себя, с собственными RP — путем подключения RP различных доменов. Каждый RP имеет одного или нескольких MSDP-партнеров RP в других доменах. Каждая пара MSDP-партнеров соединена TCP-соединением, по которому работает протокол MSDP. Вместе все MSDP-партнеры для данной многоадресной группы образуют свободную сеть, используемую в качестве сети широковещания. Сообщения MSDP транслируются через сеть партнеров RP с использованием алгоритма Reverse Path Broadcast (трансляция обратного пути), который мы обсуждали в контексте DVMRP.

MSDP передает через сеть RPs информацию об источниках — отправителях многоадресной передачи, а не информацию о членстве в группе. Когда узел присоединяется к группе, эта информация распространяется только до RP его собственного домена. Вместо этого передается информация об источниках. Каждый RP знает источники в своем домене, так как он получает сообщение Register каждый раз, когда появляется новый источник. Каждый RP периодически использует MSDP для передачи сообщений Source Active своим партнерам, указывая IP-адрес источника, адрес многоадресной группы и IP-адрес исходного RP.

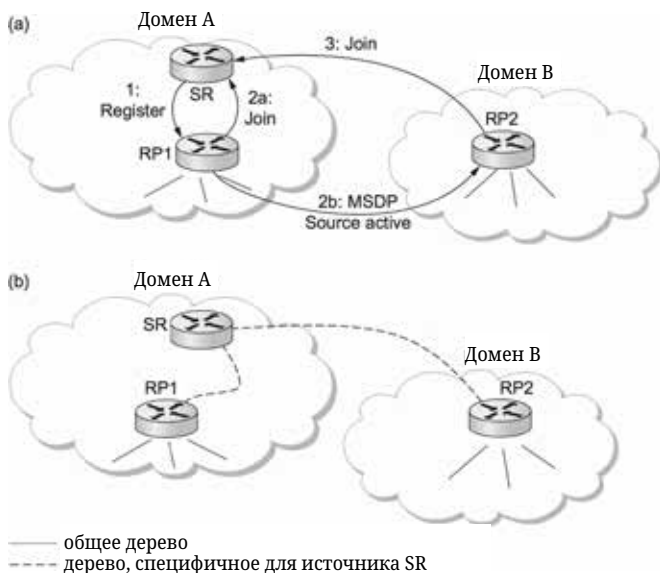


Рисунок 4.16. Операция MSDP: (а) Источник SR посылает сообщение Register на RP своего домена, RP1; затем RP1 посылает сообщение Join для SR и сообщение MSDP Source Active своему MSDP-аналогу в домене B, RP2; затем RP2 посылает сообщение Join для SR. (б) В результате RP1 и RP2 находятся в дереве, специфичном для источника SR.

Если RP-партнер MSDP, получивший одно из этих сообщений, имеет активных получателей для этой многоадресной группы, он отправляет запрос Join, специфичный для источника, от имени этого RP к узлу-источнику, как показано на рис. 4.16(a). Сообщение Join строит ветвь дерева, специфичного для источника, до этого RP, как показано на рис. 4.16 (b). В результате каждый RP, который является частью сети MSDP и имеет активных получателей для конкретной многоадресной группы, добавляется в дерево, специфичное для источника. Когда RP получает многоадресное сообщение от источника, он использует свое общее дерево для передачи многоадресного сообщения получателям в своем домене.

Многоадресная передача, специфичная для источника (PIM-SSM)

Изначальная модель сервиса PIM, как и предыдущие многоадресные протоколы, была моделью «многие ко многим». Получатели присоединялись к группе, и любой узел мог отправлять сообщения в эту группу. Однако в конце 1990-х годов было признано, что может быть полезно добавить модель «один ко многим». Многие приложения для многоадресной передачи имеют только одного законного отправителя, например, выступающего на конференции, транслируемой через Интернет. Мы уже видели, что PIM-SM может создавать деревья кратчайших путей, специфичные для источника, как оптимизацию после первоначального использования общего дерева. В исходном дизайне PIM эта оптимизация была невидима для узлов — только маршрутизаторы присоединялись к деревьям, специфичным для источника. Однако, как только была признана необходимость в модели сервиса «один ко многим», было решено создать возможность маршрутизации, специфичную для источника, явной для узлов. Оказалось, что для этого главным образом требовались изменения в IGMP и его аналоге для IPv6, MLD, а не в самом PIM. Новая явная возможность теперь известна как PIM-SSM (PIM Source-Specific Multicast).

PIM-SSM вводит новую концепцию — канал, который является комбинацией адреса источника S и адреса группы G. Адрес группы G выглядит как обычный IP-адрес для многоадресной передачи, и как IPv4, так и IPv6 выделили поддиапазоны в пространстве адресов для SSM. Чтобы использовать PIM-SSM, узел указывает как группу, так и источник в сообщении IGMP Membership Report своему локальному маршрутизатору. Этот маршрутизатор затем отправляет сообщение Join, специфичное для источника, по направлению к источнику, добавляя таким образом ветвь к себе в дерево, специфичное для источника, как было описано выше, для «обычного» PIM-SM, но обходя стадию общего дерева. Поскольку результатом является дерево, специфичное для источника, только указанный источник может отправлять пакеты в это дерево.

Введение PIM-SSM принесло значительные преимущества, особенно учитывая относительно высокий спрос на многоадресную передачу «один ко многим»:

- Многоадресные сообщения передаются более прямо к получателям.
- Адрес канала фактически является адресом многоадресной группы плюс адресом источника. Следовательно, учитывая, что определенный диапазон адресов многоадресных групп будет использоваться исключительно для SSM, несколько доменов могут использовать один и тот же адрес многоадресной группы независимо и без конфликтов, если они используют его только с источниками в своих собственных доменах.
- Поскольку только указанный источник может отправлять сообщения в группу SSM, риск атак со стороны вредоносных узлов, переполняющих маршрутизаторы или получателей поддельным многоадресным трафиком, уменьшается.
- PIM-SSM может использоваться между доменами точно так же, как и внутри домена, без зависимости от таких протоколов, как MSDP.

Таким образом, SSM является весьма полезным дополнением к модели многоадресной передачи.

Двунаправленные деревья (BIDIR-PIM)

Мы завершаем обсуждение многоадресной передачи другой модификацией PIM, известной как двунаправленный PIM (Bidirectional PIM, BIDIR-PIM). BIDIR-PIM — это недавний вариант PIM-SM, который хорошо подходит для многоточечной многоадресной передачи внутри домена, особенно когда отправители и получатели группы могут быть одними и теми же, как, например, в многопользовательской видеоконференции. Как и в PIM-SM, потенциальные получатели присоединяются к группам, отправляя сообщения IGMP Membership Report (которые не должны быть специфичными для источника), и для передачи многоадресных пакетов получателями используется общее дерево, корень которого находится в RP. Однако, в отличие от PIM-SM, общее дерево BIDIR-PIM также имеет ветви к источникам. Это не имело бы смысла в PIM-SM с его однонаправленным деревом, но деревья BIDIR-PIM двусторонние — маршрутизатор, получающий многоадресный пакет с нисходящей ветви, может пересылать его как вверх по дереву, так и вниз по другим ветвям. Маршрут, по которому доставляется пакет конкретному получателю, идет только вверх по дереву настолько, насколько это необходимо, прежде чем спуститься по ветви к этому получателю. См. пример маршрута многоадресной передачи от R1 к R2 на рис. 4.17(b). R4 пересылает многоадресный пакет вниз по дереву к R2 одновременно с тем, как он пересылает копию того же пакета вверх по дереву к R5.

Удивительный аспект BIDIR-PIM заключается в том, что на самом деле не обязательно иметь RP. Все, что нужно, — это маршрутизируемый адрес, который известен как RP-адрес, даже если это не обязательно адрес RP или вообще чего-либо. Как это возможно? Запрос Join от получателя пересылается к RP-адресу, пока не достигнет маршрутизатора с интерфейсом на связи, где будет находиться RP-адрес, где этот запрос и завершается. Рис. 4.17(a) показывает, как запрос от R2 завершается на R5, а запрос от R3 — на R6. Аналогично восходящая пересылка многоадресного пакета идет в направлении RP-адреса, пока не достигнет маршрутизатора с интерфейсом на связи, где будет находиться RP-адрес, но затем маршрутизатор пересылает многоадресный пакет на эту связь, завершая этап восходящей пересылки, обеспечивая получение пакета всеми другими маршрутизаторами на этой связи. Рис. 4.17(b) иллюстрирует поток многоадресного трафика, исходящего от R1.

Таким образом, BIDIR-PIM пока не может использоваться между доменами. С другой стороны, он имеет несколько преимуществ перед PIM-SM для многоточечной многоадресной передачи внутри домена:

- Нет процесса регистрации источников, так как маршрутизаторы уже знают, как маршрутизировать многоадресный пакет в направлении RP-адреса.
- Маршруты более прямые по сравнению с теми, которые используют общее дерево PIM-SM, так как они идут только вверх по дереву настолько, насколько это необходимо, а не до самого RP.
- Двусторонние деревья используют намного меньше состояния, чем деревья, специфичные для источников PIM-SM, потому что никогда не существует состояния, специфичного для источников. (С другой стороны, маршруты будут длиннее, чем у деревьев, специфичных для источников.)
- RP не может быть слабым местом, и фактически не нужен реальный RP.

Одним из выводов, которые можно сделать из того, что существует столько различных подходов к многоадресной передаче даже в рамках PIM, является то, что многоадресная передача представляет собой сложную область задач для нахождения оптимальных решений. Нужно решить, какие критерии вы хотите оптимизировать (использование полосы пропускания, состояние маршрутизаторов, длина пути и т. д.) и какое приложение вы пытаетесь поддержать ("один ко многим", "многие ко многим" и т. д.), прежде чем можно будет выбрать «лучший» режим многоадресной передачи для задачи.

Глава 4.4. Маршрутизация с множеством протокольных меток (MPLS)

Мы продолжаем обсуждение улучшений IP, описывая дополнение к архитектуре Интернета, которое очень широко используется, но в значительной степени скрыто от конечных пользователей. Это дополнение, называемое *маршрутизацией с множеством протокольных меток* (Multiprotocol Label Switching, MPLS), сочетает некоторые свойства виртуальных цепей с гибкостью и надежностью дейтаграмм. С одной стороны, MPLS тесно связано с архитектурой Интернета, основанной на дейтаграммах, — оно полагается на IP-адреса и протоколы маршрутизации IP для выполнения своей задачи. С другой стороны, маршрутизаторы с поддержкой MPLS также пересылают пакеты, рассматривая относительно короткие метки фиксированной длины, и эти метки имеют локальную область действия, как в сети с виртуальными цепями. Возможно, именно этот союз двух казалось бы противоположных технологий вызвал смешанную реакцию в сообществе интернет-инженеров.

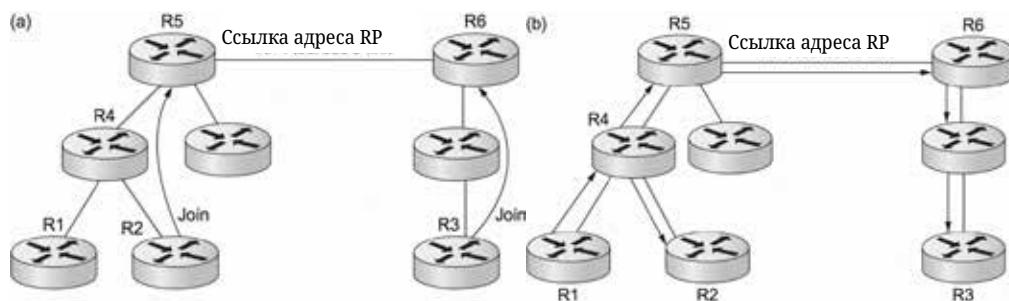


Рисунок 4.17. Работа BIDIR-PIM:

- (a) R2 и R3 посылают сообщения *Join* в направлении адреса RP, которые завершаются, когда достигают маршрутизатора на канале адреса RP.
- (b) Многоадресный пакет от R1 пересылается вверх по течению на канал RP-адреса и вниз по течению, где он пересекается с веткой члена группы.

Прежде чем рассмотреть, как работает MPLS, разумно спросить: «Для чего это нужно?» Хотя в отношении MPLS было сделано много заявлений, сегодня он используется в основном для трех задач:

- Чтобы обеспечить возможности IP на устройствах, которые не могут пересылать дейтаграммы IP обычным способом.
- Чтобы пересылать IP-пакеты по явным маршрутам — предрасчитанным маршрутам, которые не обязательно совпадают с теми, которые выбрали бы обычные протоколы маршрутизации IP.
- Чтобы поддерживать определенные типы услуг виртуальных частных сетей.

Стоит отметить, что одной из первоначальных целей — улучшение производительности — нет в списке. Это связано с достигнутыми в последние годы успехами в алгоритмах пересылки для маршрутизаторов IP и с комплексом факторов, выходящих за рамки обработки заголовков, которые определяют производительность.

Лучший способ понять, как работает MPLS — рассмотреть примеры его использования. В следующих трех главах мы рассмотрим примеры, чтобы проиллюстрировать три упомянутых выше приложения MPLS.

Глава 4.4.1. Пересылка на основе назначения

Одной из первых публикаций, представивших идею прикрепления меток к IP-пакетам, была статья Чандраниена и Варгезе, в которой описывалась идея, называемая *резьбовыми индексами*. Очень похожая идея сейчас реализована в маршрутизаторах с поддержкой MPLS. Следующий пример показывает, как работает эта идея.

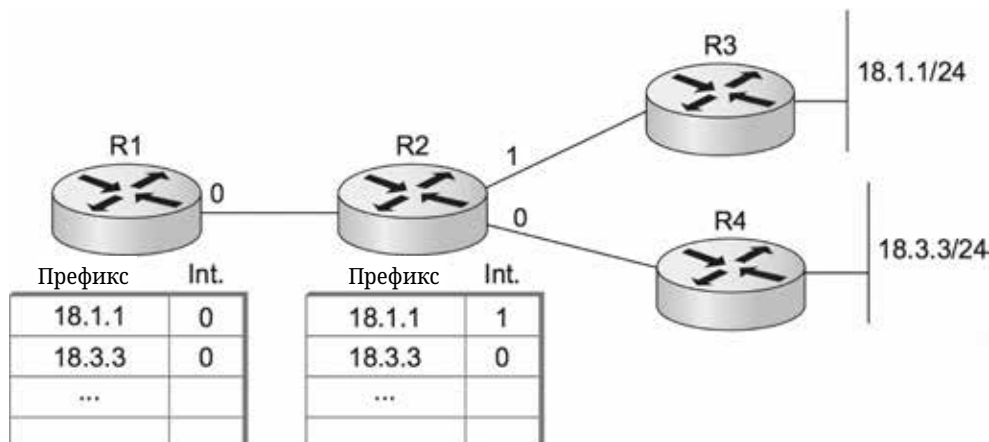


Рисунок 4.18. Таблицы маршрутизации в примере сети.

Рассмотрим сеть на рис. 4.18. Каждый из двух маршрутизаторов на крайнем правом (R3 и R4) имеет одну подключенную сеть с префиксами 18.1.1/24 и 18.3.3/24. Оставшиеся маршрутизаторы (R1 и R2) имеют таблицы маршрутизации, указывающие, какой выходной интерфейс каждый маршрутизатор будет использовать при пересылке пакетов в одну из этих двух сетей.

Когда на маршрутизаторе включен MPLS, маршрутизатор выделяет метку для каждого префикса в своей таблице маршрутизации и рекламирует как метку, так и префикс, который она представляет, своим соседним маршрутизаторам. Эта реклама передается через протокол распространения меток. Это показано на рис. 4.19. Маршрутизатор R2 выделил значение метки 15 для префикса 18.1.1 и значение метки 16 для префикса 18.3.3. Эти метки могут быть выбраны по усмотрению маршрутизатора, который их выделяет, и могут рассматриваться как индексы в таблице маршрутизации. После выделения меток R2 рекламирует привязки меток своим соседям; в данном случае мы видим, как R2 рекламирует привязку между меткой 15 и префиксом 18.1.1 для R1. Смысл такой рекламы заключается в том, что R2 как бы говорит: «Пожалуйста, прикрепите метку 15 ко всем пакетам, отправленным мне, которые предназначены для префикса 18.1.1.» R1 сохраняет метку в таблице рядом с префиксом, который она представляет, как удаленную или исходящую метку для любых пакетов, которые он отправляет на этот префикс.

На рис. 4.19(с) мы видим еще одну рекламу меток от маршрутизатора R3 к R2 для префикса 18.1.1, и R2 помещает удаленную метку, которую он узнал от R3, в соответствующее место в своей таблице.

На этом этапе мы можем рассмотреть, что происходит, когда пакет пересылается в этой сети. Допустим, пакет, предназначенный для IP-адреса 18.1.1.5, прибывает слева на маршрутизатор R1. В этом случае R1 называется *пограничным маршрутизатором меток* (Label Edge Router, LER); LER выполняет полную IP-проверку на входящих IP-пакетах.

тах, а затем применяет к ним метки в результате этой проверки. В этом случае R1 увидит, что 18.1.1.5 соответствует префиксу 18.1.1 в его таблице пересылки и что эта запись содержит как исходящий интерфейс, так и значение удаленной метки. Поэтому R1 прикрепляет удаленную метку 15 к пакету перед его отправкой.

Когда пакет прибывает на R2, R2 смотрит только на метку в пакете, а не на IP-адрес. Таблица пересылки на R2 указывает, что пакеты, прибывающие со значением метки 15, должны быть отправлены через интерфейс 1 и что они должны нести значение метки 24, как это было объявлено маршрутизатором R3. Поэтому R2 перезаписывает или меняет метку и пересылает пакет на R3.

Что было достигнуто всем этим применением и обменом меток? Обратите внимание, что когда R2 пересылал пакет в этом примере, ему никогда не нужно было фактически проверять IP-адрес. Вместо этого R2 смотрел только на входящую метку. Таким образом, мы заменили обычный поиск по IP-адресу назначения на поиск по метке. Чтобы понять, почему это важно, стоит вспомнить, что хотя IP-адреса всегда имеют одинаковую длину, IP-префиксы имеют переменную длину, и алгоритму поиска по IP-адресу назначения необходимо найти самое длинное совпадение — самый длинный префикс, который соответствует старшим битам IP-адреса пакета, который пересылается. Напротив, механизм пересылки по метке, описанный выше, является алгоритмом точного совпадения. Возможна реализация очень простого алгоритма *точного совпадения*, например, путем использования метки в качестве индекса в массив, где каждый элемент массива представляет собой одну строку в таблице пересылки.

Обратите внимание, что, несмотря на изменение алгоритма пересылки с самого длинного совпадения на точное совпадение, алгоритм маршрутизации может быть любым стандартным алгоритмом маршрутизации IP (например, OSPF). Путь, по которому будет следовать пакет в этой среде, точно такой же, каким бы он был, если бы MPLS не участвовал: путь, выбранный алгоритмами маршрутизации IP. Единственное, что изменилось, это алгоритм пересылки.

Важная фундаментальная концепция MPLS иллюстрируется этим примером. Каждая MPLS-метка ассоциируется с классом эквивалентности пересылки (forwarding equivalence class, FEC) — набором пакетов, которые должны получить одинаковую обработку пересылки в конкретном маршрутизаторе. В этом примере каждый префикс в таблице маршрутизации является FEC; то есть все пакеты, которые соответствуют префиксу 18.1.1 (независимо от младших битов IP-адреса), пересылаются по одному и тому же пути. Таким образом, каждый маршрутизатор может выделить одну метку, которая сопоставляется с 18.1.1, и любой пакет, содержащий IP-адрес, старшие биты которого совпадают с этим префиксом, может быть переслан с использованием этой метки.

Как мы увидим в последующих примерах, FEC является очень мощной и гибкой концепцией. FEC могут формироваться по любым критериям; например, все пакеты, соответствующие определенному клиенту, могут считаться входящими в один и тот же FEC.

Возвращаясь к рассматриваемому примеру, мы замечаем, что изменение алгоритма пересылки с обычной IP-пересылки на пересылку с заменой меток имеет важное следствие: устройства, которые ранее не умели пересылать IP-пакеты, могут быть использованы для пересылки IP-трафика в сети MPLS. Наиболее заметным ранним применением этого результата было использование ATM-коммутаторов, которые могут поддерживать MPLS без каких-либо изменений в их аппаратуре пересылки. ATM-коммутаторы поддерживают алгоритм пересылки с заменой меток, как описано выше, и, предоставив этим коммутаторам IP-маршрутизирующие протоколы и метод распределения привязок меток, их можно было бы превратить в *маршрутизаторы с коммутацией меток* (Label Switching Routers, LSR) — устройства, которые запускают IP-контрольные протоколы, но используют алгоритм пересылки с заменой меток. В последнее время та же идея была применена к оптическим коммутаторам.

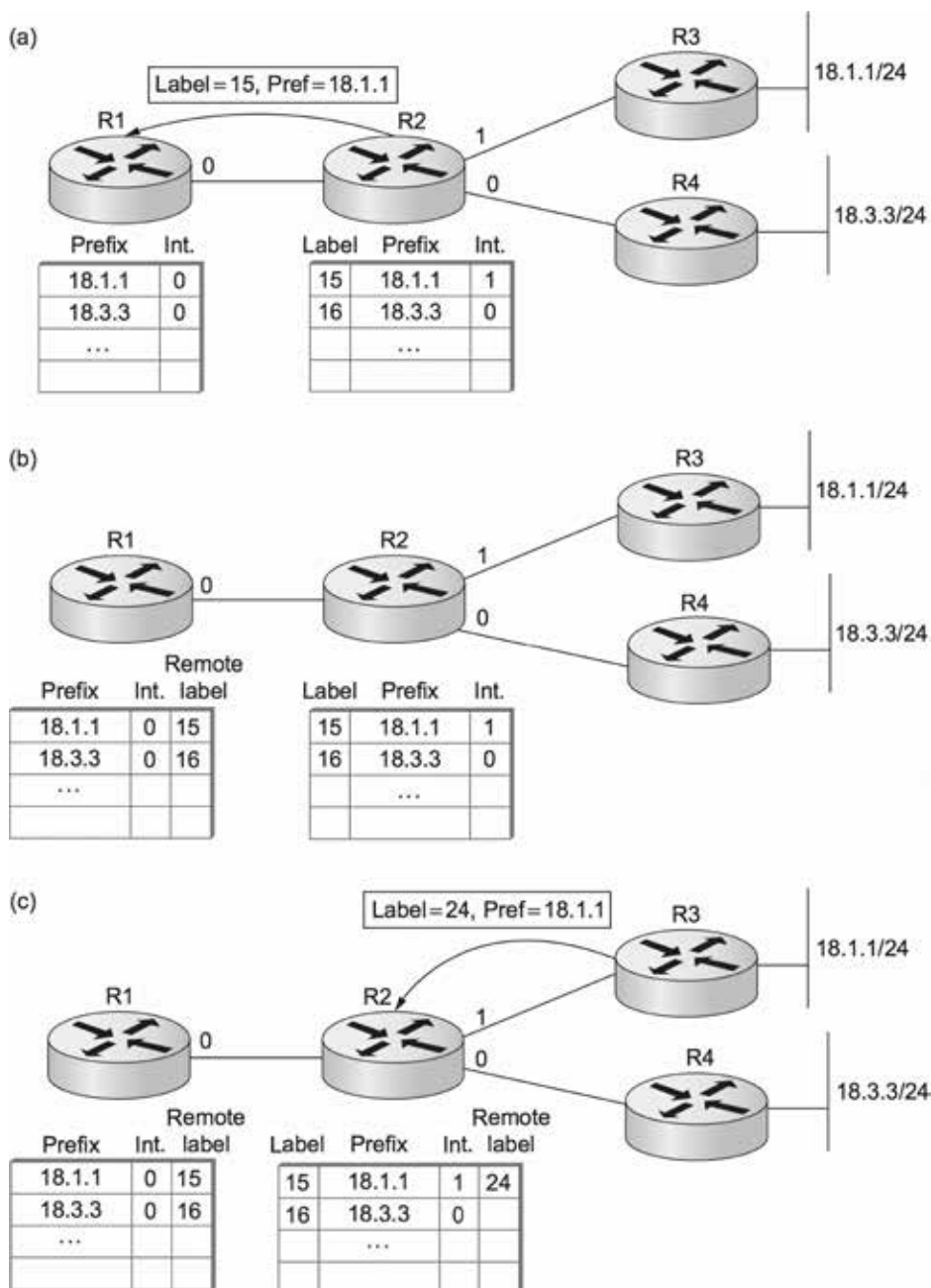


Рисунок 4.19. (a) R2 распределяет метки и рекламирует привязки для R1.

(b) R1 сохраняет полученные метки в таблице.

(c) R3 рекламирует другую привязку, и R2 сохраняет полученную метку в таблице.

Прежде чем рассматривать предполагаемые преимущества превращения АТМ-коммутатора в LSR, нам следует связать некоторые незаконченные моменты. Мы говорили, что метки «прикрепляются» к пакетам, но где именно они прикрепляются? Ответ зависит от типа канала, по которому передаются пакеты. Два распространенных мето-

да переноса меток на пакетах показаны на рис. 4.20. Когда IP-пакеты передаются в виде полных фреймов, как это происходит на большинстве типов каналов, включая Ethernet и PPP, метка вставляется в виде «шима» между заголовком уровня 2 и заголовком IP (или другим заголовком уровня 3), как показано в нижней части рисунка. Однако если ATM-коммутатор должен функционировать как LSR MPLS, то метка должна находиться в месте, где коммутатор сможет его использовать, а это значит, что она должна быть в заголовке ATM ячейки, точно там, где обычно находятся поля идентификатора виртуального канала (VCI) и идентификатора виртуального пути (VPI).

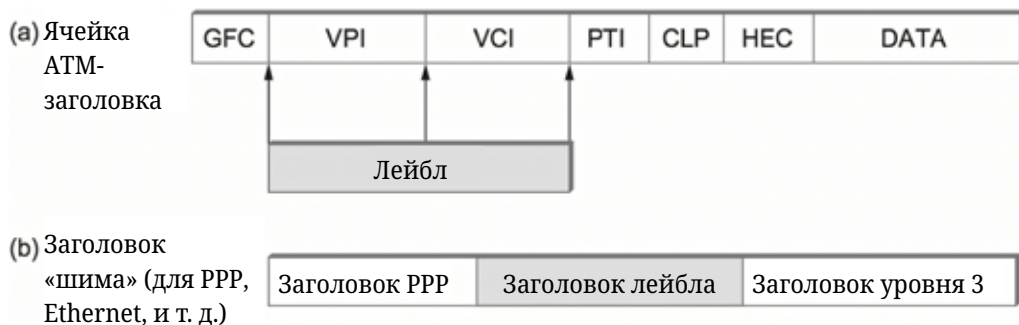


Рисунок 4.20. (a) метка (или лейбл) на ATM-инкапсулированном пакете;
(b) метка на пакете, инкапсулированном в кадр.

Теперь, когда мы разработали схему, по которой ATM-коммутатор может функционировать как LSR, что мы получили? Следует отметить, что теперь мы можем построить сеть, использующую смесь обычных IP-маршрутизаторов, пограничных маршрутизаторов с метками и ATM-коммутаторами, функционирующих как LSR, и они все будут использовать одни и те же маршрутизирующие протоколы. Чтобы понять преимущества использования одних и тех же протоколов, рассмотрим альтернативу. На рис. 4.21(a) мы видим набор маршрутизаторов, соединенных виртуальными каналами через ATM-сеть, конфигурацию, называемую *наложенной* (оверлейной) сетью. В определенный момент времени сети такого типа часто строились, потому что коммерчески доступные ATM-коммутаторы поддерживали большую общую пропускную способность, чем маршрутизаторы. Сегодня такие сети менее распространены, потому что маршрутизаторы догнали и даже превзошли ATM-коммутаторы. Однако эти сети все еще существуют из-за значительной установленной базы ATM-коммутаторов в магистральных сетях, что частично является результатом способности ATM поддерживать ряд возможностей, таких как эмуляция каналов и услуги виртуальных каналов.

В наложенной (оверлейной) сети каждый маршрутизатор потенциально может быть соединен с каждым другим маршрутизатором виртуальным каналом, но в этом случае для ясности мы показали только каналы от R1 до всех его соседних маршрутизаторов. R1 имеет пять маршрутизирующих соседей и должен обмениваться сообщениями маршрутизирующих протоколов со всеми ними — мы говорим, что у R1 пять маршрутизирующих смежностей. Напротив, на рис. 4.21(b) ATM-коммутаторы заменены на LSR. Больше нет виртуальных каналов, соединяющих маршрутизаторы. Таким образом, у R1 есть только одна смежность, с LSR1. В крупных сетях запуск MPLS на коммутаторах приводит к значительному снижению количества смежностей, которые каждый маршрутизатор должен поддерживать, и может значительно уменьшить объем работы, которую маршрутизаторы должны выполнять, чтобы держать друг друга в курсе изменений топологии.

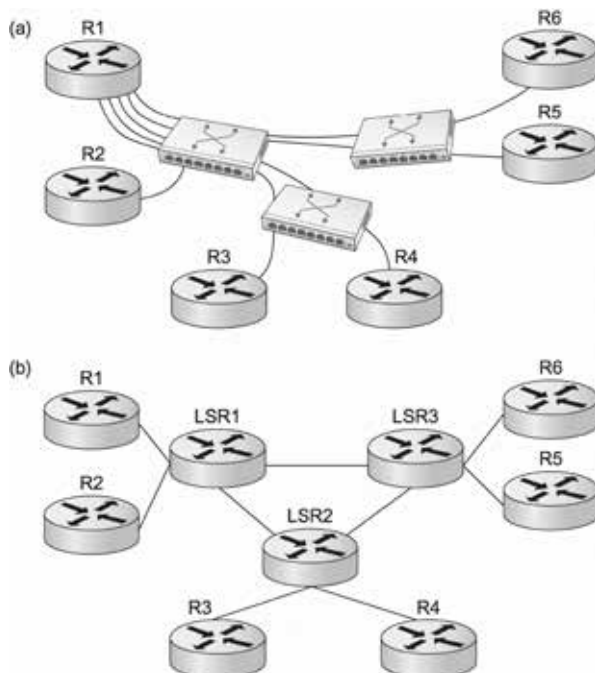


Рисунок 4.21. (а) Маршрутизаторы соединяются друг с другом с помощью наложения виртуальных цепей. (б) Маршрутизаторы напрямую взаимодействуют с LSR.

Вторым преимуществом использования одних и тех же маршрутизационных протоколов на крайних маршрутизаторах и на LSR является то, что крайние маршрутизаторы теперь имеют полное представление о топологии сети. Это означает, что в случае отказа какого-либо канала или узла внутри сети крайние маршрутизаторы будут иметь больше шансов выбрать хороший новый путь, чем если бы ATM-коммутаторы перенаправили затронутые виртуальные каналы (VC) без ведома крайних маршрутизаторов.

Заметьте, что шаг «замены» ATM-коммутаторов на LSR фактически достигается путем изменения протоколов, работающих на коммутаторах, но обычно не требуется изменения оборудования пересылки; то есть ATM-коммутатор часто можно преобразовать в MPLS LSR, обновив только его программное обеспечение. Более того, MPLS LSR может продолжать поддерживать стандартные возможности ATM одновременно с запуском MPLS контрольных протоколов, в режиме, известном как *ships in the night* (корабли в ночи).

Идея запуска IP-контрольных протоколов на устройствах, которые не могут нативно пересылать IP-пакеты, была расширена до сетей мультиплексирования с разделением по длине волны (Wavelength Division Multiplexing, WDM) и сетей мультиплексирования с временным разделением каналов (Time Division Multiplexing, TDM) (например, SONET). Это известно как *обобщенный MPLS* (Generalized MPLS, GMPLS). Часть причины развития GMPLS заключалась в предоставлении маршрутизаторам топологической информации об оптической сети, так же как в случае с ATM. Еще более важным было то, что не существовали стандартные протоколы для управления оптическими устройствами, поэтому MPLS оказался единственным разумным решением для этой задачи.

Глава 4.4.2. Явная маршрутизация

IP имеет опцию маршрутизации по источнику, но она не используется широко по нескольким причинам, включая тот факт, что можно указать только ограниченное количество переходов, и потому что она обычно обрабатывается вне «быстрого пути» на большинстве маршрутизаторов.

MPLS предоставляет удобный способ добавления возможностей, подобных маршрутизации по источнику, в IP-сети, хотя эта возможность чаще называется *явной маршрутизацией*, а не *маршрутизацией по источнику*. Одна из причин различия заключается в том, что обычно не реальный источник пакета выбирает маршрут. Чаще это один из маршрутизаторов внутри сети провайдера услуг. На рис. 4.22 показан пример того, как может быть применена возможность явной маршрутизации MPLS. Такая сеть часто называется *рыбообразной сетью* из-за своей формы (маршрутизаторы R1 и R2 образуют хвост; R7 находится в голове).

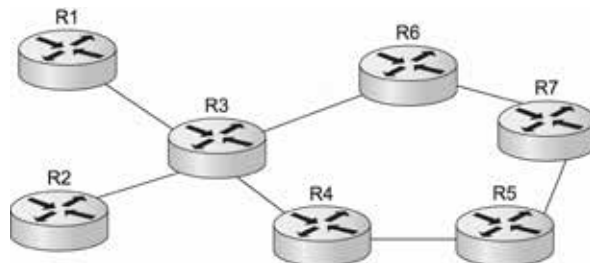


Рисунок 4.22. Сеть, требующая явной маршрутизации.

Предположим, что оператор сети на рис. 4.22 определил, что любой трафик, текущий от R1 к R7, должен следовать по пути R1-R3-R6-R7, а любой трафик, идущий от R2 к R7, должен следовать по пути R2-R3-R4-R5-R7. Одной из причин такого выбора было бы эффективное использование доступной пропускной способности по двум различным путям от R3 к R7. Мы можем рассматривать трафик от R1 к R7 как один класс эквивалентности пересылки (forwarding equivalence class, FEC), а трафик от R2 к R7 как второй FEC. Пересылка трафика в этих двух классах по разным путям затруднена при нормальной IP-маршрутизации, потому что R3 обычно не учитывает, откуда пришел трафик, при принятии решений о пересылке.

Поскольку MPLS использует подстановку меток для пересылки пакетов, достаточно легко достичь желаемой маршрутизации, если маршрутизаторы поддерживают MPLS. Если R1 и R2 прикрепляют к пакетам различные метки перед отправкой их к R3 (таким образом идентифицируя их как принадлежащие к различным FEC), то R3 может пересылать пакеты от R1 и R2 по разным путям. Вопрос, который возникает, заключается в том, как все маршрутизаторы в сети договариваются о том, какие метки использовать и как пересылать пакеты с определенными метками? Очевидно, что мы не можем использовать те же процедуры, что описаны в предыдущей главе, для распределения меток, потому что эти процедуры устанавливают метки, которые заставляют пакеты следовать по обычным путям, выбранным IP-маршрутизацией, чего мы как раз пытаемся избежать. Вместо этого необходим новый механизм. Оказывается, что протокол, используемый для этой задачи, — это *протокол резервирования ресурсов* (Resource Reservation Protocol, RSVP). Пока достаточно сказать, что можно отправить сообщение RSVP по явно указанному пути (например, R1-R3-R6-R7) и использовать его для настройки записей таблицы пересылки меток вдоль всего этого пути. Это очень похоже на процесс установления виртуального канала.

Одним из применений явной маршрутизации является *управление трафиком*, что подразумевает задачу обеспечения достаточных ресурсов в сети для удовлетворения предъявляемых к ней требований. Контроль над тем, по каким именно путям движется трафик, является важной частью управления трафиком. Явная маршрутизация также может помочь сделать сети более устойчивыми к сбоям, используя возможность *быстрой перенастройки* (fast reroute). Например, можно заранее вычислить путь от маршрутизатора А к маршрутизатору В, который явно избегает определенного канала L. В случае отказа канала L маршрутизатор А может отправлять весь трафик, предназначенный для В, по за-

ранее рассчитанному пути. Сочетание предварительного расчета резервного пути и явной маршрутизации пакетов по этому пути означает, что А не нужно ждать, пока пакеты маршрутизационного протокола пройдут через сеть, или пока маршрутизационные алгоритмы будут выполнены различными другими узлами в сети. В определенных обстоятельствах это может значительно сократить время, необходимое для перенаправления пакетов вокруг точки отказа.

Последний момент, который стоит отметить о явной маршрутизации, заключается в том, что явные маршруты не обязательно должны рассчитываться оператором сети, как в приведенном выше примере. Маршрутизаторы могут использовать различные алгоритмы для автоматического расчета явных маршрутов. Наиболее распространенным из них является алгоритм *кратчайшего пути с ограничениями* (constrained shortest path first, CSPF), который является алгоритмом состояния канала, но также учитывает различные ограничения. Например, если требуется найти путь от R1 к R7, который может выдержать предлагаемую нагрузку в 100 Мбит/с, мы могли бы сказать, что ограничение заключается в том, что каждый канал должен иметь по крайней мере 100 Мбит/с доступной пропускной способности. CSPF решает такую задачу.

Глава 4.4.3. Виртуальные частные сети и туннели

Один из способов построения виртуальных частных сетей (VPN) — использование туннелей. Оказывается, MPLS можно рассматривать как способ создания туннелей, что делает его подходящим для создания VPN различных типов.

Самая простая для понимания форма MPLS VPN — это VPN уровня 2. В этом типе VPN MPLS используется для туннелирования данных уровня 2 (например, Ethernet-кадры или ячейки ATM) через сеть маршрутизаторов с поддержкой MPLS. Одной из причин использования туннелей является предоставление какого-либо сетевого сервиса (например, многоадресной рассылки), который не поддерживается некоторыми маршрутизаторами в сети. Та же логика применима и здесь: IP-маршрутизаторы не являются ATM-коммутаторами, поэтому вы не можете предоставить услугу виртуального канала ATM через сеть обычных маршрутизаторов. Однако если бы у вас была пара маршрутизаторов, соединенных туннелем, они могли бы отправлять ячейки ATM через туннель и эмулировать виртуальный канал ATM. В терминологии IETF эта техника называется *эмуляцией псевдопроводов* (pseudowire emulation). Рис. 4.23 иллюстрирует эту идею.

Мы уже видели, как строятся IP-туннели: маршрутизатор на входе в туннель оборачивает данные, подлежащие туннелированию, в IP-заголовок (*заголовок туннеля*), который представляет собой адрес маршрутизатора на дальнем конце туннеля и отправляет данные как обычный IP-пакет. Принимающий маршрутизатор получает пакет с его собственным адресом в заголовке, снимает заголовок туннеля и находит данные, которые были туннелированы, после чего обрабатывает их. Что именно он делает с этими данными, зависит от их типа. Например, если это был другой IP-пакет, он будет переслан как обычный IP-пакет. Однако это не обязательно должен быть IP-пакет, если принимающий маршрутизатор знает, что делать с не-IP-пакетами. Мы вернемся к вопросу обработки не-IP-данных чуть позже.

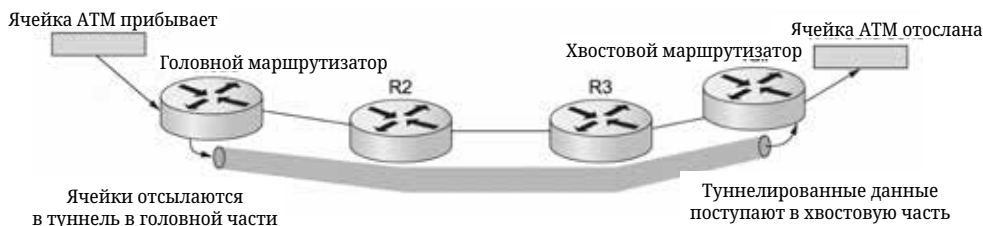


Рисунок 4.23. Схема ATM эмулируется туннелем.

Туннель MPLS мало чем отличается от IP-туннеля, за исключением того, что заголовок туннеля состоит из MPLS-заголовка, а не IP-заголовка. Возвращаясь к нашему первому примеру на рисунке 4.19, мы видели, что маршрутизатор R1 прикреплял метку (15) к каждому пакету, который он отправлял в сторону префикса 18.1.1. Такой пакет затем следовал бы по пути R1-R2-R3, при этом каждый маршрутизатор на пути исследовал бы только MPLS-метку. Таким образом, мы видим, что не было требования, чтобы R1 отправлял по этому пути только IP-пакеты — любые данные могли быть обернуты в MPLS-заголовок, и они следовали бы тем же путем, поскольку промежуточные маршрутизаторы никогда не смотрят дальше MPLS-заголовка. В этом отношении MPLS-заголовок подобен заголовку IP-туннеля (только длиной всего 4 байта вместо 20 байт). Единственный вопрос с отправкой не-IP-трафика через туннель, будь то MPLS или иной, заключается в том, что делать с не-IP-трафиком, когда он достигает конца туннеля. Общее решение заключается в том, чтобы нести в полезной нагрузке туннеля какой-то идентификатор демultipлексирования, который говорит маршрутизатору на конце туннеля, что делать. Оказывается, что метка MPLS идеально подходит для такой роли. Пример прояснит это.

Предположим, что мы хотим туннелировать ATM-ячейки от одного маршрутизатора к другому через сеть маршрутизаторов с поддержкой MPLS, как показано на рис. 4.23. Кроме того, предположим, что цель — эмулировать виртуальный канал ATM; то есть ячейки прибывают на вход в туннель на определенном входном порту с определенным VCI и должны покидать выход туннеля на определенном выходном порту с потенциально другим VCI. Это можно выполнить, настроив головной и конечный маршрутизаторы следующим образом:

- Головной маршрутизатор должен быть настроен с входным портом, входным VCI, демultiplexирующей меткой для этого эмулируемого канала и адресом маршрутизатора на конце туннеля.
- Конечный маршрутизатор должен быть настроен с выходным портом, выходным VCI и демultiplexирующей меткой.

После того как маршрутизаторы получают эту информацию, мы сможем увидеть, как будет пересылаться ячейка ATM. Рисунок 4.24 иллюстрирует эти шаги.

1. ATM-ячейка поступает на указанный входной порт с соответствующим значением VCI (в данном примере 101).
2. Головной маршрутизатор прикрепляет демultiplexирующую метку, которая идентифицирует эмулируемый канал.
3. Затем головной маршрутизатор прикрепляет вторую метку, которая является меткой туннеля и будет направлять пакет к конечному маршрутизатору. Эта метка определяется с помощью механизмов, описанных ранее в этой главе.
4. Маршрутизаторы между началом и концом туннеля пересылают пакет, используя только метку туннеля.
5. Конечный маршрутизатор снимает метку туннеля, находит демultiplexирующую метку и распознает эмулируемый канал.
6. Конечный маршрутизатор изменяет значение VCI ATM на корректное значение (в данном случае 202) и отправляет его на правильный порт.



Рисунок 4.24. Передача ячеек ATM по туннелю.

Один элемент в этом примере, который может оказаться неожиданным, — это то, что к пакету прикреплены две метки. Это одна из интересных особенностей MPLS — метки могут быть наложены на пакет в любом количестве. Это обеспечивает полезные возможности масштабирования. В этом примере это позволяет одному туннелю нести потенциально большое количество эмулируемых каналов.

Те же самые техники, описанные здесь, могут быть применены для эмуляции многих других сервисов уровня 2, включая Frame Relay и Ethernet. Стоит отметить, что практически идентичные возможности могут быть предоставлены с использованием IP-туннелей; основное преимущество MPLS здесь — более короткий заголовок туннеля.

До того как MPLS использовался для туннелирования сервисов уровня 2, он также применялся для поддержки VPN уровня 3. Мы не будем вдаваться в подробности VPN уровня 3, которые достаточно сложны, но отметим, что они представляют собой одно из самых популярных применений MPLS сегодня. VPN уровня 3 также используют стеки меток MPLS для туннелирования пакетов через IP-сеть. Однако пакеты, которые туннелируются, сами являются IP-пакетами — отсюда и название VPN уровня 3. В VPN уровня 3 один поставщик услуг управляет сетью маршрутизаторов с поддержкой MPLS и предоставляет виртуально частную IP-сеть услуг для любого числа различных клиентов. То есть каждый клиент поставщика услуг имеет несколько сайтов, и поставщик услуг создает для каждого клиента иллюзию того, что в сети нет других клиентов. Клиент видит IP-сеть, соединяющую только его собственные сайты и никакие другие сайты. Это означает, что каждый клиент изолирован от всех других клиентов в плане маршрутизации и адресации. Клиент А не может отправлять пакеты напрямую клиенту В и наоборот. Клиент А может даже использовать IP-адреса, которые также использует клиент В. Основная идея показана на рис. 4.25. Как и в VPN уровня 2, MPLS используется для туннелирования пакетов от одного сайта к другому; однако настройка туннелей выполняется автоматически с помощью довольно сложного использования BGP, что выходит за рамки этой книги.

На самом деле клиент А обычно может отправлять данные клиенту В некоторым ограниченным образом. Скорее всего, и клиент А, и клиент В имеют какое-то соединение с глобальным Интернетом, и, таким образом, вероятно, клиент А может отправлять электронные письма, например, на почтовый сервер внутри сети клиента В. «Конфиденциальность», предлагаемая VPN, предотвращает клиенту А неограниченный доступ ко всем машинам и подсетям внутри сети клиента В.

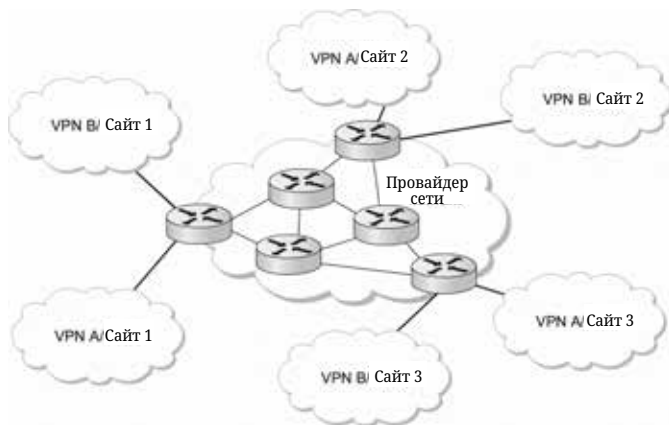


Рисунок 4.25. Пример VPN уровня 3. Клиенты А и В получают виртуально частную IP-услугу от одного провайдера.

В заключение отметим, что MPLS — это достаточно универсальный инструмент, который был применен к широкому спектру различных сетевых задач. Он сочетает в себе механизм пересылки меток, который обычно ассоциируется с сетями виртуальных кана-

лов, с маршрутными и управляющими протоколами IP-сетей дейтаграмм, создавая класс сети, который находится между двумя традиционными крайностями. Это расширяет возможности IP-сетей, позволяя, среди прочего, более точно контролировать маршрутизацию и поддерживать различные сервисы VPN.

Глава 4.5. Маршрутизация среди мобильных устройств

Вероятно, для читателя не станет большим сюрпризом тот факт, что мобильные устройства представляют некоторые вызовы для интернет-архитектуры. Интернет был разработан в эпоху, когда компьютеры были большими неподвижными устройствами, и хотя разработчики Интернета, вероятно, предполагали, что мобильные устройства могут появиться в будущем, справедливо считать, что это не было их главным приоритетом. Сегодня, конечно, мобильные компьютеры повсюду, особенно в виде ноутбуков и смартфонов, и все чаще в других формах, таких как дроны. В этой главе мы рассмотрим некоторые из проблем, вызванных появлением мобильных устройств, и некоторые текущие подходы к их решению.

Глава 4.5.1. Проблемы мобильных сетей

Сегодня достаточно просто оказаться в зоне действия беспроводной точки доступа, подключиться к Интернету, используя 802.11 или какой-либо другой протокол беспроводной сети, и получить довольно хороший интернет-сервис. Одной из ключевых технологий, которая сделала возможной точки доступа, является DHCP. Вы можете удобно устроиться в кафе, открыть свой ноутбук, получить IP-адрес для вашего ноутбука и подключиться к маршрутизатору по умолчанию и серверу доменных имен (DNS), и для большого количества сетевых приложений у вас есть все необходимое.

Однако, если мы посмотрим немного ближе, становится ясно, что для некоторых сценариев приложений простое получение нового IP-адреса каждый раз, когда вы перемещаетесь (то, что делает для вас DHCP), не всегда достаточно. Предположим, вы используете ноутбук или смартфон для телефонного звонка по технологии Voice over IP, и во время разговора по телефону вы перемещаетесь от одной точки доступа к другой или даже переключаетесь с Wi-Fi на сотовую сеть для вашего интернет-соединения.

Очевидно, что при переходе от одной сети доступа к другой вам нужно получить новый IP-адрес — тот, который соответствует новой сети. Но компьютер или телефон на другом конце вашего разговора не сразу узнает, куда вы переместились и какой у вас теперь новый IP-адрес. Следовательно, при отсутствии какого-либо другого механизма пакеты будут продолжать отправляться на адрес, где вы были раньше, а не туда, где вы находитесь сейчас. Эта проблема иллюстрируется на рис. 4.26; когда мобильный узел перемещается из сети 802.11 на рис. 4.26(a) в сотовую сеть на рис. 4.26(b), каким-то образом пакеты от узла-корреспондента должны найти путь к новой сети и затем к мобильному узлу.

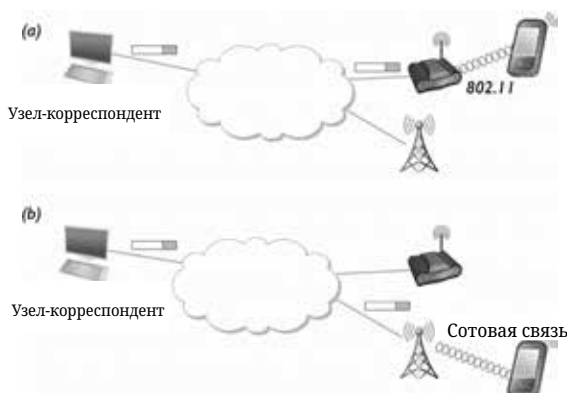


Рисунок 4.26. Передача пакетов от узла-корреспондента к мобильному узлу.

Существует множество различных способов решения описанной проблемы, и мы рассмотрим некоторые из них позже. Предполагая, что существует какой-то способ перенаправления пакетов так, чтобы они поступали на ваш новый адрес, а не на старый, следующие очевидные проблемы связаны с безопасностью. Например, если существует механизм, с помощью которого я могу сказать: «Мой новый IP-адрес — X», как предотвратить возможность того, что злоумышленник сделает такое заявление без моего разрешения, что позволит ему либо получать мои пакеты, либо перенаправлять мои пакеты никому не подозревающему третьему лицу? Таким образом, мы видим, что безопасность и мобильные устройства тесно связаны.

Одна из проблем, которую подчеркивает приведенное выше обсуждение, заключается в том, что IP-адреса фактически выполняют две задачи. Они используются в качестве идентификатора конечной точки, а также для определения местоположения конечной точки. Идентификатор можно рассматривать как долгоживущее имя для конечной точки, а локатор как некоторую, возможно, более временную информацию о том, как маршрутизировать пакеты к конечной точке. Пока устройства не перемещаются или перемещаются нечасто, использование одного адреса для обеих задач кажется вполне разумным. Но как только устройства начинают перемещаться, становится желательно иметь идентификатор, который не меняется при перемещении — это иногда называется *идентификатором конечной точки* или *идентификатором хоста*, — и отдельный локатор. Идея разделения локаторов и идентификаторов существует давно, и большинство подходов к поддержке мобильности, описанных ниже, обеспечивают такое разделение в той или иной форме.

Предположение о том, что IP-адреса не меняются, проявляется во многих различных местах. Например, транспортные протоколы, такие как TCP, исторически основывались на предположении, что IP-адрес остается постоянным на протяжении всего соединения, поэтому одним из подходов может быть перепроектирование транспортных протоколов, чтобы они могли работать с изменяющимися адресами конечных точек.

Но вместо того, чтобы пытаться изменить TCP, распространенной альтернативой является периодическое повторное установление TCP-соединения приложением в случае изменения IP-адреса клиента. Как бы странно это ни звучало, если приложение основано на HTTP (например, веб-браузер, такой как Chrome, или потоковое приложение, такое как Netflix), то именно так и происходит. Другими словами, стратегия заключается в том, чтобы приложение обходило ситуации, когда IP-адрес пользователя может измениться, вместо того чтобы пытаться поддерживать видимость его неизменности.

Хотя мы все знакомы с конечными точками, которые перемещаются, стоит отметить, что маршрутизаторы тоже могут перемещаться. Сегодня это, конечно, менее распространено, чем мобильность конечных точек, но существует множество сред, где мобильный маршрутизатор может иметь смысл. Один из примеров — команда реагирования на чрезвычайные ситуации, пытающаяся развернуть сеть после того, как природное бедствие вывело из строя всю фиксированную инфраструктуру. Когда все узлы в сети, а не только конечные точки, мобильны, возникают дополнительные соображения, о которых мы поговорим позже в этой главе.

Прежде чем мы начнем рассматривать некоторые подходы к поддержке мобильных устройств, нужно прояснить пару моментов. Часто люди путают беспроводные сети с мобильностью. В конце концов, мобильность и беспроводная связь часто встречаются вместе по очевидным причинам. Но беспроводная связь на самом деле касается передачи данных от точки А к точке В без провода, тогда как мобильность касается того, что происходит, когда узел перемещается во время передачи данных. Конечно, многие узлы, использующие беспроводные каналы связи, не являются мобильными, и иногда мобильные узлы будут использовать проводную связь (хотя это менее распространено).

Наконец, в этой главе нас в основном интересует то, что мы можем назвать *мобильностью на уровне сети*. То есть нас интересует, как обрабатывать узлы, перемещающиеся из одной сети в другую. Переход с одной точки доступа на другую в одной и той же сети

802.11 можно обрабатывать с помощью механизмов, специфичных для 802.11, и сотовые сети, конечно, также имеют способы справляться с мобильностью, но в больших гетерогенных системах, таких как Интернет, нам нужно поддерживать мобильность более широко по различным сетям.

Глава 4.5.2. Маршрутизация для мобильных узлов (Mobile IP)

Mobile IP — это основной механизм в архитектуре сегодняшнего Интернета для решения проблемы маршрутизации пакетов к мобильным узлам. Он вводит несколько новых возможностей, но не требует изменений от немобильных узлов или большинства маршрутизаторов, что делает его развертывание постепенным.

Предполагается, что мобильный узел имеет постоянный IP-адрес, называемый его *домашним адресом*, который имеет сетевой префикс, совпадающий с префиксом его *домашней сети*. Это адрес, который будут использовать другие узлы при первоначальной отправке пакетов на мобильный узел; поскольку он не изменяется, его можно использовать для долгосрочных приложений, когда узел перемещается. Мы можем рассматривать это как долгосрочный идентификатор узла.

Когда хост перемещается в новую внешнюю сеть, находящуюся вдали от его домашней сети, он обычно получает новый адрес в этой сети с помощью таких средств, как DHCP. Этот адрес будет изменяться каждый раз, когда узел перемещается в новую сеть, поэтому его можно рассматривать скорее как локатор для узла, но важно отметить, что узел не теряет своего постоянного домашнего адреса при получении нового адреса во внешней сети. Этот домашний адрес критически важен для его способности поддерживать связь при перемещении, как будет показано ниже.

Так как DHCP был разработан примерно в то же время, что и Mobile IP, оригинальные стандарты Mobile IP не требовали DHCP, но сегодня DHCP повсеместен.

Хотя большинство маршрутизаторов остаются неизменными, поддержка мобильности требует новой функциональности как минимум в одном маршрутизаторе, известном как *домашний агент* мобильного узла. Этот маршрутизатор находится в домашней сети мобильного узла. В некоторых случаях также требуется второй маршрутизатор с расширенной функциональностью, называемый *внешним агентом*. Этот маршрутизатор находится в сети, к которой подключается мобильный узел, когда он находится вдали от своей домашней сети. Сначала рассмотрим работу Mobile IP с использованием внешнего агента. Пример сети с домашним и внешним агентами показан на рис. 4.27.

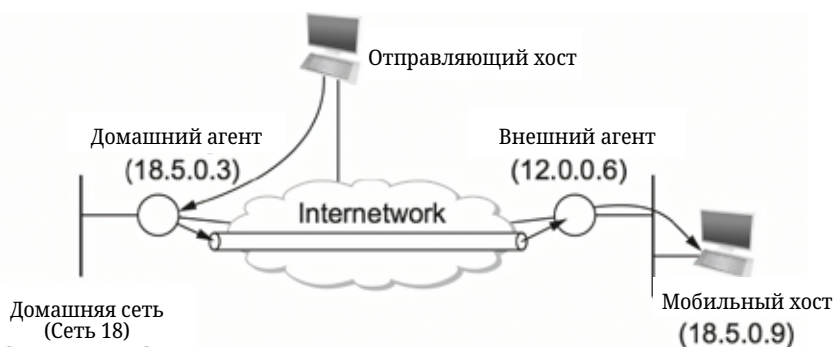


Рисунок 4.27. Мобильный хост и агенты мобильности.

Оба агента, домашний и внешний, периодически объявляют о своем присутствии в сетях, к которым они подключены, с помощью сообщений об объявлении агента. Мобильный узел также может запрашивать объявление при подключении к новой сети. Объявление домашнего агента позволяет мобильному узлу узнать адрес своего домашнего агента

перед тем, как он покинет свою домашнюю сеть. Когда мобильный узел подключается к внешней сети, он слышит объявление от внешнего агента и регистрируется у него, предоставляя адрес своего домашнего агента. Внешний агент затем связывается с домашним агентом, предоставляя *адрес временного пребывания*. Обычно это IP-адрес внешнего агента.

На данном этапе можно увидеть, что любой узел, пытающийся отправить пакет на мобильный узел, будет отправлять его с адресом назначения, равным домашнему адресу этого узла. Обычная IP-маршрутизация приведет этот пакет в домашнюю сеть мобильного узла, где находится домашний агент. Таким образом, проблему доставки пакета на мобильный узел можно разделить на три части:

1. Как домашний агент перехватывает пакет, предназначенный для мобильного узла?
2. Как домашний агент затем доставляет пакет внешнему агенту?
3. Как внешний агент доставляет пакет мобильному узлу?

Первая проблема может показаться простой, если посмотреть на рис. 4.27, где домашний агент явно является единственным путем между отправляющим узлом и домашней сетью, и поэтому он должен получать пакеты, предназначенные для мобильного узла. Но что если отправляющий узел (корреспондент) находится в сети 18, или если в сети 18 есть другой маршрутизатор, который пытается доставить пакет, минуя домашний агент? Чтобы решить эту проблему, домашний агент фактически выдает себя за мобильный узел, используя технику, называемую *прокси ARP*. Это работает точно так же, как протокол разрешения адресов (ARP), за исключением того, что домашний агент вставляет IP-адрес мобильного узла, а не свой собственный, в ARP-сообщения. Он использует свой собственный аппаратный адрес, так что все узлы в одной сети учатся связывать аппаратный адрес домашнего агента с IP-адресом мобильного узла. Один тонкий аспект этого процесса заключается в том, что информация ARP может кешироваться в других узлах сети. Чтобы гарантировать, что эти кеши будут своевременно аннулированы, домашний агент отправляет ARP-сообщение, как только мобильный узел регистрируется у внешнего агента. Поскольку ARP-сообщение не является ответом на обычный ARP-запрос, оно называется *безвозмездным ARP*.

Вторая проблема заключается в доставке перехваченного пакета внешнему агенту. Здесь мы используем описанную ранее технику туннелирования. Домашний агент просто оборачивает пакет в IP-заголовок, предназначенный для внешнего агента, и отправляет его в интернет. Все промежуточные маршрутизаторы видят IP-пакет, предназначенный для IP-адреса внешнего агента. Другими словами, между домашним агентом и внешним агентом создается IP-туннель, и домашний агент просто помещает пакеты, предназначенные для мобильного узла, в этот туннель.

Когда пакет наконец достигает внешнего агента, он снимает дополнительный IP-заголовок и обнаруживает внутри IP-пакет, предназначенный для домашнего адреса мобильного узла. Ясно, что внешний агент не может обработать этот пакет как любой другой IP-пакет, так как это приведет к отправке его обратно в домашнюю сеть. Вместо этого он должен распознать адрес как адрес зарегистрированного мобильного узла. Затем он доставляет пакет на *аппаратный* адрес мобильного узла (например, его Ethernet-адрес), который был получен в процессе регистрации.

Одно наблюдение, которое можно сделать относительно этих процедур, заключается в том, что внешний агент и мобильный узел могут находиться в одном устройстве; то есть мобильный узел может выполнять функцию внешнего агента сам. Чтобы это работало, мобильный узел должен уметь динамически получать IP-адрес, который находится в адресном пространстве внешней сети (например, используя DHCP). Этот адрес будет использоваться в качестве адреса временного пребывания. В нашем примере этот адрес будет иметь сетевой номер 12. Этот подход имеет желаемую особенность, позволяя мобильным узлам подключаться к сетям, не имеющим внешних агентов; таким образом, мобильность может быть достигнута с добавлением только домашнего агента и нового программного обеспечения на мобильный узел (при условии использования DHCP во внешней сети).

А как насчет трафика в другом направлении (то есть от мобильного узла к фиксированному узлу)? Это, оказывается, гораздо проще. Мобильный узел просто помещает IP-адрес фиксированного узла в поле назначения своих IP-пакетов, а в поле источника — свой постоянный адрес, и пакеты пересылаются фиксированному узлу обычными средствами. Конечно, если оба узла в разговоре являются мобильными, то процедуры, описанные выше, используются в каждом направлении.

Оптимизация маршрутов в Mobile IP

У вышеописанного подхода есть один значительный недостаток: маршрут от корреспондирующего узла к мобильному узлу может быть неоптимальным. Один из примеров — когда мобильный узел и корреспондирующий узел находятся в одной и той же сети, но домашняя сеть мобильного узла находится на другой стороне Интернета. Корреспондирующий узел отправляет все пакеты на домашнюю сеть; они пересекают Интернет, чтобы достичь домашнего агента, который затем туннелирует их обратно через Интернет, чтобы достигнуть внешнего агента. Очевидно, было бы неплохо, если бы корреспондирующий узел мог узнать, что мобильный узел на самом деле находится в той же сети, и доставить пакет напрямую. В общем случае цель состоит в том, чтобы доставлять пакеты как можно более прямо от корреспондирующего узла к мобильному узлу, минуя домашний агент. Это иногда называют проблемой *треугольной маршрутизации*, так как путь от корреспондирующего узла к мобильному узлу через домашний агент занимает две стороны треугольника, а не третью сторону, которая является прямым путем.

Основная идея решения проблемы треугольной маршрутизации заключается в том, чтобы позволить узлу-корреспонденту узнать адрес временного пребывания (care-of address) мобильного узла. Узел-корреспондент тогда сможет создать свой собственный туннель к внешнему агенту. Это рассматривается как оптимизация ранее описанного процесса. Если отправитель оснащен необходимым программным обеспечением для получения адреса временного пребывания и создания собственного туннеля, то маршрут может быть оптимизирован; в противном случае пакеты следуют по неоптимальному маршруту.

Когда домашний агент видит пакет, предназначенный для одного из поддерживаемых им мобильных узлов, он может предположить, что отправитель не использует оптимальный маршрут. Поэтому он отправляет сообщение «обновление привязки» (binding update) обратно к источнику, помимо пересылки пакета данных внешнему агенту. Источник, если способен, использует это обновление привязки для создания записи в кеше привязок, который состоит из списка отображений от адресов мобильных узлов к адресам временного пребывания. В следующий раз, когда этот источник будет отправлять пакет данных этому мобильному узлу, он найдет привязку в кеше и сможет туннелировать пакет непосредственно к внешнему агенту.

Очевидная проблема с этой схемой заключается в том, что кеш привязок может устареть, если мобильный узел переместится в новую сеть. Если будет использована устаревшая запись кеша, внешний агент получит туннелированные пакеты для мобильного узла, который больше не зарегистрирован в его сети. В этом случае он отправляет предупреждающее сообщение о привязке обратно отправителю, чтобы сообщить ему прекратить использование этой записи кеша. Эта схема работает только в случае, когда внешний агент не является самим мобильным узлом. По этой причине записи кеша должны удаляться через определенный промежуток времени; точное время указывается в сообщении обновления привязки.

Как отмечено выше, мобильная маршрутизация представляет собой интересные проблемы безопасности, которые становятся более понятными после изучения работы Mobile IP. Например, злоумышленник, желающий перехватить пакеты, предназначенные для другого узла в интернете, может связаться с домашним агентом этого узла и объявить себя новым внешним агентом для узла. Таким образом, очевидно, что необходимы некоторые механизмы аутентификации.

Мобильность в IPv6

Существует несколько значительных различий между поддержкой мобильности в IPv4 и IPv6. Самое важное — возможность встроить поддержку мобильности в стандарты IPv6 практически с самого начала, что позволяет избежать ряда проблем с поэтапным внедрением. (Можно более правильно сказать, что IPv6 является одной большой проблемой поэтапного внедрения, которая, будучи решенной, обеспечит поддержку мобильности как часть пакета.)

Поскольку все хосты, поддерживающие IPv6, могут получить адрес, когда они подключаются к внешней сети (используя несколько механизмов, определенных в основных спецификациях IPv6), Mobile IPv6 избавляется от потребности во внешнем агенте и включает необходимые возможности для выполнения функций внешнего агента в каждом узле.

Еще один интересный аспект IPv6, который проявляется в Mobile IP, это включение гибкого набора расширенных заголовков, как описано в другой части этой главы. Это используется в описанном выше сценарии оптимизированной маршрутизации. Вместо *туннелирования* пакета к мобильному узлу по его адресу временного пребывания узел IPv6 может отправить IP-пакет на адрес временного пребывания с домашним адресом, содержащимся в *заголовке маршрутизации*. Этот заголовок игнорируется всеми промежуточными узлами, но позволяет мобильному узлу обрабатывать пакет так, как если бы он был отправлен на домашний адрес, таким образом позволяя ему продолжать представлять протоколы более высокого уровня с иллюзией, что его IP-адрес фиксирован. Использование расширенного заголовка вместо туннеля более эффективно с точки зрения потребления полосы пропускания и обработки.

Наконец, следует отметить, что многие нерешенные проблемы остаются в мобильных сетях. Управление энергопотреблением мобильных устройств становится все более важным, чтобы можно было создавать более компактные устройства с ограниченным временем работы от батареи. Существует также проблема мобильных сетей *ad hoc* — создание сети из группы мобильных узлов в отсутствие фиксированных узлов, что создает некоторые особые трудности. Особенно сложным классом мобильных сетей являются *сенсорные сети*. Сенсоры обычно маленькие, недорогие и часто питаются от батареи, а это означает, что также должны быть учтены вопросы очень низкого энергопотребления и ограниченной вычислительной мощности. Кроме того, поскольку беспроводные коммуникации и мобильность обычно идут рука об руку, постоянное развитие беспроводных технологий продолжает создавать новые вызовы и возможности для мобильных сетей.

Перспектива: облако съедает Интернет

Облако и Интернет — это симбиотические системы. Изначально они были исторически раздельны, но сегодня граница между ними становится все более размытой. Согласно определению, Интернет обеспечивает сквозное соединение между двумя узлами (например, клиентским ноутбуком и удаленным сервером), а облако поддерживает несколько дата-центров размером со склад, каждый из которых обеспечивает экономически эффективный способ питания, охлаждения и эксплуатации большого количества серверов. Конечные пользователи подключаются к ближайшему дата-центру через Интернет точно так же, как они подключаются к серверу в удаленной серверной комнате.

Это точное описание отношений между Интернетом и облаком в первые дни коммерческих облачных провайдеров, таких как Amazon, Microsoft и Google. Например, облако Amazon примерно в 2009 году имело два дата-центра: один на восточном побережье США и один на западном побережье. Сегодня, однако, каждый из крупных облачных провайдеров управляет несколькими десятками дата-центров, разбросанных по всему миру, и неудивительно, что они стратегически расположены в непосредственной близости от точек обмена интернет-трафиком (IXP), каждая из которых обеспечивает богатую связность с остальной частью Интернета. В мире насчитывается более 150 IXP, и хотя не каждый

облачный провайдер оборудует полноценный дата-центр рядом с каждой из них (многие из этих мест — это колокационные объекты), можно сказать, что наиболее часто запрашиваемый контент облака (например, самые популярные фильмы Netflix, видео YouTube и фото Facebook) потенциально распределен в стольких местах.

Есть два последствия этого широкого распространения облака. Одно заключается в том, что сквозной путь от клиента к серверу не обязательно проходит через весь Интернет. Пользователь, скорее всего, обнаружит, что контент, который он или она хочет получить, был реплицирован в ближайшем IXP, что обычно находится всего в одном переходе AS, а не на другом конце света. Второе следствие заключается в том, что крупные облачные провайдеры не используют публичный Интернет для соединения своих распределенных дата-центров. Обычно облачные провайдеры поддерживают синхронизацию контента между распределенными дата-центрами, но они делают это через частную магистраль. Это позволяет им использовать любые оптимизации без необходимости полной интероперабельности с кем-либо еще.

Другими словами, хотя рисунки в главе 4.1 справедливо представляют общую форму Интернета, и BGP позволяет соединять любые пары узлов, на практике большинство пользователей взаимодействуют с приложениями, работающими в облаке, которое выглядит скорее как на рис. 4.28. (Одна важная деталь, которую рисунок не передает, заключается в том, что облачные провайдеры обычно не строят собственную WAN, прокладывая свое волокно, а вместо этого арендуют волокно у поставщиков услуг, что означает, что частная облачная магистраль и магистрали поставщиков услуг часто используют одну и ту же физическую инфраструктуру.)

Обратите внимание, что, несмотря на возможность *копирования контента* во многих местах облака, у нас пока нет технологии для копирования *людей*. Это означает, что когда разбросанные по всему земному шару пользователи хотят поговорить друг с другом (например, в рамках видеоконференции), именно дерево многоадресной рассылки распространяется по облаку. Другими словами, многоадресная рассылка обычно не работает в маршрутизаторах магистральных сетей поставщиков услуг.

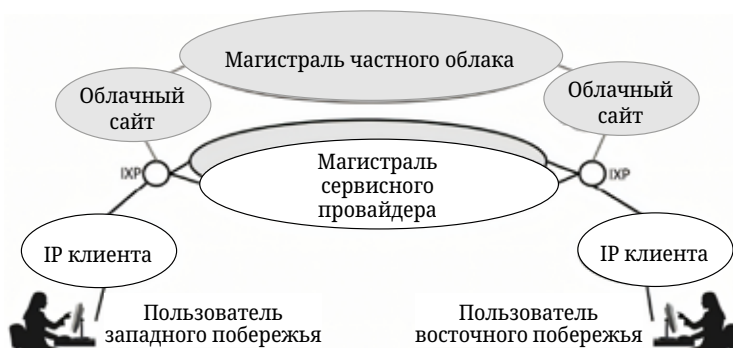


Рисунок 4.28. Облако широко распространено в Интернете с частными магистралями.

Раздел 5.

Сквозные протоколы

Победа — это прекрасный, ярко окрашенный цветок.

Транспорт — это стебель, без которого он никогда не смог бы расцвести.

Уинстон Черчилль

Проблема: обеспечение взаимодействия процессов

Для соединения множества компьютеров могут использоваться многие технологии, начиная от простых Ethernet и беспроводных сетей и заканчивая глобальными интернет-сетями. После их объединения следующая задача состоит в том, чтобы превратить эту службу доставки пакетов от хоста к хосту в канал связи от процесса к процессу. Это задача *транспортного* уровня сетевой архитектуры, который, поскольку он поддерживает связь между прикладными программами, работающими в конечных узлах, иногда называют сквозным протоколом.

Две силы формируют сквозной протокол. Сверху процессы прикладного уровня, использующие его услуги, имеют определенные требования. В следующем списке перечислены некоторые из общих свойств, которые можно ожидать от транспортного протокола:

- Гарантирует доставку сообщений
- Доставляет сообщения в том же порядке, в котором они отправлены
- Доставляет не более одной копии каждого сообщения
- Поддерживает произвольно большие сообщения
- Поддерживает синхронизацию между отправителем и получателем
- Позволяет получателю применять управление потоком к отправителю
- Поддерживает несколько прикладных процессов на каждом хосте

Обратите внимание, что в этом списке не указаны все функции, которые процессы приложений могут требовать от сети. Например, он не включает функции безопасности, такие как аутентификация или шифрование, которые обычно обеспечиваются протоколами, расположенными выше транспортного уровня. (Вопросы, связанные с безопасностью, мы обсудим в следующем разделе.)

Снизу подстилающая сеть, на которой работает транспортный протокол, имеет определенные ограничения в уровне обслуживания, который она может предоставить. Некоторые из наиболее типичных ограничений сети заключаются в том, что она может:

- Терять сообщения
- Переставлять сообщения
- Доставлять дубликаты одного и того же сообщения
- Ограничивать размер сообщений до некоторого конечного значения
- Доставлять сообщения после произвольно долгой задержки

Такая сеть, как принято говорить, предоставляет уровень обслуживания по принципу «наилучших усилий» (best-effort), как это характерно для Интернета.

Таким образом, задача состоит в том, чтобы разработать алгоритмы, которые превращают неидеальные свойства подстилающей сети в высокий уровень обслуживания, требуемый прикладными программами. Различные транспортные протоколы используют различные комбинации этих алгоритмов. В данном разделе эти алгоритмы рассматриваются в контексте четырех репрезентативных услуг — простой асинхронной демультиплексировой службы, надежной службы побайтового потока, службы запрос/ответ и службы для приложений реального времени.

В случае служб демультиплексирования и побайтового потока мы используем протокол дейтаграмм пользователя (User Datagram Protocol, UDP) и протокол управления пере-

дачей (Transmission Control Protocol, TCP) Интернета соответственно, чтобы проиллюстрировать, как эти службы предоставляются на практике. В случае службы «запрос/ответ» мы обсуждаем роль, которую она играет в службе удаленного вызова процедур (Remote Procedure Call, RPC), и какие функции это включает. В Интернете нет единого протокола RPC, поэтому мы завершаем это обсуждение описанием трех широко используемых протоколов RPC: SunRPC, DCE-RPC и gRPC.

Наконец, приложения реального времени предъявляют особые требования к транспортному протоколу, такие как необходимость передачи информации о времени, которая позволяет воспроизводить аудио или видео в нужный момент времени. Мы рассмотрели требования, предъявляемые приложениями к такому протоколу, и наиболее широко используемый пример — протокол транспортировки в реальном времени (Real-Time Transport Protocol, RTP).

Глава 5.1. Простой демультиплексор (UDP)

Самый простой транспортный протокол — это тот, который расширяет службу доставки от хоста к хосту подстилающей сети в службу связи от процесса к процессу. На любом данном хосте, вероятно, будут работать многие процессы, поэтому протоколу необходимо добавить уровень демультиплексирования, тем самым позволяя нескольким прикладным процессам на каждом хосте делить сеть. Помимо этого требования транспортный протокол не добавляет никакой другой функциональности к службе «наилучших усилий» (best-effort), предоставляемой подстилающей сетью. Протокол дейтаграмм пользователя (User Datagram Protocol, UDP) Интернета является примером такого транспортного протокола.

Единственный интересный вопрос в таком протоколе — это форма адреса, используемого для идентификации целевого процесса. Хотя процессы могут прямо идентифицировать друг друга с помощью назначенного операционной системой идентификатора процесса (pid), такой подход практичен только в закрытой распределенной системе, в которой на всех хостах работает одна операционная система, назначающая каждому процессу уникальный идентификатор. Более распространенный подход, используемый UDP, заключается в том, что процессы косвенно идентифицируют друг друга, используя абстрактный локатор, обычно называемый портом. Основная идея заключается в том, что исходный процесс отправляет сообщение на порт, а целевой процесс получает сообщение с порта.

Заголовок для сквозного протокола, реализующего эту функцию демультиплексирования, обычно содержит идентификатор (порт) как для отправителя (источника), так и для получателя (назначение) сообщения. Например, заголовок UDP показан на рис. 5.1. Обратите внимание, что поле порта UDP имеет длину всего 16 бит. Это означает, что существует до 64К возможных портов, которых явно недостаточно для идентификации всех процессов на всех хостах в Интернете. К счастью, порты не интерпретируются по всему Интернету, а только на одном хосте. То есть процесс действительно идентифицируется портом на каком-то конкретном хосте: пара (порт, хост). Эта пара составляет демультиплексирующий ключ для протокола UDP.

Следующий вопрос заключается в том, как процесс узнает порт процесса, которому он хочет отправить сообщение. Обычно клиентский процесс инициирует обмен сообщениями с серверным процессом. После того как клиент связался с сервером, сервер узнает порт клиента (из поля SrcPrt, содержащегося в заголовке сообщения) и может ответить на него. Таким образом, настоящая проблема заключается в том, как клиент узнает порт сервера в первую очередь. Распространенный подход заключается в том, что сервер принимает сообщения на *известный порт*. То есть каждый сервер получает свои сообщения на некотором фиксированном порту, который широко известен, как, например, служба экстренной помощи, доступная в Соединенных Штатах по известному номеру телефона 911. В Интернете, например, сервер доменных имен (DNS) принимает сообщения на из-

вестном порту 53 на каждом хосте, почтовая служба принимает сообщения на порту 25, а программа Unix talk принимает сообщения на известном порту 517 и так далее. Это соответствие периодически публикуется в RFC и доступно на большинстве систем Unix в файле `/etc/services`. Иногда известный порт — это всего лишь отправная точка для связи: клиент и сервер используют известный порт, чтобы договориться о другом порте, который они будут использовать для последующей связи, оставляя известный порт свободным для других клиентов.

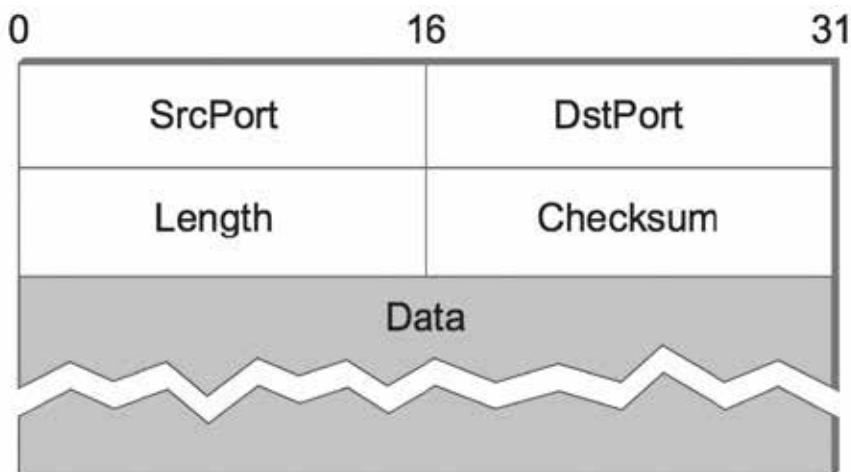


Рисунок 5.1. Формат заголовка UDP.

Альтернативная стратегия заключается в обобщении этой идеи, чтобы существовал только один известный порт — тот, через который служба *port mapper* принимает сообщения. Клиент посылает сообщение на известный порт службы *port mapper*, запрашивая порт, который он должен использовать для связи со службой «whatever», и служба *port mapper* возвращает соответствующий порт. Такая стратегия позволяет легко менять порт, связанный с различными службами, с течением времени, а также использовать на каждом хосте разные порты для одной и той же службы.

Как уже упоминалось, порт является чистой абстракцией. То, как он реализован, отличается от системы к системе, или, точнее, от операционной системы к операционной системе. Например, описанный в первом разделе API сокетов является примером реализации портов. Обычно порт реализуется в виде очереди сообщений, как показано на рис. 5.2. Когда сообщение прибывает, протокол (например, UDP) добавляет сообщение в конец очереди. Если очередь заполнена, сообщение отбрасывается. В UDP нет механизма управления потоком, чтобы сообщить отправителю о необходимости замедлить отправку. Когда прикладной процесс хочет получить сообщение, оно удаляется из начала очереди. Если очередь пуста, процесс блокируется до тех пор, пока сообщение не станет доступным.

Наконец, хотя UDP не реализует управление потоком или надежную/упорядоченную доставку, он предоставляет еще одну функцию помимо демультимплексирования сообщений для какого-то прикладного процесса — он также обеспечивает правильность сообщения с помощью контрольной суммы. (Контрольная сумма UDP является необязательной в IPv4, но она обязательна в IPv6.) Основной алгоритм контрольной суммы UDP такой же, как и для IP — то есть он складывает набор 16-битных слов, используя арифметику с дополнением до единицы, и берет дополнение до единицы от результата. Но входные данные, используемые для контрольной суммы, немного неинтуитивны.

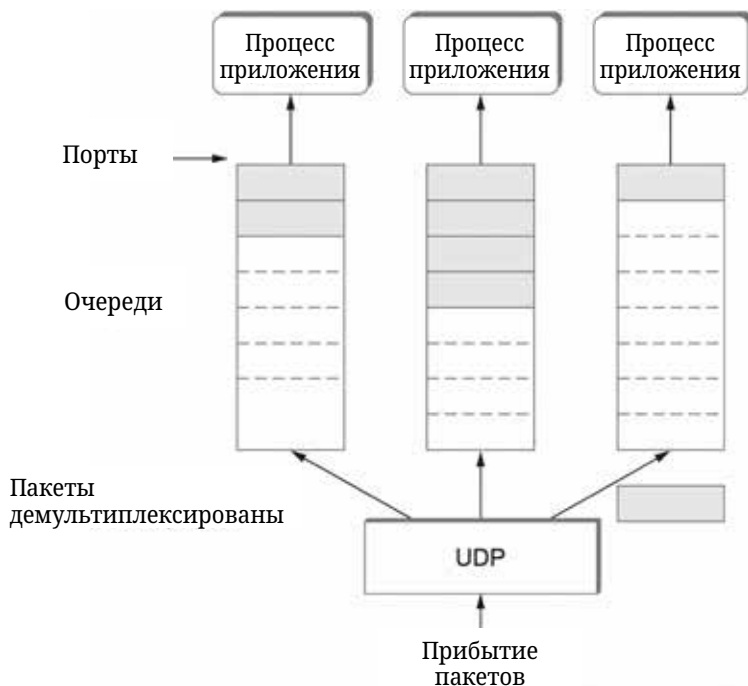


Рисунок 5.2. Очередь сообщений UDP.

Контрольная сумма UDP берет в качестве входных данных заголовок UDP, содержимое тела сообщения и нечто, называемое *псевдозаголовком*. Псевдозаголовок состоит из трех полей заголовка IP — номера протокола, IP-адреса источника и IP-адреса назначения — плюс поле длины UDP. (Да, поле длины UDP дважды включается в расчет контрольной суммы.) Мотивация для использования псевдозаголовка заключается в том, чтобы проверить, что это сообщение было доставлено между двумя правильными конечными точками. Например, если IP-адрес назначения был изменен, пока пакет находился в пути, что привело к неправильной доставке пакета, этот факт будет обнаружен контрольной суммой UDP.

Глава 5.2. Надежный побайтовый поток (TCP)

В отличие от простого протокола демультиплексирования, такого как UDP, более сложный транспортный протокол — это тот, который предлагает надежную, ориентированную на соединение побайтовую службу. Такая служба оказалась полезной для широкого круга приложений, потому что освобождает приложение от необходимости беспокоиться о пропущенных или переставленных данных. Протокол управления передачей (Transmission Control Protocol, TCP) Интернета, вероятно, является наиболее широко используемым протоколом такого типа; он также является самым тщательно настроенным. Именно по этим двум причинам этот раздел подробно изучает TCP, хотя мы выделяем и обсуждаем альтернативные варианты дизайна сети в конце раздела.

С точки зрения свойств транспортных протоколов, указанных в постановке задачи в начале этого раздела, TCP гарантирует надежную, упорядоченную доставку потока байтов. Это полнодуплексный протокол, что означает, что каждое соединение TCP поддерживает пару потоков байтов, один в каждом направлении. Он также включает механизм управления потоком для каждого из этих потоков байтов, который позволяет получателю ограничивать, сколько данных отправитель может передавать в данный момент времени. Наконец, как и UDP, TCP поддерживает механизм демультиплексирования, который

позволяет нескольким прикладным программам на любом данном хосте одновременно вести беседу со своими аналогами.

В дополнение к вышеуказанным функциям ТСП также реализует тщательно настроенный механизм управления перегрузками. Идея этого механизма заключается в том, чтобы регулировать скорость отправки данных ТСП не для того, чтобы предотвратить перегрузку получателя, а чтобы предотвратить перегрузку сети. Описание механизма управления перегрузками ТСП отложено до следующего раздела, где мы обсуждаем его в более широком контексте распределения сетевых ресурсов справедливым образом.

Поскольку многие люди путают управление перегрузками и управление потоком, мы повторим, в чем заключается различие. Управление потоком включает в себя предотвращение перегрузки отправителями возможностей получателей. Управление перегрузками включает в себя предотвращение попадания слишком большого количества данных в сеть, что может привести к перегрузке коммутаторов или каналов. Таким образом, управление потоком — это вопрос «от конца до конца», в то время как управление перегрузками касается того, как взаимодействуют хосты и сети.

Глава 5.2.1. Проблемы сквозного взаимодействия

В основе ТСП лежит алгоритм «скользящего окна». Хотя это тот же основной алгоритм, который часто используется на канальном уровне, из-за того, что ТСП работает по всему Интернету, а не по физической линии «точка-точка», существует множество важных различий. В этой подглаве выявляются эти различия и объясняется, как они усложняют ТСП. В следующих подглавах описывается, как ТСП решает эти и другие осложнения.

Во-первых, тогда как алгоритм «скользящего окна» на канальном уровне работает по одной физической линии, которая всегда соединяет одни и те же два компьютера, ТСП поддерживает логические соединения между процессами, работающими на любых двух компьютерах в Интернете. Это означает, что ТСП нуждается в явной фазе установления соединения, во время которой обе стороны соединения договариваются обмениваться данными друг с другом. Это отличие аналогично необходимости набрать номер другого абонента, а не иметь выделенную телефонную линию. ТСП также имеет явную фазу завершения соединения. Одной из задач при установлении соединения является создание общего состояния, чтобы алгоритм «скользящего окна» мог начать работать. Завершение соединения необходимо, чтобы каждый хост знал, что можно прекратить это состояние.

Во-вторых, тогда как одиночная физическая линия, которая всегда соединяет одни и те же два компьютера, имеет фиксированное время кругового пути (RTT), соединения ТСП могут иметь значительно различающиеся времена кругового пути. Например, соединение ТСП между хостом в Сан-Франциско и хостом в Бостоне, которые разделены несколькими тысячами километров, может иметь RTT в 100 мс, тогда как соединение ТСП между двумя хостами в одной комнате, расположенными всего в нескольких метрах друг от друга, может иметь RTT всего в 1 мс. Один и тот же протокол ТСП должен поддерживать оба этих соединения. Чтобы усложнить ситуацию, соединение ТСП между хостами в Сан-Франциско и Бостоне может иметь RTT в 100 мс в 3 часа ночи, но RTT в 500 мс в 3 часа дня. Вариации в RTT возможны даже во время одного соединения ТСП, которое длится всего несколько минут. Это означает, что механизм тайм-аута, который запускает повторные передачи, должен быть адаптивным. (Безусловно, тайм-аут для соединения «точка-точка» должен быть настраиваемым параметром, но нет необходимости адаптировать этот таймер для конкретной пары узлов.)

Третье отличие заключается в том, что пакеты могут быть переупорядочены при прохождении через Интернет, чего не может произойти на линии «точка-точка», где первый пакет, помещенный на одном конце линии, должен быть первым, который появится на другом конце. Пакеты, которые слегка нарушили порядок, не вызывают проблемы, так как алгоритм «скользящего окна» может правильно переупорядочить пакеты, используя порядковый номер. Настоящая проблема заключается в том, насколько сильно пакеты могут быть вне порядка или, иными словами, насколько поздно пакет может прибыть

к месту назначения. В худшем случае пакет может быть задержан в Интернете до тех пор, пока не истечет поле времени жизни (TTL) IP, после чего пакет будет отброшен (и, следовательно, нет опасности его запоздалого прибытия). Зная, что IP отбрасывает пакеты после истечения их TTL, TCP предполагает, что каждый пакет имеет максимальное время жизни. Точное время жизни, известное как *максимальное время жизни сегмента* (maximum segment lifetime, MSL), является инженерным выбором. Текущее рекомендованное значение — 120 секунд. Помните, что IP напрямую не применяет это значение в 120 секунд; это просто консервативная оценка, которую делает TCP относительно того, как долго пакет может жить в Интернете. Последствие этого значимо — TCP должен быть готов к тому, что очень старые пакеты могут неожиданно появиться у получателя, потенциально сбивая с толку алгоритм «скользящего окна».

Четвертое отличие заключается в том, что компьютеры, подключенные к линии «точка-точка», обычно проектируются для поддержки этой линии. Например, если произведение задержки и пропускной способности линии составляет 8 КБ, что означает, что размер окна выбран так, чтобы позволить до 8 КБ данных оставаться неподтвержденными в любой момент времени, то, вероятно, компьютеры на обоих концах линии имеют возможность буферизовать до 8 КБ данных. Проектировать систему иначе было бы глупо. С другой стороны, почти любой тип компьютера может быть подключен к Интернету, что делает количество ресурсов, выделенных для любого одного соединения TCP, весьма переменным, особенно учитывая, что один хост может потенциально поддерживать сотни соединений TCP одновременно. Это означает, что TCP должен включать механизм, который каждая сторона использует для «обучения», какие ресурсы (например, сколько буферного пространства) другая сторона способна выделить для соединения. Это и есть проблема управления потоком.

Пятое отличие заключается в том, что передающая сторона напрямую подключенной линии не может отправлять данные быстрее, чем позволяет пропускная способность линии, и только один хост передает данные в линию, поэтому невозможно случайно перегрузить линию. Иными словами, нагрузка на линию видна в виде очереди пакетов у отправителя. В отличие от этого отправляющая сторона TCP-соединения не знает, какие линии будут пересекаться для достижения пункта назначения. Например, отправляющий компьютер может быть напрямую подключен к относительно быстрой сети Ethernet и способен отправлять данные со скоростью 10 Гбит/с, но где-то посреди сети может находиться линия с пропускной способностью 1,5 Мбит/с. Более того, данные, генерируемые множеством различных источников, могут пытаться пройти через эту «медленную» линию. Это приводит к проблеме перегрузки сети. Обсуждение этой темы будет отложено до следующего раздела.

Мы завершаем обсуждение сквозных вопросов, сравнивая подход TCP к обеспечению надежной/упорядоченной доставки с подходом, используемым в сетях с виртуальными цепями, таких как исторически важная сеть X.25. В TCP предполагается, что основная сеть IP является ненадежной и доставляет сообщения вне порядка; TCP использует алгоритм «скользящего окна» на сквозной основе для обеспечения надежной/упорядоченной доставки. В отличие от этого сети X.25 используют протокол «скользящего окна» внутри сети на основе переходов. Предположение, лежащее в основе этого подхода, заключается в том, что если сообщения доставляются надежно и в порядке между каждой парой узлов на пути от исходного хоста до конечного хоста, то и сквозная служба также гарантирует надежную/упорядоченную доставку.

Проблема с последним подходом заключается в том, что последовательность гарантий на основе переходов не обязательно складывается в сквозную гарантию. Во-первых, если к одному концу пути добавляется гетерогенная линия (например, Ethernet), то нет гарантии, что этот переход сохранит ту же службу, что и другие переходы. Во-вторых, тот факт, что протокол скользящего окна гарантирует, что сообщения доставляются правильно от узла А к узлу В, а затем от узла В к узлу С, не гарантирует, что узел В будет работать безупречно. Например, известно, что сетевые узлы могут вносить ошибки в сообщения при их переносе из входного буфера в выходной буфер. Также известно, что они могут случайно переупорядо-

чивать сообщения. В результате этих небольших уязвимостей по-прежнему необходимо обеспечивать истинные сквозные проверки для гарантии надежной/упорядоченной службы, даже если более низкие уровни системы также реализуют эту функциональность.

Основные выводы

Это обсуждение иллюстрирует один из самых важных принципов проектирования систем — *аргумент сквозного соединения*. Если вкратце, то аргумент сквозного соединения говорит, что функция (в нашем примере обеспечение надежной/упорядоченной доставки) не должна предоставляться на нижних уровнях системы, если она не может быть полностью и правильно реализована на этом уровне. Поэтому данное правило выступает в пользу подхода TCP/IP. Однако это правило не является обязательным. Оно допускает, что функции могут быть частично реализованы на низком уровне в качестве оптимизации производительности. Именно поэтому вполне соответствует аргументу сквозного соединения выполнение проверки ошибок (например, CRC) на основе переходов; обнаружение и повторная передача одного поврежденного пакета через один переход предпочтительнее, чем повторная передача целого файла сквозным образом

Глава 5.2.2. Формат сегмента

TCP (Transmission Control Protocol) — это протокол, ориентированный на байты, и это означает, что отправитель записывает байты в TCP-соединение, а получатель считывает байты из TCP-соединения. Хотя «поток байтов» описывает услугу, которую TCP предлагает прикладным процессам, сам TCP не передает отдельные байты через Интернет. Вместо этого TCP на исходном хосте буферизует достаточно байтов от процесса-отправителя, чтобы заполнить пакет разумного размера, а затем отправляет этот пакет своему аналогу на хосте-получателе. TCP на хосте-получателе освобождает содержимое пакета в буфер приема, и процесс-получатель считывает из этого буфера по мере необходимости. Эта ситуация проиллюстрирована на рис. 5.3., который для простоты показывает поток данных только в одном направлении. Помните, что в общем случае одно TCP-соединение поддерживает потоки байтов в обоих направлениях.

Пакеты, обмениваемые между TCP-партнерами на рис. 5.3, называются *сегментами*, так как каждый из них несет сегмент потока байтов. Каждый TCP-сегмент содержит заголовок, схематически изображенный на рис. 5.4. Значение большинства из этих полей станет очевидным на протяжении изучения данного раздела. Пока что мы просто вводим их в обиход.

Поля *SrcPort* и *DstPort* идентифицируют исходные и целевые порты соответственно, так же как в UDP.

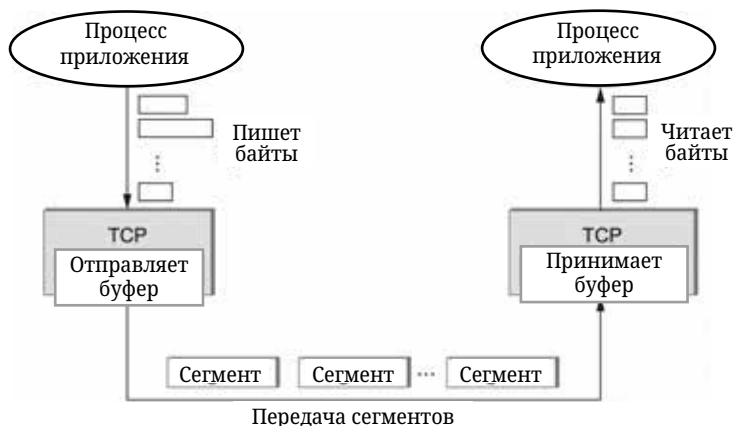


Рисунок 5.3. Как TCP управляет потоком байтов.

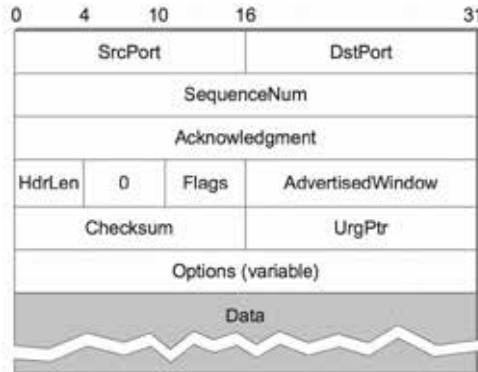


Рисунок 5.4. Формат заголовка TCP.

Эти два поля, плюс исходный и целевой IP-адреса, объединяются, чтобы уникально идентифицировать каждое TCP-соединение. То есть ключ демультиплексирования TCP задан кортежем из четырех элементов:

(SrcPort, SrcIPAddr, DstPort, DstIPAddr)

Заметьте, что поскольку TCP-соединения создаются и разрываются, возможно, что соединение между определенной парой портов будет установлено, использовано для отправки и получения данных и затем закрыто, после чего в дальнейшем для той же пары портов может быть установлено второе соединение. Мы иногда называем эту ситуацию двумя различными *воплощениями одного и того же соединения*.

Поля Acknowledgement, SequenceNum и AdvertisedWindow задействованы в алгоритме «скользящего окна» TCP. Поскольку TCP является протоколом, ориентированным на байты, каждый байт данных имеет порядковый номер. Поле SequenceNum содержит порядковый номер для первого байта данных, передаваемых в этом сегменте, а поля Acknowledgement и AdvertisedWindow несут информацию о потоке данных, идущем в другом направлении. Чтобы упростить наше обсуждение, мы игнорируем тот факт, что данные могут течь в обоих направлениях, и концентрируемся на данных, имеющих определенный SequenceNum, текущих в одном направлении, и значениях Acknowledgement и AdvertisedWindow, текущих в противоположном направлении, как показано на рис. 5.5. Использование этих трех полей более подробно описано в дальнейшем в этом разделе.

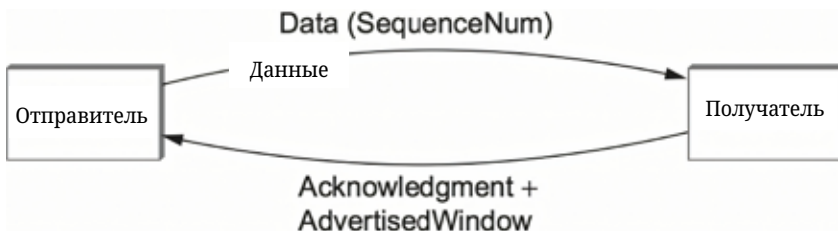


Рисунок 5.5. Упрощенная иллюстрация (показывающая только одно направление) процесса TCP, с потоком данных в одном направлении и ACK в другом.

Поле Flags длиной 6 бит используется для передачи управляющей информации между TCP-партнерами. Возможные флаги включают SYN, FIN, RESET, PUSH, URG и ACK. Флаги SYN и FIN используются при установлении и завершении TCP-соединения соответственно. Их использование описано далее. Флаг ACK устанавливается всякий раз, когда поле Acknowledgement действительно, что означает, что получатель должен обратить на него

внимание. Флаг URG означает, что этот сегмент содержит срочные данные. Когда этот флаг установлен, поле UrgPtr указывает, где начинается несрочные данные, содержащиеся в этом сегменте. Срочные данные содержатся в начале тела сегмента, до и включая значение UrgPtr байтов в сегменте. Флаг PUSH означает, что отправитель вызвал операцию push, а это указывает на то, что принимающая сторона TCP должна уведомить процесс-получатель об этом факте. Эти последние две функции более подробно обсуждаются в дальнейших подглавах. Наконец, флаг RESET означает, что получатель запутался, например, из-за получения сегмента, который он не ожидал получить, и поэтому хочет прервать соединение.

Наконец, поле Checksum используется точно так же, как и для UDP — оно вычисляется по заголовку TCP, данным TCP и псевдозаголовку, который состоит из исходного адреса, целевого адреса и полей длины из заголовка IP. Контрольная сумма обязательна для TCP как в IPv4, так и в IPv6. Также, поскольку заголовок TCP имеет переменную длину (опции могут быть добавлены после обязательных полей), включено поле HdrLen, которое указывает длину заголовка в 32-битных словах. Это поле также известно как поле Offset, поскольку оно измеряет смещение от начала пакета до начала данных.

Глава 5.2.3. Установление и завершение соединения

TCP-соединение начинается с того, что клиент (отправляющий) делает активное открытие серверу (принимающему). Предполагая, что сервер ранее сделал пассивное открытие, обе стороны начинают обмен сообщениями для установления соединения. (Напомним из первого раздела, что сторона, желающая инициировать соединение, выполняет активное открытие, в то время как сторона, готовая принять соединение, выполняет пассивное открытие.¹) Только после завершения фазы установления соединения обе стороны начинают отправлять данные. Аналогично, как только участник завершает отправку данных, он закрывает одно направление соединения, что вызывает у TCP начало раунда сообщений о завершении соединения. Заметьте, что установка соединения является асимметричной деятельностью (одна сторона делает пассивное открытие, а другая — активное), тогда как завершение соединения симметрично (каждая сторона должна закрыть соединение независимо). Поэтому возможно, что одна сторона закрыла соединение, что означает, что она больше не может отправлять данные, но другая сторона оставляет свою половину двунаправленного соединения открытой и продолжает отправлять данные.

Трехэтапное рукопожатие

Алгоритм, используемый TCP для установления и завершения соединения, называется *трехэтапным рукопожатием*. Сначала мы опишем основной алгоритм, а затем покажем, как он используется в TCP. Трехэтапное рукопожатие включает обмен тремя сообщениями между клиентом и сервером, как показано на временной шкале на рис. 5.6.

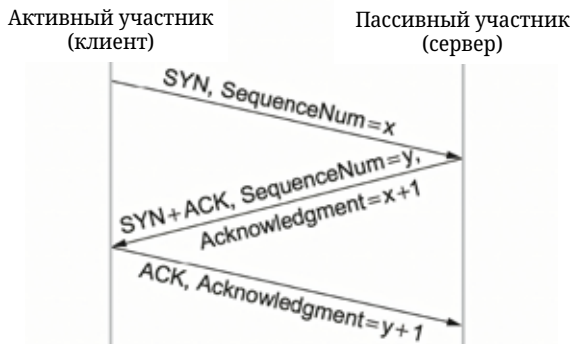


Рисунок 5.6. Временная шкала для алгоритма трехстороннего рукопожатия.

¹ Если быть более точным, TCP позволяет устанавливать соединение симметрично, когда обе стороны пытаются открыть соединение одновременно, но обычно одна сторона делает активное открытие, а другая — пассивное.

Идея состоит в том, что две стороны хотят договориться о наборе параметров, которые, в случае открытия TCP-соединения, представляют собой начальные порядковые номера, которые обе стороны планируют использовать для своих потоков байтов. В общем случае параметры могут быть любыми фактами, которые каждая сторона хочет, чтобы другая знала. Сначала клиент (активный участник) отправляет сегмент серверу (пассивному участнику), указывая начальный порядковый номер, который он планирует использовать ($\text{Flags} = \text{SYN}$, $\text{SequenceNum} = x$). Сервер затем отвечает одним сегментом, который одновременно подтверждает порядковый номер клиента ($\text{Flags} = \text{ACK}$, $\text{Ack} = x + 1$) и указывает свой собственный начальный порядковый номер ($\text{Flags} = \text{SYN}$, $\text{SequenceNum} = y$). То есть в этом втором сообщении установлены флаги как SYN, так и ACK. Наконец, клиент отвечает третьим сегментом, который подтверждает порядковый номер сервера ($\text{Flags} = \text{ACK}$, $\text{Ack} = y + 1$). Причина, по которой каждая сторона подтверждает порядковый номер, который на единицу больше отправленного, заключается в том, что поле Acknowledgement фактически указывает «следующий ожидаемый порядковый номер», тем самым косвенно подтверждая все предыдущие порядковые номера. Хотя на данной временной шкале это не показано, для каждого из первых двух сегментов устанавливается таймер, и если ожидаемый ответ не получен, сегмент передается повторно.

Вы можете спросить себя, почему клиент и сервер должны обмениваться начальными порядковыми номерами при установке соединения. Было бы проще, если бы каждая сторона просто начинала с какого-то «известного» порядкового номера, такого как 0. На самом деле спецификация TCP требует, чтобы каждая сторона соединения выбирала начальный порядковый номер случайным образом. Причина этого заключается в защите от повторного использования тех же порядковых номеров двумя инкарнациями одного и того же соединения слишком рано — то есть когда еще существует вероятность, что сегмент от предыдущей инкарнации соединения может помешать более поздней инкарнации соединения.

Диаграмма переходов состояния

TCP достаточно сложен, чтобы его спецификация включала диаграмму переходов состояния. Копия этой диаграммы приведена на рис. 5.7. Эта диаграмма показывает только состояния, связанные с открытием соединения (все выше ESTABLISHED) и с закрытием соединения (все ниже ESTABLISHED). Все, что происходит, пока соединение открыто — то есть работа алгоритма «скользящего окна», — скрыто в состоянии ESTABLISHED.

Диаграмма переходов состояния TCP довольно проста для понимания. Каждый блок обозначает состояние, в котором может находиться один конец TCP-соединения. Все соединения начинаются в состоянии CLOSED. По мере прогресса соединения оно переходит из состояния в состояние согласно дугам. Каждая дуга помечена тегом вида *событие/действие*. Таким образом, если соединение находится в состоянии LISTEN и приходит сегмент SYN (то есть сегмент с установленным флагом SYN), соединение переходит в состояние SYN_RCVD и выполняет действие ответа сегментом ACK+SYN.

Обратите внимание, что два типа событий вызывают переход состояния: (1) сегмент прибывает от пира (например, событие на дуге от LISTEN к SYN_RCVD), или (2) локальный процесс приложения вызывает операцию на TCP (например, событие *active open* на дуге от CLOSED к SYN_SENT). Другими словами, диаграмма переходов состояния TCP эффективно определяет *семантику* как однорангового, так и сервисного интерфейса. *Синтаксис* этих двух интерфейсов задается форматом сегмента (как показано на рис. 5.4) и некоторым интерфейсом прикладного программирования, таким как API сокетов.

Теперь давайте проследим типичные переходы на диаграмме на рис. 5.7. Помните, что на каждом конце соединения TCP делает разные переходы из одного состояния в другое. При открытии соединения сервер сначала выполняет операцию пассивного открытия на TCP, что переводит TCP в состояние LISTEN. В какой-то момент позже клиент вы-

полняет активное открытие, что заставляет его конец соединения отправить сегмент SYN серверу и перейти в состояние SYN_SENT. Когда сегмент SYN приходит к серверу, он переходит в состояние SYN_RCVD и отвечает сегментом SYN+ACK. Прибытие этого сегмента заставляет клиента перейти в состояние ESTABLISHED и отправить обратно серверу ACK. Когда этот ACK прибывает, сервер наконец переходит в состояние ESTABLISHED. Другими словами, мы только что проследили за трехэтапным рукопожатием.

Есть три вещи, на которые следует обратить внимание в половине диаграммы переходов состояния, связанной с установлением соединения. Во-первых, если ACK клиента серверу потерян, что соответствует третьей фазе трехэтапного рукопожатия, то соединение все равно функционирует правильно. Это потому, что сторона клиента уже находится в состоянии ESTABLISHED, поэтому локальный процесс приложения может начать отправку данных на другой конец. Каждый из этих сегментов данных будет иметь установленный флаг ACK и правильное значение в поле Acknowledgement, поэтому сервер перейдет в состояние ESTABLISHED, когда первый сегмент данных прибывает. Это на самом деле важный момент о TCP — каждый сегмент сообщает, какой порядковый номер отправитель ожидает увидеть следующим, даже если это повторяет тот же порядковый номер, содержащийся в одном или нескольких предыдущих сегментах.

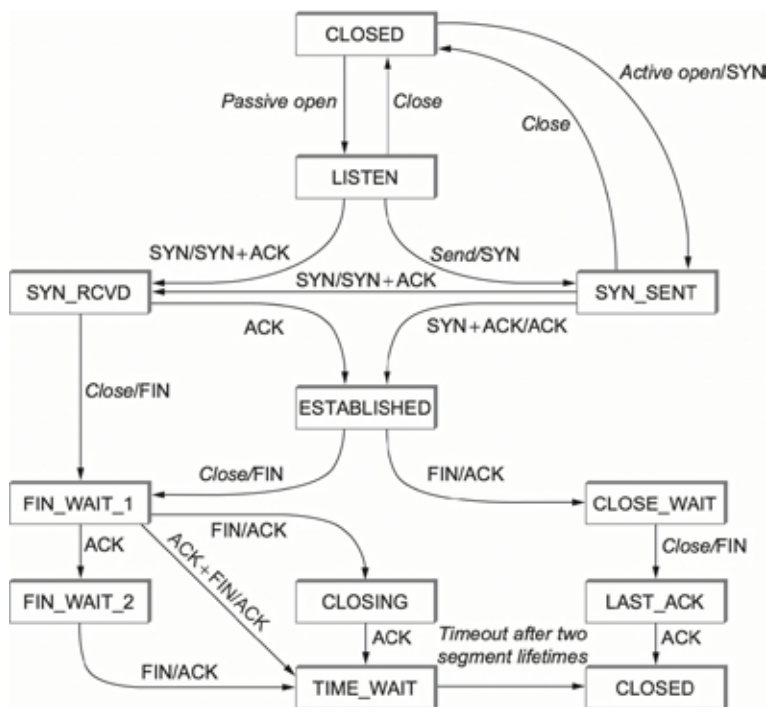


Рисунок 5.7. Диаграмма переходов между состояниями TCP.

Второе, на что следует обратить внимание в диаграмме переходов состояния, это переход из состояния LISTEN, когда локальный процесс вызывает операцию *отправки* в TCP. То есть возможно, что пассивный участник может идентифицировать оба конца соединения (то есть себя и удаленного участника, с которым он готов установить соединение), а затем изменить свое решение о том, чтобы ждать другой стороны, и вместо этого активно установить соединение. Насколько нам известно, ни один процесс приложения на самом деле не использует эту функцию TCP.

Последнее, на что следует обратить внимание в диаграмме, это дуги, которые не показаны. В частности, большинство состояний, связанных с отправкой сегмента на другую сторону, также назначают тайм-аут, который в конечном итоге вызывает повторную отправку сегмента, если ожидаемый ответ не происходит. Эти повторные передачи не отображены на диаграмме переходов состояния. Если после нескольких попыток ожидаемый ответ не приходит, TCP прекращает попытки и возвращается в состояние CLOSED.

Теперь обратим внимание на процесс завершения соединения. Важно помнить, что процесс приложения с обеих сторон соединения должен независимо закрыть свою половину соединения. Если только одна сторона закрывает соединение, это означает, что у нее больше нет данных для отправки, но она все еще доступна для получения данных от другой стороны. Это усложняет диаграмму переходов состояния, так как необходимо учитывать возможность того, что обе стороны вызывают операцию закрытия одновременно, а также возможность того, что сначала одна сторона вызывает операцию закрытия, а затем, через некоторое время, другая сторона вызывает операцию закрытия. Таким образом, на любой стороне есть три комбинации переходов, которые переводят соединение из состояния ESTABLISHED в состояние CLOSED:

- Эта сторона закрывает первой: ESTABLISHED → FIN_WAIT_1 → FIN_WAIT_2 → TIME_WAIT → CLOSED.
- Другая сторона закрывает первой: ESTABLISHED → CLOSE_WAIT → LAST_ACK → CLOSED.
- Обе стороны закрывают одновременно: ESTABLISHED → FIN_WAIT_1 → CLOSING → TIME_WAIT → CLOSED.

На самом деле есть четвертая, хотя и редкая, последовательность переходов, которая приводит к состоянию CLOSED; она следует по дуге от FIN_WAIT_1 к TIME_WAIT. Мы оставляем это в качестве упражнения для вас, чтобы вы самостоятельно выяснили, какая комбинация обстоятельств приводит к этой четвертой возможности.

Главное, что следует понимать о завершении соединения, это то, что соединение в состоянии TIME_WAIT не может перейти в состояние CLOSED, пока не пройдет время, равное двум максимальным промежуткам времени жизни IP-дейтаграммы в Интернете (то есть 120 секунд). Причина этого заключается в том, что, хотя локальная сторона соединения отправила ACK в ответ на сегмент FIN другой стороны, она не знает, был ли ACK успешно доставлен. В результате другая сторона может повторно отправить свой сегмент FIN, и этот второй сегмент FIN может задержаться в сети. Если бы соединению разрешили перейти непосредственно в состояние CLOSED, другая пара процессов приложения могла бы открыть то же самое соединение (то есть использовать ту же пару номеров портов), и задержанный сегмент FIN от предыдущей инкарнации соединения немедленно инициировал бы завершение более поздней инкарнации этого соединения.

Глава 5.2.4. Повторное рассмотрение «скользящего окна»

Теперь мы готовы обсудить вариант алгоритма «скользящего окна», используемый в TCP, который выполняет несколько задач: (1) гарантирует надежную доставку данных, (2) обеспечивает доставку данных в правильном порядке и (3) осуществляет управление потоком между отправителем и получателем. Использование алгоритма скользящего окна в TCP аналогично его применению на канальном уровне в отношении первых двух функций. Отличие TCP от канального уровня состоит в том, что в него включена функция управления потоком. В частности, вместо фиксированного размера «скользящего окна» получатель сообщает отправителю размер окна. Это делается с использованием поля AdvertisedWindow в заголовке TCP. Отправитель затем ограничен наличием не более чем значением AdvertisedWindow байтов неподтвержденных данных в любой момент времени. Получатель выбирает подходящее значение для AdvertisedWindow на основе объема памяти, выделенной для соединения с целью буферизации данных. Идея заключается в том, чтобы не позволить отправителю переполнить буфер получателя. Мы обсудим это более подробно далее.

Надежная и своевременная доставка

Чтобы понять, как взаимодействуют стороны отправки и получения TCP для реализации надежной и упорядоченной доставки, рассмотрим ситуацию, показанную на рис. 5.8. TCP на стороне отправки поддерживает буфер отправки. Этот буфер используется для хранения данных, которые были отправлены, но еще не подтверждены, а также данных, которые были записаны отправляющим приложением, но еще не переданы. На стороне получения TCP поддерживает буфер приема. Этот буфер хранит данные, которые поступили вне порядка, а также данные, которые находятся в правильном порядке (то есть в потоке нет пропущенных байтов), но которые процесс приложения еще не успел прочитать. Чтобы упростить дальнейшее обсуждение, мы первоначально игнорируем тот факт, что и буферы, и номера последовательностей имеют конечный размер и, следовательно, в конечном итоге будут за циклироваться. Также мы не различаем указатель в буфере, где хранится конкретный байт данных, и номер последовательности для этого байта.

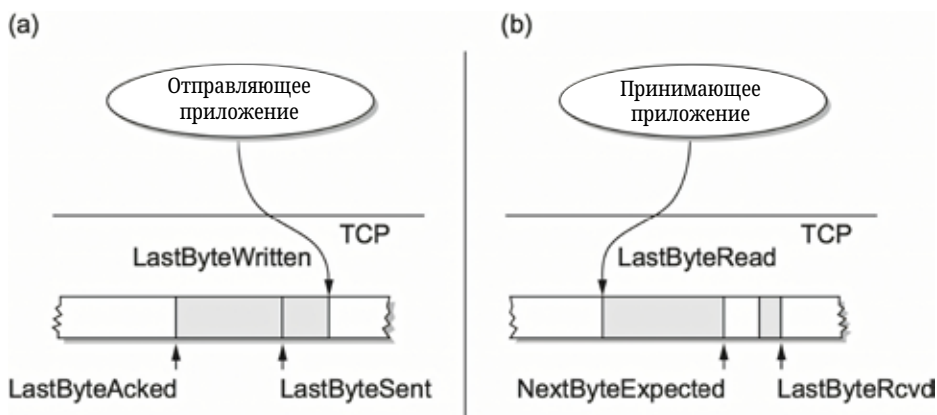


Рисунок 5.8. Взаимосвязь между буфером отправки TCP (a) и буфером получения (b).

Чтобы упростить последующее обсуждение, мы изначально игнорируем тот факт, что и буферы, и номера последовательностей имеют некоторый конечный размер и, следовательно, в конце концов обернутся. Кроме того, мы не выделяем различия между указателем на буфер, в котором хранится конкретный байт данных, и порядковым номером этого байта.

Сначала рассмотрим сторону отправки, где поддерживаются три указателя в буфер отправки, каждый из которых имеет очевидное значение: `LastByteAcked`, `LastByteSent` и `LastByteWritten`. Очевидно,

`LastByteAcked <= LastByteSent`

так как получатель не может подтвердить байт, который еще не был отправлен, и

`LastByteSent <= LastByteWritten`

так как TCP не может отправить байт, который процесс приложения еще не записал. Также обратите внимание, что никакие байты слева от `LastByteAcked` не нужно сохранять в буфере, потому что они уже были подтверждены, и никакие байты справа от `LastByteWritten` не нужно буферизировать, потому что они еще не были сгенерированы.

Аналогичный набор указателей (номеров последовательностей) поддерживается на стороне получения: `LastByteRead`, `NextByteExpected` и `LastByteRcvd`. Однако нера-

венства здесь менее интуитивны из-за проблемы доставки вне порядка. Первое соотношение

$$\text{LastByteRead} < \text{NextByteExpected}$$

верно, потому что байт не может быть прочитан приложением, пока он не получен, и все предшествующие байты также не получены. `NextByteExpected` указывает на байт сразу после последнего байта, который соответствует этому критерию. Второе соотношение,

$$\text{NextByteExpected} \leq \text{LastByteRcvd} + 1$$

так как если данные поступили в нужном порядке, `NextByteExpected` указывает на байт после `LastByteRcvd`, тогда как если данные поступили вне порядка, то `NextByteExpected` указывает на начало первого пробела в данных, как показано на рис. 5.8. Обратите внимание, что байты слева от `LastByteRead` не нужно буферизировать, потому что они уже были прочитаны локальным процессом приложения, и байты справа от `LastByteRcvd` не нужно буферизировать, потому что они еще не поступили.

Управление потоком

Большая часть вышеприведенного обсуждения аналогична тому, что можно найти в стандартном алгоритме скользящего окна; единственная реальная разница заключается в том, что на этот раз мы подробно рассмотрели тот факт, что отправляющие и принимающие процессы приложений соответственно заполняют и опустошают свои локальные буферы. (В предыдущем обсуждении был упущен тот факт, что данные, поступающие от вышестоящего узла, заполняли буфер отправки, а данные, передаваемые на нижестоящий узел, опустошали буфер приема.)

Вы должны убедиться, что поняли данный аспект, прежде чем продолжать, потому что сейчас наступает момент, где два алгоритма существенно различаются. В следующем обсуждении мы снова вводим тот факт, что оба буфера имеют конечный размер, обозначаемый как `MaxSendBuffer` и `MaxRcvBuffer`, хотя нас не волнуют детали их реализации. Иными словами, нас интересует только количество байт, находящихся в буфере, а не то, где именно эти байты хранятся.

Напомним, что в протоколе скользящего окна размер окна задает количество данных, которые можно отправить без ожидания подтверждения от получателя. Таким образом, получатель ограничивает отправителя, объявляя окно, не превышающее объем данных, которые он может буферизовать. Обратите внимание, что TCP на стороне приема должен соблюдать

$$\text{LastByteRcvd} - \text{LastByteRead} \leq \text{MaxRcvBuffer}$$

чтобы избежать переполнения своего буфера. Поэтому он объявляет размер окна

$$\text{AdvertisedWindow} = \text{MaxRcvBuffer} - ((\text{NextByteExpected} - 1) - \text{LastByteRead})$$

что представляет собой количество свободного места, оставшегося в его буфере. По мере поступления данных получатель подтверждает их, если все предыдущие байты также прибыли. Кроме того, `LastByteRcvd` сдвигается вправо (инкрементируется), а это означает, что рекламируемое окно потенциально сокращается. Сократится оно или нет, зависит от того, как быстро локальный процесс приложения потребляет данные. Если локальный процесс считывает данные так же быстро, как они поступают (что приводит к инкрементированию `LastByteRead` с той же скоростью, что и `LastByteRcvd`), то рекламируемое окно остается открытым (то есть `AdvertisedWindow = MaxRcvBuffer`). Однако если процесс приема отстает, возможно, из-за выполнения очень затратной операции на каждом байте данных, которые он считывает, то рекламируемое окно становится все меньше с каждым поступающим сегментом, пока в конечном итоге не дойдет до 0.

TCP на стороне отправки затем должен придерживаться объявленного окна, полученного от получателя. Это означает, что в любой момент времени он должен обеспечивать, чтобы

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{AdvertisedWindow}$$

Другими словами, отправитель вычисляет `EffectiveWindow`, ограничивающее объем данных, которые он может отправить:

$$\text{EffectiveWindow} = \text{AdvertisedWindow} - (\text{LastByteSent} - \text{LastByteAcked})$$

Очевидно, что `EffectiveWindow` должно быть больше 0, прежде чем источник сможет отправить больше данных. Таким образом, возможно, что сегмент прибывает с подтверждением x байт, позволяя отправителю увеличить `LastByteAcked` на x , но поскольку процесс приема не считывал никаких данных, рекламируемое окно теперь на x байт меньше, чем было раньше. В такой ситуации отправитель сможет освободить место в буфере, но не сможет отправить больше данных.

Все это время сторона отправки также должна следить за тем, чтобы локальный процесс приложения не переполнил буфер отправки, то есть

$$\text{LastByteWritten} - \text{LastByteAcked} \leq \text{MaxSendBuffer}$$

Если процесс отправки пытается записать y байт в TCP, но

$$(\text{LastByteWritten} - \text{LastByteAcked}) + y > \text{MaxSendBuffer}$$

то TCP блокирует процесс отправки и не позволяет ему генерировать больше данных.

Теперь можно понять, как медленный процесс приема в конечном итоге останавливает быстрый процесс отправки. Сначала заполняется буфер приема, что означает, что рекламируемое окно уменьшается до 0. Рекламируемое окно размером 0 означает, что сторона отправки не может передавать никакие данные, даже если ранее отправленные данные были успешно подтверждены. В конечном итоге невозможность передавать данные означает, что буфер отправки заполняется, и в конечном итоге это приводит к тому, что TCP блокирует процесс отправки. Как только процесс приема снова начинает считывать данные, TCP на стороне приема может открыть свое окно, что позволяет TCP на стороне отправки передавать данные из своего буфера. Когда эти данные в конечном итоге подтверждаются, `LastByteAcked` инкрементируется, место в буфере, занимаемое этими подтвержденными данными, освобождается, и процесс отправки разблокируется и может продолжить работу.

Остался только один нерешенный вопрос — как сторона отправки узнает, что рекламируемое окно больше не равно 0? Как упоминалось выше, TCP всегда отправляет сегмент в ответ на полученный сегмент данных, и этот ответ содержит последние значения полей `Acknowledge` и `AdvertisedWindow`, даже если эти значения не изменились с последнего раза. Проблема в следующем. Как только сторона приема рекламирует размер окна 0, отправителю не разрешается отправлять больше данных, а это означает, что он не имеет возможности узнать, что рекламируемое окно больше не равно 0, в какой-то момент в будущем. TCP на стороне приема не отправляет спонтанно сегменты без данных; он отправляет их только в ответ на поступающий сегмент данных.

TCP решает эту ситуацию следующим образом. Каждый раз, когда другая сторона рекламирует размер окна 0, сторона отправки продолжает отправлять сегмент с 1 байтом данных через определенные промежутки времени. Она знает, что эти данные, вероятно, не будут приняты, но все равно пытается, потому что каждый из этих 1-байтовых сегментов вызывает ответ, содержащий текущее рекламируемое окно. В конце концов одна из этих 1-байтовых проб вызывает ответ, сообщающий о ненулевом рекламируемом окне.

Заметьте, что эти 1-байтовые сообщения называются *Zero Window Probes*, и на практике они отправляются каждые 5–60 секунд. Что касается того, какой именно байт данных отправить в пробе: это следующий байт реальных данных за пределами окна. (Он должен быть реальными данными на случай, если он будет принят получателем.)

Основные выводы

Заметьте, что причина, по которой сторона отправки периодически отправляет этот пробный сегмент, заключается в том, что TCP спроектирован таким образом, чтобы сделать сторону приема как можно проще — она просто отвечает на сегменты от отправителя и никогда не иницирует никакую активность самостоятельно. Это пример хорошо известного (хотя и не универсально применяемого) правила проектирования протоколов, которое, за неимением лучшего названия, мы называем правилом «умный отправитель/глупый приемник». Напомним, что мы видели другой пример этого правила, когда обсуждали использование NAK в алгоритме «скользящего окна».

Защита от обхода

В этой и следующей подглавах рассматриваются размер полей SequenceNum и AdvertisedWindow и их влияние на корректность и производительность TCP. Поле SequenceNum в TCP имеет длину 32 бита, а поле AdvertisedWindow — 16 бит, и это означает, что TCP легко удовлетворяет требованию алгоритма «скользящего окна», чтобы пространство номеров последовательностей было вдвое больше размера окна: $232 \gg 2 \times 216$. Однако это требование не является самым интересным аспектом этих двух полей. Рассмотрим каждое поле по очереди.

Значимость 32-битного пространства номеров последовательностей заключается в том, что номер последовательности, используемый на данном соединении, может «обернуться» — байт с номером последовательности S может быть отправлен в одно время, а затем позже может быть отправлен второй байт с тем же номером последовательности S. Снова предположим, что пакеты не могут выживать в Интернете дольше рекомендованного MSL. Таким образом, в настоящее время нам нужно убедиться, что номер последовательности не «обернется» в течение 120 секунд. Случится это или нет, зависит от того, как быстро данные могут передаваться по Интернету — то есть как быстро можно исчерпать 32-битное пространство номеров последовательностей. (Это обсуждение предполагает, что мы пытаемся исчерпать пространство номеров последовательностей как можно быстрее, что мы, конечно, будем делать, если будем выполнять свою задачу по полной загрузке канала.) Табл. 5.1 показывает, сколько времени требуется для «обертывания» номера последовательности в сетях с различной пропускной способностью.

Таблица 5.1.

Время до того, как 32-битное пространство номеров последовательностей обернется вокруг

Пропускная способность	Время до «обертывания»
T1 (1.5 Mbps)	6,4 часа
T3 (45 Мбит/с)	13 минут
Быстрый Ethernet (100 Мбит/с)	6 минут
OC-3 (155 Мбит/с)	4 минуты
OC-48 (2,5 Гбит/с)	14 секунд
OC-192 (10 Гбит/с)	3 секунды
10GigE (10 Гбит/с)	3 секунды

Как вы видите, 32-битное пространство номеров последовательностей адекватно при умеренной пропускной способности, но учитывая, что соединения ОС-192 теперь распространены в магистральных сетях Интернета и что большинство серверов теперь оснащены интерфейсами 10Gig Ethernet (или 10 Гбит/с), мы уже давно перешли ту грань, где 32 бита — слишком мало. К счастью, IETF разработала расширение для TCP, которое эффективно расширяет пространство номеров последовательностей для защиты от «обертывания». Это и связанные с ним расширения описаны в следующем разделе.

Поддержка полной загрузки канала

Значимость 16-битного поля AdvertisedWindow заключается в том, что оно должно быть достаточно большим, чтобы позволить отправителю поддерживать полную загрузку канала. Очевидно, что получатель может не открывать окно настолько широко, насколько позволяет поле AdvertisedWindow; нас интересует ситуация, в которой у получателя достаточно буферного пространства для обработки данных в объеме, разрешенном максимальным возможным значением AdvertisedWindow.

В этом случае важна не только пропускная способность сети, но и произведение задержки на пропускную способность, которое диктует, насколько большим должно быть поле AdvertisedWindow — окно должно быть открыто настолько, чтобы позволить передать объем данных, равный произведению задержки на пропускную способность. При предположении, что RTT составляет 100 мс (типичное значение для соединения через всю страну в США), табл. 5.2 показывает произведение задержки на пропускную способность для нескольких сетевых технологий.

Таблица 5.2.
Требуемый размер окна для 100-минутного RTT

пропускная способность	задержка × пропускную способность
T1 (1.5 Mbps)	18 КБ
T3 (45 Мбит/с)	549 КБ
Быстрый Ethernet (100 Мбит/с)	1,2 МБ
ОС-3 (155 Мбит/с)	1,8 МБ
ОС-48 (2,5 Гбит/с)	29,6 МБ
ОС-192 (10 Гбит/с)	118,4 МБ
10GigE (10 Гбит/с)	118,4 МБ

Как видно, поле AdvertisedWindow в TCP находится в еще худшем положении, чем поле SequenceNum — его недостаточно даже для обработки соединения ТЗ через континентальные Соединенные Штаты, так как 16-битное поле позволяет объявлять окно только размером 64 КБ. Расширение TCP, упомянутое выше, предоставляет механизм для эффективного увеличения размера объявляемого окна.

Глава 5.2.5. Триггеры передачи

Теперь рассмотрим весьма непростой вопрос: как TCP решает передать сегмент. Как описано ранее, TCP поддерживает абстракцию потока байтов; то есть программы-приложения записывают байты в поток, и TCP решает, что у него достаточно байтов для отправки сегмента. Какие факторы определяют это решение?

Если игнорировать возможность управления потоком (то есть предположить, что окно полностью открыто, как это бывает при начале соединения), то у TCP есть три механизма для запуска передачи сегмента. Во-первых, TCP поддерживает переменную, обычно называемую *максимальным размером сегмента* (MSS), и отправляет сегмент, как только собирает MSS байтов от процесса отправки. MSS обычно устанавливается на размер самого большого сегмента, который TCP может отправить без фрагментации в локальной IP-сети. То есть MSS устанавливается на максимальную единицу передачи (MTU) в напрямую подключенной сети, минус размер заголовков TCP и IP. Второй фактор, который запускает передачу сегмента в TCP, — это явный запрос на это от процесса отправки. В частности, TCP поддерживает операцию *push*, и процесс отправки вызывает эту операцию, чтобы эффективно очистить буфер от неотправленных байтов. Последний триггер для передачи сегмента — это срабатывание таймера; результатом будет сегмент, содержащий столько байтов, сколько в данный момент буферизовано для передачи. Однако, как мы скоро увидим, этот «таймер» — не совсем то, чего вы ожидаете.

Синдром «глупого окна»

Конечно, мы не можем просто игнорировать управление потоком, которое играет очевидную роль в ограничении отправителя. Если у отправителя есть MSS байтов данных для отправки и окно открыто хотя бы настолько, то отправитель передает полный сегмент. Однако предположим, что отправитель накапливает байты для отправки, но окно в данный момент закрыто. Теперь предположим, что приходит ACK, который эффективно открывает окно, чего достаточно для отправки, скажем, MSS/2 байтов. Должен ли отправитель передать полупустой сегмент или подождать, пока окно не откроется до полного MSS? Первоначальная спецификация не касалась этого момента, и ранние реализации TCP решили передавать полупустой сегмент. В конце концов, невозможно сказать, сколько времени пройдет, прежде чем окно откроется дальше.

Оказывается, стратегия агрессивного использования любого доступного окна приводит к ситуации, известной как *синдром «глупого окна»*. Рис. 5.9 помогает визуализировать, что происходит. Если представить поток TCP как конвейер с «полными» контейнерами (сегменты данных), идущими в одном направлении, и пустыми контейнерами (ACKs), идущими в обратном направлении, то сегменты размера MSS соответствуют большому контейнеру, а сегменты размером 1 байт — очень маленьким контейнерам. Пока отправитель отправляет сегменты размера MSS, а получатель подтверждает хотя бы один MSS данных за раз, все хорошо (рис. 5.9(a)). Но что если получатель должен уменьшить окно так, что в какой-то момент отправитель не сможет отправить полный MSS данных? Если отправитель агрессивно заполняет контейнер меньшего, чем MSS, размера, как только он прибывает, то получатель подтвердит это меньшее количество байтов, и, следовательно, маленький контейнер, введенный в систему, остается в системе на неопределенный срок. То есть он немедленно заполняется и опустошается на каждом конце и никогда не объединяется с соседними контейнерами для создания больших контейнеров, как на рис. 5.9(b). Этот сценарий был обнаружен, когда ранние реализации TCP регулярно заполняли сеть маленькими сегментами.

Заметьте, что синдром «глупого окна» является проблемой только тогда, когда либо отправитель передает маленький сегмент, либо получатель открывает окно на небольшое количество. Если ни то ни другое не происходит, то маленький контейнер никогда не вводится в поток. Невозможно запретить отправку маленьких сегментов; например, приложение может выполнить операцию *push* после отправки одного байта. Однако возможно предотвратить введение маленького контейнера (то есть небольшого открытого окна) со стороны получателя. Правило заключается в том, что после объявления нулевого окна получатель должен дождаться места, равного MSS, прежде чем он объявит открытое окно.

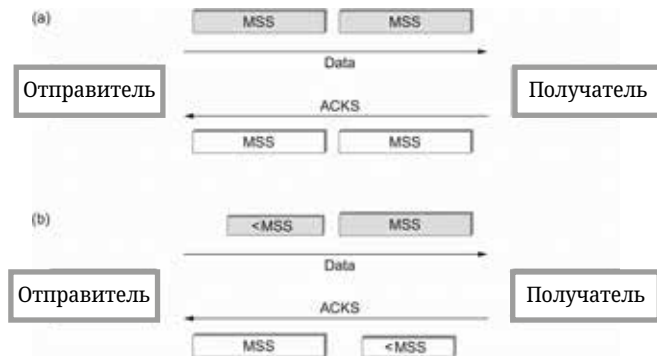


Рисунок 5.9. Синдром «глупого окна». (а) Пока отправитель посылает сегменты размером MSS, а получатель получает ACK по одному MSS за раз, система работает без сбоев. (б) Как только отправитель посылает меньше одного MSS или получатель получает ACK меньше одного MSS, в систему попадает небольшой «контейнер» и продолжает циркулировать.

Поскольку мы не можем исключить возможность появления маленького контейнера в потоке, нам также нужны механизмы для их объединения. Получатель может делать это, задерживая отправку подтверждений (ACKs) (отправляя одно комбинированное подтверждение вместо нескольких маленьких), но это лишь частичное решение, так как у получателя нет способа узнать, как долго можно безопасно задерживать ожидание другого сегмента или чтения большего количества данных приложением (тем самым открывая окно). Окончательное решение ложится на отправителя, что возвращает нас к первоначальному вопросу: когда отправитель TCP решает передать сегмент?

Алгоритм Нейгла

Возвращаясь к отправителю TCP, если есть данные для отправки, но окно открыто менее MSS, то мы можем захотеть подождать некоторое время перед отправкой имеющихся данных, но вопрос в том, как долго? Если мы будем ждать слишком долго, то навредим интерактивным приложениям, таким как Telnet. Если ждать недостаточно долго, то мы рискуем отправить кучу крошечных пакетов и впасть в синдром «глупого окна». Ответ заключается в том, чтобы ввести таймер и передавать данные по его истечении.

Хотя мы могли бы использовать таймер, работающий по часам (например, срабатывающий каждые 100 мс), Нейгл предложил элегантное *самосинхронизирующееся* решение. Идея заключается в том, что пока у TCP есть любые данные в пути, отправитель в конечном итоге получит ACK. Этот ACK можно рассматривать как срабатывание таймера, запускающего передачу дополнительных данных. Алгоритм Нейгла предоставляет простое единое правило для решения вопроса о времени передачи данных:

```

Когда приложение генерирует данные для отправки
  если и доступные данные, и окно  $\geq$  MSS
    отправить полный сегмент
  иначе
    если есть неподтвержденные (unACKed) данные в пути
      буферизовать новые данные до получения ACK
    иначе
      отправить все новые данные сейчас
  
```

Другими словами, всегда можно отправить полный сегмент, если окно это позволяет. Также допустимо немедленно отправить небольшое количество данных, если в данный момент нет сегментов в пути, но если что-то находится в пути, отправитель

должен дожидаться ACK перед передачей следующего сегмента. Таким образом, интерактивное приложение, такое как Telnet, которое постоянно отправляет по одному байту за раз, будет отправлять данные со скоростью один сегмент на RTT. Некоторые сегменты будут содержать один байт, в то время как другие будут содержать столько байтов, сколько пользователь смог напечатать за одно время кругового пути. Поскольку некоторые приложения не могут позволить себе такую задержку при каждой записи в соединение TCP, интерфейс сокетов позволяет приложению отключить алгоритм Нейгла, установив опцию TCP_NODELAY. Установка этой опции означает, что данные передаются как можно скорее.

Глава 5.2.6. Адаптивная ретрансляция (повторная передача)

Поскольку TCP гарантирует надежную доставку данных, он повторно передает каждый сегмент, если ACK не получен в определенный период времени. TCP устанавливает этот тайм-аут как функцию от RTT, который он ожидает между двумя концами соединения. К сожалению, учитывая диапазон возможных RTT между любой парой хостов в Интернете, а также вариации RTT между этими двумя хостами со временем, выбор соответствующего значения тайм-аута не так прост. Чтобы решить эту проблему, TCP использует адаптивный механизм повторной передачи. Теперь мы опишем этот механизм и то, как он эволюционировал с течением времени, по мере того как сообщество Интернета набиралось опыта в использовании TCP.

Оригинальный алгоритм

Мы начинаем с простого алгоритма вычисления значения тайм-аута между парой хостов. Это алгоритм, который был изначально описан в спецификации TCP — и ниже приводится описание в этих терминах, — но он мог бы использоваться любым сквозным протоколом.

Идея заключается в том, чтобы поддерживать текущее среднее значение RTT и затем вычислять тайм-аут как функцию этого RTT. Если говорить конкретно, то каждый раз, когда TCP отправляет сегмент данных, он записывает время. Когда приходит ACK для этого сегмента, TCP снова считывает время и затем определяет разницу между этими двумя временными отметками как `SampleRTT`. TCP затем вычисляет `EstimatedRTT` как взвешенное среднее между предыдущей оценкой и этим новым образцом. То есть:

$$\text{EstimatedRTT} = \alpha \times \text{EstimatedRTT} + (1 - \alpha) \times \text{SampleRTT}$$

Параметр `alpha` выбран для *сглаживания* `EstimatedRTT`. Малая `alpha` отслеживает изменения в RTT, но, возможно, слишком сильно подвержена временным колебаниям. С другой стороны, большое значение `alpha` более стабильно, но, возможно, недостаточно быстро адаптируется к реальным изменениям. Первоначальная спецификация TCP рекомендовала устанавливать `alpha` в пределах от 0.8 до 0.9. TCP затем использует `EstimatedRTT` для вычисления тайм-аута довольно консервативным способом:

$$\text{TimeOut} = 2 \times \text{EstimatedRTT}$$

Алгоритм Карна/Партриджа

Через несколько лет использования в Интернете был обнаружен довольно очевидный изъян в этом простом алгоритме. Проблема заключалась в том, что ACK на самом деле не подтверждает передачу; он подтверждает получение данных. Другими словами, всякий раз, когда сегмент повторно передается и затем приходит ACK к отправителю, невозможно определить, следует ли связывать этот ACK с первой или второй передачей сегмента для измерения `SampleRTT`. Необходимо знать, с какой передачей его связывать, чтобы вычислить точный `SampleRTT`. Как показано на рис. 5.10, если вы предполагаете, что ACK относится к исходной передаче, но на самом деле он относится к другой,

то SampleRTT будет слишком большим (а); если вы предполагаете, что ACK относится ко второй передаче, но на самом деле он относится к первой, то SampleRTT будет слишком маленьким (б).

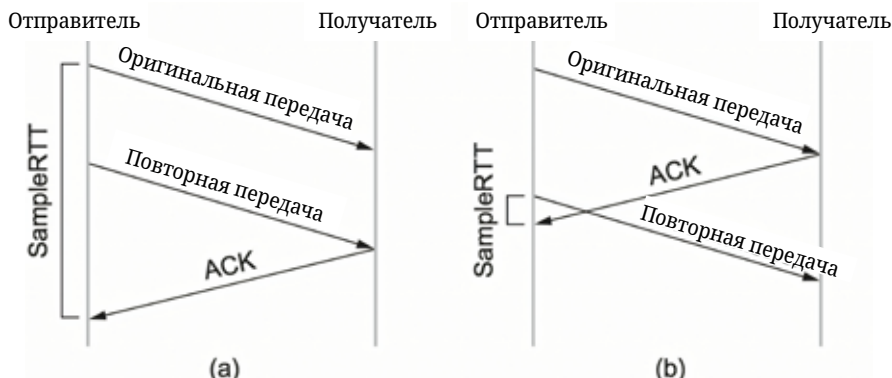


Рисунок 5.10. Ассоциирование ACK с (а) первоначальной передачей и (б) повторной передачей.

Решение, предложенное в 1987 году, удивительно простое. Всякий раз, когда TCP повторно передает сегмент, он прекращает измерение RTT; он измеряет SampleRTT только для сегментов, которые были отправлены лишь один раз. Это решение известно как алгоритм Карна/Парtridge, названный в честь его изобретателей. Их предложенное исправление также включает второе небольшое изменение в механизм тайм-аута TCP. Каждый раз, когда TCP повторно передает данные, он устанавливает следующий тайм-аут в два раза больше предыдущего, а не основываясь на последнем EstimatedRTT. То есть Карн и Парtridge предложили, чтобы TCP использовал экспоненциальное увеличение тайм-аута, аналогично тому, как это делает Ethernet. Мотивация использования экспоненциального увеличения проста: перегрузка является наиболее вероятной причиной потерь сегментов, что означает, что источник TCP не должен реагировать слишком агрессивно на тайм-аут. На самом деле чем больше раз соединение истекает по времени, тем более осторожным должен становиться источник. Мы снова увидим эту идею, воплощенную в гораздо более сложном механизме, в следующем разделе.

Алгоритм Джейкобсона/Карелса

Алгоритм Карна/Парtridge был введен в то время, когда в Интернете наблюдался высокий уровень сетевой перегрузки. Их подход был разработан для устранения некоторых причин этой перегрузки, но хотя это было улучшением, перегрузка не была устранена. В следующем году (1988) два других исследователя — Джейкобсон и Карелс — предложили более радикальное изменение TCP для борьбы с перегрузкой. Основная часть этого предложенного изменения описана в следующем разделе. Здесь мы сосредоточимся на аспекте этого предложения, связанном с решением о том, когда следует завершить таймаут и повторно передать сегмент.

Для начала следует понять, как механизм тайм-аута связан с перегрузкой — если вы завершаете тайм-аут слишком рано, вы можете без необходимости повторно передать сегмент, что только увеличит нагрузку на сеть. Другая причина, по которой нужно точное значение тайм-аута, заключается в том, что тайм-аут подразумевает перегрузку, что запускает механизм управления перегрузкой. Наконец, отметим, что в вычислении тайм-аута по Джейкобсону/Карелсу нет ничего специфичного для TCP. Оно может использоваться любым сквозным протоколом.

Основная проблема исходного вычисления заключается в том, что оно не учитывает разброс (дисперсию) выборочных значений RTT. Интуитивно, если разброс среди вы-

борок мал, то EstimatedRTT можно доверять больше, и нет необходимости умножать эту оценку на 2 для вычисления тайм-аута. С другой стороны, большой разброс в выборках предполагает, что значение тайм-аута не должно быть слишком тесно связано с EstimatedRTT.

В новом подходе отправитель измеряет новое значение SampleRTT, как и раньше. Затем он учитывает этот новый образец в вычислении тайм-аута следующим образом:

```
Difference = SampleRTT - EstimatedRTT
EstimatedRTT = EstimatedRTT + (delta x Difference)
Deviation = Deviation + delta (|Difference| - Deviation)
```

где delta — это значение в пределах от 0 до 1. То есть мы рассчитываем как среднее значение RTT, так и вариацию этого среднего.

TCP затем вычисляет значение тайм-аута как функцию, как EstimatedRTT, так и Deviation, следующим образом:

```
Timeout = mu x EstimatedRTT + phi x Deviation
```

где на основе опыта μ обычно устанавливается равным 1, а ϕ устанавливается равным 4. Таким образом, когда дисперсия мала, Timeout близок к EstimatedRTT; большая дисперсия вызывает доминирование термина Deviation в расчетах.

Реализация

Есть два важных момента, касающихся реализации тайм-аутов в TCP. Первый заключается в том, что можно реализовать вычисление EstimatedRTT и Deviation без использования арифметики с плавающей запятой. Вместо этого все вычисление масштабируется на $2n$, с delta, выбранной как $1/2n$. Это позволяет выполнять целочисленные вычисления, реализуя умножение и деление с помощью сдвигов, тем самым достигая более высокой производительности. Полученное вычисление представлено следующим фрагментом кода, где $n=3$ (т.е. $\text{delta} = 1/8$). Заметьте, что EstimatedRTT и Deviation хранятся в их масштабированных формах, в то время как значение SampleRTT в начале кода и значение Timeout в конце являются реальными немасштабированными значениями. Если вам сложно следовать за кодом, попробуйте подставить какие-нибудь реальные числа, и вы убедитесь, что он дает те же результаты, что и уравнения выше.

```
{
    SampleRTT -= (EstimatedRTT >> 3);
    EstimatedRTT += SampleRTT;
    if (SampleRTT < 0)
        SampleRTT = -SampleRTT;
    SampleRTT -= (Deviation >> 3);
    Deviation += SampleRTT;
    Timeout = (EstimatedRTT >> 3) + (Deviation >> 1);
}
```

Второй момент заключается в том, что алгоритм Джейкобсона/Карелса настолько хорош, насколько хороши часы, используемые для чтения текущего времени. В типичных реализациях Unix того времени гранулярность часов составляла до 500 мс, что значительно больше, чем среднее время кругового прохода (RTT) от 100 до 200 мс. Что еще хуже, реализация TCP в Unix проверяла, нужно ли срабатывать тайм-ауту, только каждые 500 мс и брала образец времени кругового прохода только один раз за RTT. Сочетание этих двух факторов могло означать, что тайм-аут происходил через 1 секунду после передачи сегмента. Опять же расширения TCP включают механизм, который делает расчет RTT немного точнее.

Все обсуждаемые алгоритмы повторной передачи основаны на тайм-аутах подтверждений, которые указывают на то, что сегмент, вероятно, был потерян. Заметьте, что тайм-аут, однако, не говорит отправителю, были ли успешно получены какие-либо сегменты, отправленные после потерянного сегмента. Это связано с тем, что подтверждения TCP являются кумулятивными; они идентифицируют только последний сегмент, полученный без каких-либо предшествующих разрывов. Прием сегментов, которые происходят после разрыва, становится все более частым по мере того, как более быстрые сети приводят к увеличению окон. Если бы АСК также сообщали отправителю, какие последующие сегменты, если таковые имеются, были получены, то отправитель мог бы более разумно подходить к тому, какие сегменты он повторно передает, делать лучшие выводы о состоянии перегрузки и делать более точные оценки RTT. Расширение TCP, поддерживающее это, описано в следующей главе.

Основные выводы

Есть еще один момент, касающийся вычисления тайм-аутов. Это удивительно сложное дело, настолько сложное, что существует целый RFC, посвященный этой теме: RFC 6298. Основной вывод заключается в том, что иногда полное определение протокола включает в себя столько мелочей, что грань между спецификацией и реализацией становится размытой. Это случалось не раз с TCP, что вызывает у некоторых мнение, что «реализация — это спецификация». Но это не обязательно плохо, пока эталонная реализация доступна в виде открытого программного обеспечения. В более общем смысле, индустрия наблюдает рост важности открытого программного обеспечения, тогда как значение открытых стандартов уменьшается.

Глава 5.2.7. Границы записей

Поскольку TCP является протоколом потока байтов, количество байтов, записанных отправителем, не обязательно совпадает с количеством байтов, считанных получателем. Например, приложение может записать 8 байтов, затем 2 байта, затем 20 байтов в соединение TCP, в то время как на стороне получения приложение читает по 5 байтов за раз в цикле, который повторяется 6 раз. TCP не вставляет границы записей между 8-м и 9-м байтами, а также между 10-м и 11-м байтами. Это контрастирует с протоколом, ориентированным на сообщения, таким как UDP, в котором сообщение, которое отправлено, точно такой же длины, как и сообщение, которое получено.

Несмотря на то, что TCP является протоколом потока байтов, у него есть две различные функции, которые могут использоваться отправителем для вставки границ записей в этот поток байтов, тем самым информируя получателя о том, как разбить поток байтов на записи. (Возможность пометить границы записей полезна, например, во многих приложениях баз данных.) Обе эти функции изначально были включены в TCP по совершенно другим причинам, они начали использоваться для этой цели только со временем.

Первый механизм — это функция срочных данных, реализованная с помощью флага URG и поля UrgPtr в заголовке TCP. Изначально механизм срочных данных был разработан для того, чтобы позволить отправляющему приложению отправлять внеполосные данные своему партнеру. Под «внеполосными» понимаются данные, которые отделены от нормального потока данных (например, команда прерывания уже выполняемой операции). Эти внеполосные данные идентифицировались в сегменте с использованием поля UrgPtr и должны были доставляться принимающему процессу сразу после прибытия, даже если это означало доставку перед данными с более ранним порядковым номером. Со временем, однако, эта функция перестала использоваться, и вместо того, чтобы обозначать «срочные» данные, она стала использоваться для обозначения «особых» данных, таких как маркер записи. Это использование развилось потому, что, как и при операции push, TCP на принимающей стороне должен информировать приложение о прибытии срочных данных. То есть сами по себе срочные данные не важны. Важно то, что отправляющий процесс может эффективно отправить сигнал получателю.

Второй механизм для вставки маркеров конца записи в поток байтов — это операция *push*. Изначально этот механизм был разработан для того, чтобы позволить отправляющему процессу сообщать TCP, что он должен отправить (сбросить) любые байты, которые он собрал, своему партнеру. Операция *push* может использоваться для реализации границ записей, потому что спецификация говорит, что TCP должен отправить все данные, которые он буферизовал на источнике, когда приложение говорит *push*, и, опционально, TCP на приемной стороне уведомляет приложение всякий раз, когда входящий сегмент имеет установленный флаг *PUSH*. Если принимающая сторона поддерживает эту опцию (интерфейс сокетов этого не делает), то операция *push* может использоваться для разделения потока TCP на записи.

Конечно, программа приложения всегда может вставлять границы записей без какой-либо помощи от TCP. Например, она может отправить поле, которое указывает длину записи, которая должна последовать, или она может вставлять собственные маркеры границ записей в поток данных.

Глава 5.2.8. Расширения TCP

Мы упомянули в четырех различных местах в этой главе, что существуют расширения TCP, помогающие смягчить некоторые проблемы, с которыми сталкивался TCP по мере увеличения скорости основной сети. Эти расширения разработаны таким образом, чтобы как можно меньше воздействовать на TCP. В частности, они реализованы как опции, которые могут быть добавлены в заголовок TCP. (Мы ранее упустили этот момент, но причина, по которой заголовок TCP имеет поле *HdrLen*, заключается в том, что заголовок может быть переменной длины; переменная часть заголовка TCP содержит добавленные опции.) Значимость добавления этих расширений в качестве опций, а не изменения ядра заголовка TCP, заключается в том, что хосты могут продолжать общаться с использованием TCP, даже если они не реализуют эти опции. Однако хосты, которые реализуют опциональные расширения, могут воспользоваться ими. Две стороны соглашаются использовать опции во время фазы установления соединения TCP.

Первое расширение помогает улучшить механизм тайм-аута TCP. Вместо того чтобы измерять RTT с использованием грубого события, TCP может считывать текущее системное время, когда он собирается отправить сегмент, и записывать это время (представим его как 32-битную *временную метку*) в заголовок сегмента. Получатель затем эхо-ответом возвращает этот временной штамп отправителю в своем подтверждении, и отправитель вычитает этот временной штамп из текущего времени, чтобы измерить RTT. По сути, опция временного штампа предоставляет удобное место TCP для хранения записи о том, когда сегмент был передан; она сохраняет время в самом сегменте. Заметьте, что конечные точки соединения не нуждаются в синхронизированных часах, так как временной штамп записывается и считывается на одном и том же конце соединения.

Второе расширение решает проблему слишком быстрого переполнения 32-битного поля *SequenceNum* в сети с высокой скоростью. Вместо того чтобы определять новое 64-битное поле номера последовательности, TCP использует описанный выше 32-битный временной штамп для эффективного расширения пространства номеров последовательности. Другими словами, TCP решает, принимать или отклонять сегмент, на основе 64-битного идентификатора, который имеет поле *SequenceNum* в младших 32 битах и временной штамп в старших 32 битах. Поскольку временной штамп всегда увеличивается, он служит для определения различия двух разных инкарнаций одного и того же номера последовательности. Заметьте, что временной штамп используется в этом контексте только для защиты от переполнения; он не рассматривается как часть номера последовательности для целей упорядочивания или подтверждения данных.

Третье расширение позволяет TCP рекламировать большее окно, тем самым позволяя ему заполнять большие каналы производства задержки и пропускной способности,

которые становятся возможными благодаря высокоскоростным сетям. Это расширение включает опцию, которая определяет *коэффициент масштабирования* для рекламируемого окна. То есть вместо того чтобы интерпретировать число, которое появляется в поле `AdvertisedWindow` и указывающее, сколько байтов отправителю разрешено иметь неподтвержденными, эта опция позволяет двум сторонам TCP договориться о том, что поле `AdvertisedWindow` учитывает большие блоки (например, сколько 16-байтных единиц данных отправителю разрешено иметь неподтвержденными). Другими словами, опция масштабирования окна указывает, на сколько битов каждая сторона должна сдвигать влево поле `AdvertisedWindow` перед использованием его содержимого для вычисления эффективного окна.

Четвертое расширение позволяет TCP дополнить свои кумулятивные подтверждения выборочными подтверждениями любых дополнительных сегментов, которые были получены, но не являются смежными со всеми ранее полученными сегментами. Это опция *выборочного подтверждения*, или `SACK`. Когда используется опция `SACK`, получатель продолжает подтверждать сегменты обычным образом (значение поля `Acknowledge` не изменяется) но он также использует опциональные поля в заголовке для подтверждения любых дополнительных блоков полученных данных. Это позволяет отправителю повторно передавать только те сегменты, которые отсутствуют согласно выборочному подтверждению.

Без `SACK` у отправителя есть только две разумные стратегии. Пессимистическая стратегия отвечает на тайм-аут, повторно отправляя не только сегмент, который вызвал тайм-аут, но и все сегменты, отправленные после него. По сути, пессимистическая стратегия предполагает худший сценарий: что все эти сегменты были потеряны. Недостаток пессимистической стратегии заключается в том, что она может ненужно повторно отправлять сегменты, которые были успешно получены с первого раза. Другая стратегия — оптимистическая, которая отвечает на тайм-аут, повторно отправляя только сегмент, который вызвал тайм-аут. По сути, оптимистическая стратегия предполагает самый благоприятный сценарий, то есть мы допускаем, что потерял только один сегмент. Недостаток оптимистической стратегии заключается в том, что она очень медленная, когда серия последовательных сегментов была потеряна, что может произойти при перегрузке. Это происходит потому, что потеря каждого сегмента обнаруживается только тогда, когда отправитель получает `ACK` за повторную передачу предыдущего сегмента. Таким образом, на каждый сегмент уходит одно время круговой задержки (`RTT`), пока не будут повторно переданы все сегменты в потерянной серии. С опцией `SACK` доступна лучшая стратегия для отправителя: повторно отправлять только те сегменты, которые заполняют пробелы между сегментами, которые были выборочно подтверждены.

Эти расширения, кстати, не являются полным рассуждением. Мы увидим еще несколько расширений в следующей главе, когда рассмотрим, как TCP справляется с перегрузками. `Internet Assigned Numbers Authority` (`IANA`, Управление по присвоению номеров в Интернете) отслеживает все опции, определенные для TCP (и для многих других интернет-протоколов).

Глава 5.2.9. Производительность

Напомним, что в первом разделе были введены две количественные метрики, по которым оценивается производительность сети: задержка и пропускная способность. Как упоминалось в этом обсуждении, эти метрики зависят не только от основной аппаратуры (например, задержки распространения и пропускной способности канала), но и от программных накладных расходов. Теперь, когда у нас есть полный программный протокол, включающий альтернативные транспортные протоколы, мы можем обсудить, как осмысленно измерять его производительность. Важность таких измерений заключается в том, что они представляют производительность, которую видят прикладные программы.

Мы начнем, как и должен начинаться любой отчет о результатах экспериментов, с описания нашего экспериментального метода. Это включает аппаратуру, использованную в экспериментах; в данном случае каждая рабочая станция имеет пару двухъядерных процессоров Хеоп с тактовой частотой 2,4 ГГц, работающих под управлением Linux. Для обеспечения скоростей выше 1 Гбит/с на каждой машине используется пара Ethernet-адаптеров (обозначенных как NIC, сетевая интерфейсная карта). Ethernet охватывает одну машинную комнату, поэтому распространение не является проблемой, что делает это измерением процессорных/программных накладных расходов. Тестовая программа, работающая поверх интерфейса сокетов, просто пытается передать данные как можно быстрее с одной машины на другую. Рис. 5.11 иллюстрирует настройку.

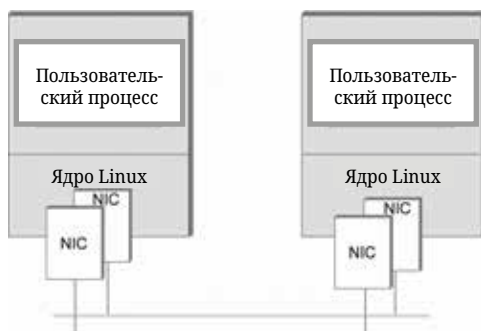


Рисунок 5.11. Измеренная система: две рабочие станции Linux и пара Гбит/с Ethernet-каналов.

Вы можете заметить, что эта экспериментальная установка не является передовой с точки зрения аппаратуры или скорости канала. Цель этой главы не показать, как быстро может работать определенный протокол, а иллюстрировать общую методологию измерения и отчетности о производительности протокола.

Тест пропускной способности проводится для различных размеров сообщений с использованием стандартного инструмента тестирования производительности под названием ТТСР. Результаты теста пропускной способности приведены на рис. 5.12. Ключевой момент, который нужно заметить на этом графике, заключается в том, что пропускная способность улучшается по мере увеличения размеров сообщений. Это логично — каждое сообщение включает определенное количество накладных расходов, поэтому основная мысль — эти накладные расходы распределяются на большее количество байтов. Кривая пропускной способности выравнивается выше 1 КБ, в этот момент накладные расходы на сообщение становятся незначительными по сравнению с большим количеством байтов, которые протокольный стек должен обработать.

Стоит отметить, что максимальная пропускная способность меньше 2 Гбит/с — доступной скорости канала в этой настройке. Для определения узкого места (или нескольких) потребуются дополнительное тестирование и анализ результатов. Например, анализ загрузки ЦП может дать представление о том, является ли ЦП узким местом или же проблема связана с пропускной способностью памяти, производительностью адаптера или чем-то другим.

Мы также отмечаем, что сеть в этом тесте практически «идеальна». В ней почти нет задержек или потерь, поэтому единственными факторами, влияющими на производительность, являются реализация ТСП и аппаратное и программное обеспечение рабочей станции. В отличие от этого, в большинстве случаев мы имеем дело с сетями, которые далеки от идеальных, особенно с ограниченными по пропускной способности каналами последней мили и подверженными потерям беспроводными соединениями. Прежде чем мы сможем полностью оценить, как эти каналы влияют на производительность ТСП, нам нужно понять, как ТСП справляется с *перегрузками*, что является темой следующего раздела.

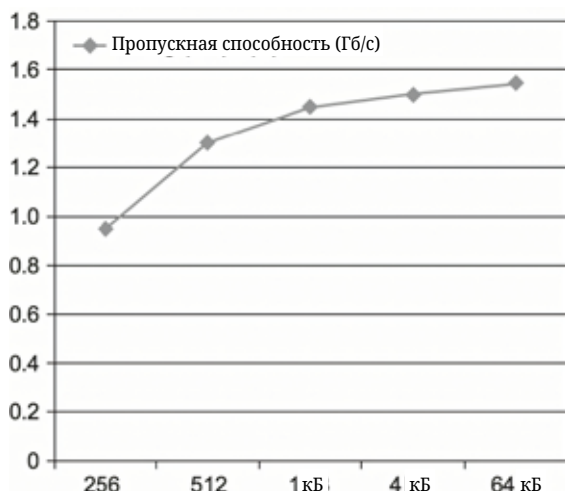


Рисунок 5.12. Измеренная пропускная способность с помощью TSP для различных размеров сообщений.

В разные времена в истории сетей постоянно увеличивающаяся скорость сетевых каналов угрожала опережать возможности доставки данных приложениям. Например, в 1989 году в Соединенных Штатах началась крупная исследовательская работа по созданию «гигабитных сетей», цель которой заключалась не только в создании каналов и коммутаторов, способных работать на скорости 1 Гбит/с и выше, но и в обеспечении такой пропускной способности до одного прикладного процесса. Существовали реальные проблемы (например, сетевые адаптеры, архитектуры рабочих станций и операционные системы должны были быть разработаны с учетом сетевой до приложений пропускной способности), а также некоторые кажущиеся проблемы, которые оказались не столь серьезными. Одной из таких проблем было беспокойство о том, что существующие транспортные протоколы, особенно TSP, могут не справиться с задачей гигабитной работы.

Как оказалось, TSP хорошо справляется с возрастающими требованиями высокоскоростных сетей и приложений. Одним из самых важных факторов стало введение масштабирования окна для работы с большими произведениями пропускной способности и задержки. Однако часто существует большая разница между теоретической производительностью TSP и тем, что достигается на практике. Относительно простые проблемы, такие как излишнее копирование данных при их передаче от сетевого адаптера к приложению, могут снизить производительность, как и недостаточный объем буферной памяти при больших произведениях пропускной способности и задержки. И динамика TSP достаточно сложна (как станет еще более очевидным в следующем разделе), поэтому тонкие взаимодействия между поведением сети, поведением приложений и самим протоколом TSP могут значительно изменить производительность.

Для наших целей стоит отметить, что TSP продолжает показывать очень хорошие результаты по мере увеличения скорости сетей, и когда он сталкивается с какими-либо ограничениями (обычно связанными с перегрузками, увеличением произведений пропускной способности и задержки или обоими факторами), программисты спешат найти решения. Мы видели некоторые из них в этом разделе и увидим в следующем.

Глава 5.2.10. Альтернативные варианты дизайна (SCTP, QUIC)

Хотя TSP зарекомендовал себя как надежный протокол, удовлетворяющий потребности широкого круга приложений, пространство для проектирования транспортных протоколов весьма обширно. TSP ни в коем случае не является единственной допустимой

точкой в этом пространстве. Мы завершаем наше обсуждение TCP рассмотрением альтернативных вариантов дизайна. Хотя мы предлагаем объяснение, почему разработчики TCP приняли именно такие решения, мы отмечаем, что существуют другие протоколы, которые сделали другие выборы, и в будущем могут появиться новые подобные протоколы.

Во-первых, с самой первой главы этой книги мы предположили, что существует как минимум два интересных класса транспортных протоколов: потоково-ориентированные протоколы, такие как TCP, и протоколы запросов/ответов, такие как RPC. Другими словами, мы неявно разделили пространство проектирования пополам и поместили TCP прямо в потоково-ориентированную половину. Далее мы можем разделить потоко-ориентированные протоколы на две группы — надежные и ненадежные, причем первые содержат TCP, а вторые больше подходят для интерактивных видеоприложений, которые предпочитают отбрасывать кадры, а не нести задержки, связанные с повторной передачей.

Это упражнение по построению таксономии транспортных протоколов интересно и могло бы продолжаться во все возрастающей детализации, но мир не настолько черно-белый, как нам бы хотелось. Рассмотрим пригодность TCP как транспортного протокола для приложений «запрос/ответ», например. TCP является полнодуплексным протоколом, поэтому было бы легко установить TCP-соединение между клиентом и сервером, отправить запрос в одном направлении и отправить ответ в другом. Однако есть две сложности. Первая заключается в том, что TCP является *побайтовым* протоколом, а не протоколом, ориентированным на сообщения, тогда как приложения «запрос/ответ» всегда работают с сообщениями. (Мы рассмотрим проблему байтов против сообщений более подробно позже.) Вторая сложность заключается в том, что в тех случаях, когда и запрос, и ответ умещаются в один сетевой пакет, хорошо спроектированный протокол «запрос/ответ» нуждается всего в двух пакетах для реализации обмена, тогда как TCP потребует по крайней мере девять: три для установления соединения, два для обмена сообщениями и четыре для завершения соединения. Конечно, если запрос или ответные сообщения достаточно большие, чтобы требовать нескольких сетевых пакетов (например, может потребоваться 100 пакетов для отправки ответа размером 100 000 байт), то накладные расходы на установку и завершение соединения незначительны. Иными словами, не всегда протокол не может поддерживать определенную функциональность; иногда бывает так, что один дизайн более эффективен, чем другой, в определенных условиях.

Во-вторых, как только что предпологалось, можно задать вопрос, почему TCP выбрал предоставление надежного побайтового сервиса, а не надежного потокового сервиса сообщений; сообщения были бы естественным выбором для базы данных, которая хочет обмениваться записями. На этот вопрос есть два ответа. Первый заключается в том, что протокол, ориентированный на сообщения, по определению должен устанавливать верхнюю границу размера сообщений. В конце концов, бесконечно длинное сообщение — это поток байтов. Для любого размера сообщения, который выберет протокол, будут приложения, которые захотят отправить более крупные сообщения, что сделает транспортный протокол бесполезным и заставит приложение реализовать свои собственные транспортноподобные сервисы. Вторая причина заключается в том, что, хотя протоколы, ориентированные на сообщения, определенно более подходящи для приложений, которые хотят отправлять записи друг другу, вы можете легко вставить границы записей в поток байтов для реализации этой функциональности.

Третье решение, принятое при разработке TCP, заключается в том, что он доставляет приложению байты *по порядку*. Это означает, что он может задерживать байты, полученные из сети вне порядка, ожидая, пока пропущенные байты заполнят пробел. Это чрезвычайно полезно для многих приложений, но оказывается совершенно бесполезным, если приложение способно обрабатывать данные вне порядка. В качестве простого примера можно привести веб-страницу, содержащую несколько встроенных объектов, для отображения которой нет необходимости доставлять все объекты по порядку. Фактически существует класс приложений, которые предпочли бы обрабатывать данные вне порядка на уровне приложения, чтобы получать данные быстрее, когда паке-

ты теряются или перепутываются в сети. Желание поддерживать такие приложения привело к созданию не одного, а двух стандартных транспортных протоколов IETF. Первым из них был SCTP, или *протокол передачи управления потоком* (Stream Control Transmission Protocol). SCTP предоставляет услугу частично упорядоченной доставки, а не строго упорядоченной, как TCP. (SCTP также принимает некоторые другие решения по дизайну, отличающиеся от TCP, включая ориентацию на сообщения и поддержку нескольких IP-адресов для одной сессии.) В последнее время IETF стандартизирует протокол, оптимизированный для веб-трафика, известный как QUIC. Подробнее о QUIC чуть позже.

Четвертое, TCP выбрал реализацию явных фаз установки/завершения соединения, но это не обязательно. В случае установки соединения можно было бы отправить все необходимые параметры соединения вместе с первым сообщением данных. TCP решил выбрать более консервативный подход, который дает получателю возможность отклонить соединение до того, как придут какие-либо данные. В случае завершения соединения можно было бы тихо закрыть соединение, которое было неактивно в течение длительного времени, но это усложнило бы приложения, такие как удаленный вход в систему, которые хотят поддерживать соединение в течение недель; такие приложения были бы вынуждены отправлять «keep alive» сообщения вне канала связи, чтобы состояние соединения на другом конце не исчезло.

Наконец, TCP является протоколом, основанным на окнах, но это не единственная возможность. Альтернатива — дизайн, *основанный на скорости*, при котором получатель сообщает отправителю скорость, выраженную в байтах или пакетах в секунду, с которой он готов принимать входящие данные. Например, получатель может сообщить отправителю, что он может обрабатывать 100 пакетов в секунду. Существует интересная двойственность между окнами и скоростью, так как количество пакетов (байтов) в окне, деленное на RTT, является именно скоростью. Например, размер окна в 10 пакетов и RTT в 100 мс означает, что отправитель может передавать данные со скоростью 100 пакетов в секунду. Путем увеличения или уменьшения размера объявленного окна получатель фактически повышает или понижает скорость, с которой отправитель может передавать данные. В TCP эта информация передается обратно отправителю в поле `AdvertisedWindow` ACK для каждого сегмента. Одним из ключевых вопросов в протоколе, основанном на скорости, является частота передачи желаемой скорости, которая может изменяться со временем, обратно к источнику: для каждого пакета, раз в RTT или только при изменении скорости? Хотя мы только что рассмотрели окна и скорость в контексте управления потоком, это еще более горячо обсуждаемый вопрос в контексте управления перегрузками, который мы обсудим в следующем разделе.

QUIC

QUIC, *Quick UDP Internet Connections*, был разработан в Google в 2012 году и на момент написания этой книги все еще проходит стандартизацию в IETF. Он уже был развернут в умеренном количестве (в некоторых веб-браузерах и на довольно большом числе популярных веб-сайтов). Сам факт того, что он добился такого успеха, является интересной частью истории QUIC, и возможность развертывания была ключевым фактором для разработчиков протокола.

Мотивация для разработки QUIC напрямую связана с вышеупомянутыми недостатками TCP: некоторые проектные решения оказались неоптимальными для ряда приложений, использующих TCP, особенно для HTTP (веб-трафика). Эти проблемы стали более заметными со временем из-за таких факторов, как рост высоколатентных беспроводных сетей, доступность множества сетей для одного устройства (например, Wi-Fi и сотовая связь) и растущее использование зашифрованных, аутентифицированных соединений в интернете. Хотя полное описание QUIC выходит за рамки нашего обсуждения, стоит рассмотреть некоторые ключевые проектные решения.

Многоканальный (Multipath) TCP

- Не всегда нужно определять новый протокол, если существующий протокол не полностью удовлетворяет конкретным требованиям. Иногда можно существенно изменить способ реализации существующего протокола, оставаясь при этом в рамках оригинальной спецификации. Multipath TCP (MPTCP) является примером такой ситуации.
- Идея MPTCP заключается в маршрутизации пакетов по нескольким путям через Интернет, например, используя два разных IP-адреса для одного из конечных узлов. Это может быть особенно полезно при доставке данных на мобильное устройство, подключенное как к Wi-Fi, так и к сотовой сети (и, следовательно, имеющее два уникальных IP-адреса). Оба сетевых соединения, будучи беспроводными, могут испытывать значительные потери пакетов, поэтому возможность использования обоих соединений для передачи пакетов может значительно улучшить пользовательский опыт. Ключевым моментом является реконструкция оригинального, упорядоченного байтового потока на принимающей стороне TCP до передачи данных приложению, которое остается неосведомленным о том, что оно работает поверх MPTCP. (Это контрастирует с приложениями, которые намеренно открывают два или более TCP-соединения для улучшения производительности.)
- Как бы просто ни звучала концепция MPTCP, ее реализация является невероятно сложной, так как нарушает многие предположения о том, как реализованы управление потоком TCP, пересборка сегментов по порядку и управление перегрузками. Мы оставляем изучение этих тонкостей на усмотрение читателя. Это отличный способ убедиться в том, что ваше базовое понимание TCP является верным.

Если задержка в сети высока (в сотни миллисекунд), то несколько RTT могут быстро привести к заметному раздражению у конечного пользователя. Установка HTTP-сеанса поверх TCP с использованием Transport Layer Security (TLS) обычно занимает три раунда обмена (один для установки сеанса TCP и два для настройки параметров шифрования) до того, как можно будет отправить первое HTTP-сообщение. Разработчики QUIC признали, что эта задержка — прямой результат многослойного подхода к проектированию протоколов — может быть значительно уменьшена, если соединение и необходимые обмены протоколами безопасности будут объединены и оптимизированы для минимального количества раундов обмена.

Обратите также внимание на то, как наличие нескольких сетевых интерфейсов может повлиять на проектирование. Если ваш мобильный телефон теряет соединение с Wi-Fi и должен переключиться на сотовое соединение, это обычно требует как тайм-аута TCP на одном соединении, так и новой серии обменов на другом. Создание соединения, которое может сохраняться при переключении между различными сетевыми соединениями, было еще одной целью разработки QUIC.

Наконец, как было упомянуто выше, модель надежного байтового потока в TCP плохо подходит для запроса веб-страницы, когда необходимо получить множество объектов, и отображение страницы могло бы начаться до того, как все объекты будут получены. Хотя одно из решений — открытие нескольких параллельных TCP-соединений, этот подход (использовавшийся в первые дни сети интернета) имеет свои недостатки, особенно в плане управления перегрузками.

Интересно, что к моменту появления QUIC многие проектные решения сделали сложной задачу развертывания нового транспортного протокола. В частности, многие «промежуточные устройства», такие как NAT и файерволы, имеют достаточное понимание существующих широко распространенных транспортных протоколов (TCP и UDP), чтобы нельзя было полагаться на их пропуск нового транспортного протокола. В результате QUIC фактически работает поверх UDP. Иными словами, это транспортный протокол, работающий поверх другого транспортного протокола. Это не так уж необычно, как может показаться при нашем фокусе на многослойности, как иллюстрируют следующие две главы.

QUIC реализует быстрое установление соединения с шифрованием и аутентификацией в первом цикле RTT. Он обеспечивает идентификатор соединения, который сохраняется при изменениях в подлежащей сети. Он поддерживает мультиплексирование нескольких потоков в одном транспортном соединении, чтобы избежать блокировки начала очереди, которая может возникнуть, когда один пакет теряется, в то время как другие полезные данные продолжают поступать. И он сохраняет свойства избежания перегрузок TSP — важный аспект транспортных протоколов, к которому мы вернемся в шестом разделе.

QUIC является самой интересной разработкой в мире транспортных протоколов. Многие ограничения TSP были известны десятилетиями, но QUIC представляет собой одну из самых успешных попыток на сегодняшний день занять иную позицию в пространстве дизайна. Поскольку QUIC был вдохновлен опытом работы с HTTP и Web, которые возникли намного позже, чем TSP утвердился в Интернете, он представляет собой увлекательное исследование непредвиденных последствий многоуровневых дизайнов и эволюции Интернета.

Глава 5.3. Вызов удаленной процедуры

Общая схема общения, используемая прикладными программами, структурированными как пара *клиент/сервер*, — это транзакция сообщения запроса/ответа: клиент отправляет сообщение-запрос серверу, и сервер отвечает сообщением-ответом, при этом клиент блокируется (приостанавливает выполнение) в ожидании ответа. Рис. 5.13 иллюстрирует основное взаимодействие между клиентом и сервером в таком обмене.

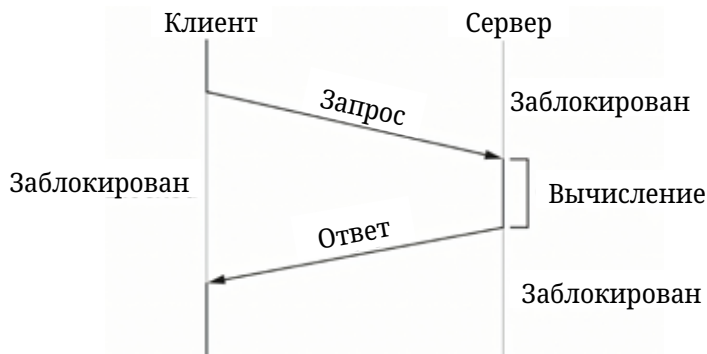


Рисунок 5.13. Временная шкала для RPC.

Транспортный протокол, который поддерживает парадигму запроса/ответа, это намного больше, чем UDP-сообщение в одном направлении, за которым следует UDP-сообщение в другом направлении. Он должен корректно идентифицировать процессы на удаленных хостах и сопоставлять запросы с ответами. Также может потребоваться преодоление некоторых или всех ограничений подлежащей сети, описанных в заявлении о проблеме в начале этого раздела. В то время как TSP преодолевает эти ограничения, предоставляя надежный байтовый потоковый сервис, он тоже не идеально соответствует парадигме запроса/ответа. Следующая глава описывает третью категорию транспортного протокола, называемую *вызов удаленной процедуры* (Remote Procedure Call, RPC), которая более точно соответствует потребностям приложения, участвующего в обмене сообщениями «запрос/ответ».

Глава 5.3.1. Основы RPC

RPC технически не является протоколом — его лучше рассматривать как общий механизм для структурирования распределенных систем. RPC популярен, потому что он основан на семантике локального вызова процедуры — прикладная программа вызывает

ет процедуру, не обращая внимания на то, локальная она или удаленная, и блокируется до возврата вызова. Разработчик приложений может в значительной степени не знать, локальная процедура или удаленная, что значительно упрощает его задачу. Когда вызываемые процедуры фактически являются методами удаленных объектов в объектно-ориентированном языке, RPC называется *удаленным вызовом метода* (Remote Method Invocation, RMI). Хотя концепция RPC проста, существуют две основные проблемы, которые делают его более сложным, чем локальные вызовы процедур:

- Сеть между вызывающим процессом и вызываемым процессом имеет гораздо более сложные свойства, чем магистраль компьютера. Например, она может ограничивать размеры сообщений и имеет тенденцию к потере и перестановке сообщений.
- Компьютеры, на которых выполняются вызывающие и вызываемые процессы, могут иметь значительно отличающиеся архитектуры и форматы представления данных.

Таким образом, полный механизм RPC фактически включает в себя два основных компонента:

1. Протокол, который управляет сообщениями, передаваемыми между клиентом и сервером, и который справляется с потенциально нежелательными свойствами подлежащей сети.
2. Поддержка языков программирования и компилятора для упаковки аргументов в сообщение запроса на клиентской машине, а затем для перевода этого сообщения обратно в аргументы на серверной машине, а также с возвращаемым значением (эта часть механизма RPC обычно называется *компилятором заглушек*).

Рис. 5.14 схематически изображает, что происходит, когда клиент вызывает удаленную процедуру. Сначала клиент вызывает локальную заглушку для процедуры, передавая ей аргументы, необходимые для процедуры. Эта заглушка скрывает факт, что процедура удаленная, переводя аргументы в сообщение запроса, а затем вызывает протокол RPC для отправки сообщения запроса на серверную машину. На сервере протокол RPC доставляет сообщение запроса серверной заглушке, которая переводит его в аргументы для процедуры и затем вызывает локальную процедуру. После завершения процедуры сервер возвращает сообщение-ответ, которое передается протоколу RPC для отправки обратно клиенту. Протокол RPC на клиенте передает это сообщение вверх клиентской заглушке, которая переводит его в возвращаемое значение и возвращает клиентской программе.

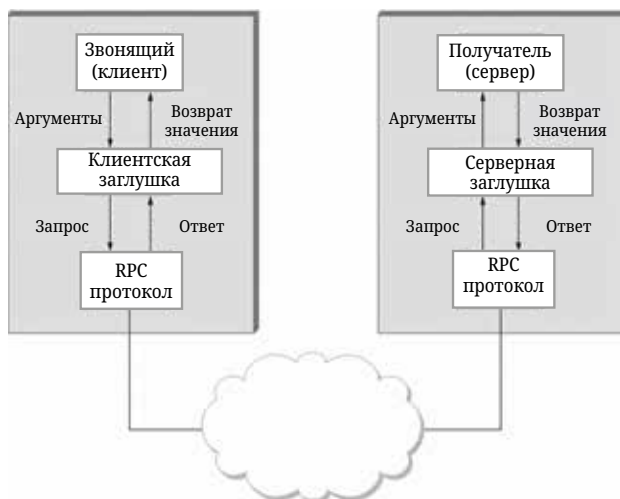


Рисунок 5.14. Полный механизм RPC.

В этой главе рассматриваются только аспекты, связанные с протоколом механизма RPC. То есть он игнорирует заглушки и фокусируется вместо этого на протоколе RPC, иногда называемом протоколом запроса/ответа, который передает сообщения между клиентом и сервером. Преобразование аргументов в сообщения и *наоборот* рассматривается в другом разделе. Также важно помнить, что клиентские и серверные программы написаны на каком-то языке программирования, а это означает, что определенный механизм RPC может поддерживать заглушки для Python, заглушки для Java, заглушки для GoLang и так далее, каждая из которых включает языковые идиомы для того, как процедуры вызываются.

Термин *RPC* относится к типу протокола, а не к конкретному стандарту, как TCP, поэтому конкретные RPC-протоколы различаются по выполняемым функциям. И, в отличие от TCP, который является доминирующим протоколом надежного байтового потока, нет одного доминирующего RPC-протокола. Таким образом, в этом разделе мы будем говорить больше об альтернативных вариантах дизайна, чем ранее.

Идентификаторы в RPC

Две функции, которые должен выполнять любой протокол RPC:

- Обеспечить пространство имен для уникальной идентификации вызываемой процедуры.
- Соотнести каждое сообщение-ответ с соответствующим сообщением-запросом.

Первая проблема имеет некоторые сходства с проблемой идентификации узлов в сети, с чем, например, справляются IP-адреса. Одним из дизайнерских выборов при идентификации объектов является решение о том, делать ли это пространство имен плоским или иерархическим. Плоское пространство имен просто назначило бы уникальный неструктурированный идентификатор (например, целое число) каждой процедуре, и этот номер переносился бы в одном поле в сообщении запроса RPC. Это потребовало бы какого-то центрального координирования, чтобы избежать назначения одного и того же номера процедуры двум разным процедурам. В качестве альтернативы протокол мог бы реализовать иерархическое пространство имен, аналогичное используемому для путей к файлам, что требует только того, чтобы «базовое имя» файла было уникальным в пределах своего каталога. Такой подход потенциально упрощает задачу обеспечения уникальности имен процедур. Иерархическое пространство имен для RPC можно реализовать, определив набор полей в формате сообщения запроса, по одному для каждого уровня именования в, скажем, двух- или трехуровневом иерархическом пространстве имен.

Ключом к сопоставлению сообщения-ответа с соответствующим запросом является уникальная идентификация пар «запрос-ответ» с помощью поля идентификатора сообщения. В сообщении ответа поле идентификатора сообщения устанавливается в то же значение, что и в сообщении запроса. Когда модуль RPC клиента получает ответ, он использует идентификатор сообщения для поиска соответствующего ожидающего запроса. Чтобы сделать транзакцию RPC похожей на локальный вызов процедуры для вызывающего абонента, вызывающий абонент блокируется до получения сообщения-ответа. Когда ответ получен, заблокированный вызывающий абонент идентифицируется по номеру запроса в ответе, возвращаемое значение удаленной процедуры получается из ответа, и вызывающий абонент разблокируется, чтобы он мог вернуть это возвращаемое значение.

Одна из повторяющихся проблем в RPC — это работа с неожиданными ответами, и мы видим это на примере идентификаторов сообщений. Например, рассмотрим следующую патологическую (но реалистичную) ситуацию. Клиентская машина отправляет сообщение-запрос с идентификатором сообщения 0, затем происходит сбой и машина перезагружается, а затем отправляет несвязанный запрос с тем же идентификатором сообщения 0. Сервер может не знать, что у клиента произошел сбой и прошла перезагрузка,

и, увидев сообщение-запрос с идентификатором сообщения 0, подтверждает его и отбрасывает как дубликат. Клиент никогда не получает ответа на запрос.

Один из способов устранить эту проблему — использовать *идентификатор загрузки* (boot ID). Идентификатор загрузки машины — это номер, который увеличивается при каждой перезагрузке машины; этот номер читается из энергонезависимой памяти (например, диска или флеш-накопителя), увеличивается и записывается обратно на устройство хранения во время процедуры запуска машины. Этот номер затем помещается в каждое сообщение, отправленное этим хостом. Если сообщение получено с устаревшим идентификатором сообщения, но новым идентификатором загрузки, оно распознается как новое сообщение. По сути, идентификатор сообщения и идентификатор загрузки объединяются, чтобы сформировать уникальный идентификатор для каждой транзакции.

Преодоление сетевых ограничений

Протоколы RPC часто выполняют дополнительные функции, чтобы справиться с тем, что сети не являются идеальными каналами. Две такие функции:

- Обеспечение надежной доставки сообщений
- Поддержка больших размеров сообщений через фрагментацию и сборку

Протокол RPC может «определить эту проблему», выбрав компиляцию поверх надежного протокола, такого как TCP, но во многих случаях протокол RPC реализует собственный уровень надежной доставки сообщений поверх ненадежного субстрата (например, UDP/IP). Такой протокол RPC, вероятно, реализует надежность, используя подтверждения и тайм-ауты, аналогично TCP.

Основной алгоритм прост, как показано на временной диаграмме на рис. 5.15. Клиент отправляет сообщение-запрос, и сервер подтверждает его. Затем, после выполнения процедуры, сервер отправляет сообщение-ответ, и клиент подтверждает ответ.

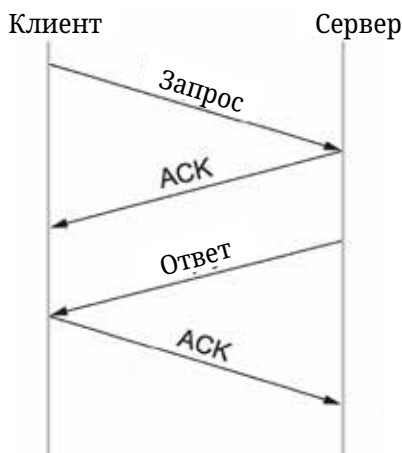


Рисунок 5.15. Простая временная шкала для надежного протокола RPC.

Любое сообщение, передающее данные (сообщение запроса или сообщение ответа) или ACK, отправленное для подтверждения этого сообщения, может быть потеряно в сети. Для учета этой возможности и клиент, и сервер сохраняют копию каждого отправленного сообщения до тех пор, пока для него не будет получено ACK. Каждая сторона также устанавливает таймер RETRANSMIT и повторно отправляет сообщение, если этот таймер исте-

кает. Обе стороны сбрасывают этот таймер и пытаются повторить процедуру снова некоторое согласованное количество раз, прежде чем сдаться и освободить сообщение.

Если клиент RPC получает сообщение-ответ, то, очевидно, соответствующее сообщение-запрос должно было быть получено сервером. Следовательно, само сообщение-ответ является *неявным подтверждением*, и любое дополнительное подтверждение со стороны сервера логически не требуется. Аналогично сообщение-запрос может неявно подтверждать предыдущее сообщение-ответ — при условии, что протокол делает транзакции «запрос-ответ» последовательными, так что одна транзакция должна завершиться, прежде чем начнется следующая. К сожалению, эта последовательность серьезно ограничила бы производительность RPC.

Выходом из этого затруднительного положения является реализация абстракции *канала* в протоколе RPC. В пределах данного канала транзакции «запрос/ответ» последовательны — в данном канале в любой момент времени может быть активна только одна транзакция — но может быть несколько каналов. Или, говоря иначе, абстракция канала позволяет *мультиплексировать* несколько транзакций «запрос/ответ» RPC между парой «клиент/сервер».

Каждое сообщение включает поле идентификатора канала, чтобы указать, к какому каналу принадлежит сообщение. Сообщение-запрос в данном канале неявно подтверждало бы предыдущее сообщение-ответ в этом канале, если оно еще не было подтверждено. Прикладная программа может открыть несколько каналов к серверу, если она хочет иметь более одной транзакции «запрос/ответ» одновременно (программа должна иметь несколько потоков). Как показано на рис. 5.16, сообщение-ответ служит для подтверждения сообщения-запроса, и последующий запрос подтверждает предшествующий ответ. Обратите внимание, что мы видели очень похожий подход (называемый *параллельными логическими каналами*) в предыдущей главе, как способ улучшения производительности механизма надежности «остановка и ожидание».

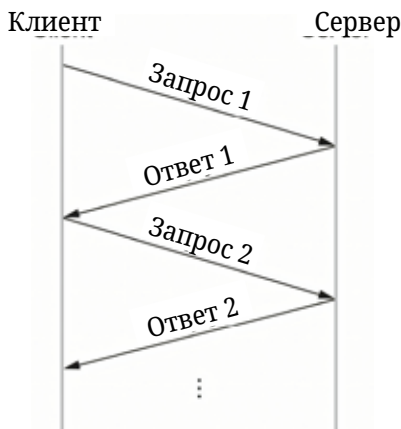


Рисунок 5.16. Временная шкала для надежного протокола RPC, использующего неявное подтверждение.

Еще одной сложностью, которую необходимо решить RPC, является то, что сервер может ждать произвольное время для получения результата, и, что еще хуже, он может дать сбой до генерации ответа. Имейте в виду, что мы говорим о периоде времени после того, как сервер подтвердил запрос, но до того, как он отправил ответ. Чтобы помочь клиенту отличить медленный сервер от «мертвого» сервера, клиентская сторона RPC может периодически отправлять серверу сообщение «Ты жив?», а сервер отвечает с помощью АСК. В качестве альтернативы сервер может отправлять клиенту сообщения «Я все еще жив»,

без того чтобы клиент сначала запрашивал их. Этот подход более масштабируемый, так как большая часть нагрузки по управлению тайм-аутом ложится на клиентов.

Надежность RPC может включать свойство, известное как *семантика «не более одного раза»* (at-most-once). Это означает, что для каждого сообщения-запроса, которое отправляет клиент, на сервер доставляется не более одной копии этого сообщения. Каждый раз, когда клиент вызывает удаленную процедуру, эта процедура вызывается не более одного раза на серверной машине. Мы говорим «не более одного раза», а не «ровно один раз», потому что всегда возможно, что либо сеть, либо серверная машина вышли из строя, делая невозможной доставку даже одной копии сообщения-запроса.

Для реализации семантики «не более одного раза» RPC на серверной стороне должен распознавать дублирующиеся запросы (и игнорировать их), даже если он уже успешно ответил на исходный запрос. Следовательно, он должен хранить некоторую информацию о состоянии, которая идентифицирует прошлые запросы. Один из подходов заключается в идентификации запросов с помощью порядковых номеров, так что серверу нужно только запомнить самый последний порядковый номер. К сожалению, это ограничит RPC одним незавершенным запросом (к данному серверу) за раз, так как один запрос должен быть завершен, прежде чем можно будет передать запрос со следующим порядковым номером. Каналы снова предоставляют решение. Сервер может распознавать дублирующиеся запросы, запоминая текущий порядковый номер для каждого канала, не ограничивая клиента одним запросом за раз.

Как бы очевидно ни звучало «не более одного раза», не все протоколы RPC поддерживают такое поведение. Некоторые поддерживают семантику, которая шуточно называется семантикой «*ноль или больше раз*»; то есть каждый вызов на клиенте приводит к тому, что удаленная процедура вызывается ноль или более раз. Нетрудно понять, как это может вызвать проблемы для удаленной процедуры, которая изменяет некоторые локальные переменные состояния (например, увеличивает счетчик) или имеет некоторый внешне видимый побочный эффект (например, запускает ракету) каждый раз, когда она вызывается. С другой стороны, если вызываемая удаленная процедура идемпотентна (множественные вызовы имеют тот же эффект, что и один вызов), тогда механизм RPC не обязательно должен поддерживать семантику «не более одного раза»; достаточно будет более простая (возможно, быстрая) реализация.

Как и в случае с надежностью, две причины, по которым протокол RPC может реализовывать фрагментацию и пересборку сообщений, заключаются в том, что это не предусмотрено базовым стеком протоколов или это может быть реализовано более эффективно протоколом RPC. Рассмотрим случай, когда RPC реализован поверх UDP/IP и полагается на IP для фрагментации и пересборки. Если хотя бы один фрагмент сообщения не успеет прибыть в определенное время, IP отбрасывает фрагменты, которые успели прибыть, и сообщение фактически теряется. В конечном итоге протокол RPC (предполагая, что он реализует надежность) исчерпает время ожидания и повторно отправит сообщение. В отличие от этого, рассмотрим протокол RPC, который реализует свою собственную фрагментацию и пересборку и агрессивно отправляет ACK или NACK (отрицательное подтверждение) для отдельных фрагментов. Потерянные фрагменты будут обнаружены и повторно отправлены быстрее, и будут повторно отправлены только потерянные фрагменты, а не все сообщения.

Синхронные и асинхронные протоколы

Один из способов охарактеризовать протокол — это определить, является ли он синхронным или *асинхронным*. Точное значение этих терминов зависит от уровня протокола, на котором они используются. На транспортном уровне точнее думать о них как о крайностях спектра, а не как о двух взаимоисключающих альтернативах. Ключевой характеристикой любой точки вдоль спектра является то, что известно процессу отправки после завершения операции отправки сообщения. Другими словами, если мы предполагаем,

что прикладная программа вызывает операцию отправки в транспортном протоколе, что именно известно программе о результате операции, когда возвращается управление после вызова отправки?

В *асинхронном* случае приложение не знает абсолютно ничего, когда `send` возвращается. Оно не знает, было ли сообщение получено его корреспондентом, и даже не знает наверняка, что сообщение успешно покинуло локальную машину. В *синхронном* случае операция `send` обычно возвращает сообщение-ответ. То есть приложение не только знает, что отправленное им сообщение было получено его корреспондентом, но и знает, что корреспондент вернул ответ. Таким образом, синхронные протоколы реализуют абстракцию запроса/ответа, в то время как асинхронные протоколы используются, если отправитель хочет иметь возможность отправлять много сообщений, не дожидаясь ответа. В соответствии с этим определением протоколы RPC обычно являются синхронными протоколами.

Хотя мы не обсуждали их в этой главе, между этими двумя крайностями есть интересные аспекты. Например, транспортный протокол может реализовать отправку таким образом, чтобы она блокировалась (не возвращала управление), пока сообщение не будет успешно получено на удаленной машине, но возвращала управление до того, как получатель на этой машине фактически обработает и ответит на него. Это иногда называют *протоколом надежных дейтаграмм*.

Глава 5.3.2. Реализации RPC (SunRPC, DCE, gRPC)

Теперь мы перейдем к обсуждению некоторых примеров реализаций протоколов RPC. Они помогут выделить различные проектные решения, которые принимали разработчики протоколов. Наш первый пример — SunRPC, широко используемый протокол RPC, также известный как Open Network Computing RPC (ONC RPC). Наш второй пример, который мы будем называть DCE-RPC, является частью Distributed Computing Environment (DCE). DCE — это набор стандартов и программного обеспечения для создания распределенных систем, определенный Open Software Foundation (OSF), консорциумом компьютерных компаний, который изначально включал IBM, Digital Equipment Corporation и Hewlett-Packard; сегодня OSF называется The Open Group. Наш третий пример — gRPC, популярный механизм RPC, который Google выпустил в качестве open-source, основанный на механизме RPC, который они использовали внутри компании для реализации облачных сервисов в своих дата-центрах.

Эти три примера представляют интересные альтернативные проектные решения в пространстве решений RPC, но если вы думаете, что это единственные варианты, мы описываем еще три механизма, похожих на RPC (WSDL, SOAP и REST) в контексте веб-сервисов в девятом разделе.

SunRPC

SunRPC стал *фактическим* стандартом благодаря широкому распространению с рабочими станциями Sun и центральной роли, которую он играет в популярной у Sun Network File System (NFS). IETF впоследствии приняла его в качестве стандартного интернет-протокола под названием ONC RPC.

SunRPC может быть реализован поверх нескольких различных транспортных протоколов. Рис. 5.17 иллюстрирует граф протоколов, когда SunRPC реализован на UDP. Как мы отмечали ранее в этой главе, строгий сторонник слоистости может не одобрить идею запуска транспортного протокола поверх транспортного протокола или утверждать, что RPC должно быть чем-то другим, кроме транспортного протокола, так как он появляется «выше» транспортного уровня. На самом деле проектное решение запустить RPC поверх существующего транспортного уровня имеет много смысла, как станет очевидно в следующем обсуждении.

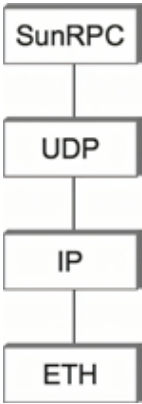


Рисунок 5.17. Граф протокола для SunRPC поверх UDP.

SunRPC использует двухуровневые идентификаторы для идентификации удаленных процедур: 32-битный номер программы и 32-битный номер процедуры. (Существует также 32-битный номер версии, но мы игнорируем это в данном обсуждении.) Например, сервер NFS был назначен номер программы `x00100003`, и в рамках этой программы `getattr` является процедурой 1, `setattr` — процедурой 2, `read` — процедурой 6, `write` — процедурой 8 и так далее. Номер программы и номер процедуры передаются в заголовке запроса SunRPC, поля которого показаны на рис. 5.18. Сервер, который может поддерживать несколько номеров программ, отвечает за вызов указанной процедуры указанной программы. Запрос SunRPC действительно представляет собой запрос на вызов указанной программы и процедуры на конкретной машине, к которой был отправлен запрос, даже если та же программа может быть реализована на других машинах в той же сети. Таким образом, адрес машины сервера (например, IP-адрес) является неявным третьим уровнем адреса RPC.

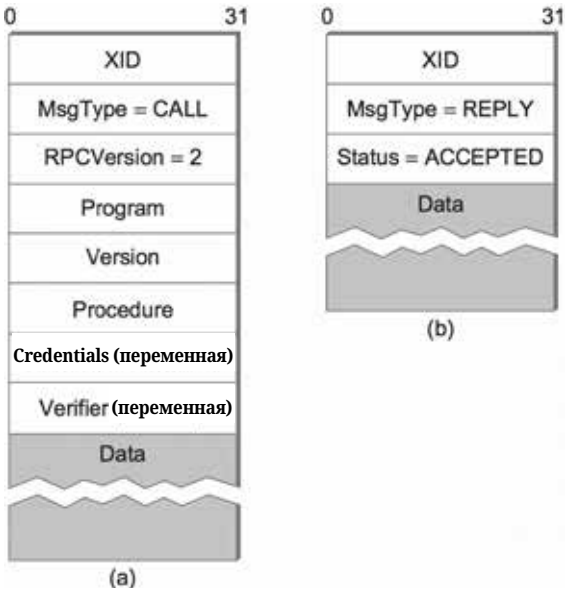


Рисунок 5.18. Форматы заголовков SunRPC: (a) запрос; (b) ответ.

Различные номера программ могут принадлежать разным серверам на одной и той же машине. Эти разные серверы имеют различные ключи демультиплексирования транспортного уровня (например, порты UDP), большинство из которых не являются хорошо известными номерами, а назначаются динамически. Эти ключи демультиплексирования называются *селекторами транспортного уровня*. Как клиент SunRPC, который хочет обратиться к определенной программе, может определить, какой селектор транспортного уровня использовать для доступа к соответствующему серверу? Решение заключается в назначении одного известного адреса *только одной программе* на удаленной машине и предоставлении этой программе задачи информирования клиентов о том, какой селектор транспортного уровня использовать для доступа к любой другой программе на машине. Исходная версия этой программы SunRPC называется *Port Mapper* (портовой картограф) и поддерживает только UDP и TCP как базовые протоколы. Ее номер программы — x00100000, а ее известный порт — 111. RPCBIND, который развился из Port Mapper, поддерживает произвольные базовые транспортные протоколы. При запуске каждого сервера SunRPC он вызывает процедуру регистрации RPCBIND на собственной домашней машине сервера, чтобы зарегистрировать свой селектор транспортного уровня и номера поддерживаемых программ. Удаленный клиент затем может вызвать процедуру поиска RPCBIND, чтобы узнать селектор транспортного уровня для конкретного номера программы.

Чтобы сделать обсуждение более конкретным, рассмотрим пример с использованием Port Mapper и UDP. Чтобы отправить сообщение запроса к процедуре чтения NFS, клиент сначала отправляет сообщение запроса к Port Mapper на известный UDP-порт 111, запрашивая вызов процедуры 3 для отображения номера программы x00100003 на UDP-порт, где в данный момент находится программа NFS. Затем клиент отправляет сообщение запроса SunRPC с номером программы x00100003 и номером процедуры 6 на этот UDP-порт, и модуль SunRPC, прослушивающий этот порт, вызывает процедуру чтения NFS. Клиент также кеширует отображение программы на номер порта, чтобы не обращаться к Port Mapper каждый раз, когда он хочет связаться с программой NFS.¹

Чтобы сопоставить сообщение ответа с соответствующим запросом, чтобы результат RPC мог быть возвращен правильному вызывающему, заголовки сообщений как запроса, так и ответа включают поле XID (идентификатор транзакции), как показано на рис. 5.18. XID — это уникальный идентификатор транзакции, используемый только для одного запроса и соответствующего ответа. После успешного ответа сервера на данный запрос он не запоминает XID. Из-за этого SunRPC не может гарантировать семантику «не более одного раза».

Детали семантики SunRPC зависят от базового транспортного протокола. Он не реализует свою собственную надежность, поэтому он надежен только в том случае, если базовый транспорт надежен. (Конечно, любое приложение, работающее поверх SunRPC, также может реализовать свои собственные механизмы надежности выше уровня SunRPC.) Возможность отправлять запросы и ответы, размер которых превышает MTU сети, также зависит от базового транспорта. Другими словами, SunRPC не предпринимает попыток улучшить базовый транспорт в плане надежности и размера сообщений. Поскольку SunRPC может работать поверх множества различных транспортных протоколов, это дает ему значительную гибкость без усложнения дизайна самого протокола RPC.

Возвращаясь к формату заголовка SunRPC, показанному на рис. 5.18, сообщение запроса содержит поля Credentials и Verifier переменной длины, которые используются клиентом для аутентификации перед сервером, то есть для предоставления доказательств того, что клиент имеет право вызывать сервер. Как клиент аутентифицируется перед сервером — это общий вопрос, который должен быть решен любым протоколом, стремящимся обеспечить разумный уровень безопасности. Эта тема обсуждается более подробно в другой главе.

¹ На практике NFS (Network File System) является настолько важной программой, что ей был выделен собственный хорошо известный UDP-порт, но для целей базовой иллюстрации мы предположим, что это не так.

DCE-RPC

DCE-RPC — это протокол удаленного вызова процедур (RPC), являющийся основой системы DCE и служивший основой для механизма RPC, лежащего в основе Microsoft's DCOM и ActiveX. Он может использоваться с компилятором заглушек Network Data Representation (NDR), описанным в другой главе, но также служит основным протоколом RPC для Common Object Request Broker Architecture (CORBA), которая является промышленным стандартом для построения распределенных объектно-ориентированных систем.

DCE-RPC, как и SunRPC, может быть реализован поверх нескольких транспортных протоколов, включая UDP и TCP. Он также похож на SunRPC тем, что определяет двухуровневую адресацию: транспортный протокол демультиплексирует до правильного сервера, DCE-RPC направляет к определенной процедуре, экспортированной этим сервером, а клиенты обращаются к «службе отображения конечных точек» (аналогично Port Mapper в SunRPC), чтобы узнать, как достичь конкретного сервера. Однако, в отличие от SunRPC, DCE-RPC реализует семантику вызовов не более одного раза. (На самом деле DCE-RPC поддерживает несколько семантик вызовов, включая идемпотентную семантику, аналогичную SunRPC, но не более одного раза — это поведение по умолчанию.) Существуют и другие различия между этими двумя подходами, которые мы выделим в следующих абзацах.

Рис. 5.19 показывает временную шкалу для типичного обмена сообщениями, где каждое сообщение помечено своим типом DCE-RPC. Клиент отправляет сообщение запроса (Request), сервер в конечном итоге отвечает сообщением ответа (Response), и клиент подтверждает (Ack) ответ. Вместо того чтобы сервер подтверждал сообщения запроса, клиент периодически отправляет серверу сообщение Ping, на которое сервер отвечает сообщением Working, указывая, что удаленная процедура все еще выполняется. Если ответ сервера получен достаточно быстро, сообщения Ping не отправляются. Хотя это не показано на рисунке, поддерживаются и другие типы сообщений. Например, клиент может отправить серверу сообщение Quit, прося его прервать предыдущий вызов, который все еще выполняется; сервер отвечает сообщением Quack (подтверждение выхода). Также сервер может ответить на сообщение запроса сообщением Reject (указывающим на то, что вызов был отклонен), и он может ответить на сообщение Ping сообщением NoCall (указывающим на то, что сервер никогда не слышал о вызывающем абоненте).

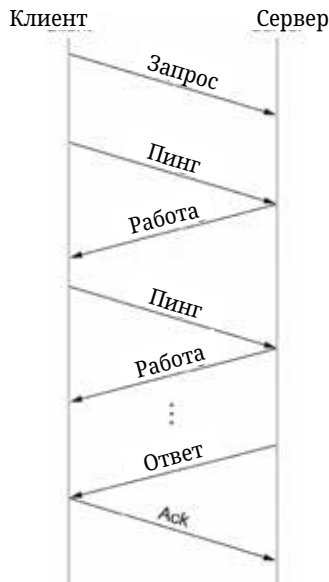


Рисунок 5.19. Типичный обмен сообщениями DCE-RPC.

Каждая транзакция «запрос/ответ» в DCE-RPC происходит в контексте активности. Активность — это логический канал «запрос/ответ» между парой участников. В любой момент времени на данном канале может быть активна только одна транзакция сообщения. Как и в подходе с параллельными логическими каналами, описанном выше, прикладные программы должны открывать несколько каналов, если они хотят иметь более одной транзакции «запрос/ответ» одновременно. Активность, к которой относится сообщение, определяется полем `ActivityId` в сообщении. Поле `SequenceNum` затем различает вызовы, сделанные в рамках одной и той же активности; оно служит той же цели, что и поле `XID` (идентификатор транзакции) в SunRPC. В отличие от SunRPC, DCE-RPC отслеживает последний использованный порядковый номер в рамках конкретной активности, чтобы обеспечить семантику не более одного раза. Чтобы различать ответы, отправленные до и после перезагрузки сервера, DCE-RPC использует поле `ServerBoot` для хранения идентификатора загрузки машины.

Еще одно отличие в проектировании DCE-RPC от SunRPC — это поддержка фрагментации и сборки в протоколе RPC. Как было отмечено выше, даже если основной протокол, такой как IP, обеспечивает фрагментацию/сборку, более сложный алгоритм, реализованный как часть RPC, может привести к более быстрому восстановлению и уменьшению потребления пропускной способности при потере фрагментов. Поле `FragmentNum` уникально идентифицирует каждый фрагмент, составляющий данное сообщение запроса или ответа. Каждому фрагменту DCE-RPC присваивается уникальный номер фрагмента (0, 1, 2, 3 и так далее). И клиент, и сервер реализуют механизм селективного подтверждения, который работает следующим образом. (Мы описываем механизм в терминах клиента, отправляющего фрагментированное сообщение запроса серверу; тот же механизм применяется, когда сервер отправляет фрагментированный ответ клиенту.)

Во-первых, каждый фрагмент, составляющий сообщение запроса, содержит как уникальный `FragmentNum`, так и флаг, указывающий, является ли этот пакет фрагментом вызова (`frag`) или последним фрагментом вызова (`last`); сообщения запроса, которые помещаются в один пакет, содержат флаг `last`. Сервер знает, что он получил полное сообщение запроса, когда у него есть пакет с флагом `last` и нет пропусков в номерах фрагментов. Во-вторых, в ответ на каждый поступающий фрагмент сервер отправляет клиенту сообщение `Frack` (подтверждение фрагмента). Это подтверждение идентифицирует наибольший номер фрагмента, который сервер успешно получил. Иными словами, подтверждение является кумулятивным, аналогично TCP. Однако, кроме того, сервер выборочно подтверждает любые более высокие номера фрагментов, которые он получил вне порядка. Это делается с помощью битовой матрицы, которая идентифицирует эти внеочередные фрагменты относительно наибольшего фрагмента, полученного в порядке. Наконец, клиент отвечает повторной передачей отсутствующих фрагментов.

Рис. 5.20 иллюстрирует, как это работает. Предположим, сервер успешно получил фрагменты до номера 20 включительно, плюс фрагменты 23, 25 и 26. Сервер отвечает сообщением `Frack`, которое идентифицирует фрагмент 20 как наибольший фрагмент, полученный в порядке, плюс битовая матрица (`SelAck`) с включенными третьим ($23=20+3$), пятым ($25=20+5$) и шестым ($26=20+6$) битами. Чтобы поддерживать (почти) произвольно длинную битовую матрицу, размер матрицы (измеряемый в 32-битных словах) указан в поле `SelAckLen`.

Учитывая поддержку DCE-RPC очень больших сообщений — поле `FragmentNum` имеет длину 16 бит, а это означает, что оно может поддерживать 64K фрагментов, — нецелесообразно, чтобы протокол отправлял все фрагменты, составляющие сообщение, так быстро, как только может, поскольку это может перегрузить получателя. Вместо этого DCE-RPC реализует алгоритм управления потоком, очень похожий на TCP. В частности, каждое сообщение `Frack` не только подтверждает полученные фрагменты, но и сообщает отправителю, сколько фрагментов он теперь может отправить. Это цель поля `WindowSize` на рис. 5.20, которое выполняет ту же функцию, что и поле `AdvertisedWindow` в TCP, только оно считает фрагменты, а не байты. DCE-RPC также реализует механизм управления перегрузками, аналогичный TCP. Учитывая сложность управления перегрузками, неудивительно, что некоторые протоколы RPC избегают ее, избегая фрагментации.

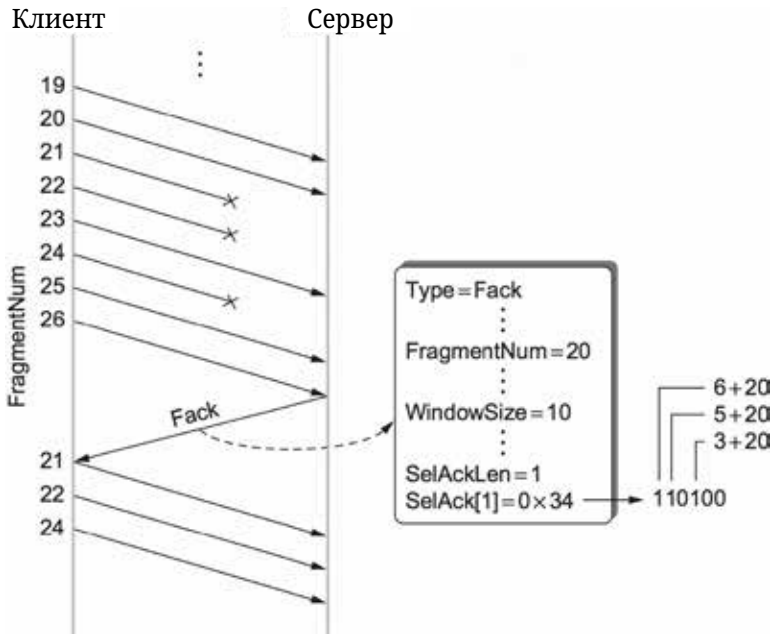


Рисунок 5.20. Фрагментация с выборочными подтверждениями.

В заключение отметим, что у дизайнеров есть довольно широкий выбор опций при проектировании протокола RPC. SunRPC использует более минималистский подход и добавляет относительно немного к основному транспорту, помимо необходимых элементов для нахождения правильной процедуры и идентификации сообщений. DCE-RPC добавляет больше функциональности, с возможностью улучшенной производительности в некоторых средах за счет большей сложности.

gRPC

Несмотря на свое происхождение в Google, gRPC не означает Google RPC. «g» обозначает что-то разное в каждой версии. В версии 1.10 это означало «glamorous» (гламурный), а в 1.18 — «goose» (гусь). Тем не менее gRPC популярен, потому что он предоставляет всем — как открытый исходный код — десятилетний опыт использования RPC внутри Google для создания масштабируемых облачных сервисов.

Прежде чем углубиться в детали, отметим несколько основных различий между gRPC и двумя другими примерами, которые мы только что рассмотрели. Самое большое отличие заключается в том, что gRPC предназначен для облачных сервисов, а не для простого клиент/серверного подхода, который предшествовал ему. Разница заключается, по сути, в дополнительном уровне косвенности. В мире «клиент/сервер» клиент вызывает метод на конкретном серверном процессе, работающем на конкретной серверной машине. Предполагается, что одного серверного процесса достаточно для обслуживания вызовов от всех клиентских процессов, которые могут к нему обращаться.

С облачными сервисами клиент вызывает метод на сервисе, который, чтобы поддерживать вызовы от произвольно большого числа клиентов одновременно, реализован масштабируемым числом серверных процессов, каждый из которых потенциально работает на другой серверной машине. Здесь вступает в игру облако: дата-центры предоставляют, казалось бы, бесконечное количество серверных машин для масштабирования облачных сервисов. Когда мы используем термин «масштабируемый», мы имеем в виду, что количество

идентичных серверных процессов, которые вы собираетесь создать, зависит от рабочей нагрузки (то есть количества клиентов, которые хотят получить услугу в любой данный момент), и это количество может динамически изменяться со временем. Еще одна деталь заключается в том, что облачные сервисы обычно не создают новый процесс как таковой, а запускают новый *контейнер*, который по сути является процессом, инкапсулированным в изолированной среде, включающей все программные пакеты, необходимые для работы процесса. Docker — это сегодняшний канонический пример платформы контейнеров.

Возвращаясь к утверждению, что сервис по сути является дополнительным уровнем сложности, наложенным на сервер, это означает лишь то, что вызывающий идентифицирует сервис, который он хочет вызвать, и *балансировщик нагрузки* направляет этот вызов одному из множества доступных серверных процессов (контейнеров), которые реализуют этот сервис, как показано на рис. 5.21. Балансировщик нагрузки может быть реализован по-разному, включая аппаратное устройство, но обычно он реализуется как прокси-процесс, который работает в виртуальной машине (также размещенной в облаке), а не как физическое устройство.

Существует набор передовых методов для реализации реального серверного кода, который в конечном итоге отвечает на этот запрос, и некоторая дополнительная облачная инфраструктура для создания/уничтожения контейнеров и распределения запросов между этими контейнерами. Kubernetes — это сегодняшний канонический пример такой системы управления контейнерами, а *микро-сервисная архитектура* — это то, что мы называем передовыми методами в построении сервисов таким облачным способом. Оба эти вопроса представляют интерес, но выходят за рамки данной книги.

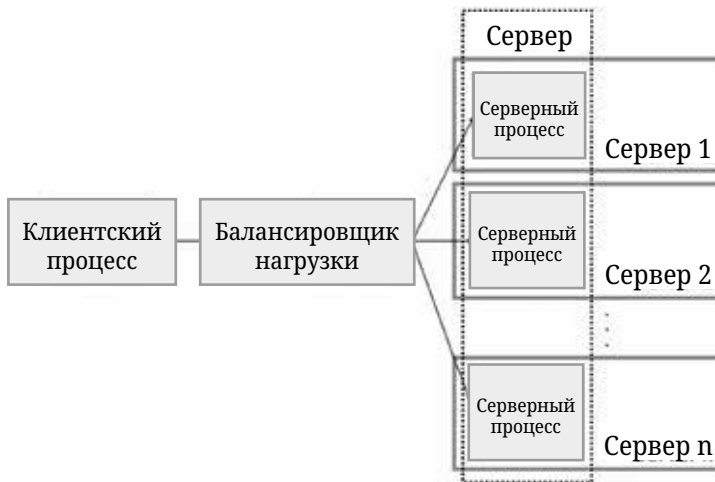


Рисунок 5.21. Использование RPC для вызова масштабируемой облачной службы.

Что нас интересует здесь, так это транспортный протокол, лежащий в основе gRPC. И здесь снова наблюдается значительное отличие от двух предыдущих примеров протоколов, не в плане фундаментальных проблем, которые нужно решить, а в плане подхода gRPC к их решению. Короче говоря, gRPC «передает» многие из этих проблем другим протоколам, оставляя gRPC по сути упаковывать эти возможности в удобную для использования форму. Давайте рассмотрим детали.

Во-первых, gRPC работает поверх TCP, а не UDP, а это означает, что он передает проблемы управления соединениями и надежной передачи запросов и ответных сообщений произвольного размера на уровень TCP. Во-вторых, gRPC фактически работает поверх защищенной версии TCP под названием *Transport Layer Security* (TLS) (тонкого слоя, который

находится над TCP в стеке протоколов), а это означает, что он передает ответственность за обеспечение безопасности канала связи, чтобы злоумышленники не могли подслушивать или захватывать обмен сообщениями. В-третьих, gRPC на самом деле работает поверх HTTP/2 (который сам по себе накладывается на TCP и TLS), а это означает, что gRPC решает еще две проблемы: (1) эффективное кодирование/сжатие двоичных данных в сообщение и (2) мультиплексирование нескольких удаленных вызовов процедур на одно TCP-соединение. Иными словами, gRPC кодирует идентификатор удаленного метода как URI, параметры запроса к удаленному методу как содержимое HTTP-сообщения, а возвращаемое значение от удаленного метода — в HTTP-ответе. Полный стек gRPC показан на рис. 5.22, который также включает элементы, специфичные для различных языков программирования. (Одним из преимуществ gRPC является широкий набор поддерживаемых языков программирования, и на рис. 5.22 показано лишь небольшое их подмножество.)

Мы обсудим TLS в разделе 8 (в контексте широкого круга вопросов безопасности) и HTTP в разделе 9 (в контексте традиционно рассматриваемых протоколов уровня приложений). Но мы сталкиваемся с интересной зависимостью: RPC — это разновидность транспортного протокола, используемого для реализации распределенных приложений, HTTP — пример протокола уровня приложений, и тем не менее gRPC работает поверх HTTP, а не наоборот.

Краткое объяснение состоит в том, что слоистая архитектура предоставляет удобный способ для людей понимать сложные системы, но на самом деле мы пытаемся решить набор проблем (например, надежную передачу сообщений произвольного размера, идентификацию отправителей и получателей, сопоставление сообщений запросов с ответами и так далее), и то, как эти решения объединяются в протоколы, а затем эти протоколы накладываются друг на друга, является следствием постепенных изменений с течением времени. Можно сказать, что это историческая случайность. Если бы Интернет начинался с механизма RPC, столь же повсеместного, как TCP, HTTP мог бы быть реализован поверх него (как и почти все другие протоколы уровня приложений, описанные в разделе 9), и Google потратил бы свое время на улучшение этого протокола, а не на создание собственного (как они и другие поступали с TCP). Вместо этого веб стал убийственным приложением Интернета, что означало, что его протокол уровня приложений (HTTP) стал повсеместно поддерживаться остальной инфраструктурой Интернета: межсетевыми экранами, балансировщиками нагрузки, шифрованием, аутентификацией, сжатием и так далее. Поскольку все эти сетевые элементы были спроектированы для работы с HTTP, HTTP фактически стал универсальным транспортным протоколом запросов/ответов в Интернете.

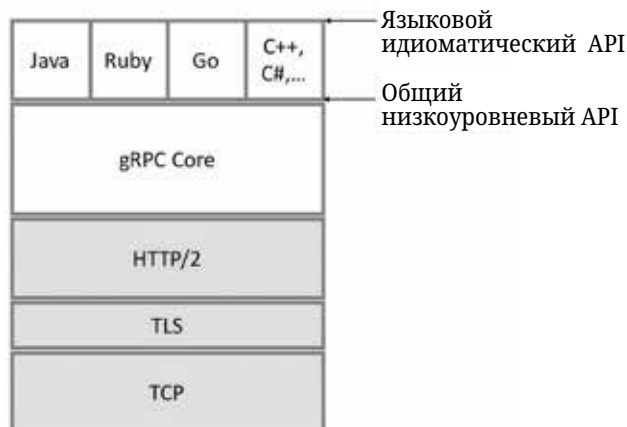


Рисунок 5.22. Ядро gRPC, уложенное поверх HTTP, TLS и TCP и поддерживающее множество языков.

Возвращаясь к уникальным характеристикам gRPC, наибольшая ценность, которую он приносит, заключается во включении потоковой передачи в механизм RPC, то есть gRPC поддерживает четыре различных шаблона запросов/ответов:

1. Простой RPC: клиент отправляет одно сообщение запроса, а сервер отвечает одним сообщением ответа.
2. Потоковый RPC сервера: клиент отправляет одно сообщение запроса, а сервер отвечает потоком сообщений ответа. Клиент завершает работу после получения всех ответов сервера.
3. Потоковый RPC клиента: клиент отправляет поток запросов серверу, и сервер отправляет обратно один ответ, как правило (но не обязательно) после получения всех запросов клиента.
4. Двусторонний потоковый RPC: вызов инициируется клиентом, но после этого клиент и сервер могут читать и писать запросы и ответы в любом порядке; потоки полностью независимы.

Эта дополнительная свобода в том, как клиент и сервер взаимодействуют, означает, что транспортный протокол gRPC должен отправлять дополнительные метаданные и управляющие сообщения — помимо собственно запросов и ответов — между двумя участниками. Примеры включают коды `Error` и `Status` (для указания успеха или причины сбоя), `Timeouts` (для указания, сколько времени клиент готов ждать ответа), `PING` (уведомление о поддержке соединения, чтобы указать, что одна или другая сторона все еще работает), `EOS` (уведомление о конце потока, чтобы указать, что больше нет запросов или ответов) и `GOAWAY` (уведомление от серверов клиентам, что они больше не будут принимать новые потоки). В отличие от многих других протоколов в этой книге, где мы показываем формат заголовка протокола, то, как эта управляющая информация передается между двумя сторонами, в значительной степени определяется основным транспортным протоколом, в данном случае HTTP/2. Например, как мы увидим в разделе 9, HTTP уже включает набор полей заголовка и кодов ответа, которыми пользуется gRPC.

Вы можете ознакомиться с обсуждением HTTP в разделе 9, прежде чем продолжить, но следующее достаточно понятно. Простой запрос RPC (без потоковой передачи) может включать следующее HTTP-сообщение от клиента к серверу:

```
HEADERS (flags = END_HEADERS)
:method = POST
:scheme = http
:path = /google.pubsub.v2.PublisherService/CreateTopic
:authority = pubsub.googleapis.com
grpc-timeout = 1S
content-type = application/grpc+proto
grpc-encoding = gzip
authorization = Bearer y235.wef315yf138vh31hv93hv8h3v
DATA (flags = END_STREAM)
<Length-Prefixed Message>
```

что приводит к следующему ответному сообщению от сервера клиенту:

```
HEADERS (flags = END_HEADERS)
:status = 200
grpc-encoding = gzip
content-type = application/grpc+proto
DATA
<Length-Prefixed Message>
```

```
HEADERS (flags = END_STREAM, END_HEADERS)
grpc-status = 0 # OK
trace-proto-bin = jher831yy13JHy3hc
```

В этом примере HEADERS и DATA — это два стандартных управляющих сообщения HTTP, которые эффективно разделяют «заголовок сообщения» и «полезную нагрузку сообщения». Если конкретизировать, то каждая строка, следующая за HEADERS (но перед DATA), является парой `attribute = value`, которая составляет заголовок (подумайте о каждой строке как об аналогичной полю заголовка); те пары, которые начинаются с двоеточия (например, `:status = 200`), являются частью стандарта HTTP (например, `status 200` указывает на успешное выполнение); а пары, которые не начинаются с двоеточия, являются специфичными для gRPC настройками (например, `grpc-encoding = gzip` указывает на то, что данные в следующем сообщении были сжаты с использованием gzip, и `grpc-timeout = 1S` указывает на то, что клиент установил тайм-аут в одну секунду).

Осталось объяснить последнюю деталь. Строка заголовка

```
content-type = application/grpc+proto
```

указывает на то, что тело сообщения (как это отмечено строкой DATA) имеет смысл только для прикладной программы (то есть метода сервера), у которой этот клиент запрашивает услугу. Если более конкретно, то строка `+proto` указывает на то, что получатель сможет интерпретировать биты в сообщении в соответствии со спецификацией интерфейса Protocol Buffer (сокращенно proto). Protocol Buffers — это способ gRPC указывать, как параметры, передаваемые на сервер, кодируются в сообщение, которое, в свою очередь, используется для генерации заглушек, расположенных между основным механизмом RPC и фактическими вызываемыми функциями (см. рис. 5.14). Это тема, которую мы рассмотрим в разделе 7.

Основные выводы

Самое главное заключается в том, что сложные механизмы, такие как RPC, когда-то были упакованы как монолитные пакеты программного обеспечения (как SunRPC и DCE-RPC), а в наши дни создаются путем сборки набора более мелких компонентов, каждый из которых решает узкую проблему. gRPC является одновременно примером такого подхода и инструментом, который способствует дальнейшему внедрению этого подхода. Архитектура микросервисов, упомянутая ранее в этом разделе, применяет стратегию «собрана из маленьких частей» к целым облачным приложениям (например, Uber, Lyft, Netflix, Yelp, Spotify), где gRPC часто является механизмом коммуникации, используемым этими маленькими частями для обмена сообщениями друг с другом.

Глава 5.4. Транспортный протокол реального времени (RTP)

На заре развития пакетной коммутации большинство приложений было связано с передачей файлов, хотя уже в 1981 году проводились эксперименты по передаче трафика в реальном времени, например оцифрованных образцов голоса. Мы называем приложение «приложением реального времени», если оно предъявляет высокие требования к своевременной доставке информации. Передача голоса по IP (VoIP) — это классический пример приложения реального времени, потому что вы не можете спокойно вести разговор с кем-то, если на получение ответа уходит больше доли секунды. Как мы вскоре увидим, приложения реального времени предъявляют особые требования к транспортному протоколу, которые не могут быть удовлетворены протоколами, рассмотренными до сих пор в этом разделе.

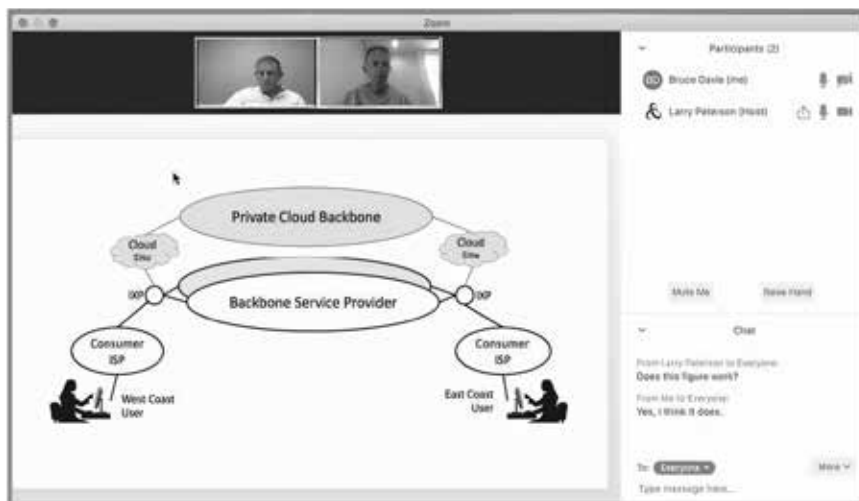


Рисунок 5.23. Пользовательский интерфейс инструмента для проведения видеоконференций.

Мультимедийные приложения (те, которые включают видео, аудио и данные) иногда делятся на два класса: *интерактивные* приложения и *поточные* приложения. На рис. 5.23 показаны авторы, использующие пример инструмента для видеоконференций, типичного для интерактивного класса. Наряду с VoIP это те приложения, которые имеют самые строгие требования реального времени.

Поточные приложения обычно передают аудио- или видеопотоки от сервера к клиенту и типичны для таких коммерческих продуктов, как Spotify. Поточное видео, типичное для YouTube и Netflix, стало одной из доминирующих форм трафика в Интернете. Поскольку поточные приложения не включают взаимодействие между людьми, они предъявляют менее строгие требования к реальному времени к основным протоколам. Однако своевременность все же важна — например, вы хотите, чтобы видео начало воспроизводиться вскоре после нажатия «воспроизвести», и как только оно начнет воспроизводиться, поздние пакеты либо вызовут его остановку, либо создадут визуальные искажения. Таким образом, хотя поточные приложения не являются строго реальными, у них все же достаточно общего с интерактивными мультимедийными приложениями, чтобы рассматривать общий протокол для обоих типов приложений.

Теперь должно быть понятно, что разработчики транспортного протокола для приложений реального времени и мультимедийных приложений сталкиваются с реальной проблемой в определении требований достаточно широко, чтобы удовлетворить потребности очень разных приложений. Им также нужно обращать внимание на взаимодействие между различными приложениями, например, синхронизацию аудио- и видеопотоков. Мы увидим далее, как эти соображения повлияли на разработку основного протокола транспортного времени, используемого сегодня: *транспортного протокола реального времени* (Real-time Transport Protocol, RTP).

Многие функции RTP фактически происходят из функциональности протокола, которая изначально была встроена в само приложение. Два первых таких приложения были *vis* и *vat*, первое из которых поддерживало видео в реальном времени, а второе — аудио в реальном времени. Оба приложения изначально работали напрямую через UDP, в то время как разработчики выясняли, какие функции необходимы для обработки реального времени связи. Позже они поняли, что эти функции могут быть полезны многим другим приложениям, и определили протокол с этими функциями. Этот протокол был в конечном итоге стандартизирован как RTP.

RTP может работать поверх множества нижележащих протоколов, но все же часто работает поверх UDP. Это приводит к стеку протоколов, показанному на рис. 5.24. Заметьте, что мы запускаем транспортный протокол поверх транспортного протокола. Против этого нет никаких правил, и на самом деле это имеет смысл, поскольку UDP предоставляет минимальный уровень функциональности, а базовая демultipлексация на основе номеров портов как раз соответствует тому, что нужно RTP в качестве отправной точки. Поэтому, вместо того чтобы воссоздавать номера портов в RTP, RTP передает функцию демultipлексации на UDP.

Приложение
RTP
UDP
IP
Подсеть

Рисунок 5.24. Стек протоколов для мультимедийных приложений, использующих RTP.

Глава 5.4.1. Требования

Основное требование для универсального мультимедийного протокола заключается в том, что он должен позволять аналогичным приложениям взаимодействовать друг с другом. Например, два независимо реализованных приложения для аудиоконференций должны иметь возможность общаться друг с другом. Это сразу же предполагает, что приложения должны использовать один и тот же метод кодирования и сжатия голоса; в противном случае данные, отправленные одной стороной, будут непонятны принимающей стороне. Поскольку существует довольно много различных схем кодирования голоса, каждая из которых имеет свои компромиссы между качеством, требованиями к пропускной способности и вычислительными затратами, вероятно, было бы плохой идеей постановить, что можно использовать только одну такую схему. Вместо этого наш протокол должен предоставить способ, чтобы отправитель мог сообщить получателю, какую схему кодирования он хочет использовать, и возможно провести переговоры до тех пор, пока не будет идентифицирована схема, доступная обеим сторонам.

Так же как и с аудио, существуют многие различные схемы кодирования видео. Таким образом, мы видим, что первой общей функцией, которую может предоставить RTP, является возможность сообщать о выборе схемы кодирования. Заметьте, что это также служит для идентификации типа приложения (например, аудио или видео); как только мы знаем, какой алгоритм кодирования используется, мы также знаем, какой тип данных кодируется.

Еще одно важное требование заключается в том, чтобы получатель потока данных мог определить временные соотношения между полученными данными. Приложения реального времени должны помещать полученные данные в *буфер воспроизведения*, чтобы сгладить джиттер, который мог быть введен в поток данных во время передачи по сети. Таким образом, для того чтобы позволить получателю воспроизводить данные в нужное время, необходимо использовать какую-то форму временной метки.

С временными метками одного медиапотока связана проблема синхронизации нескольких медиа в конференции. Очевидным примером этого является синхронизация аудио- и видеопотоков, исходящих от одного и того же отправителя. Как мы увидим далее, это несколько более сложная проблема, чем определение времени воспроизведения для одного потока.

Еще одна важная функция, которую необходимо обеспечить, это указание потери пакетов. Заметьте, что приложение с жесткими временными ограничениями обычно не может использовать надежный транспорт, такой как TCP, поскольку повторная передача данных для исправления потерь, вероятно, приведет к тому, что пакет прибывает слишком поздно, чтобы быть полезным. Таким образом, приложение должно уметь справляться с отсутствующими пакетами, и первым шагом в этом является определение того, что пакеты *действительно* отсутствуют. Например, видеоприложение, использующее кодирование MPEG, может предпринять различные действия при потере пакета, в зависимости от того, пришел ли пакет из I-кадра, В-кадра или Р-кадра.

Потеря пакетов также может служить индикатором перегрузки. Поскольку мультимедийные приложения обычно не работают поверх TCP, они также не используют функции избегания перегрузок TCP. Тем не менее многие мультимедийные приложения способны реагировать на перегрузки, например, изменяя параметры алгоритма кодирования для уменьшения потребляемой пропускной способности. Очевидно, что для этого необходимо, чтобы получатель уведомлял отправителя о возникновении потерь, чтобы отправитель мог корректировать параметры кодирования.

Еще одной общей функцией для мультимедийных приложений является концепция указания границ кадров. Кадр в данном контексте является специфическим для приложения. Например, полезно уведомлять видеоприложение, что определенный набор пакетов соответствует одному кадру. В аудиоприложении полезно отмечать начало «речевого всплеска» — набора звуков или слов, за которым следует тишина. Получатель может затем идентифицировать промежутки тишины между всплесками и использовать их как возможности для перемещения точки воспроизведения. Это следует из наблюдения, что небольшое сокращение или увеличение промежутков между словами не воспринимается пользователями, тогда как сокращение или увеличение самих слов заметно и раздражает.

Последняя функция, которую мы могли бы захотеть включить в протокол, — это способ идентификации отправителей, более удобный для пользователя, чем IP-адрес. Как показано на рис. 5.23, аудио- и видеоконференц-приложения могут отображать строки, такие как имена пользователей, на своих панелях управления, и поэтому протокол должен поддерживать ассоциацию такой строки с потоком данных.

Кроме функциональности, требуемой от нашего протокола, мы отмечаем дополнительное требование: он должен разумно и эффективно использовать пропускную способность. Иными словами, мы не хотим вводить много дополнительных битов, которые нужно отправлять с каждым пакетом в виде длинного заголовка. Причина этого в том, что аудиопакеты, являющиеся одним из наиболее распространенных типов мультимедийных данных, обычно маленькие, чтобы уменьшить время, необходимое для их заполнения образцами. Длинные аудиопакеты означали бы высокую задержку из-за упаковки, что негативно сказывается на восприятии качества разговоров. (Это было одним из факторов при выборе длины ячейек ATM.) Поскольку сами данные в пакетах короткие, длинный заголовок означал бы, что относительно большая часть пропускной способности канала будет использоваться заголовками, что уменьшает доступную емкость для «полезных» данных. Мы увидим несколько аспектов дизайна RTP, которые были нужны из-за необходимости сокращения заголовка.

Можно спорить о том, действительно ли каждая из описанных функций должна быть в протоколе транспортировки в реальном времени, и можно найти еще несколько, которые можно было бы добавить. Ключевая идея здесь состоит в том, чтобы облегчить жизнь разработчикам приложений, предоставив им полезный набор абстрактных и строительных блоков для их приложений. Например, введя механизм временной метки в RTP, мы избавляем каждого разработчика приложения в реальном времени от необходимости изобретать свой собственный. Мы также увеличиваем шансы, что два разных приложения в реальном времени смогут взаимодействовать.

Глава 5.4.2. Проектирование RTP

Теперь, когда мы рассмотрели довольно длинный список требований к нашему транспортному протоколу для мультимедиа, перейдем к деталям протокола, который был разработан для удовлетворения этих требований. Этот протокол, RTP, был разработан в IETF и получил широкое распространение. Стандарт RTP на самом деле определяет пару протоколов: RTP и Протокол управления реального времени (Real-time Transport Control Protocol, RTCP). Первый используется для обмена мультимедийными данными, в то время как второй периодически отправляет управляющую информацию, связанную с определенным потоком данных. При работе через UDP RTP-поток данных и связанный с ним RTCP-поток управления используют последовательные порты транспортного уровня. RTP-данные используют четный номер порта, а управляющая информация RTCP использует следующий по порядку (нечетный) номер порта.

Поскольку RTP разработан для поддержки широкого спектра приложений, он предоставляет гибкий механизм, с помощью которого можно разрабатывать новые приложения без необходимости постоянно пересматривать сам протокол RTP. Для каждого класса приложений (например, аудио) RTP определяет *профиль* и один или несколько *форматов*. Профиль предоставляет набор информации, обеспечивающий общее понимание полей заголовка RTP для этого класса приложений, как будет ясно, когда мы детально рассмотрим заголовок. Спецификация формата объясняет, как должны интерпретироваться данные, следующие за заголовком RTP. Например, за заголовком RTP может просто следовать последовательность байтов, каждый из которых представляет собой одноканальный аудиосэмпл, сделанный через определенный интервал после предыдущего. В качестве альтернативы формат данных может быть гораздо более сложным; например, для MPEG-кодированного видеопотока потребуется иметь сложную структуру для представления различных типов информации.

Основные выводы

Проектирование RTP воплощает архитектурный принцип, известный как *фрейминг на уровне приложения* (Application Level Framing, ALF). Этот принцип был предложен Кларком и Тенненхаусом в 1990 году как новый способ проектирования протоколов для новых мультимедийных приложений. Они признали, что эти новые приложения вряд ли будут хорошо обслуживаться существующими протоколами, такими как TCP, и что, более того, они могут не быть хорошо обслужены каким-либо универсальным протоколом. В основе этого принципа лежит убеждение, что приложение лучше всего понимает свои собственные потребности. Например, MPEG-видеоприложение знает, как лучше восстанавливаться после потери кадров и как по-разному реагировать, если потерян I-кадр или B-кадр. То же приложение также лучше всего понимает, как сегментировать данные для передачи — например, лучше отправлять данные различных кадров в разных дейтаграммах, чтобы потеря пакета повредила только один кадр, а не два. Именно по этой причине RTP оставляет так много деталей протокола документам профиля и формата, специфичным для приложения.

Формат заголовка

На рис. 5.25 показан формат заголовка, используемого RTP. Первые 12 байтов всегда присутствуют, тогда как идентификаторы источника, вносящего вклад, используются только в определенных случаях. После этого заголовка могут следовать опциональные расширения заголовка, как описано далее. Наконец, за заголовком следует полезная нагрузка RTP, формат которой определяется приложением. Цель этого заголовка — содержать только те поля, которые, вероятно, будут использоваться многими различными приложениями, так как все, что очень специфично для одного приложения, будет более эффективно передаваться в полезной нагрузке RTP только для этого приложения.

V=2	P	X	СС	M	РТ	Порядковый номер
Временная метка						
Идентификатор источника синхронизации (SSRC)						
Идентификаторы источника вклада (SSRC)						
⋮						
Заголовок расширения						
Полезная нагрузка RTPS						

Рисунок 5.25. Формат заголовка RTP.

Первые два бита являются идентификатором версии, который содержит значение 2 в версии RTP, используемой на момент написания книги. Можно подумать, что разработчики протокола были достаточно смелыми, полагая, что двух бит будет достаточно для всех будущих версий RTP, но напомним, что биты в заголовке RTP являются дефицитными. Кроме того, использование профилей для различных приложений снижает вероятность необходимости частых пересмотров основного протокола RTP. В любом случае, если потребуется версия RTP, превышающая версию 2, можно будет рассмотреть изменение формата заголовка, чтобы стало возможным существование нескольких будущих версий. Например, новый заголовок RTP со значением 3 в поле версии может иметь поле «подверсии» (subversion) где-то еще в заголовке.

Следующий бит — это бит *заполнения* (P, padding), который устанавливается в тех случаях, когда полезная нагрузка RTP была дополнена по какой-то причине. Данные RTP могут быть дополнены для заполнения блока определенного размера, как требуется, например, алгоритмом шифрования. В таком случае полная длина заголовка RTP, данных и заполнения будет передаваться заголовком нижележащего протокола (например, заголовком UDP), а последний байт заполнения будет содержать количество байтов, которые следует игнорировать. Это показано на рис. 5.26. Обратите внимание, что такой подход к заполнению устраняет необходимость в поле длины в заголовке RTP (что служит цели сокращения длины заголовка); в случае отсутствия заполнения длина определяется из заголовка нижележащего протокола.

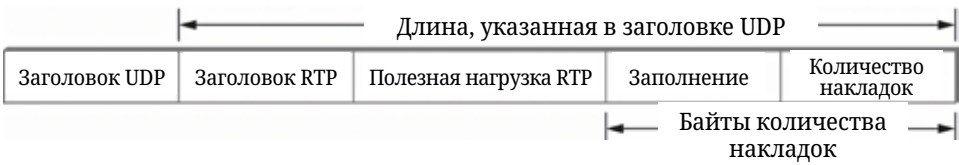


Рисунок 5.26. Вставка в пакет RTP.

Бит *расширения* (X) используется для указания на наличие заголовка расширения, который будет определен для конкретного приложения и следовать за основным заголовком. Такие заголовки используются редко, поскольку обычно возможно определить заголовок, специфичный для полезной нагрузки, как часть определения формата полезной нагрузки для конкретного приложения.

Бит X сопровождается 4-битным полем, которое подсчитывает количество *источников*, вносящих вклад, если они включены в заголовок. Источники, вносящие вклад, обсуждаются ниже.

Мы отметили выше частую необходимость в каком-либо указании кадров; это обеспечивается битом маркера, который имеет специфическое для профиля применение. Например, для голосового приложения он может быть установлен в начале голосового фрагмента. Далее следует 7-битное поле типа полезной нагрузки; оно указывает, какой тип мультимедийных данных передается в этом пакете. Одно из возможных применений этого поля — позволить приложению переключаться с одной схемы кодирования на другую на основе информации о доступности ресурсов в сети или обратной связи о качестве приложения. Точное использование типа полезной нагрузки также определяется профилем приложения.

Обратите внимание, что тип полезной нагрузки обычно не используется в качестве демultiplexирующего ключа для направления данных к различным приложениям (или к различным потокам в рамках одного приложения, таким как аудио- и видеопоток для видеоконференции). Это связано с тем, что такое демultiplexирование обычно осуществляется на нижележащем уровне (например, с помощью UDP, как описано в предыдущей главе). Таким образом, два медиапотока, использующих RTP, обычно будут использовать разные номера портов UDP.

Поле номера последовательности используется для того, чтобы позволить получателю потока RTP обнаруживать отсутствующие и неправильно упорядоченные пакеты. Отправитель просто увеличивает значение на единицу для каждого переданного пакета. Заметьте, что RTP не предпринимает никаких действий при обнаружении потерянного пакета, в отличие от TCP, который и корректирует потерю (путем повторной передачи), и интерпретирует потерю как признак перегрузки (что может привести к уменьшению размера окна). Вместо этого приложение должно самостоятельно решать, что делать при потере пакета, так как это решение, вероятно, сильно зависит от приложения. Например, видеоприложение может решить, что лучше всего при потере пакета воспроизвести последний кадр, который был корректно получен. Некоторые приложения также могут решить изменить свои алгоритмы кодирования для уменьшения потребности в пропускной способности в ответ на потерю пакета, но это не является функцией RTP. Было бы неразумно, если бы RTP решил, что скорость передачи данных должна быть уменьшена, так как это могло бы сделать приложение бесполезным.

Функция поля метки времени состоит в том, чтобы позволить получателю воспроизводить сэмплы через соответствующие интервалы и синхронизировать различные медиапотоки. Поскольку различные приложения могут требовать разной точности измерения времени, сам RTP не указывает единицы измерения времени. Вместо этого метка времени является просто счетчиком «тиков», где время между тиками зависит от используемого кодирования. Например, аудиоприложение, которое берет сэмплы данных каждые 125 мкс, может использовать это значение в качестве разрешения своих часов. Точность часов — это одна из деталей, которая указывается в профиле RTP или формате полезной нагрузки для приложения.

Значение метки времени в пакете представляет собой число, которое соответствует времени, когда был сгенерирован первый сэмпл в пакете. Метка времени не отражает время суток; имеют значение только различия между метками времени. Например, если интервал выборки составляет 125 мкс, а первый сэмпл в пакете n+1 был сгенерирован через 10 мс после первого сэмпла в пакете n, то количество интервалов выборки между этими двумя сэмплами равно

$$\begin{aligned} & \text{TimeBetweenPackets} / \text{TimePerSample} \\ &= (10 \times 10^{-3}) / (125 \times 10^{-6}) = 80 \end{aligned}$$

Предполагая, что точность часов такая же, как и интервал выборки, метка времени в пакете n+1 будет больше, чем в пакете n, на 80. Заметьте, что из-за методов сжатия, таких как обнаружение тишины, могло быть отправлено меньше чем 80 сэмплов, и тем не менее метка времени позволяет получателю воспроизводить сэмплы с правильными временными отношениями.

Источник синхронизации (SSRC) — это 32-битное число, которое уникально идентифицирует один источник потока RTP. В данной мультимедийной конференции каждый отправитель выбирает случайный SSRC и должен разрешать конфликты в маловероятном случае, если два источника выберут одно и то же значение. Благодаря тому, что идентификатор источника не является сетевым или транспортным адресом источника, RTP обеспечивает независимость от протокола нижнего уровня. Это также позволяет одному узлу с несколькими источниками (например, несколькими камерами) различать эти источники. Когда один узел генерирует разные медиапотоки (например, аудио и видео), он не обязан использовать один и тот же SSRC в каждом потоке, так как в RTCP (описанном ниже) существуют механизмы, позволяющие синхронизацию между медиаданными.

Источник, вносящий вклад (CSRC), используется только тогда, когда несколько потоков RTP проходят через микшер. Микшер может использоваться для уменьшения требований к пропускной способности для конференции, получая данные от множества источников и отправляя их в виде одного потока. Например, аудиопотоки от нескольких одновременных ораторов могут быть декодированы и перекодированы в виде одного аудиопотока. В этом случае микшер указывает себя как источник синхронизации, но также перечисляет источники, вносящие вклад — значения SSRC ораторов, которые внесли вклад в данный пакет.

Глава 5.4.3. Протокол управления

RTCP предоставляет поток управления, который связан с потоком данных для мультимедийного приложения. Этот поток управления выполняет три основные функции:

1. Обратная связь о производительности приложения и сети.
2. Способ корреляции и синхронизации различных медиапотоков, поступающих от одного отправителя.
3. Способ передачи идентичности отправителя для отображения на пользовательском интерфейсе.

Первая функция может быть полезна для обнаружения и реагирования на перегрузку. Некоторые приложения могут работать на разных скоростях и могут использовать данные о производительности для решения о применении более агрессивной схемы сжатия для уменьшения перегрузки, например, или для отправки потока более высокого качества, когда перегрузка минимальна. Обратная связь о производительности также может быть полезна при диагностике проблем в сети.

Можно подумать, что вторая функция уже обеспечивается идентификатором источника синхронизации (SSRC) RTP, но на самом деле это не так. Как уже отмечалось, несколько камер с одного узла могут иметь разные значения SSRC. Кроме того, нет требования, чтобы аудио- и видеопоток с одного узла использовали одинаковый SSRC. Поскольку могут происходить столкновения значений SSRC, может возникнуть необходимость изменить значение SSRC потока. Для решения этой проблемы RTCP использует концепцию *канонического имени* (CNAME), которое присваивается отправителю и ассоциируется с различными значениями SSRC, которые могут использоваться этим отправителем.

Просто корреляция двух потоков — это лишь часть проблемы межмедийной синхронизации. Поскольку различные потоки могут иметь совершенно разные часы (с разной точностью и даже разными уровнями дрейфа), необходимо найти способ точно синхронизировать потоки друг с другом. RTCP решает эту проблему, передавая временную информацию, которая коррелирует фактическое время суток с зависящими от частоты часов метками времени, которые передаются в RTP-пакетах данных.

RTCP определяет несколько различных типов пакетов, включая:

- Отчеты отправителя, которые позволяют активным отправителям сеанса сообщать статистику передачи и приема.
- Отчеты получателя, которые используются получателями, не являющимися отправителями, для сообщения статистики приема.

- Описания источника, которые содержат CNAME и другую информацию об отправителе.
- Специфичные для приложений пакеты управления.

Эти различные типы пакетов RTCP отправляются через протокол нижнего уровня, который, как уже отмечалось, обычно является UDP. Несколько пакетов RTCP могут быть упакованы в одну PDU протокола нижнего уровня. Требуется, чтобы в каждой PDU нижнего уровня отправлялись по крайней мере два пакета RTCP: один из них — отчетный пакет, другой — пакет описания источника. Другие пакеты могут быть включены до пределов размера, установленных протоколами нижнего уровня.

Прежде чем подробнее рассмотреть содержимое пакета RTCP, отметим, что существует потенциальная проблема, связанная с тем, что каждый член группы многоканальной передачи посылает периодический управляющий трафик. Если не предпринять меры для его ограничения, этот управляющий трафик может стать значительным потребителем пропускной способности. Например, в аудиоконференции в любой момент времени, вероятно, не более двух или трех отправителей будут передавать аудиоданные, так как нет смысла всем говорить одновременно. Но нет такого социального ограничения на отправку управляющего трафика всеми участниками, и это может стать серьезной проблемой в конференции с тысячами участников. Для решения этой проблемы RTCP имеет набор механизмов, с помощью которых участники уменьшают частоту своих отчетов по мере увеличения числа участников. Эти правила достаточно сложны, но основная цель такова: ограничить общий объем трафика RTCP небольшой долей (обычно 5%) от трафика данных RTP. Для достижения этой цели участники должны знать, какой объем пропускной способности данных будет использоваться (например, объем для отправки трех аудиопотоков) и количество участников. Они узнают первое из внешних средств (известных как *управление сеансами*, обсуждаемое в конце этой главы), а второе — из отчетов RTCP других участников. Поскольку отчеты RTCP могут отправляться с очень низкой частотой, возможно получить только приблизительное количество текущих участников, но этого обычно достаточно. Также рекомендуется выделить больше пропускной способности RTCP активным отправителям, исходя из предположения, что большинство участников хотели бы видеть отчеты от них (например, чтобы узнать, кто говорит).

Как только участник определил, какое количество пропускной способности он может использовать для трафика RTCP, он начинает отправлять периодические отчеты с соответствующей частотой. Отчеты отправителя и отчеты получателя различаются лишь тем, что первые включают дополнительную информацию о отправителе. Оба типа отчетов содержат информацию о данных, полученных от всех источников за последний отчетный период.

Дополнительная информация в отчете отправителя состоит из:

- Метки времени, содержащей фактическое время суток, когда был создан этот отчет.
- Метки времени RTP, соответствующей времени создания отчета.
- Кумулятивного количества пакетов и байтов, отправленных этим отправителем с момента начала передачи.

Следует отметить, что первые два параметра могут быть использованы для синхронизации различных медиапотоков от одного источника, даже если эти потоки используют разные единицы измерения времени в своих потоках данных RTP, поскольку это дает ключ к преобразованию времени суток в метки времени RTP.

И отчеты отправителя, и отчеты получателя содержат один блок данных на каждый источник, о котором было известно с момента последнего отчета. Каждый блок содержит следующую статистику для соответствующего источника:

- Его SSRC.
- Доля потерянных пакетов данных от этого источника с момента отправки последнего отчета (вычисляется путем сравнения количества полученных пакетов с количеством ожидаемых пакетов; это последнее значение можно определить по номерам последовательностей RTP).

- Общее количество потерянных пакетов от этого источника с момента первого получения данных от него.
- Наивысший номер последовательности, полученный от этого источника (расширен до 32 бит, чтобы учитывать переполнение номера последовательности).
- Оценка интервалов между прибытием пакетов для этого источника (вычисляется путем сравнения интервалов между прибытием полученных пакетов с ожидаемыми интервалами во время передачи).
- Последняя фактическая метка времени, полученная через RTCP для этого источника.
- Задержка с момента последнего отчета отправителя, полученного через RTCP для этого источника.

Как можно представить, получатели этой информации могут узнать множество данных о состоянии сеанса. В частности, они могут увидеть, получают ли другие получатели гораздо лучшее качество от какого-то отправителя, чем они сами, что может указывать на необходимость резервирования ресурсов или на проблему в сети, требующую внимания. Кроме того, если отправитель заметит, что многие получатели испытывают высокие потери его пакетов, он может решить, что следует уменьшить скорость отправки или использовать схему кодирования, более устойчивую к потерям.

Последний аспект RTCP, который мы рассмотрим, это пакет описания источника. Такой пакет содержит, как минимум, SSRC отправителя и CNAME отправителя. Каноническое имя (CNAME) формируется таким образом, чтобы все приложения, генерирующие медиапотoki, которые могут потребовать синхронизации (например, отдельно генерируемые аудио- и видеопотоки от одного пользователя), выбирали одно и то же CNAME, даже если они выбирают разные значения SSRC. Это позволяет получателю идентифицировать медиапоток, пришедший от одного и того же отправителя. Наиболее распространенный формат CNAME — это где *host* — полное доменное имя отправляющей машины. Таким образом, приложение, запущенное пользователем с именем *jdoe*, работающее на этой машине, будет использовать эту строку в качестве своего CNAME. Большое и переменное количество байтов, используемых в этом представлении, делает его плохим выбором для формата SSRC, поскольку SSRC отправляется с каждым пакетом данных и должен обрабатываться в реальном времени. Позволяя связывать CNAME с SSRC в периодических сообщениях RTCP, удастся достичь компактного и эффективно формата для SSRC.

В пакете описания источника могут быть включены и другие элементы, такие как настоящее имя и адрес электронной почты пользователя. Эти данные используются в пользовательских интерфейсах и для контакта с участниками, но они менее важны для работы RTP, чем CNAME.

Подобно TCP, RTP и RTCP представляют собой довольно сложную пару протоколов. Эта сложность во многом обусловлена желанием облегчить жизнь разработчикам приложений. Поскольку существует бесконечное количество возможных приложений, задача при разработке транспортного протокола заключается в том, чтобы сделать его достаточно универсальным для удовлетворения разнообразных потребностей различных приложений, при этом не делая сам протокол слишком сложным для реализации. В этом отношении RTP оказался очень успешным, став основой для многих мультимедийных приложений реального времени, работающих в Интернете сегодня.

Перспектива: HTTP — это новая «узкая талия»

Интернет часто описывают как архитектуру с *узкой талией*, где один универсальный протокол (IP) находится посередине, расширяясь, чтобы поддерживать множество транспортных и прикладных протоколов выше него (например, TCP, UDP, RTP, SunRPC, DCE-RPC, gRPC, SMTP, HTTP, SNMP) и работая на основе многих сетевых технологий ниже (например, Ethernet, PPP, Wi-Fi, SONET, ATM). Эта общая структура была ключевым фактором для повсеместного распространения Интернета: благодаря минималистичному IP-

слою, с которым все должны были согласиться, множество решений могли развиваться как выше, так и ниже него. Это теперь широко признанная стратегия для любой платформы, стремящейся к универсальному принятию.

Но за последние 30 лет произошло нечто другое. Поскольку не были учтены все проблемы, с которыми Интернет столкнулся по мере роста (например, безопасность, перегрузка, мобильность, отзывчивость в реальном времени и так далее), стало необходимо вводить ряд дополнительных функций в архитектуру Интернета. Наличие универсальных адресов IP и модели обслуживания по принципу «лучшее усилие» было необходимым условием для принятия, но недостаточным фундаментом для всех приложений, которые люди хотели создать.

Некоторые из этих решений мы еще увидим — в следующих разделах будет описано, как Интернет управляет перегрузками (раздел 6), обеспечивает безопасность (раздел 8) и поддерживает мультимедийные приложения в реальном времени (разделы 7 и 9), — но уже сейчас полезно сопоставить ценность универсальной «узкой талии» с эволюцией, которая неизбежно происходит в любой долгоживущей системе: «фиксированная точка», вокруг которой эволюционирует остальная часть архитектуры, переместилась на новое место в программном стеке. Короче говоря, HTTP стал новой «узкой талией»; тем общим/предполагаемым элементом глобальной инфраструктуры, который делает возможным все остальное. Это произошло не мгновенно и не по указу, хотя некоторые предвидели, что это случится. «Узкая талия» медленно поднялась по протокольному стеку как следствие эволюции (смешивая геонаучные и биологические метафоры).

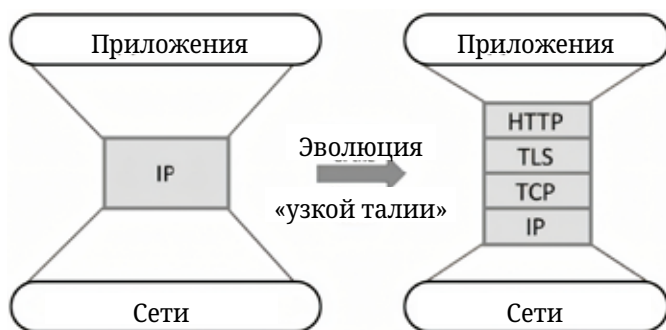


Рисунок 5.27. HTTP (плюс TLS, TCP и IP) образует «узкую талию» современной архитектуры Интернета.

Отнесение ярлыка «узкой талии» исключительно к HTTP — это упрощение. На самом деле это командная работа, с комбинацией HTTP/TLS/TCP/IP, теперь служащей общей платформой Интернета.

- HTTP предоставляет глобальные идентификаторы объектов (URI) и простой интерфейс GET/PUT.
- TLS обеспечивает безопасность связи от конца до конца.
- TCP обеспечивает управление соединениями, надежную передачу данных и контроль перегрузок.
- IP предоставляет глобальные адреса узлов и уровень абстракции сети.

Иными словами, хотя вы при желании можете изобрести свой собственный алгоритм контроля перегрузок, TCP решает эту проблему довольно хорошо, поэтому имеет смысл повторно использовать это решение. Аналогично, хотя вы свободны изобретать свой собственный протокол RPC, HTTP предоставляет вполне пригодный вариант (который, благодаря встроенной проверенной безопасности, дополнительно не блокируется корпоративными брандмауэрами), поэтому опять-таки имеет смысл его использовать, а не изобретать колесо заново.

Менее очевидно, но HTTP также предоставляет хорошую основу для работы с мобильностью. Если ресурс, к которому вы хотите получить доступ, переместился, HTTP может вернуть *ответ с перенаправлением*, указывающим клиенту новое местоположение. Аналогично HTTP позволяет внедрять *кеширующие прокси-серверы* между клиентом и сервером, что позволяет реплицировать популярный контент в нескольких местоположениях и экономить клиентам время, уходящее на пересечение всего Интернета для получения какой-либо информации. (Обе эти возможности обсуждаются в главе 9.1.) Наконец, HTTP используется для доставки мультимедиа в реальном времени, в подходе, известном как *адаптивная потоковая передача*. (Подробнее об этом — в главе 7.2.)

Раздел 6.

Контроль перегрузок

Десница, создавшая вас прекрасной, создала вас и добродетельной.

Уильям Шекспир

Проблема: распределение ресурсов

К настоящему моменту мы рассмотрели достаточно уровней иерархии сетевых протоколов, чтобы понять, как данные могут передаваться между процессами в гетерогенных сетях. Теперь мы перейдем к проблеме, охватывающей весь протокольный стек — как эффективно и справедливо распределять ресурсы среди множества конкурирующих пользователей. Ресурсы, которые распределяются, включают пропускную способность каналов связи и буферы на маршрутизаторах или коммутаторах, где пакеты ставятся в очередь в ожидании передачи. Пакеты *конкурируют* на маршрутизаторе за использование канала, и каждый конкурирующий пакет помещается в очередь, ожидая своей очереди на передачу по каналу. Когда слишком много пакетов конкурируют за один и тот же канал, очередь заполняется, и происходят два нежелательных события: пакеты испытывают увеличенную задержку от конца до конца, а в худшем случае очередь переполняется, и пакеты приходится сбрасывать. Когда длинные очереди продолжаются и сбросы становятся обычным явлением, говорят, что сеть *перегружена*. Большинство сетей имеют *механизм контроля перегрузок* для решения такой ситуации.

Контроль перегрузок и распределение ресурсов — это две стороны одной медали. С одной стороны, если сеть берет на себя активную роль в распределении ресурсов (например, планируя, какой виртуальный канал будет использовать данный физический канал в определенный период времени), тогда перегрузки можно избежать, делая контроль перегрузок ненужным. Однако распределение сетевых ресурсов с любой точностью затруднительно, потому что ресурсы, о которых идет речь, распределены по всей сети; необходимо планировать использование множества каналов, соединяющих серию маршрутизаторов. С другой стороны, всегда можно позволить источникам пакетов отправлять столько данных, сколько они хотят, а затем устранять перегрузки, если они возникнут. Это более простой подход, но он может быть разрушительным, поскольку множество пакетов может быть отброшено сетью до того, как перегрузки станут контролируруемыми. Более того, именно в те моменты, когда сеть перегружена, то есть ресурсы стали дефицитными по сравнению со спросом, необходимость в распределении ресурсов среди конкурирующих пользователей ощущается наиболее остро. Существуют также промежуточные решения, при которых принимаются неточные решения о распределении, но перегрузки все равно могут возникнуть, и поэтому все равно нужен механизм для их устранения. Называете ли вы такое смешанное решение контролем перегрузок или распределением ресурсов, не имеет значения. В каком-то смысле это и то и другое.

Контроль перегрузок и распределение ресурсов включают как хосты, так и сетевые элементы, такие как маршрутизаторы. В сетевых элементах можно использовать различные дисциплины очередей для управления порядком, в котором пакеты передаются, и определения того, какие пакеты сбрасываются. Дисциплина очереди также может сегрегировать трафик, чтобы пакеты одного пользователя не оказывали чрезмерного влияния на пакеты другого пользователя. На конечных хостах механизм контроля перегрузок регулирует, с какой скоростью источники могут отправлять пакеты. Это делается с целью предотвратить возникновение перегрузок, а в случае их возникновения помочь устранить их.

Этот раздел начинается с обзора контроля перегрузок и распределения ресурсов. Затем мы обсудим различные дисциплины очередей, которые могут быть реализованы на маршрутизаторах внутри сети, после чего идет описание алгоритма контроля перегрузок, предоставляемого TCP на хостах. В четвертой подглаве рассматриваются различные техники,

включающие как маршрутизаторы, так и хосты, которые направлены на предотвращение перегрузок до того, как они станут проблемой. Наконец, мы рассмотрим широкую область *качества обслуживания*. Мы рассмотрим потребности приложений в получении различных уровней распределения ресурсов в сети и опишем несколько способов, с помощью которых они могут запросить эти ресурсы, а сеть может удовлетворить эти запросы.

Глава 6.1. Вопросы распределения ресурсов

Распределение ресурсов и контроль перегрузок — сложные вопросы, которые изучаются с момента создания первой сети. Они до сих пор являются активными областями исследований. Один из факторов, делающих эти вопросы сложными, заключается в том, что они не ограничиваются одним уровнем иерархии протоколов. Распределение ресурсов частично реализуется в маршрутизаторах, коммутаторах и каналах связи внутри сети, а частично в транспортном протоколе, работающем на конечных хостах. Конечные системы могут использовать сигнальные протоколы для передачи своих требований к ресурсам узлам сети, которые отвечают информацией о доступности ресурсов. Одной из главных целей этой главы является определение рамок, в которых эти механизмы можно понять, а также предоставление соответствующих деталей о представительном наборе механизмов.

Следует уточнить нашу терминологию, прежде чем двигаться дальше. Под *распределением ресурсов* мы понимаем процесс, с помощью которого сетевые элементы пытаются удовлетворить конкурирующие требования приложений к сетевым ресурсам — в первую очередь к пропускной способности канала и буферному пространству в маршрутизаторах или коммутаторах. Конечно, часто не удается удовлетворить все требования, а это означает, что некоторые пользователи или приложения могут получать меньше сетевых ресурсов, чем они хотят. Часть проблемы распределения ресурсов заключается в решении, когда и кому сказать «нет».

Мы используем термин *«управление перегрузками»* для описания усилий сетевых узлов по предотвращению или реагированию на условия перегрузки. Поскольку перегрузка обычно вредна для всех, первоочередной задачей является уменьшение перегрузки или предотвращение ее возникновения. Это может быть достигнуто просто путем убеждения нескольких хостов прекратить отправку данных, тем самым улучшая ситуацию для всех остальных. Однако чаще механизмы контроля перегрузок имеют аспект справедливости, то есть они пытаются равномерно распределить неудобства среди всех пользователей, а не причинять большие неудобства немногим. Таким образом, мы видим, что многие механизмы контроля перегрузок имеют встроенные элементы распределения ресурсов.

Также важно понимать разницу между управлением потоком (flow control) и контролем перегрузок. Управление потоком предполагает предотвращение ситуации, когда быстрый отправитель перегружает медленного получателя. Контроль перегрузок, напротив, направлен на предотвращение ситуации, когда набор отправителей отправляет слишком много данных *в сеть* из-за нехватки ресурсов в какой-то точке. Эти два понятия часто путают; как мы увидим, они также имеют общие механизмы.

Глава 6.1.1. Модель сети

Начнем с определения трех важных особенностей сетевой архитектуры. По большей части это сводка материала, представленного в предыдущих главах, который имеет отношение к проблеме распределения ресурсов.

Сеть с пакетной коммутацией

Мы рассматриваем распределение ресурсов в пакетно-коммутируемой сети (или интернете), состоящей из множества каналов и коммутаторов (или маршрутизаторов). Поскольку большинство механизмов, описанных в этой главе, были разработаны

для использования в Интернете и поэтому изначально определялись в терминах маршрутизаторов, мы будем использовать термин *«маршрутизатор»* на протяжении всего обсуждения. Проблема по существу одинакова, будь то в сети или в интернете.

В такой среде у данного источника может быть достаточно емкости на исходящем канале для отправки пакета, но где-то в середине сети его пакеты сталкиваются с каналом, который используется многими различными источниками трафика. На рис. 6.1 иллюстрируется эта ситуация — два высокоскоростных канала питают низкоскоростной канал. Это в отличие от сетей с общим доступом, таких как Ethernet и беспроводные сети, где источник может напрямую наблюдать за трафиком в сети и соответственно решать, отправлять пакет или нет. Мы уже видели алгоритмы, используемые для распределения пропускной способности в сетях с общим доступом (например, Ethernet и Wi-Fi). Эти алгоритмы управления доступом в некотором смысле аналогичны алгоритмам контроля перегрузок в коммутируемой сети.

Основные выводы

Важно отметить, что контроль перегрузок — это другая проблема, не связанная с маршрутизацией. Хотя верно, что перегруженный канал может быть назначен с большим весом ребра протоколом маршрутизации и, как следствие, маршрутизаторы будут обходить его, «обход» перегруженного канала обычно не решает проблему перегрузки. Чтобы понять это, достаточно взглянуть на простую сеть, изображенную на рис. 6.1, где весь трафик должен проходить через один и тот же маршрутизатор, чтобы достичь пункта назначения. Хотя это и утрированный пример, часто встречается ситуация, когда определенный маршрутизатор невозможно обойти. Этот маршрутизатор может стать перегруженным, и механизм маршрутизации ничего не сможет с этим поделать. Такой перегруженный маршрутизатор иногда называют *«узким местом»* (bottleneck router).

Потоки без подключения

На протяжении большей части нашего обсуждения мы предполагаем, что сеть по сути не имеет соединений, а все сервисы, ориентированные на соединения, реализованы в транспортном протоколе, который запущен на конечных узлах. (Это именно та модель Интернета, в которой IP предоставляет услугу доставки дейтаграмм без соединения, а TCP реализует абстракцию сквозного соединения. Обратите внимание, что это предположение не выполняется в сетях виртуальных каналов, таких как ATM и X.25. В таких сетях сообщение об установке соединения проходит через всю сеть, когда устанавливается цепь. Это сообщение резервирует набор буферов для соединения на каждом маршрутизаторе, тем самым обеспечивая форму контроля перегрузки — соединение устанавливается только в том случае, если на каждом маршрутизаторе для него может быть выделено достаточно буферов. Основным недостатком этого подхода заключается в том, что он приводит к неполному использованию ресурсов — буферы, зарезервированные для конкретного канала, недоступны для использования другим трафиком, даже если в данный момент они не используются этим каналом. Основное внимание в этом разделе уделяется подходам к распределению ресурсов, которые применяются в интернет-сети, и поэтому мы в основном рассматриваем сети без соединений.

Нам нужно уточнить термин *«без подключений»*, потому что наша классификация сетей как «без подключений» или «с подключениями» несколько ограничена; существует серая зона между ними. В частности, предположение о том, что все дейтаграммы полностью независимы в сети без подключений, слишком сильное. Дейтаграммы, конечно, переключаются независимо, но обычно поток дейтаграмм между определенной парой хостов проходит через определенный набор маршрутизаторов. Эта идея потока — последовательность пакетов, отправляемых между парой «источник/назначение» и следующих по одному и тому же маршруту через сеть — является важной абстракцией в контексте распределения ресурсов; это понятие, которое мы будем использовать в данном разделе.

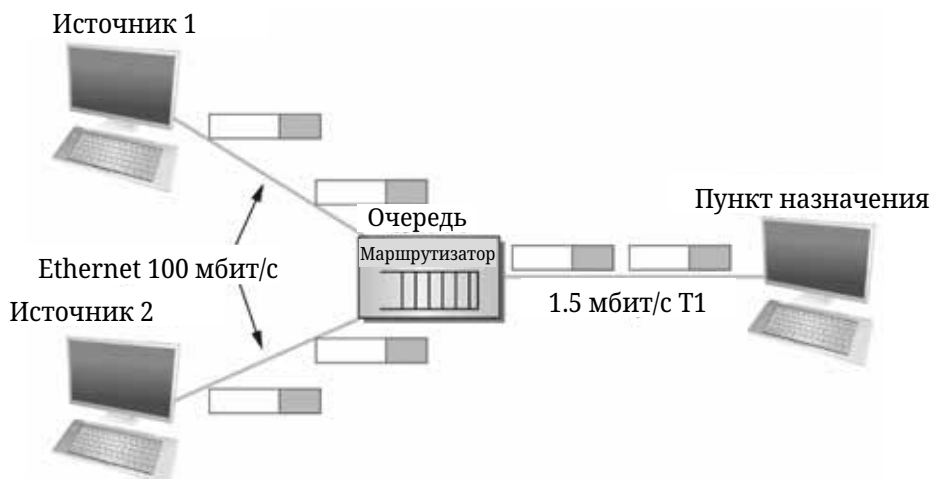


Рисунок 6.1. Потенциальный маршрутизатор «узкого места».

Одно из преимуществ абстракции потока заключается в том, что потоки могут быть определены на разных уровнях гранулярности. Например, поток может быть от хоста к хосту (т.е. иметь одинаковые адреса «источник/назначение») или от процесса к процессу (т.е. иметь одинаковые пары «хост/порт» — «источник/назначение»). В последнем случае поток по сути такой же, как канал, как мы использовали этот термин на протяжении всей книги. Причина, по которой мы вводим новый термин, заключается в том, что поток виден маршрутизаторам внутри сети, тогда как канал — это сквозная абстракция. На рис. 6.2 иллюстрируются несколько потоков, проходящих через серию маршрутизаторов.

Поскольку множество связанных пакетов проходит через каждый маршрутизатор, иногда имеет смысл сохранять некоторую информацию о состоянии для каждого потока, информацию, которая может быть использована для принятия решений о распределении ресурсов для пакетов, принадлежащих этому потоку. Это состояние иногда называют *мягким состоянием* (soft state). Основное отличие мягкого состояния от *жесткого состояния* (hard state) заключается в том, что мягкое состояние не всегда должно быть явно создано и удалено с помощью сигнальной передачи. Мягкое состояние представляет собой промежуточное положение между сетью без подключений, которая не *хранит состояние* в маршрутизаторах, и полностью подключенной сетью, которая хранит жесткое состояние в маршрутизаторах. В общем, правильная работа сети не зависит от наличия мягкого состояния (каждый пакет все равно маршрутизируется корректно без учета этого состояния), но когда пакет принадлежит потоку, для которого маршрутизатор в данный момент поддерживает мягкое состояние, маршрутизатор лучше справляется с обработкой этого пакета.

Заметьте, что поток может быть либо неявно определенным, либо явно установленным. В первом случае каждый маршрутизатор следит за пакетами, которые проходят между одной и той же парой «источник/получатель» — маршрутизатор делает это, проверяя адреса в заголовке — и обрабатывает эти пакеты как принадлежащие одному и тому же потоку для целей избежания перегрузок. Во втором случае источник отправляет сообщение об установке потока через сеть, заявляя, что поток пакетов вот-вот начнется. Хотя явные потоки, возможно, ничем не отличаются от подключения в сети с коммутацией каналов, мы обращаем внимание на этот случай, потому что, даже будучи явно установленным, поток не подразумевает каких-либо сквозных семантик и, в частности, не подразумевает надежную и упорядоченную доставку виртуального канала. Он просто существует для целей распределения ресурсов. В этой главе мы увидим примеры как неявных, так и явных потоков.

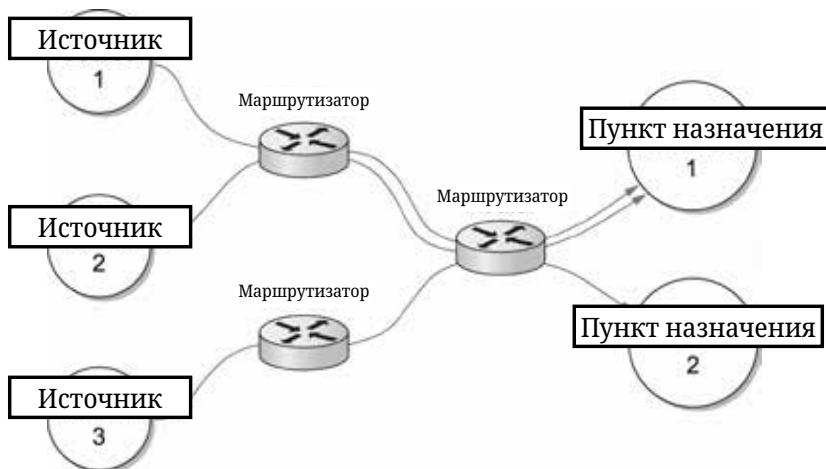


Рисунок 6.2. Несколько потоков, проходящих через набор маршрутизаторов.

Модель обслуживания

В начале этой главы мы сосредоточимся на механизмах, которые предполагают модель обслуживания по принципу наилучшего усилия (best effort) в Интернете. При обслуживании по принципу наилучшего усилия все пакеты получают в основном равное отношение, и конечные хосты не имеют возможности просить у сети, чтобы некоторые пакеты или потоки получали определенные гарантии или привилегированное обслуживание. Определение модели обслуживания, которая поддерживает некий вид предпочтительного обслуживания или гарантии — например, гарантирование полосы пропускания, необходимой для видеопотока — является предметом обсуждения в следующей главе. Такая модель обслуживания предполагает предоставление нескольких уровней качества обслуживания (QoS). Как мы увидим, существует целый спектр возможностей, начиная от модели обслуживания по принципу наилучшего усилия до такой, в которой отдельные потоки получают количественные гарантии QoS. Одной из самых больших задач является определение модели обслуживания, которая удовлетворяет потребностям широкого спектра приложений и даже корректно работает при приложениях, которые будут изобретены в будущем.

Глава 6.1.2. Таксономия

Существует бесчисленное множество способов, которыми механизмы распределения ресурсов отличаются, поэтому создание тщательной таксономии является сложной задачей. Пока мы описываем три измерения, по которым можно охарактеризовать механизмы распределения ресурсов; более тонкие различия будут отмечены в ходе этой главы.

Ориентированность на маршрутизатор против ориентированности на хост

Механизмы распределения ресурсов можно классифицировать на две широкие группы: те, которые решают проблему изнутри сети (то есть на маршрутизаторах или коммутаторах), и те, которые решают ее с краев сети (то есть на хостах, возможно, внутри транспортного протокола). Поскольку и маршрутизаторы внутри сети, и хосты на краях сети участвуют в распределении ресурсов, реальный вопрос заключается в том, где лежит основное бремя.

В дизайне, ориентированном на маршрутизатор, каждый маршрутизатор берет на себя ответственность за решение, когда пакеты будут переданы, за выбор, какие пакеты будут отброшены, а также за информирование хостов, которые генерируют сетевой трафик, о том, сколько пакетов им разрешено отправлять. В дизайне, ориентированном на хост, конечные хосты наблюдают за состоянием сети (например, сколько пакетов им удастся успешно передать через сеть) и соответствующим образом корректируют свое поведение. Заметьте, что эти две группы не являются взаимоисключающими. Например, сеть, которая возлагает основное бремя управления перегрузками на маршрутизаторы, все равно ожидает, что конечные хосты будут соблюдать любые рекомендательные сообщения, которые маршрутизаторы отправляют, тогда как маршрутизаторы в сетях, использующих сквозной контроль перегрузок, все равно имеют какую-то политику, насколько бы простой она ни была, для решения, какие пакеты отбросить, когда их очереди переполняются.

Механизмы на основе резервирования против механизмов на основе обратной связи

Второй способ классификации механизмов распределения ресурсов — это использование *резервирования* или *обратной связи*. В системе на основе резервирования некая сущность (например, конечный хост) запрашивает у сети определенное количество пропускной способности для потока. Каждый маршрутизатор затем выделяет достаточно ресурсов (буферов и/или процента полосы пропускания канала), чтобы удовлетворить этот запрос. Если запрос не может быть удовлетворен на каком-то маршрутизаторе, потому что это приведет к перегрузке его ресурсов, то маршрутизатор отклоняет резервирование. Это аналогично получению сигнала «занято» при попытке совершить телефонный звонок. В подходе на основе обратной связи конечные хосты начинают отправлять данные, не резервируя предварительно пропускную способность, и затем регулируют свою скорость отправки в соответствии с полученной обратной связью. Эта обратная связь может быть либо *явной* (т.е. перегруженный маршрутизатор отправляет хосту сообщение «пожалуйста, замедлитесь»), либо *неявной* (т.е. конечный хост регулирует свою скорость отправки в соответствии с внешне наблюдаемым поведением сети, таким как потери пакетов).

Заметьте, что система на основе резервирования всегда подразумевает механизм, который ориентирован на маршрутизатор. Это потому, что каждый маршрутизатор несет ответственность за отслеживание доступности своей пропускной способности и принимает решение о возможности принятия новых резервирований. Маршрутизаторы также должны убедиться, что каждый хост соблюдает сделанное им резервирование. Если хост отправляет данные быстрее, чем указано при резервировании, то пакеты этого хоста становятся хорошими кандидатами для отбрасывания в случае перегрузки маршрутизатора. С другой стороны, система на основе обратной связи может подразумевать как ориентированный на маршрутизатор, так и ориентированный на хост механизм. Обычно, если обратная связь явная, то маршрутизатор участвует в схеме распределения ресурсов в какой-то степени. Если обратная связь неявная, то почти все бремя ложится на конечный хост; маршрутизаторы тихо отбрасывают пакеты при перегрузке.

Резервирование не обязательно должно выполняться конечными хостами. Администратор сети может распределять ресурсы для потоков или для более крупных агрегатов трафика, как мы увидим в следующей главе.

На основе окна либо на основе скорости

Третий способ охарактеризовать механизмы распределения ресурсов — это использование *окна* или *скорости*. Это одна из областей, упомянутых выше, где схожие ме-

механизмы и терминология используются как для управления потоком, так и для управления перегрузками. Оба механизма управления потоком и распределения ресурсов нуждаются в способе выразить отправителю, сколько данных ему разрешено передавать. Существует два общих способа сделать это: с помощью *окна* или *скорости*. Мы уже видели транспортные протоколы на основе окна, такие как TCP, в которых получатель рекламирует окно отправителю. Это окно соответствует тому, сколько места в буфере у получателя, и ограничивает количество данных, которые отправитель может передать; то есть это поддерживает управление потоком. Похожий механизм — реклама окна — может использоваться внутри сети для резервирования места в буфере (то есть для поддержки распределения ресурсов). Механизмы управления перегрузками в TCP основаны на окнах.

Также возможно контролировать поведение отправителя с помощью скорости — то есть сколько бит в секунду может принять получатель или сеть. Контроль на основе скорости имеет смысл для многих мультимедийных приложений, которые обычно генерируют данные с некоторой средней скоростью и которым требуется хотя бы минимальная пропускная способность для нормальной работы. Например, видеокodeк может генерировать видео со средней скоростью 1 Мбит/с с пиковой скоростью 2 Мбит/с. Как мы увидим позже в этой главе, характеристика потоков на основе скорости является логическим выбором в системе на основе резервирования, поддерживающей разные уровни качества обслуживания — отправитель делает резервирование на определенное количество бит в секунду, и каждый маршрутизатор на пути определяет, может ли он поддерживать эту скорость с учетом других потоков, на которые он уже взял обязательства.

Резюме таксономии распределения ресурсов

Классификация подходов к распределению ресурсов по двум различным точкам в рамках каждого из трех измерений, как мы только что сделали, предполагает до восьми уникальных стратегий. Хотя восемь различных подходов, безусловно, возможны, на практике мы отмечаем, что наиболее распространены две общие стратегии; эти две стратегии связаны с базовой моделью обслуживания сети.

С одной стороны, модель обслуживания по принципу наилучшего усилия обычно предполагает использование обратной связи, поскольку такая модель не позволяет пользователям резервировать пропускную способность сети. Это, в свою очередь, означает, что большая часть ответственности за контроль перегрузок ложится на конечные хосты, возможно, с некоторой помощью от маршрутизаторов. На практике такие сети используют информацию на основе окон. Это общая стратегия, принятая в Интернете.

С другой стороны, модель обслуживания, основанная на качестве обслуживания (QoS), вероятно, подразумевает какую-то форму резервирования. Поддержка этих резервирований, вероятно, требует значительного участия маршрутизаторов, таких как различная очередь для пакетов в зависимости от уровня резервированных ресурсов, которые им требуются. Более того, естественно выражать такие резервирования в терминах скорости, поскольку окна лишь косвенно связаны с тем, сколько полосы пропускания требуется пользователю от сети. Мы обсудим эту тему в следующей главе.

Глава 6.1.3. Критерии оценки

Последний вопрос заключается в том, как узнать, является ли механизм распределения ресурсов хорошим или нет. Напомним, что в постановке задачи в начале этого раздела мы задали вопрос, как сеть *эффективно* и *справедливо* распределяет свои ресурсы. Это предполагает по крайней мере два глобальных критерия, по которым можно оценить схему распределения ресурсов. Рассмотрим каждый из них по очереди.

Эффективное распределение ресурсов

Хорошим началом для оценки эффективности схемы распределения ресурсов является рассмотрение двух основных показателей сетей: пропускной способности и задержки. Очевидно, что мы хотим иметь как можно большую пропускную способность и как можно меньшую задержку. К сожалению, эти цели часто противоречат друг другу.

Один из надежных способов для алгоритма распределения ресурсов, помогающих увеличить пропускную способность — это позволить как можно большему количеству пакетов попасть в сеть, чтобы довести использование всех каналов до 100 %. Мы делали бы это, чтобы избежать ситуации, когда канал простаивает, потому что простаивающий канал обязательно снижает пропускную способность. Проблема с этой стратегией заключается в том, что увеличение количества пакетов в сети также увеличивает длину очередей на каждом маршрутизаторе. Более длинные очереди, в свою очередь, означают, что пакеты дольше задерживаются в сети.

Для описания этой зависимости некоторые сетевые разработчики предложили использовать отношение пропускной способности к задержке в качестве меры оценки эффективности схемы распределения ресурсов. Это отношение иногда называют *мощностью сети*:

$$\text{Мощность (Power)} = \text{Пропускная способность (Throughput)} / \text{Задержка (Delay)}$$

Заметьте, что не очевидно, что мощность является правильной метрикой для оценки эффективности распределения ресурсов. Во-первых, теория, лежащая в основе мощности, основана на сетях с бесконечными очередями M/M/1;¹ реальные сети имеют конечные буферы и иногда вынуждены отбрасывать пакеты. Во-вторых, мощность обычно определяется относительно одного соединения (потока); не ясно, как это распространяется на несколько конкурирующих соединений. Несмотря на эти довольно серьезные ограничения, никакие другие альтернативы не получили широкого признания, и поэтому мощность продолжает использоваться.

Цель заключается в максимизации этого отношения, которое зависит от того, какую нагрузку вы возлагаете на сеть. Нагрузка, в свою очередь, устанавливается механизмом распределения ресурсов.

На рис. 6.3 представлена типичная кривая мощности, где, в идеале, механизм распределения ресурсов должен работать на пике этой кривой. Слева от пика механизм действует слишком консервативно; то есть он не позволяет отправить достаточно пакетов, чтобы занять каналы. Справа от пика в сеть допускается так много пакетов, что увеличения задержки из-за очередей начинают доминировать над любыми небольшими приростами пропускной способности.

Интересно, что эта кривая мощности очень похожа на кривую пропускной способности системы в многозадачной вычислительной системе. Пропускная способность системы улучшается по мере того, как в систему допускается больше заданий, до тех пор, пока не наступит момент, когда заданий станет так много, что система начинает *зацикливаться* (тратит все время на обмен страницами памяти), и пропускная способность начинает падать.

¹ Поскольку это не книга по теории очередей, мы приводим лишь краткое описание очереди M/M/1. 1 означает, что в ней один сервер, а Ms означает, что распределение времени прибытия и обслуживания пакетов является *марковским*, то есть экспоненциальным.



Рисунок 6.3. Отношение пропускной способности к задержке в зависимости от нагрузки.

Как мы увидим в последующих главах этого раздела, многие схемы контроля перегрузок способны управлять нагрузкой только очень грубыми способами; то есть просто невозможно «повернуть ручку» немного и позволить только небольшому количеству дополнительных пакетов войти в сеть. В результате проектировщики сетей должны беспокоиться о том, что происходит, даже когда система работает при экстремально высокой нагрузке — то есть в крайней правой части кривой на рис. 6.3. Было бы идеально, если бы мы могли избежать ситуации, в которой пропускная способность системы становится нулевой из-за заикливания системы. В терминологии сетей мы хотим, чтобы система была *стабильной* — чтобы пакеты продолжали проходить через сеть, даже когда она работает при высокой нагрузке. Если механизм не является стабильным, сеть может столкнуться с коллапсом перегрузки.

Справедливое распределение ресурсов

Эффективное использование сетевых ресурсов — это не единственный критерий для оценки схемы распределения ресурсов. Мы также должны рассмотреть вопрос справедливости. Однако мы сразу сталкиваемся с трудностями, когда пытаемся определить, что именно представляет собой справедливое распределение ресурсов. Например, схема распределения ресурсов, основанная на резервировании, предоставляет явный способ создания контролируемой несправедливости. С такой схемой мы можем использовать резервирование, чтобы позволить видеопотоку получать 1 Мбит/с через какой-то канал, в то время как передача файла получает только 10 Кбит/с через тот же канал.

При отсутствии явной информации об обратном, когда несколько потоков делят определенный канал, мы хотим, чтобы каждый поток получал равную долю пропускной способности. Это определение предполагает, что *справедливая* доля пропускной способности означает равную долю пропускной способности. Но даже при отсутствии резервирования равные доли могут не соответствовать справедливым долям. Следует ли нам также учитывать длину сравниваемых путей? Например, как показано на рис. 6.4, что является справедливым, когда один четырехходовой поток конкурирует с тремя одноходовыми потоками?

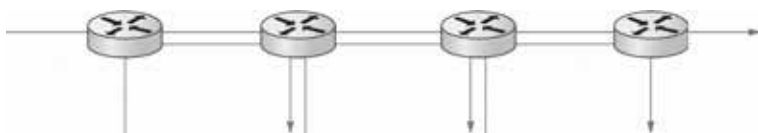


Рисунок 6.4. Один четырехходовой поток конкурирует с тремя одноходовыми потоками.

Предполагая, что справедливость означает равенство и что все пути имеют одинаковую длину, исследователь сетей Радж Джайн предложил метрику, которую можно использовать для количественной оценки справедливости механизма контроля перегрузок. Индекс справедливости Джайна определяется следующим образом. Учитывая набор пропускных способностей потоков (измеренных в согласованных единицах, таких как биты/секунда), следующая функция назначает индекс справедливости потокам:

$$(x_1, x_2, \dots, x_n)$$

(измеряется в последовательных единицах, таких как бит/секунду), следующая функция присваивает потокам индекс справедливости:

$$f(x_1, x_2, \dots, x_n) = \frac{\sum_{i=1}^n x_i}{n \sum_{i=1}^n x_i^2}$$

Индекс справедливости всегда дает число от 0 до 1, причем 1 представляет наибольшую справедливость. Чтобы понять интуицию, стоящую за этой метрикой, рассмотрим случай, когда все n потоков получают пропускную способность 1 единица данных в секунду. Мы видим, что индекс справедливости в этом случае равен:

$$\frac{n^2}{n \times n} = 1$$

Теперь предположим, что один поток получает пропускную способность $1 + \Delta$. Теперь индекс справедливости равен:

$$\frac{((n-1)+1+\Delta)^2}{n(n-1+(1+\Delta)^2)} = \frac{n^2 + 2n\Delta + \Delta^2}{n^2 + 2n\Delta + n\Delta^2}$$

Заметьте, что знаменатель превышает числитель на $((n-1)\Delta^2)$. Таким образом, независимо от того, получал ли выделенный поток больше или меньше, чем все остальные потоки (положительный или отрицательный Δ), индекс справедливости теперь снизился ниже единицы. Другой простой случай для рассмотрения — это когда только k из n потоков получают равную пропускную способность, а остальные $n-k$ пользователей получают нулевую пропускную способность, в этом случае индекс справедливости падает до k/n .

Глава 6.2. Дисциплины очередей

Независимо от того, насколько простым или сложным является остальной механизм распределения ресурсов, каждый маршрутизатор должен реализовать некоторую дисциплину очередей, которая управляет тем, как пакеты буферизуются в ожидании передачи. Алгоритм очереди можно рассматривать как распределение и пропускной способности (какие пакеты передаются), и буферного пространства (какие пакеты отбрасываются). Он также напрямую влияет на задержку, которую испытывает пакет, определяя, сколько времени пакет ждет передачи. В следующей главе рассматриваются два распространенных алгоритма очередей — «первый пришел, первым обслужен» (FIFO) и «справедливая очередь» (FQ) — а также несколько их вариаций.

Глава 6.2.1. FIFO

Идея очереди FIFO, также называемой обслуживанием в порядке очереди (FCFS), проста: первый пакет, который приходит к маршрутизатору, является первым пакетом, который передается. Это иллюстрируется на рис. 6.5(а), который показывает FIFO с «ячейками» для хранения до восьми пакетов. Учитывая, что объем буферного пространства

в каждом маршрутизаторе конечен, если пакет прибывает и очередь (буферное пространство) заполнена, маршрутизатор отбрасывает этот пакет, как показано на рис. 6.5(b). Это делается без учета того, к какому потоку принадлежит пакет или насколько он важен. Это иногда называется «отбрасывание с конца» (tail drop), так как пакеты, которые прибывают в конец очереди FIFO, отбрасываются.

Заметьте, что «отбрасывание с конца» и FIFO — это две разные идеи. FIFO — это дисциплина планирования, которая определяет порядок, в котором пакеты передаются. «Отбрасывание с конца» — это политика отбрасывания, которая определяет, какие пакеты будут отброшены. Поскольку FIFO и «отбрасывание с конца» являются самыми простыми вариантами дисциплины планирования и политики отбрасывания соответственно, их иногда рассматривают как единое целое — «ванильная» реализация очереди. К сожалению, этот комплекс часто называют просто *очередью FIFO*, тогда как его следует точнее называть *FIFO с «отбрасыванием с конца»*. В следующей главе приводится пример другой политики отбрасывания, использующей более сложный алгоритм, чем просто «Есть ли свободный буфер?», чтобы решить, когда отбрасывать пакеты. Такая политика отбрасывания может использоваться с FIFO или с более сложными дисциплинами планирования.

FIFO с «отбрасыванием с конца» как самый простой из всех алгоритмов очередей является наиболее широко используемым в интернет-маршрутизаторах на момент написания этой книги. Этот простой подход к очередям возлагает всю ответственность за контроль перегрузок и распределение ресурсов на края сети. Таким образом, распространенная форма контроля перегрузок в Интернете в настоящее время предполагает отсутствие помощи со стороны маршрутизаторов: TCP берет на себя ответственность за обнаружение и реакцию на перегрузки. Мы увидим, как это работает, в следующей главе.

Простым вариантом базовой очереди FIFO является приоритетная очередь. Идея заключается в том, чтобы маркировать каждый пакет приоритетом; метка может передаваться, например, в заголовке IP, как мы обсудим в следующей главе. Затем маршрутизаторы реализуют несколько очередей FIFO, по одной для каждого класса приоритетов. Маршрутизатор всегда передает пакеты из очереди с наивысшим приоритетом, если эта очередь не пуста, прежде чем переходить к следующей очереди приоритетов. В пределах каждого приоритета пакеты по-прежнему управляются в порядке FIFO. Эта идея является небольшим отступлением от модели доставки с наилучшим усилием, но она не идет так далеко, чтобы давать гарантии для какого-либо конкретного класса приоритетов. Она просто позволяет пакетам с высоким приоритетом прорваться вперед в очереди.

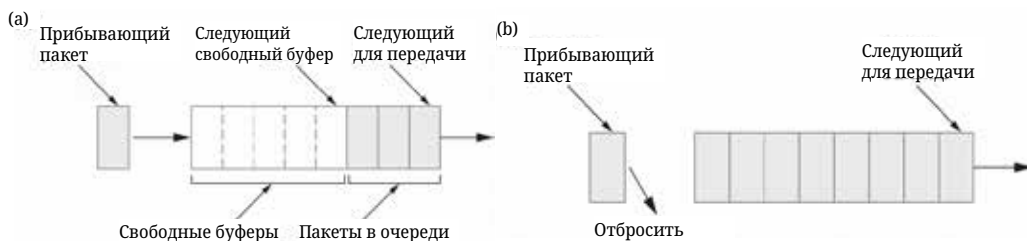


Рисунок 6.5. Очередь FIFO (a) и отбрасывание хвоста в очереди FIFO (b).

Проблема приоритетной очереди, конечно, в том, что очередь с высоким приоритетом может вытеснить все остальные очереди; то есть пока в очереди с высоким приоритетом есть хотя бы один пакет с высоким приоритетом, очереди с низким приоритетом не обслуживаются. Чтобы это было жизнеспособно, необходимо установить жесткие ограничения на количество трафика с высоким приоритетом, добавляемого в очередь. Должно быть сразу ясно, что нельзя позволить пользователям устанавливать собственные пакеты на высокий приоритет неконтролируемым образом; мы должны либо полностью предотвратить это, либо обеспечить некоторую форму «ответной реакции» на пользователей. Очевидным способом сделать это является использование экономики — сеть может взимать больше

за доставку пакетов с высоким приоритетом, чем за пакеты с низким приоритетом. Однако существуют значительные трудности в реализации такой схемы в децентрализованной среде, такой как Интернет.

Одна из ситуаций, в которой приоритетная очередь используется в Интернете, — это защита самых важных пакетов — обычно это обновления маршрутизации, которые необходимы для стабилизации таблиц маршрутизации после изменения топологии. Часто существует специальная очередь для таких пакетов, которые можно идентифицировать по точке кода дифференцированных услуг (Differentiated Services Code Point, ранее поле TOS) в заголовке IP. Это, по сути, простой случай идеи «Дифференцированных услуг».

Глава 6.2.2. Справедливая очередь (Fair queuing)

Основная проблема с очередью FIFO заключается в том, что она не различает разные источники трафика, или, говоря языком предыдущей главы, не разделяет пакеты в зависимости от потока, к которому они принадлежат. Эта проблема возникает на двух уровнях. На одном уровне неясно, сможет ли алгоритм управления перегрузкой, реализованный полностью на источнике, адекватно контролировать перегрузки с такой незначительной помощью от маршрутизаторов. Мы отложим обсуждение этого вопроса до следующей главы, когда будем говорить о контроле перегрузок в ТСП. На другом уровне, поскольку весь механизм управления перегрузками реализован на источниках, а очередь FIFO не предоставляет способа контролировать, насколько хорошо источники придерживаются этого механизма, возможно, что недобросовестный источник (поток) сможет захватить произвольно большую долю пропускной способности сети. Рассматривая Интернет, можно сказать, что определенное приложение может не использовать ТСП и, как следствие, обойти его механизм управления перегрузками. (Такие приложения, как интернет-телефония, делают это сегодня.) Такое приложение может «затопить» маршрутизаторы Интернета своими пакетами, вызывая тем самым отбрасывание пакетов других приложений.

Алгоритм честной очереди (Fair queuing, FQ) был разработан для решения этой проблемы. Идея FQ заключается в том, чтобы поддерживать отдельную очередь для каждого потока, в данный момент обрабатываемого маршрутизатором. Затем маршрутизатор обслуживает эти очереди в некотором роде кругового алгоритма, как показано на рис. 6.6. Когда поток отправляет пакеты слишком быстро, его очередь заполняется. Когда очередь достигает определенной длины, дополнительные пакеты, принадлежащие этой очереди, отбрасываются. Таким образом, данный источник не может произвольно увеличить свою долю пропускной способности сети за счет других потоков.



Рисунок 6.6. Обслуживание по круговой системе четырех потоков на маршрутизаторе.

Заметьте, FQ не предполагает, что маршрутизатор сообщает источникам трафика что-либо о состоянии маршрутизатора или каким-либо образом ограничивает скорость отправки пакетов конкретным источником. Иными словами, FQ по-прежнему предназначен для использования в сочетании с механизмом управления перегрузками от конца до конца. Он просто разделяет трафик, чтобы недобросовестные источники трафика не мешали тем, кто добросовестно реализует алгоритм от конца до конца. FQ также обеспечивает справедливость среди коллекции потоков, управляемых добросовестным алгоритмом управления перегрузками.

Несмотря на то, что основная идея проста, все же существует небольшое количество деталей, которые нужно правильно реализовать. Основная сложность заключается в том, что пакеты, обрабатываемые маршрутизатором, не обязательно одинаковой длины. Чтобы действительно распределять пропускную способность исходящего канала справедливо, необходимо учитывать длину пакетов. Например, если маршрутизатор управляет двумя потоками, один из которых имеет пакеты длиной 1000 байт, а другой — 500 байт (возможно, из-за фрагментации выше по потоку), то простое круговое обслуживание пакетов из каждой очереди потоков даст первому потоку две трети пропускной способности канала, а второму потоку только одну треть.

На самом деле нам нужен побитовый круговой алгоритм, при котором маршрутизатор передает бит из потока 1, затем бит из потока 2 и так далее. Очевидно, что невозможно чередовать биты из разных пакетов. Механизм FQ поэтому имитирует такое поведение, сначала определяя, когда конкретный пакет завершился бы передачу, если бы он передавался побитовым круговым методом, а затем использует это время завершения для упорядочивания пакетов для передачи.

Чтобы понять алгоритм, который использует побитовый круговой метод, рассмотрим поведение одного потока и представим часы, которые тикают каждый раз, когда один бит передается из всех активных потоков. (Поток активен, когда у него есть данные в очереди.) Для этого потока пусть P_i обозначает длину пакета i , S_i обозначает время, когда маршрутизатор начинает передавать пакет i , и F_i обозначает время, когда маршрутизатор завершает передачу пакета i . Если P_i выражен в терминах того, сколько тактов требуется для передачи пакета i (помня, что время идет на один такт вперед каждый раз, когда этот поток получает обслуживание на 1 бит), то легко увидеть, что $F_i = P_i + S_i$.

Когда мы начинаем передачу пакета i ? Ответ на этот вопрос зависит от того, прибыл ли пакет i до или после того, как маршрутизатор завершил передачу пакета $i-1$ из этого потока. Если это было до, то логично, что первый бит пакета i передается сразу после последнего бита пакета $i-1$. С другой стороны, возможно, что маршрутизатор завершил передачу пакета $i-1$ задолго до прибытия i , а это означает, что был период времени, в течение которого очередь этого потока была пуста, так что круговой механизм не мог передавать пакеты из этого потока. Если мы обозначим время прибытия пакета i в маршрутизатор как A_i , то $S_i = \max(F_{i-1}, A_i)$. Таким образом, мы можем вычислить

$$F_i = \max(F_{i-1}, A_i) + P_i$$

Теперь перейдем к ситуации, когда существует более одного потока, и обнаружим, что есть одна загвоздка в определении A_i . Нельзя просто прочитать время на настенных часах при поступлении пакета. Как отмечено выше, мы хотим, чтобы время двигалось на один тик каждый раз, когда все активные потоки получают один бит обслуживания в режиме побитного кругового алгоритма, поэтому нам нужны часы, которые движутся медленнее при большем количестве потоков. В частности, часы должны двигаться на один тик, когда передается n бит, если есть n активных потоков. Эти часы будут использоваться для вычисления A_i .

Теперь в каждом потоке мы вычисляем F_i для каждого поступающего пакета, используя приведенную выше формулу. Затем мы рассматриваем все F_i как метки времени, и следующий пакет на передачу — это всегда пакет с самой низкой меткой времени — пакет, который, основываясь на вышеизложенных рассуждениях, должен завершить передачу раньше всех остальных.

Обратите внимание: это означает, что пакет может прибыть на поток и, поскольку он короче, чем пакет из другого потока, который уже в очереди, он может быть вставлен в очередь перед этим более длинным пакетом. Однако это не означает, что вновь прибывший пакет может вытеснить пакет, который уже передается. Отсутствие вытеснения позволяет реализовать FQ, описанный выше, который не точно имитирует побитовый круговой алгоритм, который мы пытаемся иллюстрировать.

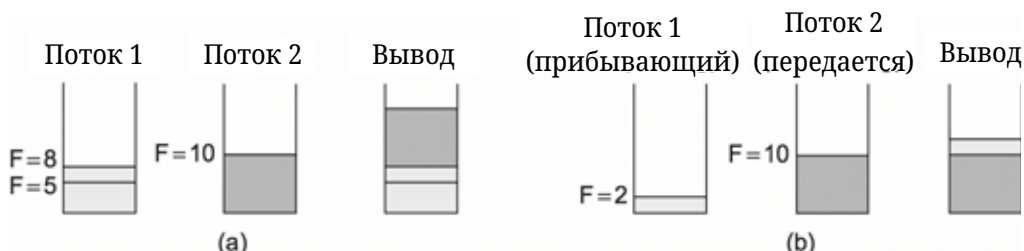


Рисунок 6.7. Пример справедливой очереди в действии: (а) пакеты с более ранним временем завершения отправляются первыми; (б) отправка пакета, который уже находится в процессе, завершается.

Чтобы лучше понять, как работает эта реализация справедливой очереди, рассмотрим пример, приведенный на рис. 6.7. В части (а) показаны очереди для двух потоков; алгоритм выбирает оба пакета из потока 1 для передачи перед пакетом в очереди потока 2 из-за их более раннего времени завершения. В (б) маршрутизатор уже начал передачу пакета из потока 2, когда пакет из потока 1 прибывает. Хотя прибывший пакет из потока 1 завершился бы раньше потока 2, если бы мы использовали идеальную побитовую справедливую очередь, реализация не вытесняет пакет из потока 2.

Есть две вещи, на которые стоит обратить внимание в справедливой очереди. Во-первых, канал никогда не остается неиспользованным, если в очереди есть хотя бы один пакет. Любая схема очереди с этой характеристикой называется *экономичной* (work conserving). Один из эффектов работоспособности заключается в том, что если я делю канал с множеством потоков, которые не отправляют данные, то я могу использовать всю пропускную способность канала для своего потока. Как только другие потоки начнут отправлять данные, они начнут использовать свою долю, и доступная пропускная способность для моего потока уменьшится.

Во-вторых, если канал полностью загружен и существует n потоков, отправляющих данные, я не могу использовать более $1/n$ -ной части пропускной способности канала. Если я попытаюсь отправить больше, мои пакеты будут получать все более крупные метки времени, из-за чего они будут дольше находиться в очереди в ожидании передачи. В конце концов очередь переполнится — хотя будет ли отброшен мой пакет или чей-то еще, это решение не зависит от того, что мы используем справедливую очередь. Это определяется политикой отбрасывания; FQ — это алгоритм планирования, который, как и FIFO, может быть совмещен с различными политиками отбрасывания.

Поскольку FQ работоспособен, любая пропускная способность, не используемая одним потоком, автоматически доступна другим потокам. Например, если через маршрутизатор проходит четыре потока и все они отправляют пакеты, то каждый получит одну четверть пропускной способности. Но если один из них будет бездействовать достаточно долго, чтобы все его пакеты ушли из очереди маршрутизатора, то доступная пропускная способность будет разделена между оставшимися тремя потоками, каждый из которых теперь получит одну треть пропускной способности. Таким образом, можно считать, что FQ предоставляет гарантированную минимальную долю пропускной способности для каждого потока, с возможностью получения большего, если другие потоки не используют свои доли.

Возможно реализовать вариант FQ, называемый *взвешенной справедливой очередью* (Weighted Fair Queuing, WFQ), который позволяет назначать вес каждому потоку (очереди). Этот вес логически указывает, сколько бит передается каждый раз, когда маршрутизатор обслуживает эту очередь, что эффективно контролирует процент пропускной способности канала, который получит этот поток. Простая FQ дает каждой очереди вес 1, и это означает, что логически передается только 1 бит из каждой очереди каждый раз. Это приводит к тому, что каждый поток получает $1/n$ -ную часть пропускной способности канала при наличии n потоков. Однако с WFQ одна очередь может иметь вес 2, вторая очередь может иметь вес 1, а третья очередь может иметь вес 3. При условии, что каждая очередь всегда содержит пакет, ожидающий передачи, первый поток будет получать одну треть доступной пропускной способности, второй будет получать одну шестую доступной пропускной способности, а третий будет получать половину доступной пропускной способности.

Хотя мы описали WFQ в терминах потоков, заметьте, что его можно реализовать для *классов* трафика, где классы определяются не так, как простые потоки, введенные в начале этой главы. Например, можно использовать некоторые биты в заголовке IP для идентификации классов и выделять очередь и вес для каждого класса. Это именно то, что предлагается в архитектуре Differentiated Services (дифференцированных услуг), описанной в последующей главе.

Заметьте, что маршрутизатор, выполняющий взвешенную справедливую очередь (WFQ), должен получать информацию о том, какие веса назначать каждой очереди, либо путем ручной настройки, либо с помощью какого-то сигнального механизма от источников. В последнем случае мы движемся к модели, основанной на резервировании. Простое назначение веса очереди представляет собой довольно слабую форму резервирования, поскольку эти веса лишь косвенно связаны с пропускной способностью, которую получает поток. (Пропускная способность, доступная потоку, также зависит, например, от того, сколько других потоков делят этот канал.) Мы увидим в следующем разделе, как WFQ может использоваться как компонент механизма распределения ресурсов на основе резервирования.

Основные выводы

Наконец, мы отмечаем, что вся эта дискуссия об управлении очередями иллюстрирует важный принцип системного проектирования, известный как *разделение политики и механизма*. Идея заключается в том, чтобы рассматривать каждый механизм как черный ящик, который предоставляет многофункциональную услугу, управляемую набором рычагов. Политика задает определенное положение этих рычагов, но не знает (и не заботится) о том, как этот черный ящик реализован. В данном случае механизмом является дисциплина очередей, а политика — это конкретная настройка, определяющая, какой поток получает какой уровень обслуживания (например, приоритет или вес). Мы обсудим некоторые политики, которые могут быть использованы с механизмом WFQ, в следующей главе.

Глава 6.3. Управление перегрузками TCP

В этой главе описывается основной пример управления перегрузками от конца до конца, который используется сегодня, реализованный в TCP. Основная стратегия TCP заключается в отправке пакетов в сеть без резервирования и последующей реакции на наблюдаемые события. TCP предполагает использование только FIFO-очередей в маршрутизаторах сети, но также работает со справедливой очередью.

Управление перегрузками TCP было введено в Интернет в конце 1980-х годов Ваном Джейкобсоном, примерно через восемь лет после того, как стек протоколов TCP/IP начал функционировать. Непосредственно перед этим Интернет страдал от коллапса перегрузок: хосты отправляли свои пакеты в Интернет так быстро, как позволяли рекламируемые окна, происходила перегрузка на каком-то маршрутизаторе (что приводило к потере

пакетов), хосты ожидали тайм-аута и повторно отправляли свои пакеты, что вызывало еще большую перегрузку.

В общем смысле идея управления перегрузками TCP заключается в том, чтобы каждый источник определял, сколько пропускной способности доступно в сети, чтобы знать, сколько пакетов можно безопасно держать в транзите. Как только у источника есть это количество пакетов в транзите, он использует прибытие ACK как сигнал о том, что один из его пакетов покинул сеть и поэтому можно безопасно вставить в сеть новый пакет, не увеличивая уровень перегрузки. Когда ACK используется для управления передачей пакетов, говорят, что TCP *саморегулируется*. Конечно, первоначальное определение доступной пропускной способности — непростая задача. К тому же, поскольку другие соединения приходят и уходят, доступная пропускная способность меняется со временем, а это означает, что каждый источник должен уметь регулировать количество пакетов в транзите. В этой главе описаны алгоритмы, используемые TCP для решения этих и других проблем.

Заметьте, что хотя мы описываем механизмы управления перегрузками TCP один за другим, создавая впечатление, что говорим о трех независимых механизмах, только в совокупности они представляют собой управление перегрузками TCP. Также, хотя мы начинаем с варианта управления перегрузками TCP, который чаще всего называется *стандартным* TCP, мы увидим, что существует довольно много вариантов управления перегрузками TCP, используемых сегодня, и исследователи продолжают изучать новые подходы к решению этой проблемы. Некоторые из этих новых подходов обсуждаются далее.

Глава 6.3.1. Аддитивное увеличение/мультипликативное уменьшение

TCP поддерживает новую переменную состояния для каждого соединения, называемую *CongestionWindow*, которая используется источником для ограничения количества данных, которые ему разрешено держать в транзите в данный момент времени. Окно перегрузки является аналогом окна управления потоком (*flow control's advertised window*). TCP модифицирован таким образом, что максимальное количество байтов неподтвержденных данных теперь является минимумом между окном перегрузки и рекламируемым окном. Таким образом, используя переменные, определенные в предыдущей главе, эффективное окно TCP пересматривается следующим образом:

$$\text{MaxWindow} = \min(\text{CongestionWindow}, \text{AdvertisedWindow})$$

$$\text{EffectiveWindow} = \text{MaxWindow} - (\text{LastByteSent} - \text{LastByteAcked})$$

То есть *MaxWindow* заменяет *AdvertisedWindow* в расчете *EffectiveWindow*. Таким образом, источнику TCP разрешено отправлять данные не быстрее, чем это может принять самый медленный компонент — сеть или конечный узел.

Конечно, проблема заключается в том, как TCP узнает подходящее значение для *CongestionWindow*. В отличие от *AdvertisedWindow*, который отправляется принимающей стороной соединения, нет никого, кто мог бы отправить подходящее значение *CongestionWindow* отправляющей стороне TCP. Ответ заключается в том, что источник TCP устанавливает *CongestionWindow* на основе уровня перегрузки, который он воспринимает в сети. Это включает уменьшение окна перегрузки при увеличении уровня перегрузки и увеличение окна перегрузки при его уменьшении. В совокупности этот механизм обычно называют *аддитивным увеличением/мультипликативным уменьшением* (AIMD); причина такого названия станет ясна позже.

Ключевой вопрос заключается в том, как источник определяет, что сеть перегружена и что следует уменьшить окно перегрузки? Ответ основан на наблюдении, что основной причиной недовозки пакетов и возникновения тайм-аутов является потеря пакета из-за перегрузки. Редко пакет теряется из-за ошибки при передаче. Поэтому TCP интерпретирует тайм-ауты как признак перегрузки и уменьшает скорость передачи. Если говорить конкретно, то каждый раз при возникновении тайм-аута источник устанавливает *CongestionWindow* в половину от его предыдущего значения. Это уменьшение

CongestionWindow вдвое при каждом тайм-ауте соответствует части «мультипликативное уменьшение» в AIMD.

Хотя CongestionWindow определяется в байтах, проще понять мультипликативное уменьшение, если думать в терминах целых пакетов. Например, предположим, что CongestionWindow в данный момент установлен на 16 пакетов. Если обнаружена потеря, CongestionWindow устанавливается на 8. (Обычно потеря обнаруживается при возникновении тайм-аута, но, как мы увидим далее, TCP имеет другой механизм для обнаружения потерянных пакетов.) Дополнительные потери уменьшают CongestionWindow до 4, затем до 2 и, наконец, до 1 пакета. CongestionWindow не может быть меньше размера одного пакета, или, в терминологии TCP, *максимального размера сегмента* (MSS).

Стратегия управления перегрузками, которая только уменьшает размер окна, очевидно, слишком консервативна. Нам также необходимо иметь возможность увеличить окно перегрузки, чтобы воспользоваться новой доступной пропускной способностью сети. Это и есть «аддитивное увеличение» части AIMD, и работает оно следующим образом. Каждый раз, когда источник успешно и полностью отправляет пакет в окно перегрузки – то есть каждый пакет, отправленный в течение последнего времени обхода (RTT), был подтвержден, — он добавляет эквивалент 1 пакета в окно перегрузки. Это линейное увеличение показано на рисунке 6.8. Обратите внимание, что на практике TCP не ждет, пока в окно перегрузки поступит целый пакет ACK, чтобы добавить к нему 1 пакет, а вместо этого увеличивает CongestionWindow на небольшую величину для каждого пришедшего ACK. В частности, окно перегрузки увеличивается следующим образом каждый раз, когда приходит ACK:

$$\text{Increment} = \text{MSS} \times (\text{MSS} / \text{CongestionWindow}) \quad \text{CongestionWindow} += \text{Increment}$$

То есть вместо увеличения CongestionWindow на целые MSS байты каждый RTT мы увеличиваем его на долю MSS при каждом получении ACK. Предполагая, что каждый ACK подтверждает получение MSS байтов, то эта доля равна:

Источник Пункт назначения

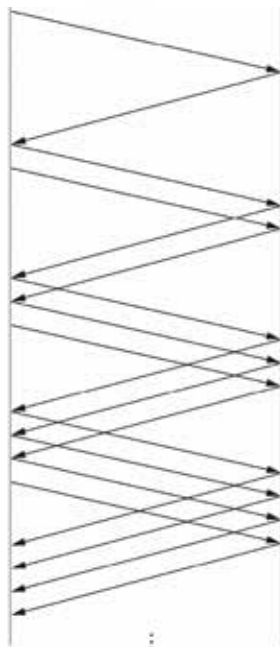


Рисунок 6.8. Пакеты в пути во время аддитивного увеличения, когда каждый RTT добавляется по одному пакету.

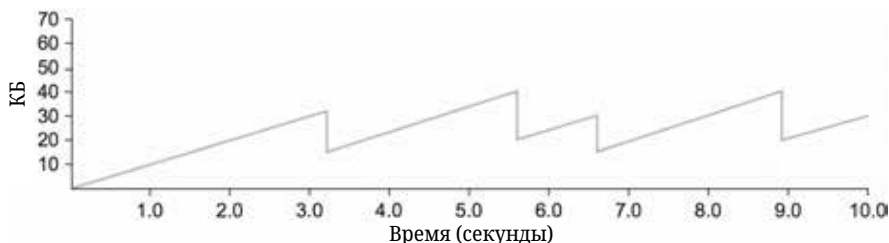


Рисунок 6.9. Типичный пилообразный шаблон TCP.

Этот шаблон постоянного увеличения и уменьшения окна перегрузки продолжается на протяжении всей продолжительности соединения. На самом деле, если построить график текущего значения окна перегрузки (CongestionWindow) в зависимости от времени, получится пилообразный узор, как показано на рис. 6.9. Важно понять, что при алгоритме AIMD (Additive Increase/Multiplicative Decrease) источник готов уменьшить свое окно перегрузки намного быстрее, чем готов его увеличить. Это контрастирует со стратегией аддитивного увеличения/аддитивного уменьшения, при которой окно увеличивалось бы на 1 пакет при получении ACK и уменьшалось бы на 1 при истечении тайм-аута. Было показано, что AIMD является необходимым условием для стабильности механизма контроля перегрузок. Одной из интуитивных причин агрессивного уменьшения окна и консервативного его увеличения является то, что последствия слишком большого окна гораздо хуже, чем слишком маленького. Например, когда окно слишком большое, пакеты, которые теряются, будут повторно передаваться, что усугубляет перегрузку; таким образом, важно быстро выйти из этого состояния.

Наконец, поскольку тайм-аут является признаком перегрузки, который запускает мультипликативное уменьшение, TCP нуждается в максимально точном механизме тайм-аута, который он может позволить себе. Мы уже рассматривали механизм тайм-аута TCP в предыдущем разделе, поэтому не будем повторять его здесь. Две основные вещи, которые нужно помнить об этом механизме, заключаются в следующем: (1) тайм-ауты устанавливаются как функция и среднего значения RTT, и стандартного отклонения этого среднего значения, и (2) из-за стоимости измерения каждой передачи с точными часами TCP измеряет время кругового маршрута (RTT) только один раз за RTT (вместо одного раза на каждый пакет) с использованием грубых (500 мс) часов.

Глава 6.3.2. Медленный старт

Механизм аддитивного увеличения, описанный выше, является правильным подходом, когда источник работает близко к доступной емкости сети, но он слишком долго разгоняет соединение при его запуске с нуля. Поэтому TCP предоставляет второй механизм, иронически называемый *медленным стартом*, который используется для быстрого увеличения окна перегрузки с холодного старта. Медленный старт эффективно увеличивает окно перегрузки экспоненциально, а не линейно.

Если говорить более конкретно, то источник начинает с установки CongestionWindow на один пакет. Когда приходит ACK для этого пакета, TCP добавляет 1 к CongestionWindow и затем отправляет два пакета. По получении двух соответствующих ACK TCP увеличивает CongestionWindow на 2 — по одному для каждого ACK — и отправляет четыре пакета. В конечном итоге TCP удваивает количество пакетов в транзите каждый RTT. На рис. 6.10 показан рост числа пакетов в транзите во время медленного старта. Сравните это с линейным ростом аддитивного увеличения, показанным на рис. 6.8.

Почему любой экспоненциальный механизм назвали бы «медленным», сначала кажется загадкой, но это можно объяснить, если учесть исторический контекст. Нужно сравнить медленный старт не с линейным механизмом предыдущей подглавы, а с ис-

ходным поведением ТСП. Рассмотрите, что происходит, когда устанавливается соединение и источник впервые начинает отправлять пакеты — то есть когда у него в настоящее время нет пакетов в транзите. Если источник отправляет столько пакетов, сколько позволяет рекламируемое окно — что ТСП и делал до разработки медленного старта, — то даже если в сети доступна довольно большая полоса пропускания, маршрутизаторы могут не справиться с этим всплеском пакетов. Все зависит от того, сколько буферного пространства доступно маршрутизаторам. Поэтому медленный старт был разработан для того, чтобы пакеты распределялись так, чтобы этого всплеска не происходило. Другими словами, хотя его экспоненциальный рост быстрее, чем линейный рост, медленный старт намного «медленнее», чем отправка сразу всего объема данных, указанного в рекламируемом окне.

На самом деле существуют две разные ситуации, в которых применяется медленный старт. Первая — это самое начало соединения, когда источник не имеет представления о том, сколько пакетов он сможет иметь в транзите в данный момент. (Имейте в виду, что сегодня ТСП работает на всем, от 1-Мбит/с до 40-Гбит/с каналов, поэтому источник не может знать емкость сети.) В этой ситуации медленный старт продолжает удваивать `CongestionWindow` каждое `RTT`, пока не произойдет потеря, в этот момент тайм-аут вызывает мультипликативное уменьшение, которое делит `CongestionWindow` на 2.



Рисунок 6.10. Пакеты в пути во время медленного старта.

Вторая ситуация, в которой используется медленный старт, немного более тонкая; она происходит, когда соединение становится неактивным, ожидая истечения тайм-аута. Вспомните, как работает алгоритм скользящего окна ТСП — когда пакет теряется, источник в конечном итоге достигает точки, где он отправил столько данных, сколько позволяет рекламируемое окно, и поэтому блокируется, ожидая ACK, который не придет. В конечном итоге происходит тайм-аут, но к этому времени нет пакетов в транзите, а это означает, что источник не получит ACK для «синхронизации» передачи новых пакетов. Вместо этого источник получит один кумулятивный ACK, который откроет все рекламируемое окно, но, как объяснено выше, источник затем использует медленный старт для перезапуска потока данных, а не отправляет сразу весь объем данных рекламируемого окна.

Хотя источник снова использует медленный старт, теперь он знает больше информации, чем в начале соединения. В частности, у источника есть текущее (и полезное) значение `CongestionWindow`; это значение `CongestionWindow`, которое существовало до по-

следней потери пакета, разделенное на 2 в результате потери. Мы можем рассматривать это как целевое окно перегрузки. Медленный старт используется для быстрого увеличения скорости отправки до этого значения, а затем применяется аддитивное увеличение. Обратите внимание, что у нас есть небольшая проблема с учетом, так как мы хотим запомнить целевое окно перегрузки, полученное в результате мультипликативного уменьшения, а также фактическое окно перегрузки, используемое медленным стартом. Для решения этой проблемы TCP вводит временную переменную для хранения целевого окна, обычно называемую `CongestionThreshold`, которая устанавливается равной значению `CongestionWindow`, полученному в результате мультипликативного уменьшения. Затем переменная `CongestionWindow` сбрасывается до одного пакета и увеличивается на один пакет за каждый полученный ACK, пока не достигнет `CongestionThreshold`, после чего она увеличивается на один пакет за RTT.

Другими словами, TCP увеличивает окно перегрузки, как определено в следующем фрагменте кода:

```
{
    u_int cw = state->CongestionWindow;
    u_int incr = state->maxseg;
    if (cw > state->CongestionThreshold) {
        incr = incr * incr / cw;
    }
    state->CongestionWindow = MIN(cw + incr, TCP_MAXWIN);
}
```

где `state` представляет состояние определенного TCP соединения и определяет верхнюю границу роста окна перегрузки.

На рис. 6.11 показано, как `CongestionWindow` TCP увеличивается и уменьшается с течением времени, иллюстрируя взаимодействие медленного старта и аддитивного увеличения/мультипликативного уменьшения. Этот график был взят из реального TCP соединения и показывает текущее значение `CongestionWindow` с течением времени.

Есть несколько вещей, на которые стоит обратить внимание на этом графике. Первое — это быстрое увеличение окна перегрузки в начале соединения. Это соответствует начальному этапу медленного старта. Этап медленного старта продолжается до тех пор, пока несколько пакетов не теряются примерно через 0,4 секунды с начала соединения, в это время `CongestionWindow` выравнивается на уровне около 34 КБ. (Почему так много пакетов теряется во время медленного старта, обсуждается ниже.) Причина, по которой окно перегрузки выравнивается, заключается в том, что из-за потери нескольких пакетов не поступают ACK. Фактически в это время не отправляются новые пакеты, что обозначено отсутствием хеш-меток в верхней части графика. В конечном итоге происходит тайм-аут примерно через 2 секунды, в это время окно перегрузки делится на 2 (т. е. сокращается с примерно 34 КБ до около 17 КБ) и `CongestionThreshold` устанавливается на это значение. Медленный старт затем вызывает сброс `CongestionWindow` до одного пакета и его последующее увеличение оттуда.

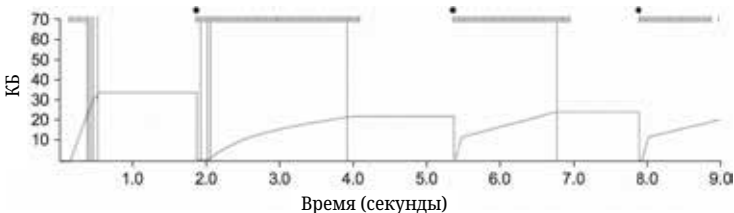


Рисунок 6.11. Поведение системы управления перегрузками TCP. Кривая линия = значение `CongestionWindow` с течением времени; сплошные пули в верхней части графика = тайм-ауты; хеш-метки в верхней части графика = время передачи каждого пакета; вертикальные полосы = время, когда пакет, который в итоге был повторно передан, был передан первым.

На трассировке недостаточно деталей, чтобы точно увидеть, что происходит, когда несколько пакетов теряются сразу после 2 секунд, поэтому мы переходим к линейному увеличению окна перегрузки, которое происходит между 2 и 4 секундами. Это соответствует аддитивному увеличению. Примерно через 4 секунды `CongestionWindow` снова выравнивается из-за потери пакета. Теперь, примерно через 5,5 секунды:

1. Происходит тайм-аут, в результате чего окно перегрузки делится на 2, уменьшаясь с примерно 22 КБ до 11 КБ, и `CongestionThreshold` устанавливается на эту величину.
2. `CongestionWindow` сбрасывается до одного пакета, так как отправитель входит в медленный старт.
3. Медленный старт вызывает экспоненциальный рост `CongestionWindow`, пока оно не достигнет `CongestionThreshold`.
4. Затем `CongestionWindow` падает линейно.

Та же схема повторяется примерно через 8 секунд, когда происходит еще один тайм-аут.

Теперь вернемся к вопросу, почему так много пакетов теряется в начальный период медленного старта. В этот момент ТСП пытается узнать, сколько пропускной способности доступно в сети. Это очень сложная задача. Если источник не будет агрессивен на этом этапе (например, если он будет увеличивать окно перегрузки только линейно), то ему потребуется много времени, чтобы выяснить, сколько пропускной способности доступно. Это может значительно повлиять на достигнутую пропускную способность для этого соединения. С другой стороны, если источник будет агрессивен на этом этапе, как ТСП во время экспоненциального роста, то он рискует потерять половину пакетов окна, отправленных сетью.

Чтобы понять, что может произойти во время экспоненциального роста, рассмотрим ситуацию, когда источнику удалось успешно отправить 16 пакетов через сеть, что привело к удвоению окна перегрузки до 32. Предположим, однако, что у сети есть ровно столько пропускной способности, чтобы поддержать 16 пакетов от этого источника. Вероятный результат заключается в том, что 16 из 32 пакетов, отправленных под новым окном перегрузки, будут потеряны сетью; на самом деле это наихудший исход, так как некоторые пакеты будут буферизованы в каком-то маршрутизаторе. Эта проблема станет все более серьезной по мере увеличения произведения задержки на пропускную способность сетей. Например, произведение задержки на пропускную способность 500 КБ означает, что каждое соединение имеет потенциал потерять до 500 КБ данных в начале каждого соединения. Конечно, это предполагает, что и источник, и получатель реализуют расширение «больших окон».

Также исследованы альтернативы медленному старту, при которых источник пытается оценить доступную пропускную способность более сложными методами. Один из примеров называется *быстрый старт* (quick-start). Основная идея заключается в том, что отправитель ТСП может запросить начальную скорость отправки, превышающую ту, которую позволяет медленный старт, указав запрашиваемую скорость в своем SYN-пакете как IP-опцию. Маршрутизаторы по пути могут рассмотреть эту опцию, оценить текущий уровень перегрузки на исходящем канале для этого потока и решить, приемлема ли эта скорость, приемлема ли более низкая скорость или следует использовать стандартный медленный старт. К тому времени, когда SYN достигает получателя, он будет содержать либо скорость, приемлемую для всех маршрутизаторов на пути, либо указание на то, что один или несколько маршрутизаторов на пути не могут поддержать запрос быстрого старта. В первом случае отправитель ТСП использует эту скорость для начала передачи; во втором случае он возвращается к стандартному медленному старту. Если ТСП разрешено начать отправки на более высокой скорости, сеанс может быстрее достичь точки заполнения канала, а не тратить на это много времени кругового маршрута (RTT).

Очевидно, что одной из проблем для такого рода улучшений ТСП является то, что они требуют значительно большего сотрудничества со стороны маршрутизаторов, чем стандартный ТСП. Если хотя бы один маршрутизатор на пути не поддерживает быстрый старт, то система возвращается к стандартному медленному старту. Таким образом,

может пройти много времени, прежде чем эти типы улучшений смогут быть внедрены в Интернет; на данный момент они, более вероятно, будут использоваться в контролируемых сетевых средах (например, исследовательских сетях).

Глава 6.3.3. Быстрая повторная передача (ретрансляция) и быстрое восстановление

Механизмы, описанные до сих пор, были частью первоначального предложения по добавлению контроля перегрузки в ТСП. Однако вскоре было обнаружено, что грубая реализация тайм-аутов в ТСП приводит к длительным периодам времени, в течение которых соединение становилось неактивным, ожидая истечения таймера. Из-за этого в ТСП был добавлен новый механизм, называемый *быстрой повторной передачей* (fast retransmit). Быстрая повторная передача — это эвристика, которая иногда запускает повторную передачу потерянного пакета раньше, чем это делает обычный механизм тайм-аута. Механизм быстрой повторной передачи не заменяет обычные тайм-ауты; он просто улучшает эту функцию.

Идея быстрой повторной передачи проста. Каждый раз, когда пакет данных прибывает на принимающую сторону, получатель отвечает подтверждением, даже если этот порядковый номер уже был подтвержден. Таким образом, когда пакет приходит не по порядку — когда ТСП не может подтвердить данные, содержащиеся в пакете, потому что более ранние данные еще не прибыли — ТСП повторно отправляет то же самое подтверждение, которое оно отправило в прошлый раз. Это вторичное отправление того же самого подтверждения называется *дублированным АСК* (duplicate ACK). Когда отправляющая сторона видит дублированный АСК, она знает, что другая сторона должна была получить пакет не по порядку, что предполагает, что более ранний пакет мог быть потерян. Поскольку также возможно, что более ранний пакет был просто задержан, а не потерян, отправитель ждет, пока не увидит некоторое количество дублированных АСК, а затем повторно отправляет пропавший пакет. На практике ТСП ждет, пока не увидит три дублированных АСК, прежде чем повторно отправить пакет.

Рис. 6.12 иллюстрирует, как дублированные АСК приводят к быстрой повторной передаче. В этом примере получатель получает пакеты 1 и 2, но пакет 3 теряется в сети. Таким образом, получатель отправит дублированный АСК для пакета 2, когда придет пакет 4, снова — когда придет пакет 5 и так далее. (Чтобы упростить этот пример, мы рассматриваем пакеты 1, 2, 3 и так далее, вместо того чтобы беспокоиться о порядковых номерах для каждого байта.) Когда отправитель видит третий дублированный АСК для пакета 2 — тот, который был отправлен, потому что получатель получил пакет 6, — он повторно отправляет пакет 3. Обратите внимание, что когда повторно отправленная копия пакета 3 прибывает к получателю, получатель затем отправляет кумулятивный АСК для всех пакетов до и включая пакет 6 обратно источнику.

Рис. 6.13 иллюстрирует поведение версии ТСП с механизмом быстрой повторной передачи. Интересно сравнить эту трассировку с той, которая приведена на рис. 6.11, где быстрая повторная передача не была реализована — долгие периоды, в течение которых окно перегрузки остается на одном уровне и пакеты не отправляются, были устранены. В целом эта техника позволяет устранить около половины грубых тайм-аутов на типичном ТСП-соединении, что приводит к примерно 20% улучшению пропускной способности по сравнению с тем, что могло быть достигнуто иначе. Обратите внимание, что стратегия быстрой повторной передачи не устраняет все грубые тайм-ауты. Это потому, что для небольшого размера окна не будет достаточно пакетов в транзите, чтобы доставить достаточное количество дублированных АСК. При достаточно большом количестве потерянных пакетов (например, как это происходит во время начального этапа медленного старта) алгоритм «скользящего окна» в конечном итоге блокирует отправителя до тех пор, пока не произойдет тайм-аут. На практике механизм быстрой повторной передачи ТСП может обнаружить до трех потерянных пакетов на окно.

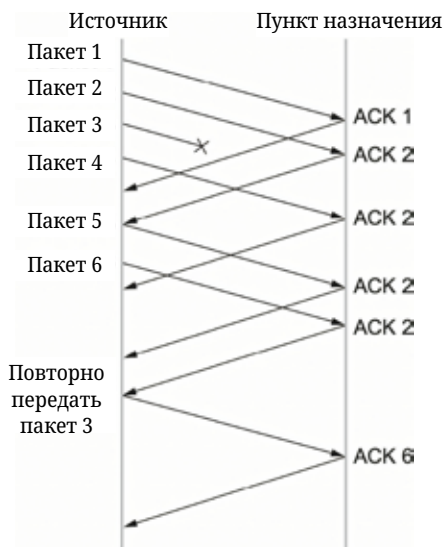


Рисунок 6.12. Быстрая ретрансляция на основе дублирования ACK.

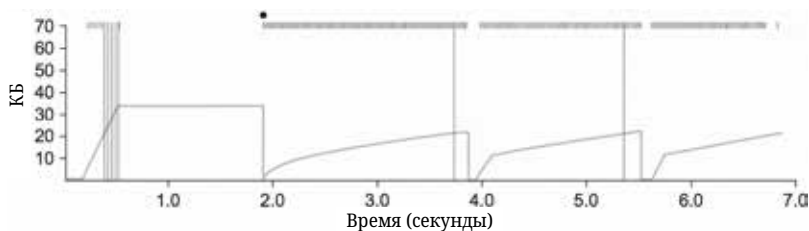


Рисунок 6.13. Трассировка TCP с быстрой ретрансляцией. Кривая линия = окно перегрузки; сплошная пуля = тайм-аут; хеш-метки = время передачи каждого пакета; вертикальные полосы = время, когда пакет, который в итоге был повторно передан, был передан первым.

Когда механизм быстрой повторной передачи сигнализирует о перегрузке, вместо того чтобы сбрасывать окно перегрузки до одного пакета и запускать медленный старт, можно использовать подтверждения (ACK), которые все еще находятся в пути, чтобы управлять отправкой пакетов. Этот механизм, который называется **быстрым восстановлением**, эффективно устраняет фазу медленного старта, которая происходит между моментом, когда быстрая повторная передача обнаруживает потерю пакета, и началом аддитивного увеличения. Например, быстрое восстановление избегает периода медленного старта между 3.8 и 4 секундами на рис. 6.13 и просто уменьшает окно перегрузки вдвое (с 22 КБ до 11 КБ) и возобновляет аддитивное увеличение. Другими словами, медленный старт используется только в начале соединения и всякий раз, когда происходит грубый тайм-аут. На протяжении остального времени окно перегрузки следует чистому паттерну аддитивного увеличения/мультипликативного уменьшения.

Глава 6.3.4. TCP CUBIC

Вариант стандартного алгоритма TCP, описанного выше, называется CUBIC и является алгоритмом управления перегрузкой по умолчанию, распространяемым с Linux. Основная цель CUBIC заключается в поддержке сетей с большим произведением задержки на пропускную способность, которые иногда называют сетями с **длинными жирными ка-**

налами (long-fat networks). Такие сети страдают от того, что оригинальный алгоритм TCP требует слишком много круговых путей, чтобы достичь доступной пропускной способности конечного пути. CUBIC делает это, будучи более агрессивным в том, как он увеличивает размер окна, но, конечно, задача заключается в том, чтобы быть более агрессивным, не оказывая при этом отрицательного воздействия на другие потоки.

Одним из важных аспектов подхода CUBIC является регулировка окна перегрузки через регулярные промежутки времени, основываясь на количестве времени, прошедшем с момента последнего события перегрузки (например, прибытия дублированного АСК), а не только когда прибывают АСК (что является функцией RTT). Это позволяет CUBIC вести себя справедливо, когда он конкурирует с потоками с коротким RTT, у которых АСК прибывают чаще.

Вторым важным аспектом CUBIC является использование кубической функции для регулировки окна перегрузки. Основную идею легче всего понять, взглянув на общую форму кубической функции, которая имеет три фазы: замедление роста, плоское плато, увеличение роста. Пример показан на рис. 6.14, который аннотирован дополнительной информацией: максимальный размер окна перегрузки, достигнутый непосредственно перед последним событием перегрузки, обозначается как W_{max} . Идея заключается в том, чтобы начинать быстро, но замедлять темп роста по мере приближения к W_{max} , быть осторожным и иметь почти нулевой рост близко к W_{max} , а затем увеличивать темп роста по мере удаления от W_{max} . Последняя фаза по сути является поиском новой достижимой W_{max} .

Специфически CUBIC вычисляет окно перегрузки как функцию времени (t) с момента последнего события перегрузки:

$$CWND(t) = C \times (t - K)^3 + W_{max}$$

где

$$K = \sqrt[3]{W_{max} \times \frac{(1-\beta)}{C}}$$

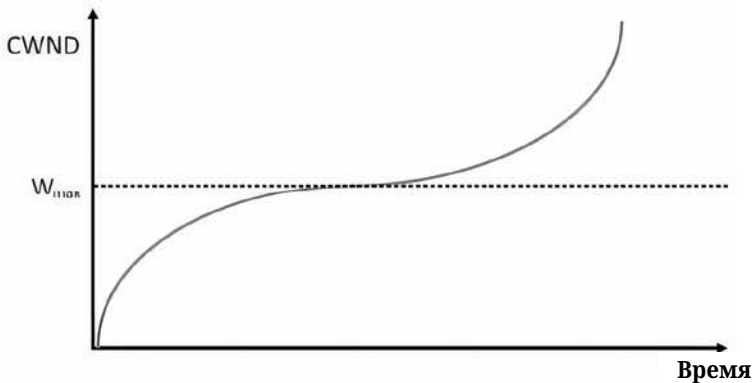


Рисунок 6.14. Общая кубическая функция, иллюстрирующая изменение окна перегрузки в зависимости от времени.

C является масштабирующим коэффициентом, а β — множителем мультипликативного уменьшения. CUBIC устанавливает последний на уровне 0.7 вместо 0.5, который используется в стандартном TCP. Рис. 6.14 иллюстрирует, почему CUBIC часто описывают как переходящий от вогнутой функции к выпуклой (в то время как аддитивная функция стандартного TCP является только выпуклой).

Глава 6.4. Расширенное управление перегрузкой

В этой главе более глубоко рассматривается управление перегрузкой. При этом важно понимать, что стандартная стратегия ТСП заключается в управлении перегрузкой после того, как она произошла, а не в попытке избежать перегрузки изначально. Фактически ТСП многократно увеличивает нагрузку, которую он накладывает на сеть, чтобы найти точку, в которой происходит перегрузка, а затем отступает от этой точки. Другими словами, ТСП нужно создавать потери, чтобы найти доступную пропускную способность соединения. Привлекательная альтернатива заключается в предсказании, когда перегрузка вот-вот произойдет, и затем в снижении скорости отправки данных хостами непосредственно перед началом сброса пакетов. Мы называем такую стратегию *избеганием перегрузки*, чтобы отличить ее от *управления перегрузкой*, но, вероятно, наиболее правильно считать «избегание» разновидностью «управления».

Мы описываем два разных подхода к избеганию перегрузки. Первый добавляет небольшое количество дополнительных функций в маршрутизатор для помощи конечному узлу в предвосхищении перегрузки. Этот подход часто называют *активным управлением очередью* (AQM). Второй подход пытается избежать перегрузки исключительно с конечных хостов. Этот подход реализован в ТСП, делая его вариантом механизмов управления перегрузкой, описанных в предыдущей главе.

Глава 6.4.1. Активное управление очередью (DECbit, RED, ECN)

Первый подход требует изменений в маршрутизаторах, что никогда не было предпочтительным способом внедрения новых функций в Интернете, но, тем не менее, является постоянным источником раздражения на протяжении последних 20 лет. Проблема в том, что, хотя разработчики в общем согласны с тем, что маршрутизаторы находятся в идеальной позиции для обнаружения начала перегрузки — т.е. их очереди начинают заполняться, — не было достигнуто единого мнения о том, какой алгоритм является наилучшим. Далее описаны два классических механизма, и в конце кратко обсуждается, на каком этапе сейчас находятся эти разработки.

DECbit

Первый механизм был разработан для использования в Digital Network Architecture (DNA), бесконтактной сети с ориентированным на соединение транспортным протоколом. Этот механизм можно было бы также применить к ТСП и IP. Как отмечено выше, идея здесь заключается в более равномерном распределении ответственности за управление перегрузками между маршрутизаторами и конечными узлами. Каждый маршрутизатор контролирует нагрузку, которую он испытывает, и явно уведомляет конечные узлы, когда перегрузка вот-вот произойдет. Это уведомление реализуется установкой бинарного бита перегрузки в пакеты, проходящие через маршрутизатор, отсюда и название *DECbit*. Узел назначения затем копирует этот бит перегрузки в ACK, который он отправляет обратно источнику. Наконец, источник корректирует свою скорость отправки, чтобы избежать перегрузки. Далее алгоритм обсуждается более подробно, начиная с того, что происходит в маршрутизаторе.

Один бит перегрузки добавляется в заголовок пакета. Маршрутизатор устанавливает этот бит в пакете, если его средняя длина очереди больше или равна 1 в момент прибытия пакета. Эта средняя длина очереди измеряется за временной интервал, охватывающий последний цикл занятости плюс простоя, плюс текущий цикл занятости. (Маршрутизатор *занят*, когда он передает, и *простаивает*, когда нет.) Рис. 6.15 показывает длину очереди в маршрутизаторе как функцию времени. По существу маршрутизатор вычисляет площадь под кривой и делит это значение на временной интервал, чтобы вычислить среднюю длину очереди. Использование длины очереди, равной 1, в качестве триггера для установки бита перегрузки является компромиссом между значительной очередью

(и, следовательно, более высокой пропускной способностью) и увеличенным временем простоя (и, следовательно, меньшей задержкой). Другими словами, длина очереди, равная 1, кажется оптимальной для функции мощности.

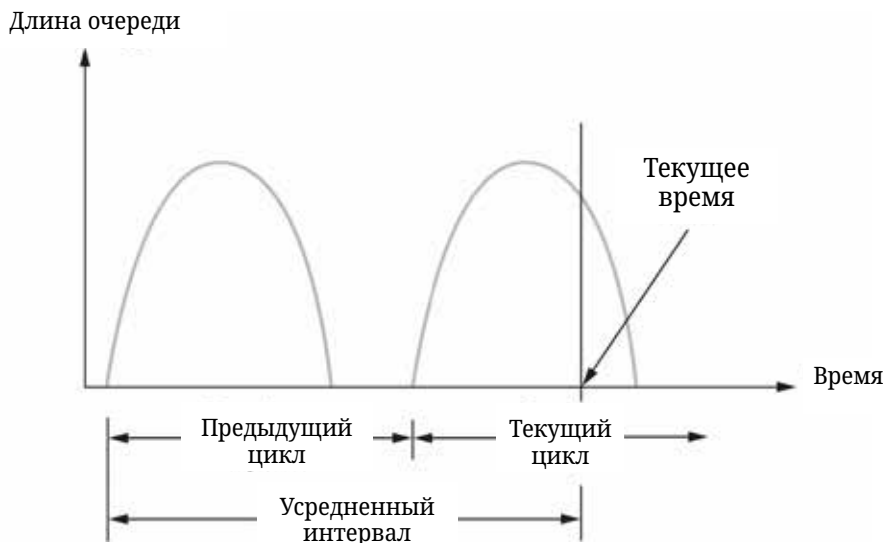


Рисунок 6.15. Вычисление средней длины очереди на маршрутизаторе.

Теперь, обращая внимание на часть механизма, относящуюся к узлу, источник записывает, сколько из его пакетов привели к установке маршрутизатором бита перегрузки. В частности, источник поддерживает окно перегрузки, так же как в TCP, и наблюдает, какая доля пакетов из последнего окна привела к установке бита. Если менее 50% пакетов имели установленный бит, то источник увеличивает окно перегрузки на один пакет. Если 50% или более пакетов из последнего окна имели установленный бит перегрузки, то источник уменьшает окно перегрузки до 0.875 от предыдущего значения. Значение 50% было выбрано в качестве порога на основе анализа, который показал, что оно соответствует пику функции мощности. Правило «увеличение на 1, уменьшение на 0.875» было выбрано потому, что аддитивное увеличение/мультипликативное уменьшение делает механизм стабильным.

Случайное раннее обнаружение (RED)

Второй механизм, называемый случайным *ранним обнаружением* (random early detection, RED), аналогичен схеме DECbit в том, что каждый маршрутизатор запрограммирован на мониторинг длины своей очереди, и когда он обнаруживает, что перегрузка неизбежна, уведомляет источник, чтобы тот скорректировал свое окно перегрузки. RED, изобретенный Салли Флойд и Ваном Джейкобсоном в начале 1990-х годов, отличается от схемы DECbit двумя основными свойствами.

Первое отличие заключается в том, что вместо явной отправки сообщения о перегрузке источнику RED чаще всего реализуется так, что он *косвенно* уведомляет источник о перегрузке, отбрасывая один из его пакетов. Таким образом, источник фактически уведомляется о перегрузке последующим тайм-аутом или дублирующим ACK. Как вы уже могли догадаться, RED предназначен для использования в сочетании с TCP, который в настоящее время обнаруживает перегрузку с помощью тайм-аутов (или других средств обнаружения потери пакетов, таких как дублирующие ACK). Как следует из части аббревиатуры RED «раннее», шлюз сбрасывает пакет раньше, чем это было бы необходимо, чтобы уведомить источник о том, что он должен уменьшить свое окно перегрузки раньше, чем это обычно происходит.

Другими словами, маршрутизатор отбрасывает несколько пакетов до того, как полностью исчерпает свое буферное пространство, чтобы заставить источник замедлиться, надеясь, что это позволит избежать необходимости сбрасывать много пакетов позже.

Второе отличие между RED и DEChit заключается в деталях того, как RED решает, когда сбросить пакет и какой пакет сбросить. Чтобы понять основную идею, рассмотрим простую очередь FIFO. Вместо того чтобы ждать, пока очередь полностью заполнится, а затем вынужденно сбрасывать каждый прибывающий пакет (политика сброса хвоста из предыдущей главы), мы могли бы решить сбрасывать каждый прибывающий пакет с некоторой вероятностью сброса всякий раз, когда длина очереди превышает некоторый уровень сброса. Эта идея называется *ранним случайным сбросом*. Алгоритм RED определяет детали того, как контролировать длину очереди и когда сбросить пакет.

В следующих подглавах мы описываем алгоритм RED в том виде, в котором он был первоначально предложен Флойдом и Джейкобсоном. Отметим, что с тех пор было предложено несколько модификаций как самими изобретателями, так и другими исследователями. Тем не менее основные идеи совпадают с теми, что представлены ниже, и большинство современных реализаций близки к алгоритму, который приведен далее.

Сначала RED вычисляет среднюю длину очереди, используя взвешенное скользящее среднее, аналогичное тому, что использовалось в первоначальном вычислении тайм-аута TCP. То есть $AvgLen$ вычисляется как:

$$AvgLen = (1 - Weight) \times AvgLen + Weight \times SampleLen$$

где $0 < Weight < 1$, а $SampleLen$ — это длина очереди в момент проведения выборочного измерения. В большинстве программных реализаций длина очереди измеряется каждый раз, когда новый пакет прибывает на шлюз. В аппаратных средствах она может рассчитываться с фиксированным интервалом выборки.

Причина использования средней длины очереди вместо мгновенной заключается в том, что она более точно отражает понятие перегрузки. Из-за взрывного характера интернет-трафика очереди могут быстро заполняться, а затем снова опустошаться. Если очередь большую часть времени пуста, то, вероятно, не следует делать вывод о том, что маршрутизатор перегружен, и сообщать узлам, чтобы они замедлялись. Таким образом, вычисление взвешенного скользящего среднего пытается обнаружить длительную перегрузку, как показано в правой части рис. 6.16, фильтруя краткосрочные изменения длины очереди. Вы можете рассматривать скользящее среднее как низкочастотный фильтр, где $Weight$ определяет временную постоянную фильтра. Вопрос о том, как выбрать эту временную постоянную, обсуждается далее.

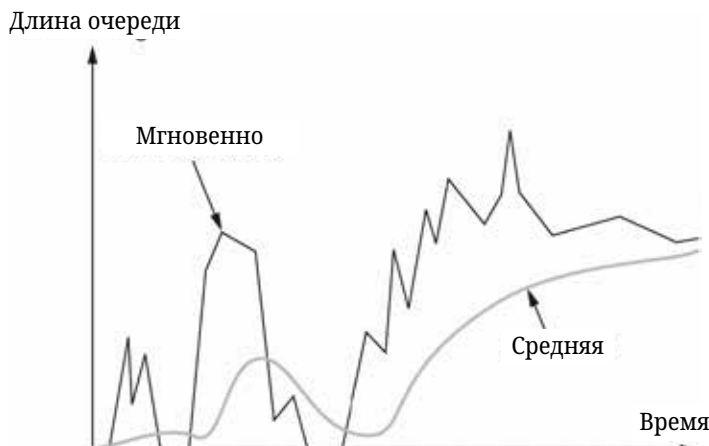


Рисунок 6.16. Средневзвешенная длина очереди.

Во-вторых, RED имеет два порога длины очереди, которые вызывают определенные действия: `MinThreshold` и `MaxThreshold`. Когда пакет прибывает на шлюз, RED сравнивает текущую `AvgLen` с этими двумя порогами в соответствии со следующими правилами:

```
if (AvgLen <= MinThreshold) {
    поместить пакет в очередь;
}
if (MinThreshold < AvgLen && AvgLen < MaxThreshold) {
    вычислить вероятность P;
    отбросить прибывающий пакет с вероятностью P;
}
if (MaxThreshold <= AvgLen) {
    отбросить прибывающий пакет;
}
```

Если средняя длина очереди меньше нижнего порога, никаких действий не предпринимается, а если средняя длина очереди больше верхнего порога, то пакет всегда отбрасывается. Если средняя длина очереди находится между двумя порогами, то вновь прибывающий пакет отбрасывается с некоторой вероятностью P . Эта ситуация изображена на рис. 6.17. Приблизительное соотношение между P и `AvgLen` показано на рис. 6.18. Обратите внимание, что вероятность сброса увеличивается медленно, когда `AvgLen` находится между двумя порогами, достигая `MaxP` на верхнем пороге, после чего она скачком увеличивается до единицы. Смысл этого в том, что если `AvgLen` достигает верхнего порога, то мягкий подход (сброс нескольких пакетов) не работает, и требуются радикальные меры: сброс всех прибывающих пакетов. Некоторые исследователи предположили, что более плавный переход от случайного сброса к полному сбросу, а не дискретный подход, показанный здесь, может быть уместен.

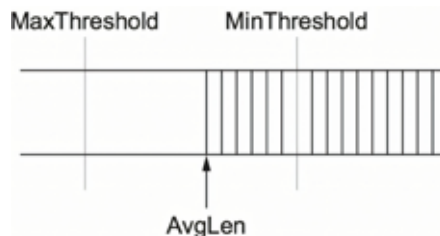


Рисунок 6.17. Пороги RED для очереди FIFO.

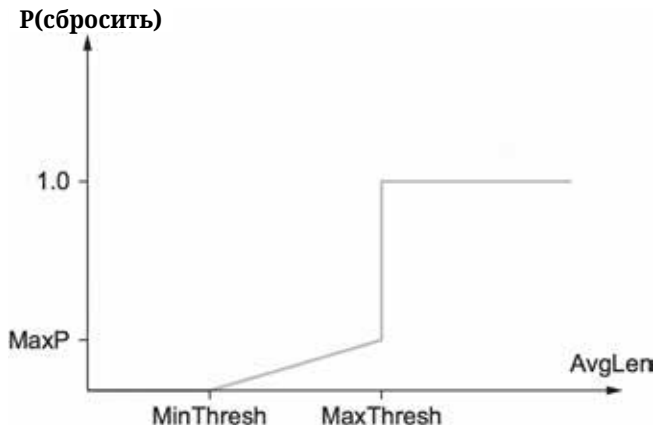


Рисунок 6.18. Функция вероятности падения для RED.

Хотя рис. 6.18 показывает вероятность сброса в зависимости только от AvgLen, на самом деле ситуация немного сложнее. В действительности P является функцией как AvgLen, так и того, сколько времени прошло с момента последнего сброса пакета. Если говорить конкретно, то она вычисляется следующим образом:

$$\begin{aligned} \text{TempP} &= \text{MaxP} \times (\text{AvgLen} - \text{MinThreshold}) / (\text{MaxThreshold} - \text{MinThreshold}) \\ P &= \text{TempP} / (1 - \text{count} \times \text{TempP}) \end{aligned}$$

TempP — это переменная, которая отображается на оси y на рис. 6.18, count отслеживает, сколько вновь прибывших пакетов было помещено в очередь (не сброшено), и AvgLen находился между двумя порогами. P увеличивается медленно по мере увеличения count, что делает сброс более вероятным по мере увеличения времени с последнего сброса. Этот дополнительный шаг в вычислении P был введен изобретателями RED, когда они обнаружили, что без него сбросы пакетов неравномерно распределялись во времени, а вместо этого происходили кластерами. Поскольку прибытие пакетов из определенного соединения, скорее всего, происходит взрывами, это кластерное распределение сбросов может вызвать множественные сбросы в одном соединении. Это нежелательно, поскольку одного сброса на время оборота достаточно, чтобы заставить соединение уменьшить размер своего окна, в то время как множественные сбросы могут вернуть его в медленный старт.

В качестве примера предположим, что мы установили MaxP на 0.02, а count инициализирован до нуля. Если средняя длина очереди была на полпути между двумя порогами, то TempP и начальное значение P будут половиной от MaxP, или 0.01. Прибывающий пакет, конечно, имеет шанс 99 из 100 попасть в очередь на этом этапе. С каждым последующим пакетом, который не сброшен, P медленно увеличивается, и к моменту, когда 50 пакетов придут без сброса, P удвоится до 0.02. В маловероятном случае, если 99 пакетов придут без потерь, P достигнет 1, гарантируя, что следующий пакет будет сброшен. Важным аспектом этой части алгоритма является то, что он обеспечивает приблизительно равномерное распределение сбросов во времени.

Идея заключается в том, что если RED сбрасывает небольшой процент пакетов, когда AvgLen превышает MinThreshold, это приведет к тому, что несколько TCP соединений уменьшат размер своих окон, что, в свою очередь, снизит скорость поступления пакетов к маршрутизатору. Если все пойдет хорошо, AvgLen уменьшится, и перегрузка будет предотвращена. Длина очереди может оставаться короткой, в то время как пропускная способность остается высокой, так как мало пакетов сбрасывается.

Важно отметить, что поскольку RED работает на основе средней длины очереди за определенный период времени, мгновенная длина очереди может быть намного больше, чем AvgLen. В этом случае, если пакет прибывает и нет места для его размещения, он будет сброшен. Когда это происходит, RED работает в режиме сброса хвоста. Одной из целей RED является предотвращение поведения сброса хвоста, если это возможно.

Случайный характер RED придает алгоритму интересное свойство. Поскольку RED сбрасывает пакеты случайным образом, вероятность того, что RED решит сбросить пакет(ы) определенного потока, примерно пропорциональна доле пропускной способности, которую этот поток в данный момент получает на этом маршрутизаторе. Это связано с тем, что поток, который отправляет относительно большое количество пакетов, предоставляет больше кандидатов для случайного сброса. Таким образом, в RED заложено некоторое «чувство» справедливого распределения ресурсов, хотя это далеко не точно. Хотя RED, можно сказать, справедлив, поскольку он «наказывает» потоки с высокой пропускной способностью больше, чем потоки с низкой пропускной способностью, он увеличивает вероятность перезапуска TCP, что вдвойне болезненно для потоков с высокой пропускной способностью.

Основные выводы

Следует отметить, что значительное количество анализа было проведено для настройки различных параметров RED (например, веса (Weight)) с целью оптимизации функции мощности (соотношение пропускной способности к задержке). Производительность этих параметров также была подтверждена с помощью симуляции, и было показано, что алгоритм не является чрезмерно чувствительным к ним. Важно помнить, что весь этот анализ и симуляция зависят от определенной характеристики сетевой нагрузки. Реальный вклад RED заключается в механизме, с помощью которого маршрутизатор может более точно управлять длиной своей очереди. Определение того, что именно является оптимальной длиной очереди, зависит от микса трафика и до сих пор является предметом исследований, с реальной информацией, которая теперь собирается из реального внедрения RED в Интернет.

Рассмотрим установку двух порогов, MinThreshold и MaxThreshold. Если трафик довольно взрывной, то MinThreshold должен быть достаточно большим, чтобы поддерживать использование канала на приемлемо высоком уровне. Также разница между двумя порогами должна быть больше типичного увеличения вычисленной средней длины очереди за одно время кругового обхода (RTT). Установка MaxThreshold в два раза больше MinThreshold кажется разумным эмпирическим правилом, учитывая микс трафика в сегодняшнем Интернете. Кроме того, поскольку мы ожидаем, что средняя длина очереди будет колебаться между двумя порогами в периоды высокой нагрузки, должно быть достаточно свободного буферного пространства выше MaxThreshold, чтобы поглотить естественные взрывы трафика в Интернете без вынуждения маршрутизатора переходить в режим сброса хвоста.

Мы уже упомянули, что вес (Weight) определяет временную постоянную для низкочастотного фильтра скользящего среднего, и это дает нам подсказку, как можно выбрать подходящее значение для него. Напомним, что RED пытается отправлять сигналы TCP-потокам, сбрасывая пакеты во время перегрузки. Предположим, что маршрутизатор сбрасывает пакет из некоторого TCP-соединения, а затем сразу же пересылает еще несколько пакетов из того же соединения. Когда эти пакеты достигают получателя, он начинает отправлять дублирующие ACK отправителю. Когда отправитель видит достаточное количество дублирующих ACK, он уменьшает размер своего окна. Таким образом, с момента сброса пакета маршрутизатором до момента, когда этот же маршрутизатор начинает видеть некоторое облегчение от затронутого соединения в виде уменьшенного размера окна, должно пройти по крайней мере одно время кругового обхода (RTT) для этого соединения. Вероятно, нет особого смысла в том, чтобы маршрутизатор реагировал на перегрузку во временных масштабах намного меньше времени кругового обхода соединений, проходящих через него. Как было отмечено ранее, 100 мс — неплохая оценка среднего времени кругового обхода в Интернете. Таким образом, вес (Weight) должен быть выбран таким образом, чтобы изменения длины очереди во временных масштабах намного меньше 100 мс были отфильтрованы.

Поскольку RED работает, отправляя сигналы TCP-потокам, чтобы они замедлились, вы можете задаться вопросом, что произойдет, если эти сигналы будут игнорироваться. Это часто называют проблемой *безответных потоков* (unresponsive flow). Безответные потоки используют больше сетевых ресурсов, чем их справедливая доля, и могут вызвать коллапс перегрузки, если их будет достаточно много, как во времена, когда не было контроля перегрузки TCP. Некоторые из техник, описанных в следующем разделе, могут помочь с этой проблемой, изолируя определенные классы трафика от других. Также возможно, что вариант RED мог бы более сильно *сбрасывать* пакеты из потоков, которые не реагируют на первоначальные подсказки, которые он посылает.

Явное уведомление о перегрузке (ECN)

RED является наиболее изученным механизмом активного управления очередью (AQM), но он не был широко внедрен, отчасти из-за того, что не приводит к идеальному поведению во всех обстоятельствах. Однако он является эталоном для понимания поведе-

ния AQM. Другое, что появилось из RED, это признание того, что TCP мог бы работать лучше, если бы маршрутизаторы отправляли более явный сигнал о перегрузке.

Итак, вместо того чтобы сбрасывать пакет и полагаться на то, что TCP в конечном итоге это заметит (например, из-за прибытия дублирующего ACK), RED (или любой другой AQM-алгоритм, если на то пошло) может сделать лучше, если он вместо этого пометит пакет и продолжит его отправку к месту назначения. Эта идея была закреплена в изменениях заголовков IP и TCP, известных как *явное уведомление о перегрузке* (Explicit Congestion Notification, ECN).

Если говорить более конкретно, то эта обратная связь реализуется путем использования двух битов в поле TOS IP как битов ECN. Один бит устанавливается источником для обозначения того, что он способен к ECN, то есть способен реагировать на уведомление о перегрузке. Этот бит называется битом ECT (ECN-Capable Transport). Другой бит устанавливается маршрутизаторами вдоль сквозного пути при возникновении перегрузки, как это вычисляется любым работающим AQM-алгоритмом. Этот бит называется битом CE (Congestion Encountered).

В дополнение к этим двум битам в заголовке IP (которые не зависят от транспорта) ECN также включает добавление двух дополнительных флагов в заголовок TCP. Первый, ECE (ECN-Echo), сообщает от получателя отправителю, что он получил пакет с установленным битом CE. Второй, CWR (Congestion Window Reduced), сообщает от отправителя получателю, что он уменьшил окно перегрузки.

Хотя ECN теперь является стандартной интерпретацией двух из восьми битов в поле TOS заголовка IP и поддержка ECN настоятельно рекомендуется, она не является обязательной. Более того, нет единого рекомендованного AQM-алгоритма, но есть список требований, которым должен соответствовать хороший AQM-алгоритм. Как и алгоритмы контроля перегрузки TCP, каждый AQM-алгоритм имеет свои преимущества и недостатки, поэтому нам нужно много знать о них. Однако есть один конкретный сценарий, в котором алгоритм контроля перегрузки TCP и AQM-алгоритм разработаны для совместной работы: центр обработки данных. Мы вернемся к этому случаю использования далее.

Глава 6.4.2. Подходы, основанные на источниках (Vegas, BBR, DCTCP)

В отличие от предыдущих схем избежания перегрузок, которые зависели от сотрудничества с маршрутизаторами, мы теперь опишем стратегию обнаружения начальных стадий перегрузки — до того, как произойдут потери — с конечных хостов. Сначала мы кратко рассмотрим набор связанных механизмов, которые используют разную информацию для обнаружения ранних стадий перегрузки, а затем подробно опишем два конкретных механизма.

Общая идея этих техник заключается в том, чтобы наблюдать за сигналами от сети, что очередь какого-то маршрутизатора начинает заполняться и что перегрузка случится вскоре, если ничего не предпринять. Например, источник может заметить, что по мере заполнения очередей пакетов в маршрутизаторах сети происходит заметное увеличение RTT для каждого последующего пакета, который он отправляет. Один из алгоритмов использует это наблюдение следующим образом: окно перегрузки обычно увеличивается, как в TCP, но каждые две задержки при круговом обходе алгоритм проверяет, превышает ли текущий RTT среднее значение между минимальным и максимальным RTT, наблюдаемыми до сих пор. Если это так, алгоритм уменьшает окно перегрузки на одну восьмую.

Второй алгоритм делает нечто подобное. Решение о том, изменять ли текущий размер окна, основано на изменениях как RTT, так и размера окна. Окно регулируется один раз каждые два задержки при круговом обходе на основе произведения

$$(\text{CurrentWindow} - \text{OldWindow}) \times (\text{CurrentRTT} - \text{OldRTT})$$

Если результат положительный, источник уменьшает размер окна на одну восьмую; если результат отрицательный или равен нулю, источник увеличивает окно на одну максимальную величину пакета. Обратите внимание, что окно изменяется при каждой настройке; то есть оно колеблется вокруг своей оптимальной точки.

Еще одно изменение, наблюдаемое по мере приближения сети к перегрузке, заключается в выравнивании скорости отправки. Третья схема использует это обстоятельство. Каждый RTT она увеличивает размер окна на один пакет и сравнивает достигнутую пропускную способность с пропускной способностью, когда окно было на один пакет меньше. Если разница составляет менее половины пропускной способности, достигнутой, когда только один пакет был в пути — как это было в начале соединения, — алгоритм уменьшает окно на один пакет. Эта схема вычисляет пропускную способность, деля количество байтов, находящихся в сети, на RTT.

TCP Vegas

Механизм, который мы собираемся описать более подробно, похож на последний алгоритм тем, что он анализирует изменения скорости передачи или, если более конкретно, изменения скорости отправки. Однако он отличается от предыдущего алгоритма тем, как он вычисляет пропускную способность, и вместо того чтобы искать изменение наклона пропускной способности, он сравнивает измеренную скорость передачи с ожидаемой скоростью передачи. Алгоритм, TCP Vegas, не очень широко используется в Интернете сегодня, но стратегия, которую он использует, была принята другими алгоритмами, которые сейчас внедряются.

Интуиция, лежащая в основе алгоритма Vegas, может быть видна на трассировке стандартного TCP, представленной на рис. 6.19. Верхний график, показанный на рис. 6.19, отслеживает окно перегрузки соединения; он показывает ту же информацию, что и трассировки, приведенные ранее в этой главе. Средний и нижний графики представляют новую информацию: средний график показывает среднюю скорость отправки, измеренную на источнике, а нижний график показывает среднюю длину очереди, измеренную на маршрутизаторе узкого места. Все три графика синхронизированы по времени. В период между 4,5 и 6,0 секундами (затененная область) окно перегрузки увеличивается (верхний график). Мы ожидаем, что наблюдаемая пропускная способность также увеличится, но вместо этого она остается на одном уровне (средний график). Это происходит потому, что пропускная способность не может увеличиваться сверх доступной полосы пропускания. За пределами этой точки любое увеличение размера окна приводит лишь к тому, что пакеты занимают место в буфере маршрутизатора узкого места (нижний график).

Полезная метафора, описывающая явление, иллюстрируемое на рис. 6.19, — это вождение по льду. Спидометр (окно перегрузки) может показывать, что вы едете со скоростью 30 миль в час, но, глядя в окно автомобиля и видя, как вас обгоняют люди пешком (измеренная скорость отправки), вы понимаете, что едете не быстрее 5 миль в час. Лишняя энергия поглощается шинами автомобиля (буферами маршрутизатора).

TCP Vegas использует эту идею для измерения и контроля количества лишних данных, которые это соединение имеет в пути, где под «лишними данными» мы подразумеваем данные, которые источник не отправлял бы, если бы пытался точно соответствовать доступной полосе пропускания сети. Цель TCP Vegas — поддерживать «правильное» количество лишних данных в сети. Очевидно, что если источник отправляет слишком много лишних данных, это вызовет долгие задержки и, возможно, приведет к перегрузке. Менее очевидно, что если соединение отправляет слишком мало лишних данных, оно не сможет быстро реагировать на временное увеличение доступной полосы пропускания сети. Действия по избежанию перегрузки в TCP Vegas основаны на изменениях в оценочном количестве лишних данных в сети, а не только на сброшенных пакетах. Теперь мы опишем алгоритм подробно.

Во-первых, определим базовое RTT (BaseRTT) данного потока как RTT пакета, когда поток не перегружен. На практике TCP Vegas устанавливает BaseRTT как минимум из всех

измеренных времен кругового обхода; обычно это RTT первого пакета, отправленного соединением, до того как очереди маршрутизаторов увеличатся из-за трафика, создаваемого этим потоком. Если мы предполагаем, что не переполняем соединение, то ожидаемая пропускная способность определяется как

$$\text{ExpectedRate} = \text{CongestionWindow} / \text{BaseRTT}$$

где *CongestionWindow* — это окно перегрузки TCP, которое мы предполагаем (для целей данного обсуждения) равным количеству байтов в пути.

Во-вторых, TCP Vegas вычисляет текущую скорость отправки, *ActualRate*. Это делается путем записи времени отправки определенного пакета, записи количества байтов, переданных между временем отправки этого пакета и получением его подтверждения, вычисления выборочного RTT для определенного пакета при получении его подтверждения и деления количества переданных байтов на выборочный RTT. Этот расчет выполняется один раз за время кругового обхода.

В-третьих, TCP Vegas сравнивает *ActualRate* с *ExpectedRate* и соответственно регулирует окно. Пусть $\text{Diff} = \text{ExpectedRate} - \text{ActualRate}$. Обратите внимание, что *Diff* положительный или равен нулю по определению, так как если $\text{ActualRate} > \text{ExpectedRate}$, это означает, что нам нужно изменить *BaseRTT* на последнее выборочное RTT. Также мы определяем два порога, $\alpha < \beta$, которые примерно соответствуют наличию слишком малого и слишком большого количества лишних данных в сети соответственно. Когда $\text{Diff} < \alpha$, TCP Vegas линейно увеличивает окно перегрузки в течение следующего RTT, а когда $\text{Diff} > \beta$, TCP Vegas линейно уменьшает окно перегрузки в течение следующего RTT. TCP Vegas оставляет окно перегрузки неизменным, когда $\alpha < \text{Diff} < \beta$.

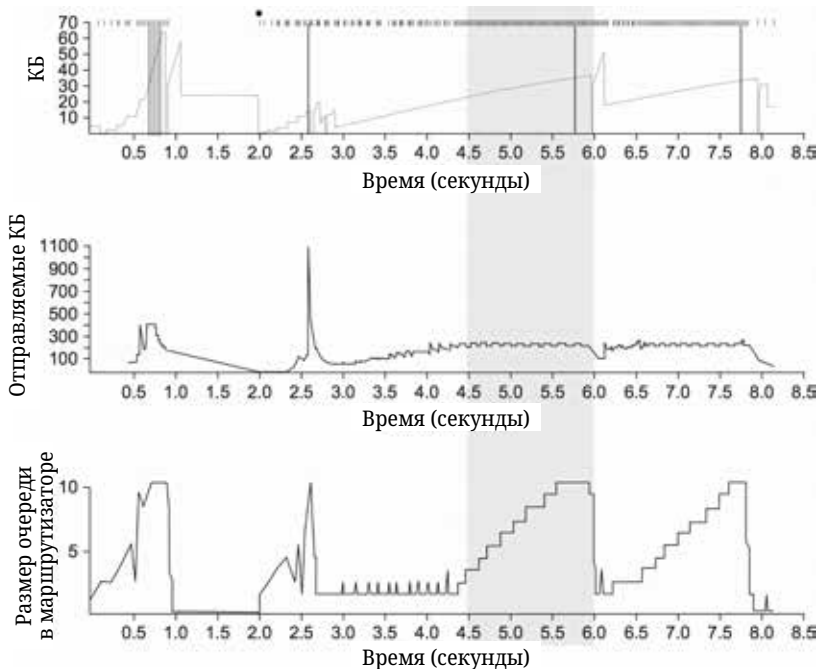


Рисунок 6.19. Окно перегрузки в сравнении с наблюдаемой пропускной способностью (три графика синхронизированы). Вверху — окно перегрузки; в середине — наблюдаемая пропускная способность; внизу — буферное пространство, занимаемое маршрутизатором. Кривая линия = *CongestionWindow*; сплошная пуля = тайм-аут; хеш-метки = время передачи каждого пакета; вертикальные полосы = время, когда пакет, который в итоге был повторно передан, был передан первым.

Интуитивно мы можем видеть, что чем дальше фактическая пропускная способность отклоняется от ожидаемой пропускной способности, тем больше перегрузка в сети, а это означает, что скорость отправки должна быть уменьшена. Порог β запускает это уменьшение. С другой стороны, когда фактическая пропускная способность становится слишком близкой к ожидаемой пропускной способности, соединение рискует не использовать доступную полосу пропускания. Порог α запускает это увеличение. Общая цель состоит в том, чтобы удерживать количество лишних байтов в сети между α и β .

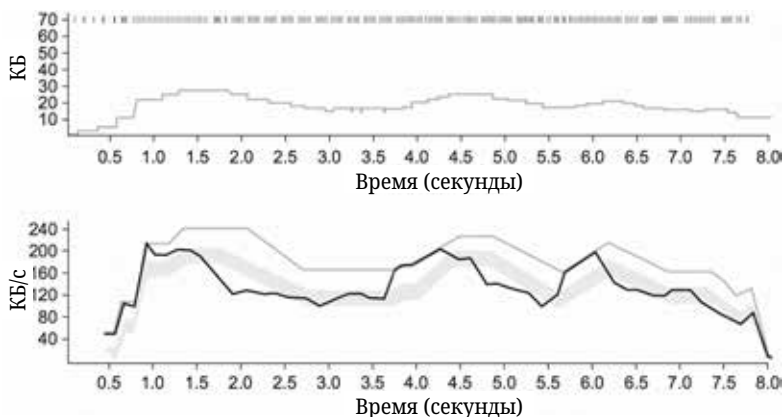


Рисунок 6.20. Трассировка механизма предотвращения перегрузок TCP Vegas. Вверху — окно перегрузки; внизу — ожидаемая (светлая линия) и фактическая (черная линия) пропускная способность. Затрихованная область — это область между α и β порогами.

Рис. 6.20 иллюстрирует алгоритм избегания перегрузок TCP Vegas. Верхний график отслеживает окно перегрузки, показывая ту же информацию, что и другие трассировки, приведенные в этой главе. Нижний график отслеживает ожидаемые и фактические пропускные способности, которые определяют, как устанавливается окно перегрузки. Именно этот нижний график лучше всего иллюстрирует работу алгоритма. Светлая линия отслеживает ExpectedRate, а черная линия отслеживает ActualRate. Широкая затененная полоса показывает область между порогами α и β ; верхняя часть затененной полосы находится на расстоянии α КБ/с от ExpectedRate, а нижняя часть затененной полосы находится на расстоянии β КБ/с от ExpectedRate. Цель состоит в том, чтобы удерживать ActualRate между этими двумя порогами, внутри затененной области. Всякий раз, когда ActualRate опускается ниже затененной области (то есть отклоняется слишком далеко от ExpectedRate), TCP Vegas уменьшает окно перегрузки, так как опасается, что слишком много пакетов буферизуется в сети. Точно так же, когда ActualRate поднимается выше затененной области (то есть приближается к ExpectedRate), TCP Vegas увеличивает окно перегрузки, поскольку опасается, что сеть используется недостаточно.

Поскольку алгоритм, как представлено, сравнивает разницу между фактической и ожидаемой пропускными способностями с порогами α и β , эти два порога определены в терминах КБ/с. Однако, возможно, более точно думать в терминах количества лишних буферов, которые соединение занимает в сети. Например, в соединении с BaseRTT 100 мс и размером пакета 1 КБ, если $\alpha = 30$ КБ/с и $\beta = 60$ КБ/с, мы можем считать, что α указывает, что соединение должно занимать не менее 3 лишних буферов в сети, а β указывает, что соединение должно занимать не более 6 лишних буферов в сети. На практике установка α на 1 буфер и β на 3 буфера работает хорошо.

Наконец, вы заметите, что TCP Vegas линейно уменьшает окно перегрузки, что, казалось бы, противоречит правилу, согласно которому мультипликативное уменьшение необходимо для обеспечения стабильности. Объяснение заключается в том,

что TCP Vegas действительно использует мультипликативное уменьшение при возникновении тайм-аута; описанное линейное уменьшение является ранним уменьшением окна перегрузки, которое должно происходить до того, как возникнет перегрузка и начнется потеря пакетов.

TCP BBR

BBR (Bottleneck Bandwidth and RTT) — это новый алгоритм управления перегрузкой TCP, разработанный исследователями из Google. Подобно Vegas, BBR основан на задержке, а это означает, что он пытается обнаружить рост буфера, чтобы избежать перегрузки и потери пакетов. И BBR, и Vegas используют минимальное RTT и максимальное RTT, рассчитанные за определенный интервал времени, в качестве основных управляющих сигналов.

BBR также вводит новые механизмы для улучшения производительности, включая распределение пакетов по времени (packet pacing), зондирование пропускной способности (bandwidth probing) и зондирование RTT (RTT probing). Распределение пакетов по времени размещает пакеты с учетом оценки доступной пропускной способности. Это устраняет всплески и ненужные очереди, что приводит к сигналу обратной связи. BBR также периодически увеличивает свою скорость, тем самым зондируя доступную пропускную способность. Аналогично BBR периодически уменьшает свою скорость, тем самым зондируя новое минимальное RTT. Механизм зондирования RTT пытается быть самосинхронизирующимся, это означает, что при наличии нескольких потоков BBR их зондирование RTT происходит одновременно. Это дает более точное представление о фактическом RTT незагруженного пути, что решает одну из основных проблем механизмов управления перегрузкой на основе задержки: наличие точных данных о RTT незагруженного пути.

BBR активно разрабатывается и быстро развивается. Один из основных акцентов делается на справедливость. Например, некоторые эксперименты показывают, что потоки CUBIC получают в 100 раз меньше пропускной способности при конкуренции с потоками BBR, а другие эксперименты показывают, что даже среди потоков BBR возможно несправедливое распределение. Другой важной задачей является избегание высоких уровней повторной передачи, когда в некоторых случаях повторно передается до 10% пакетов.

DCTCP

Мы завершаем рассмотрение примером ситуации, когда вариант алгоритма управления перегрузкой TCP был разработан для работы в сочетании с ECN: в облачных центрах обработки данных. Эта комбинация называется DCTCP, что означает Data Center TCP (TCP для центров обработки данных). Ситуация уникальна тем, что центр обработки данных является автономным, и поэтому можно развернуть специализированную версию TCP, которая не должна заботиться о справедливом отношении к другим потокам TCP. Центры обработки данных также уникальны тем, что они построены с использованием недорогих коммутаторов, и поскольку нет необходимости беспокоиться о длинных каналах, пересекающих континент, коммутаторы обычно оснащены без избытка буферов.

Идея проста. DCTCP адаптирует ECN, оценивая долю байтов, которые сталкиваются с перегрузкой, а не просто обнаруживая, что перегрузка вот-вот произойдет. На конечных узлах DCTCP затем масштабирует окно перегрузки на основе этой оценки. Стандартный алгоритм TCP все еще активируется, если пакет действительно потерян. Подход разработан для достижения высокой устойчивости к всплескам, низкой задержки и высокой пропускной способности с коммутаторами с небольшими буферами.

Ключевая задача, с которой сталкивается DCTCP, заключается в оценке доли байтов, сталкивающихся с перегрузкой. Каждый коммутатор прост. Если пакет прибывает и коммутатор видит, что длина очереди (K) превышает определенный порог; например,

$$K > \frac{(RTT \times C)}{7}$$

где C — это скорость соединения в пакетах в секунду, тогда коммутатор устанавливает бит CE в заголовке IP. Сложность RED не требуется.

Приемник затем поддерживает булеву переменную для каждого потока, которую мы будем обозначать как $SeenCE$, и реализует следующую машину состояний в ответ на каждый полученный пакет:

- Если бит CE установлен и $SeenCE=False$, установите $SeenCE$ в $True$ и отправьте немедленный ACK.
- Если бит CE не установлен и $SeenCE=True$, установите $SeenCE$ в $False$ и отправьте немедленный ACK.
- В противном случае игнорируйте бит CE .

Неочевидным следствием случая «иначе» является то, что приемник продолжает отправлять задержанные ACK каждые n пакетов, независимо от того, установлен ли бит CE . Это оказалось важным для поддержания высокой производительности.

Наконец, отправитель вычисляет долю байтов, которые столкнулись с перегрузкой в течение предыдущего окна наблюдения (обычно выбранного примерно равным RTT), как отношение общего количества переданных байтов к байтам, подтвержденным с установленным флагом ECE . DCTCP увеличивает окно перегрузки точно так же, как стандартный алгоритм, но уменьшает окно пропорционально количеству байтов, столкнувшихся с перегрузкой в течение последнего окна наблюдения.

Глава 6.5. Качество обслуживания

Обещание универсальных сетей с коммутацией пакетов заключается в поддержке всех видов приложений и данных, включая мультимедийные приложения, передающие оцифрованные аудио- и видеопотоки. В ранние годы одним из препятствий для выполнения этого обещания была необходимость в высокоскоростных каналах связи. Сейчас это уже не проблема, но для передачи аудио и видео по сети требуется нечто большее, чем просто обеспечение достаточной пропускной способности.

Участники телефонного разговора, например, ожидают возможности общаться так, чтобы один человек мог ответить на что-то сказанное другим и быть услышанным почти сразу. Таким образом, своевременность доставки может быть очень важной. Мы называем приложения, чувствительные к своевременности данных, *приложениями реального времени*. Голосовые и видеоприложения обычно являются каноническими примерами, но существуют и другие, такие как промышленное управление — команда, отправленная к роботу, должна достичь его до того, как рука робота столкнется с чем-то. Даже приложения для передачи файлов могут иметь ограничения по своевременности, такие как требование завершить обновление базы данных за ночь до того, как начнет работать бизнес, которому нужны эти данные.

Отличительной характеристикой приложений реального времени является то, что им требуется определенная гарантия от сети, что данные, вероятно, будут доставлены вовремя (по какому-то определению «вовремя»). Тогда как приложения, которые не работают в режиме реального времени, могут использовать стратегию повторной передачи от конца до конца, чтобы убедиться, что данные доставлены корректно, такая стратегия не может обеспечить своевременность: повторная передача только увеличивает общую задержку, если данные прибывают поздно. Своевременная доставка должна быть обеспечена самой сетью (маршрутизаторами), а не только на краях сети (хостами). Следовательно, мы заключаем, что модель «лучшего усилия», в которой сеть пытается доставить ваши данные, но не дает никаких обещаний и оставляет операцию восста-

новления на краях, недостаточна для приложений реального времени. Нам нужна новая модель обслуживания, в которой приложения, нуждающиеся в больших гарантиях, могут запросить их у сети.

Сеть может ответить, предоставив гарантию, что она постарается сделать лучше, или, возможно, сказав, что в данный момент она не может обещать ничего лучше. Заметьте, что такая модель обслуживания является надмножеством оригинальной модели: приложения, удовлетворенные обслуживанием «лучшего усилия», должны иметь возможность использовать новую модель обслуживания; их требования просто менее строгие. Это подразумевает, что сеть будет обрабатывать некоторые пакеты иначе, чем другие — что не делается в модели «лучшего усилия». Сеть, которая может предоставить эти разные уровни обслуживания, часто называется поддерживающей качество обслуживания (QoS).

Глава 6.5.1. Требования приложений

Прежде чем рассматривать различные протоколы и механизмы, которые могут быть использованы для обеспечения качества обслуживания приложений, мы должны попытаться понять, какие потребности у этих приложений есть. Для начала мы можем разделить приложения на два типа: приложения реального времени и приложения, которые не работают в режиме реального времени. Последние иногда называются *традиционными* приложениями передачи данных, поскольку они традиционно были основными приложениями, найденными в сетях передачи данных. Они включают большинство популярных приложений, таких как SSH, передача файлов, электронная почта, веб-серфинг и так далее. Все эти приложения могут работать без гарантий своевременной доставки данных. Другой термин для этого класса приложений, которые не работают в режиме реального времени, — эластичные, поскольку они способны гибко адаптироваться к увеличению задержек. Заметьте, что эти приложения могут выигрывать от более коротких задержек, но они не становятся неработоспособными по мере увеличения задержек. Также заметьте, что их требования к задержкам варьируются от интерактивных приложений, таких как SSH, до более асинхронных, таких как электронная почта, с интерактивной передачей больших объемов данных, как передача файлов, посередине.



Рисунок 6.21. Аудиоприложение.

Пример передачи аудио в реальном времени

В качестве конкретного примера приложения реального времени рассмотрим аудиоприложение, аналогичное тому, что изображено на рис. 6.21. Данные генерируются путем сбора образцов с микрофона и их оцифровки с помощью аналого-цифрового (A-to-D) преобразователя. Цифровые образцы помещаются в пакеты, которые передаются по сети и принимаются на другом конце. На принимающем хосте данные должны воспроизводиться с соответствующей скоростью. Например, если голосовые образцы собираются с частотой один раз в 125 мкс, их следует воспроизводить с той же частотой. Таким образом, можно считать, что у каждого образца есть определенное *время воспроизведения*: момент времени, в который он необходим на принимающем хосте. В примере с голосом каждый образец имеет время воспроизведения, которое

на 125 мкс позже, чем у предыдущего образца. Если данные прибывают после соответствующего времени воспроизведения, будь то из-за задержки в сети или потому, что они были потеряны и затем переданы повторно, они становятся бесполезными. Полная бесполезность поздних данных характеризует приложения реального времени. В эластичных приложениях данные могут быть полезными, даже если они не приходят вовремя.

Одним из способов обеспечения работы голосового приложения было бы гарантировать, что все образцы проходят по сети за одинаковое время. Тогда, поскольку образцы вводятся с частотой один раз в 125 мкс, они будут появляться у получателя с той же частотой, готовые к воспроизведению. Однако, как правило, сложно гарантировать, что все данные, проходящие через сеть с коммутацией пакетов, будут испытывать точно одинаковую задержку. Пакеты сталкиваются с очередями в коммутаторах или маршрутизаторах, и длина этих очередей варьируется со временем, что означает, что задержки также варьируются со временем и, следовательно, могут быть разными для каждого пакета в аудиопотоке. Способ справиться с этим на стороне получателя заключается в буферизации некоторого объема данных в резерве, обеспечивая таким образом наличие пакетов, ожидающих воспроизведения в нужное время. Если пакет задерживается на короткое время, он попадает в буфер до тех пор, пока не наступит его время воспроизведения. Если он задерживается на долгое время, ему не нужно долго храниться в буфере получателя перед воспроизведением. Таким образом, мы фактически добавили постоянное смещение к времени воспроизведения всех пакетов как форму страховки. Мы называем это смещение *точкой воспроизведения*. Единственный момент, когда у нас возникают проблемы, это когда пакеты задерживаются в сети настолько долго, что прибывают после времени их воспроизведения, вызывая истощение буфера воспроизведения.

Работа буфера воспроизведения иллюстрируется на рис. 6.22. Левая диагональная линия показывает генерацию пакетов с равномерной скоростью. Волнистая линия показывает время прибытия пакетов, которое зависит от того, что они встречают в сети, и может варьироваться. Правая диагональная линия показывает воспроизведение пакетов с равномерной скоростью после их нахождения в буфере воспроизведения в течение некоторого времени. Пока линия воспроизведения достаточно сдвинута вправо по времени, вариации задержки в сети не заметны для приложения. Однако если мы немного сдвинем линию воспроизведения влево, некоторые пакеты начнут прибывать слишком поздно, чтобы быть полезными.

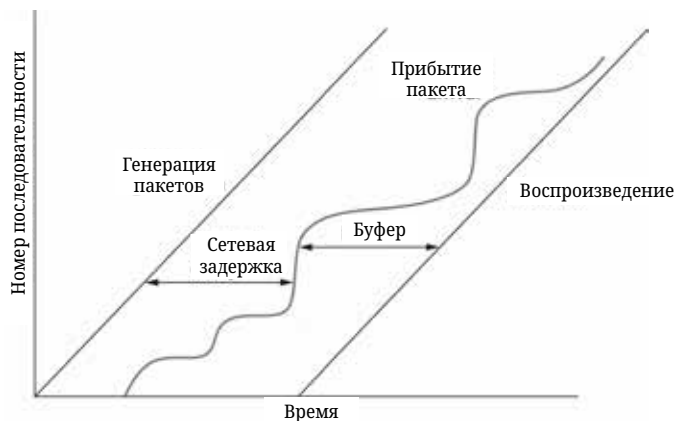


Рисунок 6.22. Буфер воспроизведения.

Для нашего аудиоприложения существуют ограничения на то, как долго мы можем откладывать воспроизведение данных. Трудно вести разговор, если время между тем, когда вы говорите, и тем, когда ваш собеседник вас слышит, составляет более 300 мс. Таким образом, в этом случае нам нужна от сети гарантия, что все наши данные будут доставлены в течение 300 мс. Если данные прибывают рано, мы буферизуем их до их корректного времени воспроизведения. Если они прибывают поздно, они бесполезны, и мы должны их отбросить.

Чтобы лучше понять, насколько может варьироваться задержка в сети, на рисунке 6.23 показана односторонняя задержка, измеренная по определенному пути через Интернет в течение одного дня. Хотя точные числа будут варьироваться в зависимости от пути и даты, ключевым фактором здесь является *изменчивость* задержки, которая постоянно наблюдается почти на любом пути в любое время. Как указано на графике накопительными процентами в верхней части, 97% пакетов в этом случае имели задержку 100 мс или менее. Это означает, что если бы наше аудиоприложение установило точку воспроизведения на 100 мс, в среднем 3 из каждых 100 пакетов прибывали бы слишком поздно, чтобы быть полезными. Важно заметить, что хвост кривой (насколько далеко она простирается вправо) очень длинный. Нам бы пришлось установить точку воспроизведения на более чем 200 мс, чтобы гарантировать, что все пакеты прибывают вовремя.

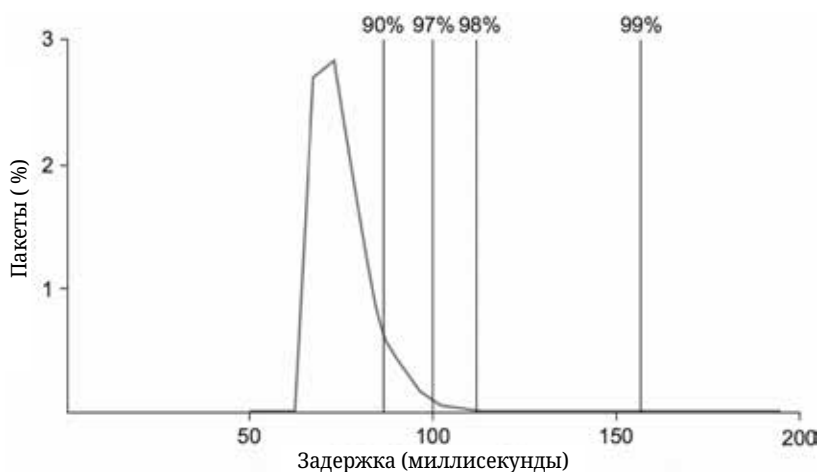


Рисунок 6.23. Пример распределения задержек для интернет-соединения.

Таксономия приложений реального времени

Теперь, когда у нас есть конкретное представление о том, как работают приложения реального времени, мы можем рассмотреть различные классы приложений, которые мотивируют нашу модель обслуживания. Следующая таксономия во многом обязана работе Кларка, Брейдена, Шенкера и Чжана, чьи статьи по этой теме можно найти в главе «Дополнительное чтение» для этого раздела. Таксономия приложений суммирована на рис. 6.24.

Первой характеристикой, по которой можно классифицировать приложения, является их терпимость к потере данных, где «потеря» может произойти из-за того, что пакет прибыл слишком поздно для воспроизведения, из-за обычных причин в сети. С одной стороны, один потерянный аудиосэмпл можно интерполировать из окружающих сэмплов с относительно небольшим влиянием на воспринимаемое качество аудио. Только по мере того, как теряется все больше и больше сэмплов, качество снижается до точки, когда речь становится непонятной. С другой стороны, программа управления роботом,

вероятно, является примером приложения реального времени, которое не может терпеть потерь — потеря пакета, содержащего команду остановить руку робота, недопустима. Таким образом, мы можем классифицировать приложения реального времени как *терпимые* или *нетерпимые* в зависимости от того, могут ли они терпеть случайные потери. (Заметим, что многие приложения реального времени более терпимы к случайным потерям, чем приложения не реального времени; например, сравните наше аудиоприложение с передачей файлов, где непоправимая потеря одного бита может сделать файл полностью бесполезным.)

Вторым способом охарактеризовать приложения реального времени является их адаптивность. Например, аудиоприложение может адаптироваться к задержке, которую пакеты испытывают при прохождении через сеть. Если мы замечаем, что пакеты почти всегда прибывают в течение 300 мс после отправки, то мы можем установить точку воспроизведения соответственно, буферизуя любые пакеты, которые прибывают менее чем за 300 мс. Предположим, что мы впоследствии наблюдаем, что все пакеты прибывают в течение 100 мс после отправки. Если мы переместим нашу точку воспроизведения на 100 мс, пользователи приложения, вероятно, заметят улучшение. Процесс смещения точки воспроизведения фактически потребует воспроизведения образцов с увеличенной скоростью в течение некоторого времени. В голосовом приложении это можно сделать так, что будет едва заметно, просто сокращая паузы между словами. Таким образом, настройка точки воспроизведения в этом случае довольно проста и была эффективно реализована для нескольких голосовых приложений, таких как программа аудиоконференций, известная как *vat*. Заметьте, что настройка точки воспроизведения может происходить в любом направлении, но это фактически связано с искажением воспроизводимого сигнала в течение периода настройки, и эффекты этого искажения будут сильно зависеть от того, как конечный пользователь использует данные.

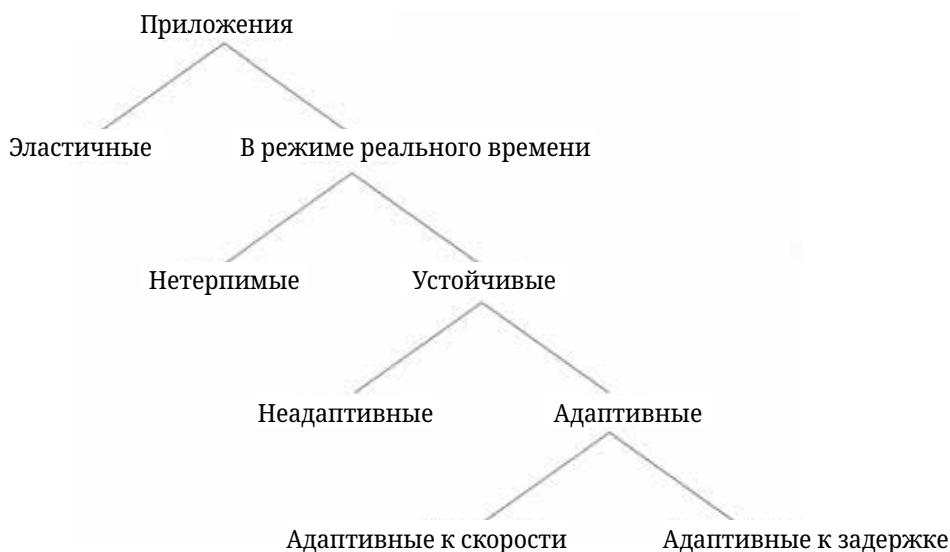


Рисунок 6.24. Таксономия приложений.

Обратите внимание, что если мы установим точку воспроизведения, предполагая, что все пакеты придут в течение 100 мс, а затем обнаружим, что некоторые пакеты прибывают чуть позже, нам придется их отбросить, тогда как мы не должны были бы их отбросить, если бы оставили точку воспроизведения на 300 мс. Таким образом, мы должны сдвигать точку воспроизведения только тогда, когда это дает ощутимое преимущество

и когда у нас есть некоторые доказательства того, что количество поздних пакетов будет приемлемо малым. Мы можем сделать это на основе наблюдаемой недавней истории или из-за некоторых гарантий от сети.

Мы называем приложения, которые могут регулировать свою точку воспроизведения, *адаптивными к задержке* (delay-adaptive). Другой класс адаптивных приложений — это *адаптивные к скорости* (rate adaptive). Например, многие алгоритмы кодирования видео могут менять битрейт в зависимости от качества. Таким образом, если мы обнаружим, что сеть может поддерживать определенную пропускную способность, мы можем установить параметры кодирования соответствующим образом. Если позже станет доступна большая пропускная способность, мы можем изменить параметры для улучшения качества.

Подходы к поддержке QoS

Учитывая это разнообразие требований приложений, нам нужна более сложная модель обслуживания, которая удовлетворяла бы потребности любого приложения. Это приводит нас к модели обслуживания с не одним классом (лучшее усилие), а несколькими классами, каждый из которых доступен для удовлетворения потребностей определенных наборов приложений. Для достижения этой цели мы теперь готовы рассмотреть некоторые подходы, которые были разработаны для предоставления различных уровней качества обслуживания (QoS). Эти подходы можно разделить на две широкие категории:

- Подходы с *тонкой* настройкой, которые предоставляют QoS для отдельных приложений или потоков.
- Подходы с *грубой* настройкой, которые предоставляют QoS для больших классов данных или агрегированного трафика.

В первой категории мы находим *интегрированные сервисы* (Integrated Services), архитектуру QoS, разработанную в IETF и часто связанную с протоколом резервирования ресурсов (RSVP). Во второй категории находятся *дифференцированные сервисы* (Differentiated Services), которые, вероятно, являются самым широко развернутым механизмом QoS сегодня. Мы обсудим их по очереди позже.

Наконец, как мы предположили в начале этого раздела, добавление поддержки QoS в сеть не обязательно является окончанием истории о поддержке приложений реального времени. Мы завершаем наше обсуждение, возвращаясь к тому, что конечные хосты могут сделать для лучшей поддержки потоков реального времени, независимо от того, насколько широко развернуты механизмы QoS, такие как интегрированные или дифференцированные сервисы.

Глава 6.5.2. Интегрированные сервисы (RSVP)

Термин *интегрированные сервисы* (часто сокращенно называемые IntServ) относится к работе, выполненной IETF в период с 1995 по 1997 годы. Рабочая группа IntServ разработала спецификации ряда *классов обслуживания*, предназначенных для удовлетворения потребностей некоторых из описанных выше типов приложений. Она также определила, как RSVP может быть использован для создания резервов, используя эти классы обслуживания. Следующие абзацы предоставляют обзор этих спецификаций и механизмов, которые используются для их реализации.

Классы обслуживания

Один из классов обслуживания предназначен для нетерпимых приложений. Эти приложения требуют, чтобы пакет никогда не опаздывал. Сеть должна гарантировать, что максимальная задержка, которую может испытать любой пакет, имеет заданное значение; приложение может затем установить свою точку воспроизведения так, чтобы ни один пакет никогда не прибыл после своего времени воспроизведения. Мы предполагаем, что раннее прибытие пакетов всегда может быть обработано буферизацией. Этот сервис называется *гарантированным обслуживанием*.

В дополнение к гарантированному обслуживанию IETF рассматривала несколько других сервисов, но в конечном итоге остановилась на одном, предназначенном для терпимых, адаптивных приложений. Этот сервис известен как *управляемая нагрузка* (controlled load) и был мотивирован наблюдением, что существующие приложения этого типа работают довольно хорошо в сетях, которые не сильно загружены. Аудиоприложения, например, регулируют свою точку воспроизведения по мере изменения сетевой задержки и обеспечивают разумное качество звука, пока уровни потерь остаются на уровне 10% или ниже.

Цель сервиса управляемой нагрузки состоит в том, чтобы эмулировать слабозагруженную сеть для тех приложений, которые запрашивают этот сервис, даже если сеть в целом может быть сильно загружена. Трюк заключается в использовании механизма очереди, такого как WFQ, чтобы изолировать трафик управляемой нагрузки от остального трафика, и некоторой формы контроля допуска, чтобы ограничить общее количество трафика управляемой нагрузки на канале так, чтобы нагрузка оставалась относительно низкой. Мы обсудим контроль допуска более подробно ниже.

Очевидно, что эти два класса обслуживания являются подмножеством всех классов, которые могут быть предоставлены. На самом деле другие сервисы были специфицированы, но так и не были стандартизированы в рамках работы IETF. До сих пор два описанных выше сервиса (наряду с традиционным лучшим усилием) оказались достаточно гибкими, чтобы удовлетворить потребности широкого круга приложений.

Обзор механизмов

Теперь, когда мы дополнили нашу модель обслуживания на основе лучшего усилия новыми классами обслуживания, следующий вопрос заключается в том, как реализовать сеть, которая предоставляет эти услуги приложениям. В этой главе описываются ключевые механизмы. Имейте в виду, что описываемые механизмы все еще дорабатываются сообществом разработчиков Интернета. Основная идея, которую нужно вынести из этого обсуждения, — общее понимание компонентов, участвующих в поддержке вышеописанной модели обслуживания.

Во-первых, в то время как при обслуживании на основе лучшего усилия мы просто указываем сети, куда мы хотим отправить наши пакеты, и на этом все, служба реального времени предполагает предоставление сети дополнительной информации о типе требуемого сервиса. Мы можем дать ей качественную информацию, такую как «используй службу контролируемой нагрузки» или количественную информацию, такую как «мне нужна максимальная задержка в 100 мс». Помимо описания того, что мы хотим, нам нужно сообщить сети что-то о том, что мы будем в нее внедрять, так как приложение с низкой пропускной способностью потребует меньше сетевых ресурсов, чем приложение с высокой пропускной способностью. Набор информации, который мы предоставляем сети, называется *спецификацией потока* (flowspec). Это название происходит от идеи, что набор пакетов, связанных с одним приложением и имеющих общие требования, называется потоком, что согласуется с нашим использованием этого термина в предыдущем разделе, в котором описывались соответствующие проблемы.

Во-вторых, когда мы просим сеть предоставить нам определенную услугу, сеть должна решить, может ли она действительно предоставить эту услугу. Например, если 10 пользователей запрашивают услугу, при которой каждый будет постоянно использовать 2 Мбит/с пропускной способности канала, и они все делят канал с пропускной способностью 10 Мбит/с, сеть должна будет отказать некоторым из них. Процесс принятия решения о том, когда сказать «нет», называется *контролем допуска*.

В-третьих, нам нужен механизм, с помощью которого пользователи сети и компоненты самой сети обмениваются информацией, такой как запросы на обслуживание, спецификации потоков и решения о контроле допуска. Это иногда называется

сигнализацией, но так как у этого слова несколько значений, мы называем этот процесс *резервированием ресурсов*, и он осуществляется с помощью протокола резервирования ресурсов.

Наконец, когда потоки и их требования описаны, и решения о контроле допуска приняты, коммутаторы и маршрутизаторы сети должны удовлетворить требования потоков. Ключевая часть удовлетворения этих требований — управление очередями пакетов и их расписанием для передачи в коммутаторах и маршрутизаторах. Этот последний механизм называется *планированием пакетов*.

Спецификации потоков

Спецификация потока имеет две отдельные части: часть, описывающую характеристики трафика потока (называемую *TSpec*), и часть, описывающую запрашиваемую услугу от сети (называемую *RSpec*). *RSpec* очень специфична для услуги и относительно легко описывается. Например, для службы контролируемой нагрузки *RSpec* тривиальна: приложение просто запрашивает службу контролируемой нагрузки без дополнительных параметров. Для гарантированной службы вы можете указать целевую задержку или предел. (В спецификации гарантированной службы IETF вы указываете не задержку, а другую величину, из которой можно вычислить задержку.)

TSpec немного сложнее. Как показано в приведенном выше примере, нам нужно предоставить сети достаточно информации о пропускной способности потока, чтобы можно было принять разумные решения о контроле допуска. Однако для большинства приложений пропускная способность не является единственным числом; она постоянно варьируется. Например, видеоприложение будет генерировать больше бит в секунду, когда сцена быстро меняется, чем когда она статична. Просто знания долгосрочной средней пропускной способности недостаточно, как показывает следующий пример. Предположим, что у нас есть 10 потоков, которые поступают в коммутатор на отдельных входных портах и все выходят на одном канале с пропускной способностью 10 Мбит/с. Предположим, что за некоторый достаточно длинный интервал каждый поток может отправлять не более 1 Мбит/с. Вы можете подумать, что это не представляет проблемы. Однако, если это приложения с переменной битовой скоростью, такие как сжатое видео, они иногда будут отправлять больше своих средних скоростей. Если достаточно источников будут отправлять выше своих средних скоростей, то общая скорость, с которой данные поступают в коммутатор, будет больше 10 Мбит/с. Эти избыточные данные будут помещены в очередь, прежде чем их можно будет отправить по каналу. Чем дольше сохраняется это состояние, тем длиннее становится очередь. Пакеты могут быть сброшены, и даже если до этого не дойдет, данные, находящиеся в очереди, будут задержаны. Если пакеты будут задержаны достаточно долго, запрашиваемая услуга не будет предоставлена.

Как именно мы управляем нашими очередями, чтобы контролировать задержку и избегать потери пакетов, будет обсуждаться далее. Однако здесь важно отметить, что нам нужно знать что-то о том, как пропускная способность наших источников варьируется со временем. Один из способов описать характеристики пропускной способности источников называется *фильтром токенов* (token bucket filter). Такой фильтр описывается двумя параметрами: скорость токенов r и глубина ведра B . Он работает следующим образом. Чтобы отправить байт, мне нужен токен. Чтобы отправить пакет длиной n , мне нужно n токенов. Я начинаю без токенов и накапливаю их со скоростью r в секунду. Я могу накопить не более B токенов. Это означает, что я могу отправить в сеть залпом до B байт так быстро, как захочу, но за достаточно долгий интервал я не могу отправить более r байт в секунду. Оказывается, эта информация очень полезна для алгоритма контроля допуска, когда он пытается выяснить, может ли он удовлетворить новый запрос на обслуживание.

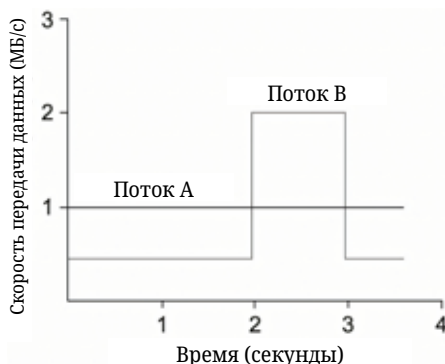


Рисунок 6.25. Два потока с одинаковыми средними скоростями, но разными описаниями токенов

Рис. 6.25 иллюстрирует, как фильтр токенов может быть использован для описания требований потока к пропускной способности. Для простоты предположим, что каждый поток может отправлять данные как отдельные байты, а не как пакеты. Поток А генерирует данные со стабильной скоростью 1 Мбайт/с, поэтому его можно описать фильтром токенов со скоростью $r = 1$ Мбайт/с и глубиной ведра 1 байт. Это означает, что он получает токены со скоростью 1 Мбайт/с, но не может накапливать более 1 токена — он тратит их немедленно. Поток В также отправляет данные со средней скоростью 1 Мбайт/с в долгосрочной перспективе, но делает это, отправляя данные со скоростью 0,5 Мбайт/с в течение 2 секунд, а затем со скоростью 2 Мбайт/с в течение 1 секунды. Поскольку скорость фильтра токенов r , в некотором смысле, является долгосрочной средней скоростью, поток В можно описать фильтром токенов со скоростью 1 Мбайт/с. В отличие от потока А, поток В требует глубины ведра B не менее 1 Мбайт, чтобы он мог накапливать токены, когда отправляет данные с меньшей скоростью, и использовать их, когда отправляет данные со скоростью 2 Мбайт/с. В первые 2 секунды в этом примере он получает токены со скоростью 1 Мбайт/с, но тратит их только со скоростью 0,5 Мбайт/с, поэтому он может накопить $2 \times 0,5 = 1$ Мбайт токенов, которые он затем тратит в третью секунду (вместе с новыми токенами, которые продолжают накапливаться в эту секунду) для отправки данных со скоростью 2 Мбайт/с. В конце третьей секунды, израсходовав избыточные токены, он снова начинает их накапливать, отправляя данные со скоростью 0,5 Мбайт/с.

Интересно отметить, что один поток можно описать множеством различных фильтров токенов. В качестве тривиального примера поток А можно было бы описать тем же фильтром токенов, что и поток В, со скоростью 1 Мбайт/с и глубиной ведра 1 Мбайт. Тот факт, что ему никогда не нужно накапливать токены, не делает это описание неточным, но это означает, что мы не передали сети полезную информацию — факт, что поток А на самом деле очень стабилен в своих потребностях в пропускной способности. В общем, хорошо быть как можно более явным в описании потребностей приложения в пропускной способности, чтобы избежать перераспределения ресурсов в сети.

Контроль допуска

Идея контроля допуска проста: когда новый поток хочет получить определенный уровень обслуживания, контроль допуска смотрит на TSрес и RSрес потока и пытается решить, можно ли предоставить желаемую услугу этому объему трафика, учитывая доступные в настоящее время ресурсы, без ухудшения обслуживания ранее принятых потоков. Если услуга может быть предоставлена, поток принимается; если нет, то ему отказывают. Сложная часть заключается в том, чтобы решить, когда сказать «да» и когда сказать «нет».

Контроль допуска сильно зависит от типа запрашиваемой услуги и дисциплины очередей, применяемой в маршрутизаторах; мы обсудим этот вопрос позже в данной главе. Для гарантированной услуги нужен хороший алгоритм для принятия однозначного решения «да/нет». Решение относительно простое, если в каждом маршрутизаторе используется взвешенное справедливое планирование. Для службы контролируемой нагрузки решение может быть основано на эвристиках, таких как «В прошлый раз, когда я допустил поток с этим TSpec в этот класс, задержки для класса превысили допустимый предел, поэтому я лучше скажу нет» или «Мои текущие задержки настолько малы, что я должен быть в состоянии допустить еще один поток без проблем».

Контроль допуска не следует путать с *полицией*. Первое — это решение на уровне потока о приеме нового потока или нет. Последнее — это функция, применяемая на уровне пакетов, чтобы убедиться, что поток соответствует TSpec, который использовался для резервирования. Если поток не соответствует своему TSpec — например, если он отправляет в два раза больше байт в секунду, чем заявил, — то он, вероятно, будет мешать предоставлению услуги другим потокам, и необходимо предпринять корректирующие действия. Есть несколько вариантов, очевидный из которых — отбросить нарушающие пакеты. Однако другой вариант — проверить, действительно ли пакеты мешают обслуживанию других потоков. Если они не мешают, пакеты могут быть отправлены с меткой, которая говорит, по сути, «Это не соответствующий пакет. Отбросьте его первым, если потребуется отбросить любые пакеты».

Контроль допуска тесно связан с важным вопросом политики. Например, администратор сети может пожелать допускать резервирования, сделанные генеральным директором его компании, при этом отклоняя резервирования, сделанные менее важными сотрудниками. Конечно, запрос на резервирование генерального директора все равно может быть отклонен, если запрашиваемые ресурсы недоступны, поэтому мы видим, что вопросы политики и доступности ресурсов могут рассматриваться вместе при принятии решений о контроле допуска. Применение политики к сетям — это область, которая привлекает много внимания на момент написания этой книги.

Протокол резервирования

Хотя сети с коммутацией каналов всегда нуждались в каком-то установочном протоколе для создания необходимого состояния виртуальной цепи в коммутаторах, сети без коммутации каналов, такие как Интернет, не имели таких протоколов. Однако, как указывалось в этой главе, нам нужно предоставить гораздо больше информации нашей сети, когда мы хотим получить от нее услуги в реальном времени. Хотя было предложено несколько установочных протоколов для Интернета, наибольшее внимание уделяется RSVP (Resource ReSerVation Protocol). Этот протокол особенно интересен, поскольку он существенно отличается от традиционных сигнальных протоколов для сетей с коммутацией каналов.

Одно из ключевых предположений, лежащих в основе RSVP, заключается в том, что он не должен снижать надежность, которую мы наблюдаем в сегодняшних сетях без коммутации каналов. Поскольку сети без коммутации каналов полагаются на то, что в самой сети хранится мало или вообще не хранится состояние, возможны ситуации, когда маршрутизаторы могут падать и перезагружаться, а каналы — работать с перебоями, но при этом сквозная связь сохраняется. RSVP пытается сохранить эту надежность, используя идею *мягкого состояния* в маршрутизаторах. В отличие от жесткого состояния, найденного в сетях с коммутацией каналов, мягкое состояние не нужно явно удалять, когда оно больше не требуется. Вместо этого оно истекает через некоторое довольно короткое время (скажем, через минуту), если оно не обновляется периодически. Позже мы увидим, как это помогает в поддержании надежности.

Еще одной важной характеристикой RSVP является то, что он стремится поддерживать мультикастовые (многопоточные) потоки так же эффективно, как и уникастовые

(однопоточные). Это неудивительно, поскольку многие из первых приложений, которые могли бы выиграть от улучшенного качества обслуживания, также были мультикастовыми приложениями — например, инструменты для видеоконференций. Одной из идей разработчиков RSVP является то, что большинство мультикастовых приложений имеют гораздо больше приемников, чем отправителей, как это характерно для большой аудитории и одного лектора. Кроме того, у приемников могут быть разные требования. Например, один приемник может хотеть получать данные только от одного отправителя, в то время как другие могут хотеть получать данные от всех отправителей. Вместо того чтобы заставлять отправителей отслеживать потенциально большое количество приемников, логичнее позволить приемникам отслеживать свои собственные потребности. Это предполагает подход, *ориентированный на приемника*, который был принят в RSVP. В отличие от этого, в сетях с коммутацией каналов обычно резервирование ресурсов оставляют отправителю, так же как инициатор телефонного звонка вызывает выделение ресурсов в телефонной сети.

Мягкое состояние и ориентированный на приемника характер RSVP придают ему ряд хороших свойств. Одним из таких свойств является то, что очень просто увеличивать или уменьшать уровень выделения ресурсов для приемника. Поскольку каждый приемник периодически отправляет сообщения-обновления, чтобы поддерживать мягкое состояние, легко отправить новое резервирование, которое запрашивает новый уровень ресурсов. Более того, мягкое состояние гибко справляется с возможностью отказов сети или узлов. В случае сбоя хоста ресурсы, выделенные этим хостом потоку, естественным образом истекают и освобождаются. Чтобы увидеть, что происходит в случае сбоя маршрутизатора или канала, нам нужно более внимательно рассмотреть механику создания резервирования.

Изначально рассмотрим случай одного отправителя и одного приемника, пытающихся получить резервирование для трафика, текущего между ними. Прежде чем приемник сможет сделать резервирование, необходимо выполнить два условия. Во-первых, приемник должен знать, какой трафик, вероятно, будет отправлять отправитель, чтобы сделать соответствующее резервирование. То есть ему нужно знать TSPEC отправителя. Во-вторых, он должен знать, какой путь будут проходить пакеты от отправителя к приемнику, чтобы установить резервирование ресурсов на каждом маршрутизаторе на этом пути. Оба этих требования можно выполнить, отправив сообщение от отправителя к приемнику, содержащее TSPEC. Очевидно, это доставит TSPEC к приемнику. Другая вещь, которая происходит, заключается в том, что каждый маршрутизатор рассматривает это сообщение (называемое PATH-сообщением) по мере его прохождения, и он выясняет *обратный путь*, который будет использоваться для отправки резервирований от приемника обратно к отправителю, чтобы обеспечить резервирование на каждом маршрутизаторе на пути. Создание мультикастового дерева изначально выполняется с помощью механизмов, которые описаны в другом разделе.

Получив PATH-сообщение, приемник отправляет резервирование обратно вверх по мультикастовому дереву в RESV-сообщении. Это сообщение содержит TSPEC отправителя и RSPEC, описывающий требования этого приемника. Каждый маршрутизатор на пути рассматривает запрос на резервирование и пытается выделить необходимые ресурсы для его удовлетворения. Если резервирование может быть выполнено, запрос RESV передается следующему маршрутизатору. Если нет, возвращается сообщение об ошибке приемнику, сделавшему запрос. Если все идет хорошо, правильное резервирование устанавливается на каждом маршрутизаторе между отправителем и приемником. Пока приемник хочет сохранить резервирование, он отправляет то же самое RESV-сообщение примерно раз в 30 секунд.

Теперь мы можем увидеть, что происходит при отказе маршрутизатора или канала. Протоколы маршрутизации адаптируются к сбою и создают новый путь от отправителя к получателю. PATH-сообщения отправляются примерно каждые 30 секунд, и могут быть отправлены раньше, если маршрутизатор обнаружит изменение в своей таблице марш-

рутизации, поэтому первое сообщение после стабилизации нового маршрута достигнет получателя по новому пути. Следующее RESV-сообщение от получателя последует по новому пути и, если все пройдет хорошо, установит новое резервирование на новом пути. Между тем маршрутизаторы, которые больше не находятся на пути, перестанут получать RESV-сообщения, и эти резервирования истекнут и будут освобождены. Таким образом, RSVP хорошо справляется с изменениями топологии, если изменения маршрутизации не происходят слишком часто.

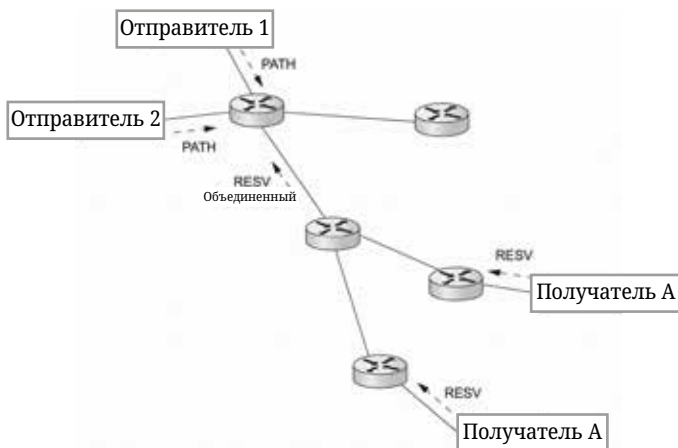


Рисунок 6.26. Резервирование в мультикастовом (многоадресном) дереве.

Следующее, что нужно рассмотреть, — это как справляться с мультикастом, где может быть несколько отправителей в группе и несколько получателей. Эта ситуация иллюстрируется на рис. 6.26. Во-первых, давайте разберемся с несколькими получателями для одного отправителя. Когда RESV-сообщение поднимается по мультикастовому дереву, оно, вероятно, встретит участок дерева, где уже установлено резервирование для другого получателя. Возможно, что ресурсы, зарезервированные выше этой точки, достаточны для обслуживания обоих получателей. Например, если получатель А уже сделал резервирование, обеспечивающее задержку менее 100 мс, а новый запрос от получателя В требует задержки менее 200 мс, то новое резервирование не требуется. С другой стороны, если новый запрос требует задержки менее 50 мс, то маршрутизатор сначала должен проверить, может ли он принять запрос; если да, то он отправит запрос выше по цепочке. В следующий раз, когда получатель А запросит минимальную задержку в 100 мс, маршрутизатору не потребуется передавать этот запрос дальше. В общем, резервирования могут быть объединены таким образом, чтобы удовлетворить потребности всех получателей ниже точки объединения.

Если в дереве также есть несколько отправителей, получатели должны собрать TSPEC всех отправителей и сделать резервирование, достаточное для учета трафика от всех отправителей. Однако это не обязательно означает, что TSPEC нужно складывать. Например, в аудиоконференции с 10 участниками нет смысла выделять ресурсы для передачи 10 аудиопотоков, поскольку результат одновременного разговора 10 человек будет непонятным. Таким образом, можно представить резервирование, достаточное для учета двух участников и не более. Расчет правильного общего TSPEC от всех TSPEC отправителей явно зависит от конкретного приложения. Кроме того, нас может интересовать только получение данных от подмножества всех возможных отправителей; RSVP имеет различные стили резервирования для обработки таких опций, как «Резервировать ресурсы для всех отправителей», «Резервировать ресурсы для любых n отправителей» и «Резервировать ресурсы только для отправителей А и В».

Классификация и планирование пакетов

После того как мы описали наш трафик и желаемую сетевую услугу и установили подходящее резервирование на всех маршрутизаторах на пути, остается только одно — чтобы маршрутизаторы действительно предоставили запрашиваемую услугу пакетам данных. Для этого нужно сделать две вещи:

- Ассоциировать каждый пакет с соответствующим резервированием, чтобы он мог быть обработан правильно, этот процесс называется *классификацией* пакетов.
- Управлять пакетами в очередях так, чтобы они получали запрашиваемую услугу, этот процесс называется *планированием* пакетов.

Первая часть выполняется путем анализа до пяти полей в пакете: адрес отправителя, адрес получателя, номер протокола, порт отправителя и порт получателя. (В IPv6 возможно использование поля FlowLabel в заголовке, чтобы выполнить поиск на основе одного, более короткого ключа.) На основе этой информации пакет может быть отнесен к соответствующему классу. Например, он может быть классифицирован в классы с контролируемой нагрузкой или быть частью гарантированного потока, который должен обрабатываться отдельно от всех других гарантированных потоков. Если вкратце, то существует сопоставление между информацией, специфичной для потока в заголовке пакета, и идентификатором класса, который определяет, как пакет обрабатывается в очереди. Для гарантированных потоков это может быть сопоставление «один к одному», в то время как для других сервисов это может быть сопоставление «многие к одному». Детали классификации тесно связаны с деталями управления очередью.

Очевидно, что такая простая вещь, как очередь FIFO в маршрутизаторе, будет недостаточной для предоставления множества различных услуг и уровней задержки внутри каждой услуги. В предыдущей главе были обсуждены несколько более сложных дисциплин управления очередями, и вероятно, что в маршрутизаторе будет использоваться комбинация этих методов.

Детали планирования пакетов идеальным образом не должны быть указаны в модели сервиса. Это область, где разработчики могут применять творческие подходы для эффективной реализации модели сервиса. В случае гарантированного сервиса было установлено, что дисциплина *взвешенного справедливого планирования* (weighted fair queuing), при которой каждый поток получает свою собственную очередь с определенной долей пропускной способности, обеспечит гарантированную конечную задержку, которую можно легко рассчитать. Для контролируемой нагрузки могут использоваться более простые схемы. Одной из возможностей является рассмотрение всего трафика с контролируемой нагрузкой как единого агрегированного потока (с точки зрения механизма планирования), с весом для этого потока, установленным на основе общего объема трафика, допущенного в класс контролируемой нагрузки. Проблема усложняется, если учесть, что в одном маршрутизаторе вероятно предоставление множества различных услуг одновременно, и каждая из этих услуг может требовать разного алгоритма планирования. Таким образом, необходим общий алгоритм управления очередями для управления ресурсами между различными услугами.

Проблемы масштабируемости

Хотя архитектура интегрированных услуг и RSVP представляли собой значительное улучшение модели обслуживания IP по принципу «лучшее усилие», многие интернет-провайдеры считали, что это не та модель, которую они должны внедрять. Причина этого нежелания связана с одной из фундаментальных целей проектирования IP: масштабируемостью. В модели обслуживания по принципу «лучшее усилие» маршрутизаторы в Интернете хранят мало или вовсе не хранят информацию о каждом потоке, проходящем через них. Таким образом, по мере роста Интернета единственное, что маршрутизаторам нужно делать для того, чтобы справляться с этим ростом, это перемещать больше бит

в секунду и обрабатывать большие таблицы маршрутизации. Однако RSVP предполагает, что каждый поток, проходящий через маршрутизатор, может иметь соответствующее резервирование. Чтобы понять серьезность этой проблемы, представим, что каждый поток на канале OC-48 (2,5 Гбит/с) представляет собой аудиопоток с пропускной способностью 64 Кбит/с. Количество таких потоков составит:

$$2,5 \times \frac{10^9}{64} \times 10^3 = 39\,000$$

Каждое из этих резервирований требует некоторого объема памяти для хранения и периодического обновления. Маршрутизатору необходимо классифицировать, контролировать и ставить в очередь каждый из этих потоков. Решения по управлению доступом должны приниматься каждый раз, когда такой поток запрашивает резервирование, и необходим механизм для «откатов» пользователям (например, списание средств с их кредитных карт), чтобы они не делали произвольно больших резервирований на длительные периоды времени.

Эти проблемы масштабируемости ограничили широкое распространение IntServ. Из-за этих проблем были разработаны другие подходы, которые не требуют такого большого количества состояния «на поток». В следующей главе обсуждается ряд таких подходов.

Глава 6.5.3. Дифференцированные услуги (EF, AF)

В то время как архитектура интегрированных сервисов выделяет ресурсы для отдельных потоков, модель дифференцированных сервисов (часто называемая DiffServ) выделяет ресурсы для небольшого числа классов трафика. На самом деле некоторые предложенные подходы к DiffServ просто делят трафик на два класса. Это весьма разумный подход: если учесть, какие трудности испытывают операторы сетей, пытаясь поддерживать работу интернета на уровне «лучшее усилие», то добавление новой модели обслуживания небольшими шагами кажется логичным.

Предположим, мы решили улучшить модель обслуживания «лучшее усилие», добавив всего один новый класс, который мы назовем «премиум». Очевидно, нам потребуется способ различения премиум-пакетов и обычных пакетов с обслуживанием «лучшее усилие». Вместо использования протокола, такого как RSVP, для информирования всех маршрутизаторов о том, что некоторый поток отправляет премиум-пакеты, было бы гораздо проще, если бы пакеты могли сами идентифицировать себя маршрутизатору при прибытии. Это можно сделать, используя бит в заголовке пакета: если этот бит равен 1, пакет является премиум-пакетом; если он равен 0, пакет обслуживается по принципу «лучшее усилие». В связи с этим возникают два вопроса, на которые нужно ответить:

- Кто устанавливает премиум-бит и при каких обстоятельствах?
- Что маршрутизатор делает по-другому, когда видит пакет с установленным битом?

Существует множество возможных ответов на первый вопрос, но общий подход заключается в установке бита на административной границе. Например, маршрутизатор на границе сети интернет-провайдера может устанавливать бит для пакетов, поступающих на интерфейс, соединенный с сетью конкретной компании. Интернет-провайдер может делать это, потому что эта компания оплатила более высокий уровень обслуживания, чем «лучшее усилие». Возможно также, что не все пакеты будут отмечены как премиум; например, маршрутизатор может быть настроен на пометку пакетов как премиум до определенной максимальной скорости, а все избыточные пакеты оставлять как «лучшее усилие».

Предположим, что пакеты были помечены каким-то образом. Что делают маршрутизаторы, которые сталкиваются с помеченными пакетами? Здесь также есть множество ответов. На самом деле IETF стандартизировал набор действий маршрутизатора, которые применяются к помеченным пакетам. Эти действия называются *поведением*

на каждый переход (per-hop behaviors, PHBs), и это означает, что они определяют поведение отдельных маршрутизаторов, а не сквозные сервисы. Поскольку существует более одного нового поведения, требуется более одного бита в заголовке пакета, чтобы указать маршрутизаторам, какое поведение применять. IETF решил использовать старый байт TOS из заголовка IP, который не был широко использован, и переопределить его. Шесть бит этого байта были выделены для кодовых точек дифференцированных сервисов (DiffServ code points, DSCPs), где каждый DSCP — это 6-битное значение, идентифицирующее конкретное PHB, которое должно быть применено к пакету. (Оставшиеся два бита используются для ECN.)

Поведение с ускоренной передачей (EF)

Одним из самых простых для объяснения PHB является *поведение с ускоренной передачей* (expedited forwarding, EF). Пакеты, помеченные для обработки по EF, должны передаваться маршрутизатором с минимальной задержкой и потерями. Единственный способ, которым маршрутизатор может гарантировать это для всех EF пакетов, заключается в том, чтобы скорость поступления EF пакетов на маршрутизатор была строго ограничена и была ниже скорости, с которой маршрутизатор может передавать EF пакеты. Например, маршрутизатор с интерфейсом на 100 Мбит/с должен быть уверен, что скорость поступления EF пакетов, предназначенных для этого интерфейса, никогда не превышает 100 Мбит/с. Он также может хотеть убедиться, что эта скорость будет несколько ниже 100 Мбит/с, чтобы у него иногда было время для отправки других пакетов, таких как обновления маршрутизации.

Ограничение скорости EF пакетов достигается путем настройки маршрутизаторов на границе административного домена, чтобы позволить определенную максимальную скорость поступления EF пакетов в домен. Простой, хотя и консервативный подход заключается в обеспечении того, чтобы сумма скоростей всех EF пакетов, входящих в домен, была меньше пропускной способности самого медленного канала в домене. Это гарантирует, что даже в худшем случае, когда все EF пакеты сходятся на самом медленном канале, он не будет перегружен и сможет обеспечить правильное поведение.

Существует несколько возможных стратегий реализации поведения EF. Одна из них заключается в предоставлении EF пакетам строгого приоритета перед всеми другими пакетами. Другая — выполнение взвешенной справедливой очереди между EF пакетами и другими пакетами, при этом вес EF установлен достаточно высоко, чтобы все EF пакеты могли быть доставлены быстро. Это имеет преимущество перед строгим приоритетом: не-EF пакеты могут быть уверены в получении доступа к каналу, даже если объем EF трафика чрезмерен. Это может означать, что EF пакеты не смогут получить точно определенное поведение, но это также может предотвратить блокировку важного маршрутизирующего трафика из сети в случае чрезмерной нагрузки EF трафиком.

Поведение с гарантированной передачей (AF)

Поведение с *гарантированной передачей* (Assured Forwarding, AF) берет начало из подхода, известного как RED с пометкой «внутри» и «снаружи» (RED with In and Out, RIO) или *взвешенный RED*, оба из которых являются улучшениями базового алгоритма RED, описанного ранее. На рис. 6.27 показано, как работает RIO; как и в случае RED, вероятность отброса на оси y увеличивается по мере увеличения средней длины очереди по оси x . Но теперь для наших двух классов трафика у нас есть две отдельные кривые вероятности отброса. RIO называет два класса «внутри» и «снаружи» по причинам, которые вскоре станут ясны. Поскольку кривая «снаружи» имеет более низкий MinThreshold, чем кривая «внутри», ясно, что при низких уровнях перегрузки только пакеты, помеченные как «снаружи», будут отброшены алгоритмом RED. Если перегрузка станет более серьезной, больший процент пакетов «снаружи» будет отброшен, и если средняя длина очереди превысит MaxThreshold, RED начнет отбрасывать пакеты «внутри».

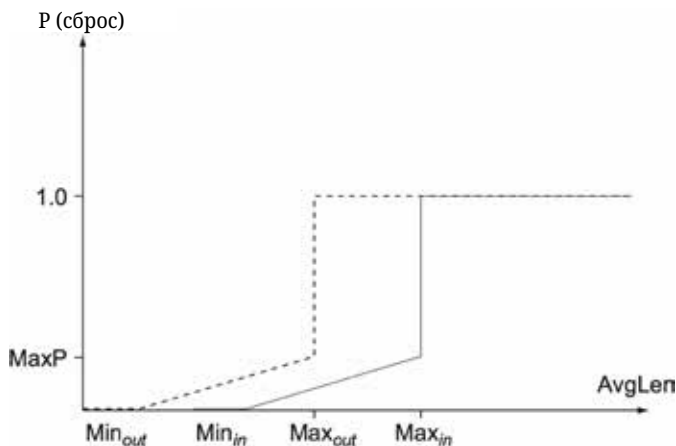


Рисунок 6.27. RED с вероятностями падения на входе и выходе.

Причина, по которой два класса пакетов называются «внутри» и «снаружи», связана с тем, как помечаются пакеты. Мы уже отмечали, что пометка пакетов может выполняться маршрутизатором на границе административного домена. Мы можем представить себе этот маршрутизатор на границе между сетевым провайдером и каким-либо клиентом этой сети. Клиент может быть любой другой сетью — например, сетью корпорации или другого сетевого провайдера. Клиент и сетевой провайдер договариваются о каком-либо профиле для гарантированного обслуживания (и, возможно, клиент платит провайдеру за этот профиль). Профиль может быть чем-то вроде «Клиент X может отправлять до Y Мбит/с гарантированного трафика» или может быть значительно сложнее. Независимо от профиля, пограничный маршрутизатор может четко пометать пакеты, поступающие от этого клиента, как находящиеся «внутри» или «снаружи» профиля. В упомянутом примере, пока клиент отправляет меньше Y Мбит/с, все его пакеты будут помечены как «внутри», но как только он превысит эту скорость, избыточные пакеты будут помечены как «снаружи».

Сочетание измерителя профиля на границе и RIO во всех маршрутизаторах сети провайдера должно обеспечить клиенту высокую уверенность (но не гарантию), что пакеты в пределах его профиля могут быть доставлены. В частности, если большинство пакетов, включая те, которые отправляются клиентами, не заплатившими дополнительно за установление профиля, являются пакетами «снаружи», то обычно механизм RIO будет действовать так, чтобы поддерживать перегрузку на таком уровне, при котором пакеты «внутри» редко будут отброшены. Очевидно, что в сети должно быть достаточно пропускной способности, чтобы пакеты «внутри» сами по себе редко могли вызвать перегрузку канала до такой степени, чтобы RIO начал отбрасывать пакеты «внутри».

Как и RED, эффективность механизма RIO зависит в некоторой степени от правильного выбора параметров, и для RIO существует значительно больше параметров для настройки. Точно неизвестно, насколько хорошо эта схема будет работать в реальных сетях.

Одним из интересных свойств RIO является то, что он не изменяет порядок пакетов «внутри» и «снаружи». Например, если TCP-соединение отправляет пакеты через профильный измеритель, и некоторые пакеты помечены как «внутри», а другие как «снаружи», эти пакеты получают разные вероятности отбрасывания в очередях маршрутизатора, но они будут доставлены получателю в том порядке, в котором они были отправлены. Это важно для большинства реализаций TCP, которые работают намного лучше, когда пакеты приходят по порядку, даже если они разработаны для работы с непоследовательной доставкой. Также стоит отметить, что механизмы, такие как быстрая повторная передача, могут ошибочно срабатывать при изменении порядка.

Идея RIO может быть обобщена для предоставления более чем двух кривых вероятности отброса, и это идея, лежащая в основе подхода, известного как взвешенный RED (WRED). В этом случае значение поля DSCP используется для выбора одной из нескольких кривых вероятности отброса, чтобы можно было предоставить несколько различных классов обслуживания.

Третий способ предоставления дифференцированных сервисов заключается в использовании значения DSCP для определения, в какую очередь поместить пакет в диспетчере взвешенной справедливой очереди. В самом простом случае мы можем использовать один код DSCP для обозначения очереди *наилучшего усилия* и второй код для выбора *премиальной* очереди. Затем нам нужно выбрать вес для премиальной очереди, чтобы премиальные пакеты получали лучшее обслуживание, чем пакеты наилучшего усилия. Это зависит от предложенной нагрузки премиальных пакетов. Например, если мы дадим премиальной очереди вес 1, а очереди наилучшего усилия вес 4, это обеспечит доступную пропускную способность для премиальных пакетов:

$$B_{\text{premium}} = W_{\text{premium}} + W_{\text{best-effort}} = \frac{1}{(1 + 4)} = 0.2$$

То есть мы фактически зарезервировали 20% канала для премиальных пакетов, так что если предложенная нагрузка премиального трафика в среднем составляет всего 10% от пропускной способности канала, тогда премиальный трафик будет работать так, как будто он находится в сильно недогруженной сети, и обслуживание будет очень хорошим. В частности, задержка, испытываемая премиальным классом, может быть низкой, поскольку WFQ будет пытаться передавать премиальные пакеты, как только они придут в этой ситуации. С другой стороны, если нагрузка премиального трафика составит 30%, это будет похоже на сильно загруженную сеть, и задержка для премиальных пакетов может быть очень высокой — даже хуже, чем для так называемых пакетов наилучшего усилия. Таким образом, знание предложенной нагрузки и внимательная настройка весов важны для этого типа обслуживания. Однако стоит отметить, что безопасный подход заключается в том, чтобы быть очень консервативным при настройке веса для премиальной очереди. Если этот вес установлен очень высоким относительно ожидаемой нагрузки, это обеспечивает запас прочности и не мешает трафику наилучшего усилия использовать любую пропускную способность, зарезервированную для премиальных пакетов, но не используемую ими.

Точно так же, как в WRED, мы можем обобщить этот подход, основанный на WFQ, чтобы позволить более чем двум классам предоставляться различными кодами. Более того, мы можем объединить идею выбора очереди с предпочтением отброса. Например, с 12 кодами мы можем иметь четыре очереди с разными весами, каждая из которых имеет три предпочтения отброса. Именно это и сделала IETF в определении «гарантированного сервиса».

Глава 6.5.4. Управление перегрузками на основе уравнений

Мы завершаем наше обсуждение QoS, возвращаясь к управлению перегрузками TCP, но на этот раз в контексте приложений реального времени. Напомним, что TCP регулирует окно перегрузки отправителя (и, следовательно, скорость, с которой он может передавать данные) в ответ на события ACK и тайм-ауты. Одним из достоинств этого подхода является то, что он не требует сотрудничества со стороны маршрутизаторов сети; это чисто хостовая стратегия. Такой подход дополняет механизмы QoS, которые мы рассматриваем, потому что (1) приложения могут использовать хостовые решения, не зависящие от поддержки маршрутизаторов, и (2) даже при полной развертке DiffServ все равно возможно, что очередь маршрутизатора будет переполнена, и мы хотели бы, чтобы приложения реального времени реагировали на это разумным образом.

Хотя мы хотели бы воспользоваться алгоритмом избежания перегрузок в TCP, сам TCP не подходит для приложений реального времени. Одна из причин заключается в том, что TCP — это надежный протокол, а приложения реального времени часто не могут позволить себе задержки, вызванные повторной передачей. Однако что если бы мы могли отделить TCP от его механизма избежания перегрузок, чтобы добавить TCP-подобное управление перегрузками к ненадежному протоколу, такому как UDP? Смогли бы приложения реального времени использовать такой протокол?

С одной стороны, это привлекательная идея, потому что она заставила бы потоки реального времени честно конкурировать с потоками TCP. Альтернатива (которая существует сегодня) заключается в том, что видеоприложения используют UDP без какой-либо формы избежания перегрузок и, как следствие, отбирают полосу пропускания у потоков TCP, которые уменьшают скорость передачи данных в условиях перегрузки. С другой стороны, поведение алгоритма избежания перегрузок TCP в виде зубчатой пилы не подходит для приложений реального времени; это означает, что скорость передачи данных приложением постоянно увеличивается и уменьшается. В отличие от этого, приложения реального времени лучше всего работают, когда они могут поддерживать плавную скорость передачи данных в течение относительно долгого периода времени.

Возможно ли добиться лучшего из обоих миров: совместимости с управлением перегрузками TCP ради справедливости, одновременно поддерживая плавную скорость передачи данных ради приложений? Недавние исследования показывают, что ответ — да. В частности, было предложено несколько так называемых TCP-дружественных алгоритмов управления перегрузками. Эти алгоритмы имеют две основные цели. Одна из них — медленно адаптировать окно перегрузок. Это достигается за счет адаптации в течение относительно более длительных периодов времени (например, RTT), а не на основе каждого пакета. Это сглаживает скорость передачи данных. Вторая цель — быть дружественным к TCP в смысле справедливости по отношению к конкурирующим потокам TCP. Это свойство часто обеспечивается за счет того, что поведение потока соответствует уравнению, моделирующему поведение TCP. Таким образом, этот подход иногда называют *управлением перегрузками на основе уравнений*.

$$\text{Rate} \propto \left(\frac{1}{RTT \times \sqrt{p}} \right)$$

Мы видели упрощенную форму уравнения скорости TCP в предыдущей главе. Для наших целей достаточно отметить, что уравнение имеет следующий общий вид:

что означает, что для того чтобы быть дружественным к TCP, скорость передачи данных должна быть обратно пропорциональна времени кругового путешествия (RTT) и квадратному корню из коэффициента потерь (p). Другими словами, чтобы построить механизм управления перегрузками на основе этого соотношения, получатель должен периодически сообщать отправителю о проценте потерь, которые он испытывает (например, он может сообщить, что не получил 10% из последних 100 пакетов), а отправитель затем корректирует свою скорость передачи вверх или вниз, так чтобы это соотношение продолжало соблюдаться. Конечно, адаптация к этим изменениям доступной скорости остается задачей приложения, но, как мы увидим в следующем разделе, многие приложения реального времени довольно адаптивны.

Перспектива: программно-определяемая инженерия трафика

Основная проблема, рассматриваемая в этом разделе, заключается в том, как распределить доступную полосу пропускания сети между набором сквозных потоков. Независимо от того, идет ли речь об управлении перегрузками TCP, интегрированных услугах или дифференцированных услугах, существует предположение, что основная полоса про-

пускания сети, подлежащая распределению, является фиксированной: связь с пропускной способностью 1 Гбит/с между узлом А и узлом В всегда остается связью с пропускной способностью 1 Гбит/с, и алгоритмы сосредоточены на том, как лучше всего разделить эту полосу пропускания между конкурирующими пользователями. Но что если это не так? Что если бы вы могли «мгновенно» получить дополнительную емкость, так что связь с пропускной способностью 1 Гбит/с была бы обновлена до связи с пропускной способностью 10 Гбит/с, или, возможно, вы могли бы добавить новую связь между двумя узлами, которые ранее не были связаны?

Эта возможность реальна, и это тема, которая обычно называется *инженерией трафика* — термин, который восходит к ранним дням сетей, когда операторы анализировали рабочие нагрузки трафика в своих сетях и периодически переоснащали свои сети, добавляя емкость, когда существующие связи становились хронически перегруженными. В те ранние дни решение о добавлении емкости не принималось легкомысленно; нужно было быть уверенным, что наблюдаемый вами тренд использования не является просто временным всплеском, так как изменение сети требовало значительного количества времени и денег. В худшем случае это могло включать прокладку кабеля через океан или запуск спутника в космос.

Но с появлением таких технологий, как DWDM (глава 3.1) и MPLS (глава 4.4), нам не всегда нужно прокладывать больше оптоволокон, вместо этого можно включить дополнительные длины волн или установить новые каналы между любой парой узлов. (Эти узлы не обязательно должны быть напрямую соединены оптоволоконным. Например, длина волны между Бостоном и Сан-Франциско может проходить через ROADMs в Чикаго и Денвере, но с точки зрения топологии сети L2/L3 Бостон и Сан-Франциско соединены прямой связью.) Это значительно снижает время доступности, но перенастройка оборудования все еще требует ручного вмешательства, и поэтому наше определение «мгновенно» все еще измеряется днями, если не неделями. В конце концов, нужно заполнить формы на закупку в трех экземплярах!

Но, как мы видели снова и снова, как только вы предоставляете правильные программные интерфейсы, программное обеспечение может быть привлечено к решению проблемы, и «мгновенно» может, для всех практических целей, стать действительно мгновенным. Это фактически то, что делают облачные провайдеры с частными магистральями, которые они строят для соединения своих центров обработки данных. Например, Google публично описал свою частную глобальную сеть, называемую B4, которая полностью построена с использованием коммутаторов «белой коробки» и SDN. B4 не добавляет/удаляет длины волн для настройки межузловой пропускной способности — она динамически строит сквозные туннели, используя технику, называемую *Equal-Cost Multipath* (ECMP), альтернативу CSPF, введенной в главе 4.4 — но гибкость, которую она обеспечивает, аналогична.

Программа управления инженерией трафика (ТЕ) затем конфигурирует сеть в соответствии с потребностями различных классов приложений. B4 определяет три таких класса: (1) копирование пользовательских данных (например, электронной почты, документов, аудио/видео) в удаленные центры обработки данных для обеспечения доступности; (2) доступ к удаленному хранилищу вычислительными процессами, работающими с распределенными источниками данных; и (3) передача данных большого объема для синхронизации состояния между несколькими центрами обработки данных. Эти классы расположены в порядке увеличения объема, снижения чувствительности к задержке и снижения общего приоритета. Например, пользовательские данные представляют собой самый низкий объем на B4, являются наиболее чувствительными к задержке и имеют самый высокий приоритет.

Централизовав процесс принятия решений, что является одним из заявленных преимуществ SDN, Google смог достичь почти 100% загрузки своих каналов. Это в два-три раза лучше, чем 30–40% средней загрузки каналов глобальных сетей, которые обычно обеспечиваются для того, чтобы эти сети могли справляться как с всплесками трафика, так

и с отказами каналов/коммутаторов. Если вы можете централизованно решать, как распределить ресурсы по всей сети, то можно управлять сетью гораздо ближе к максимальной загрузке. Учтите, что конфигурирование каналов в сети осуществляется для классов приложений с грубой градацией. Управление перегрузками ТСП все еще работает на основе каждого соединения, и маршрутизация по-прежнему осуществляется на основе топологии В4. (Кстати, стоит отметить, что поскольку В4 является частной глобальной сетью, Google может использовать свой собственный алгоритм управления перегрузками, такой как BBR, не опасаясь, что он несправедливо поставит в невыгодное положение другие алгоритмы.)

Один из уроков, который можно извлечь из систем, подобных В4, заключается в том, что граница между инженерией трафика и управлением перегрузками (а также между инженерией трафика и маршрутизацией) размыта. Существует множество различных механизмов, работающих над решением одной и той же общей проблемы, и поэтому нет жесткой границы, которая определяла бы, где заканчивается один механизм и начинается другой. Обобщая вышеизложенное, отметим, что границы уровней становятся мягкими (и легко перемещаемыми), когда уровни реализованы в программном обеспечении, а не в оборудовании. Это все чаще становится нормой.

Раздел 7.

Сквозные данные

Теоретизировать, не имея данных, — большая ошибка.

Сэр Артур Конан Дойл

Проблема: что делать с данными?

С точки зрения сети программы приложений отправляют сообщения друг другу. Каждое из этих сообщений является просто непроинтерпретированной строкой байтов. Однако с точки зрения приложения эти сообщения содержат различные виды *данных* — массивы целых чисел, видеокadres, строки текста, цифровые изображения и так далее. Другими словами, эти байты имеют значение. Теперь мы рассмотрим проблему того, как лучше всего кодировать различные виды данных, которые программы приложений хотят обменивать, в строки байтов. Во многих отношениях это похоже на проблему кодирования строк байтов в электромагнитные сигналы, которую мы рассматривали в предыдущем разделе.

Возвращаясь к нашему обсуждению кодирования, отметим, что существует, по сути, две задачи. Первая — чтобы приемник мог извлечь из сигнала то же сообщение, которое отправил передатчик; это проблема фрейминга (или кадрирования). Вторая — сделать кодирование как можно более эффективным. Оба этих аспекта также присутствуют при кодировании данных приложений в сетевые сообщения.

Для того чтобы приемник мог извлечь сообщение, отправленное передатчиком, обе стороны должны договориться о формате сообщения, который часто называют *форматом представления*. Если отправитель хочет отправить приемнику массив целых чисел, например, то обе стороны должны договориться, как выглядит каждое целое число (сколько бит оно занимает, в каком порядке байты расположены и идет ли самый значимый бит первым или последним, например) и сколько элементов в массиве. Первый раздел описывает различные кодировки традиционных компьютерных данных, таких как целые числа, числа с плавающей запятой, строковые данные, массивы и структуры. Хорошо установленные форматы также существуют для мультимедийных данных: видео, например, обычно передается в одном из форматов, созданных Moving Picture Experts Group (MPEG), а неподвижные изображения обычно передаются в формате Joint Photographic Experts Group (JPEG). Особые проблемы, возникающие при кодировании мультимедийных данных, обсуждаются в следующей главе.

Типы мультимедийных данных требуют от нас думать как о представлении, так и о *сжатии*. Хорошо известные форматы для передачи и хранения аудио и видео решают обе эти задачи: они обеспечивают, чтобы что-то, что было записано, сфотографировано или услышано на стороне отправителя, могло быть правильно интерпретировано приемником, и делают это таким образом, чтобы не перегружать сеть огромным количеством мультимедийных данных.

Сжатие и, в более общем смысле, эффективность кодирования имеют богатую историю, восходящую к новаторской работе Шеннона по теории информации в 1940-х годах. Фактически здесь действуют две противоположные силы. В одном направлении вы хотите иметь как можно больше избыточности в данных, чтобы приемник мог извлечь правильные данные, даже если в сообщении были введены ошибки. Коды обнаружения и исправления ошибок, которые мы рассматривали в предыдущем разделе, добавляют избыточную информацию в сообщения именно с этой целью. В другом направлении, мы хотели бы удалить как можно больше избыточности из данных, чтобы закодировать их в как можно меньшее количество бит. Оказывается, мультимедийные данные предлагают множество возможностей для сжатия благодаря тому, как наши чувства и мозг обрабатывают визуальные и звуковые сигналы. Мы не слышим высокие частоты так же

хорошо, как низкие, и мы не замечаем мелкие детали так же, как общую картину в изображении, особенно если изображение движется.

Сжатие важно для разработчиков сетей по многим причинам, не только потому, что мы редко сталкиваемся с избытием полосы пропускания во всей сети. Например, то, как мы разрабатываем алгоритм сжатия, влияет на нашу чувствительность к потерянными или задержанным данным и, следовательно, может повлиять на разработку механизмов распределения ресурсов и сквозных протоколов. Напротив, если основная сеть не может гарантировать фиксированное количество полосы пропускания на время видеоконференции, мы можем выбрать разработку алгоритмов сжатия, которые могут адаптироваться к изменяющимся условиям сети.

Наконец, важным аспектом как форматирования представления, так и сжатия данных является то, что они требуют от отправляющего и принимающего хостов обработки каждого байта данных в сообщении. Именно по этой причине форматирование представления и сжатие иногда называют функциями *манипулирования данными*. Это контрастирует с большинством протоколов, которые мы рассмотрели до этого момента, которые обрабатывают сообщение, не заглядывая в его содержимое. Из-за необходимости читать, вычислять и записывать каждый байт данных в сообщении манипуляции с данными влияют на сквозную пропускную способность сети. В некоторых случаях эти манипуляции могут быть ограничивающим фактором.

Глава 7.1. Форматирование представления

Одной из самых распространенных трансформаций сетевых данных является преобразование представления, используемого программой приложения, в форму, пригодную для передачи по сети, и *наоборот*. Это преобразование обычно называют *форматированием представления*. Как показано на рис. 7.1, отправляющая программа переводит данные, которые она хочет передать, из представления, используемого внутри нее, в сообщение, которое может быть передано по сети; то есть данные *кодируются* в сообщении. На принимающей стороне приложение переводит это поступающее сообщение в представление, которое оно может затем обработать; то есть сообщение *декодируется*. Этот процесс иногда называют *маршалингом аргументов* или *сериализацией*. Эта терминология происходит из мира удаленного вызова процедур (Remote Procedure Call, RPC), где клиент считает, что он вызывает процедуру с набором аргументов, но эти аргументы затем «собираются вместе и упорядочиваются в соответствующий и эффективный способ» для формирования сетевого сообщения.

Вы можете спросить, что делает эту задачу сложной. Одна из причин заключается в том, что компьютеры представляют данные по-разному. Например, некоторые компьютеры представляют числа с плавающей запятой в формате IEEE стандарт 754, в то время как некоторые старые машины все еще используют свои собственные нестандартные форматы. Даже для таких простых вещей, как целые числа, разные архитектуры используют разные размеры (например, 16-битные, 32-битные, 64-битные). Более того, на некоторых машинах целые числа представлены в *big-endian* форме (самый значимый бит слова находится в байте с самым высоким адресом), тогда как на других машинах целые числа представлены в *little-endian* форме (самый значимый бит находится в байте с самым низким адресом). Например, процессоры PowerPC являются *big-endian* машинами, а семейство Intel x86 является архитектурой *little-endian*. Сегодня многие архитектуры поддерживают оба представления (и поэтому называются *bi-endian*), но суть в том, что вы никогда не можете быть уверены, как хост, с которым вы общаетесь, хранит целые числа. *Big-endian* и *little-endian* представления целого числа 34,677,374 показаны на рис. 7.2.

Еще одной причиной, по которой маршалинг является сложной задачей, стало то, что программы приложений написаны на разных языках, и даже если вы используете один язык, может быть более одного компилятора. Например, компиляторы имеют значительную свободу в том, как они располагают структуры (записи) в памяти, например,

сколько отступов они вставляют между полями, составляющими структуру. Таким образом, вы не могли бы просто передать структуру с одной машины на другую, даже если обе машины имеют одинаковую архитектуру и программа написана на одном и том же языке, потому что компилятор на целевой машине может выравнивать поля в структуре по-разному.

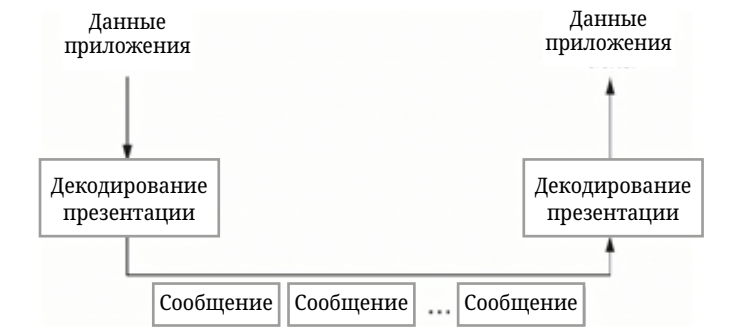


Рисунок 7.1. Форматирование презентации включает в себя кодирование и декодирование данных приложения.

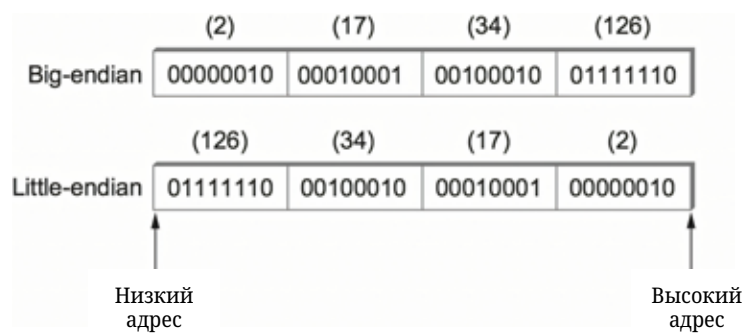


Рисунок 7.2. Big-endian и little-endian порядок байтов для целого числа 34 677 374.

Глава 7.1.1. Таксономия

Хотя маршalling аргументов не принадлежит к области ядерной физики (это небольшая задача по манипулированию битами), существует удивительное количество вариантов проектных решений, которые необходимо учитывать. Начнем с простой таксономии систем маршalling аргументов. Следующее, безусловно, не является единственной возможной таксономией, но этого достаточно для охвата большинства интересных альтернатив.

Типы данных

Первый вопрос — какие типы данных будет поддерживать система. В общем, мы можем разделить типы, поддерживаемые механизмом маршalling аргументов, на три уровня. Каждый уровень усложняет задачу, стоящую перед системой маршallingа.

На самом низком уровне система маршallingа работает с некоторым набором базовых типов. Обычно базовые типы включают целые числа, числа с плавающей запятой и символы. Система также может поддерживать порядковые типы и логические значения. Как описано выше, суть набора базовых типов заключается в том, что процесс кодирования должен уметь конвертировать каждый базовый тип из одного представления в другое — например, конвертировать целое число из big-endian в little-endian.

На следующем уровне находятся *плоские типы* — структуры и массивы. Хотя на первый взгляд плоские типы могут показаться не усложняющими маршалинг аргументов, на самом деле это не так. Проблема заключается в том, что компиляторы, используемые для компиляции программ приложений, иногда вставляют отступы между полями, составляющими структуру, чтобы выровнять эти поля по границам слов. Система маршалинга обычно *упаковывает* структуры так, чтобы в них не было отступов.

На самом высоком уровне система маршалинга может столкнуться со сложными типами — теми типами, которые построены с использованием указателей. То есть структура данных, которую одна программа хочет отправить другой, может не содержаться в одной структуре, а может включать указатели из одной структуры в другую. Дерево является хорошим примером сложного типа, включающего указатели. Очевидно, что кодировщик данных должен подготовить структуру данных для передачи по сети, потому что указатели реализуются в виде адресов памяти, и тот факт, что структура находится по определенному адресу памяти на одной машине, не означает, что она будет находиться по тому же адресу на другой машине. Другими словами, система маршалинга должна *сериализовать* (упрощать) сложные структуры данных.

В итоге, в зависимости от сложности системы типов, задача маршалинга аргументов обычно включает преобразование базовых типов, упаковку структур и линеаризацию сложных структур данных, чтобы сформировать непрерывное сообщение, которое можно передать по сети. Рис. 7.3 иллюстрирует эту задачу.

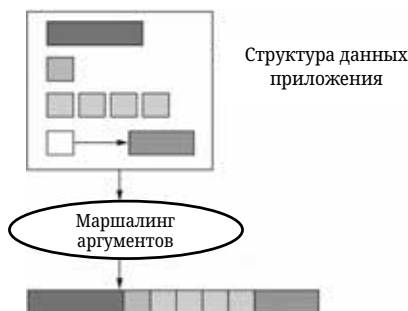


Рисунок 7.3. Маршалинг аргументов: преобразование, упаковка и линеаризация

Стратегия преобразования

Как только система типов установлена, следующая проблема заключается в выборе стратегии преобразования, которую будет использовать маршалер аргументов. Существует два основных варианта: *каноническая промежуточная форма* и подход «*получатель-приводит-в-порядок*». Рассмотрим каждый из них по очереди.

Идея канонической промежуточной формы заключается в том, чтобы установить внешнее представление для каждого типа; отправляющий хост переводит свои внутренние представления в это внешнее представление перед отправкой данных, а принимающий хост переводит это внешнее представление в свое локальное представление при получении данных. Для иллюстрации идеи рассмотрим целочисленные данные; другие типы обрабатываются аналогичным образом. Можно установить, что *big-endian* формат будет использоваться в качестве внешнего представления для целых чисел. Отправляющий хост должен переводить каждое отправляемое целое число в *big-endian* формат, а принимающий хост должен переводить *big-endian* целые числа в то представление, которое он использует. (Именно так делается в Интернете для заголовков протоколов.) Конечно, данный хост может уже использовать *big-endian* формат, в этом случае преобразование не требуется.

Альтернатива — подход «получатель-приводит-в-порядок» — заключается в том, что отправитель передает данные в своем внутреннем формате; отправитель не преобразует базовые типы, но обычно должен упаковывать и упрощать более сложные структуры данных. Затем получатель несет ответственность за преобразование данных из формата отправителя в свой собственный локальный формат. Проблема с этой стратегией заключается в том, что каждый хост должен быть готов преобразовывать данные из всех других архитектур машин. В сетевых технологиях это известно как *решение N-на-N*: каждая из N архитектур машин должна уметь обрабатывать все N архитектур. В отличие от этого, в системе, использующей каноническую промежуточную форму, каждому хосту нужно знать, как преобразовывать между своим собственным представлением и одним другим представлением — внешним.

Использование общего внешнего формата, очевидно, правильный подход, не так ли? Это было традиционной мудростью в сообществе сетевых технологий более 30 лет. Однако ответ не столь однозначен. Оказывается, существует не так много различных представлений для различных базовых классов, или, другими словами, N не так велико. Кроме того, наиболее распространенный случай — когда две машины одного типа общаются друг с другом. В этой ситуации кажется нелепым переводить данные из представления архитектуры одной машины в какой-то внешний формат, только чтобы потом переводить данные обратно в то же самое представление архитектуры на приемнике.

Существует третий вариант, хотя нам неизвестно о существовании системы, которая бы его использовала: применять стратегию «получатель-приводит-в-порядок», если отправитель знает, что у приемника такая же архитектура; отправитель использовал бы некоторую каноническую промежуточную форму, если бы машины применяли разные архитектуры. Как отправитель может узнать архитектуру приемника? Он может получить эту информацию либо от сервера имен, либо сначала использовать простой тест, чтобы убедиться, что он получит нужный ему результат.

Теги

Третий вопрос в маршалинге аргументов заключается в том, как приемник узнает, какие данные содержатся в полученном сообщении. Существует два распространенных подхода: данные с *тегами* и *без тегов*. Подход с тегами более интуитивен, поэтому сначала мы опишем его.

Тег (или метка) — это любая дополнительная информация, включенная в сообщение, помимо конкретного представления базовых типов, которая помогает приемнику декодировать сообщение. Существует несколько возможных тегов, которые могут быть включены в сообщение. Например, каждый элемент данных может быть дополнен *типовым* тегом. Типовой тег указывает, какое именно значение следует: целое число, число с плавающей запятой или что-то еще. Другим примером является тег *длины*. Такой тег используется для указания количества элементов в массиве или размера целого числа. Третьим примером является тег архитектуры, который может использоваться в сочетании со стратегией «получатель-приводит-в-порядок» для указания архитектуры, на которой были сгенерированы данные, содержащиеся в сообщении. Рис. 7.4а изображает, как простой 32-битный целочисленный тег может быть закодирован в сообщении с тегами.

type = INT	len = 4	value = 417892

Рисунок 7.4. 32-битное целое число, закодированное в тегированном сообщении.

Альтернатива, конечно, заключается в том, чтобы не использовать теги. Как же тогда приемник узнает, как декодировать данные? Он знает это, потому что был запрограммирован на это. Другими словами, если вы вызываете удаленную процедуру, которая при-

нимает два целых числа и число с плавающей точкой в качестве аргументов, то нет необходимости проверять теги, чтобы узнать, что было получено. Он просто предполагает, что сообщение содержит два целых числа и одно число с плавающей точкой и декодирует его соответствующим образом. Обратите внимание, что хотя это работает для большинства случаев, единственным исключением является передача массивов переменной длины. В таком случае обычно используется тег длины для указания длины массива.

Также стоит отметить, что подход без тегов означает, что форматирование представления действительно выполняется от конца до конца. Невозможно, чтобы какой-то промежуточный агент интерпретировал сообщение, если данные не содержат тегов. Почему промежуточный агент должен интерпретировать сообщение, можете спросить вы? Странные вещи иногда случались, в основном в результате *временных* решений для неожиданных проблем, с которыми система не была спроектирована для работы. Плохой дизайн сети выходит за рамки этой книги.

Заглушки

Заглушка — это часть кода, которая реализуетmarshalling аргументов. Заглушки обычно используются для поддержки RPC (удаленного вызова процедур). На стороне клиента заглушка преобразует аргументы процедуры в сообщение, которое может быть передано с помощью протокола RPC. На стороне сервера заглушка преобразует сообщение обратно в набор переменных, которые могут использоваться в качестве аргументов для вызова удаленной процедуры. Заглушки могут быть интерпретируемыми или компилируемыми.

В подходе на основе компиляции каждая процедура имеет индивидуальную заглушку на стороне клиента и сервера. Хотя заглушки можно писать вручную, обычно они генерируются компилятором заглушек на основе описания интерфейса процедуры. Эта ситуация проиллюстрирована на рис. 7.5. Поскольку заглушка компилируется, она очень эффективна. В подходе на основе интерпретации система предоставляет универсальные заглушки клиента и сервера, параметры которых задаются описанием интерфейса процедуры. Поскольку изменить это описание легко, интерпретируемые заглушки обладают преимуществом — гибкостью. На практике более распространены компилируемые заглушки.

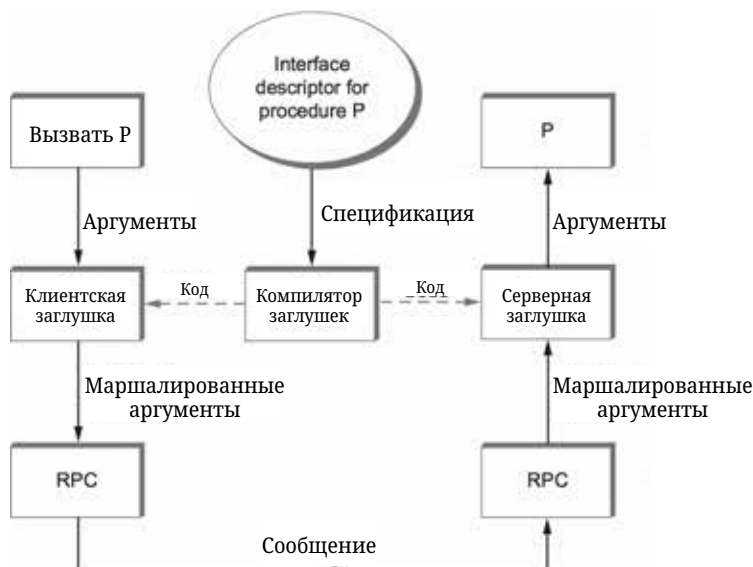


Рисунок 7.5. Компилятор заглушек принимает на вход описание интерфейса и выдает заглушки клиента и сервера.

Глава 7.1.2. Примеры (XDR, ASN.1, NDR, ProtoBufs)

Теперь мы кратко опишем четыре популярных представления сетевых данных с точки зрения этой таксономии. Для иллюстрации работы каждой системы мы используем базовый тип `integer`.

XDR

Внешнее представление данных (XDR) — это сетевой формат, используемый с SunRPC. В только что введенной таксономии XDR:

- Поддерживает всю систему типов языка C, за исключением указателей на функции.
- Определяет каноническую промежуточную форму.
- Не использует теги (кроме указания длин массивов).
- Использует компилируемые заглушки.

Целое число в XDR — это 32-битный элемент данных, кодирующий целое число языка C. Оно представлено в формате с дополнением до двух, с самым значащим байтом целого числа языка C в первом байте целого числа XDR и наименее значащим байтом целого числа языка C в четвертом байте целого числа XDR. То есть XDR использует формат *big-endian* для целых чисел. XDR поддерживает как знаковые, так и беззнаковые целые числа, как это делает язык C.

XDR представляет массивы переменной длины, сначала указывая беззнаковое целое число (4 байта), которое дает количество элементов в массиве, затем следует столько же элементов соответствующего типа. XDR кодирует компоненты структуры в порядке их объявления в структуре. Для массивов и структур размер каждого элемента/компонента кратен 4 байтам. Меньшие типы данных дополняются до 4 байт нулями. Исключение из правила «дополнения до 4 байт» сделано для символов, которые кодируются по одному на байт.



Рисунок 7.6. Пример кодирования структуры в XDR.

Следующий фрагмент кода дает пример C-структуры (`item`) и XDR-рутины, которая кодирует/декодирует эту структуру (`xdr_item`). Рис. 7.6 схематически изображает представление этой структуры в XDR на проводе, когда поле `name` состоит из семи символов, а массив `list` содержит три значения.

В этом примере `xdr_array`, `xdr_int` и `xdr_string` — это три примитивные функции, предоставляемые XDR для кодирования и декодирования массивов, целых чисел и строк символов соответственно. Аргумент `xdrs` является контекстной переменной, которую XDR использует для отслеживания местоположения в сообщении, которое обрабатывается; она включает флаг, указывающий, используется ли эта рутинка для кодирования или декодирования сообщения. Другими словами, такие рутинки, как `xdr_item`, используются как на стороне клиента, так и на стороне сервера. Заметьте, что программист приложения может либо написать рутину `xdr_item` вручную, либо использовать компилятор заглушек, называемый `grscgen` (не показан), чтобы сгенерировать эту рутину кодирования/декодирования. В последнем случае `grscgen` принимает удаленную процедуру, определяющую структуру данных `item`, как входные данные и выводит соответствующую заглушку.

```

#define MAXNAME 256
#define MAXLIST 100

struct item {
    int count;
    char name[MAXNAME];
    int list[MAXLIST];
};

bool_t
xdr_item(XDR *xdrs, struct item *ptr)
{
    return (xdr_int(xdrs, &ptr->count) &&
            xdr_string(xdrs, &ptr->name, MAXNAME) &&
            xdr_array(xdrs, &ptr->list, &ptr->count, MAXLIST,
                      sizeof(int), xdr_int));
}

};

```

Насколько хорошо XDR выполняет свою работу, зависит, конечно, от сложности данных. В простом случае массива целых чисел, где каждое целое число должно быть преобразовано из одного порядка байтов в другой, в среднем требуется три инструкции на каждый байт, а это означает, что преобразование всего массива, вероятно, будет ограничено пропускной способностью памяти машины. Более сложные преобразования, требующие значительно большего числа инструкций на байт, будут ограничены производительностью процессора и, следовательно, выполняться с более низкой скоростью, чем пропускная способность памяти.

ASN.1

Abstract Syntax Notation One (ASN.1) — это стандарт ISO, который определяет, среди прочего, представление для данных, передаваемых по сети. Специфическая часть представления ASN.1 называется *Basic Encoding Rules* (базовые правила кодирования, BER). ASN.1 поддерживает систему типов языка C без указателей на функции, определяет каноническую промежуточную форму и использует типовые теги. Заглушки могут быть как интерпретируемыми, так и компилируемыми. Одним из достижений ASN.1 BER является то, что он используется в Интернет-стандарте Simple Network Management Protocol (SNMP).

ASN.1 представляет каждый элемент данных в виде тройки:

(tag, length, value)

Тег обычно представляет собой 8-битное поле, хотя ASN.1 позволяет определять многобайтовые теги. Поле длины указывает, сколько байтов составляет значение; длину мы обсудим позже. Сложные типы данных, такие как структуры, могут быть созданы путем вложения примитивных типов, как показано на рис. 7.7.

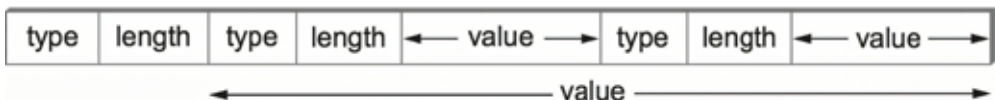


Рисунок 7.7. Составные типы, созданные с помощью вложенности в ASN.1 BER.

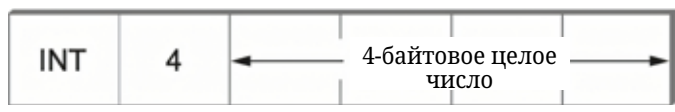


Рисунок 7.8. Представление ASN.1 BER для 4-байтового целого числа.

Если длина значения составляет 127 байт или меньше, то длина указывается в одном байте. Таким образом, например, 32-битное целое число кодируется как 1-байтовый тип, 1-байтовая длина и 4 байта, которые кодируют целое число, как показано на рис. 7.8. Само значение, в случае целого числа, представлено в формате с дополнением до двух и в формате big-endian, как и в XDR. Имейте в виду, что хотя значение целого числа представлено точно так же и в XDR, и в ASN.1, представление XDR не содержит ни типовых, ни длиновых тегов, связанных с этим целым числом. Эти два тега занимают место в сообщении и, что более важно, требуют обработки приmarshalingи demarshaling. Это одна из причин, почему ASN.1 не так эффективен, как XDR. Еще одной причиной является то, что сам факт наличия поля длины перед каждым значением данных означает, что значение данных, скорее всего, не будет выровнено по естественной границе байтов (например, целое число, начинающееся на границе слова). Это усложняет процесс кодирования/декодирования.

Если значение составляет 128 байт или больше, то для указания его длины используются несколько байтов. В этот момент вы можете задаться вопросом, почему байт может указывать длину до 127 байт, а не 256. Причина в том, что 1 бит поля длины используется для обозначения длины самого поля длины. Ноль в восьмом бите указывает на однобайтовое поле длины. Чтобы указать более обширную длину, восьмой бит устанавливается в 1, а остальные 7 бит указывают, сколько дополнительных байтов составляют длину. На рис. 7.9 показаны простая однобайтовая длина и многобайтовая длина.

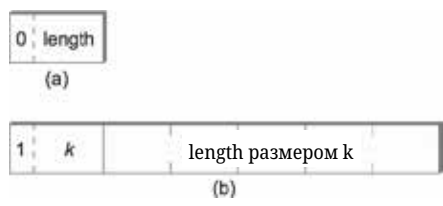


Рисунок 7.9. Представление ASN.1 BER для длины: (a) 1 байт; (b) несколько байт.

NDR

Network Data Representation (NDR) — это стандарт кодирования данных, используемый в Distributed Computing Environment (DCE). В отличие от XDR и ASN.1, NDR использует стратегию «получатель-делает-правильно». Это достигается путем вставки архитектурного тега в начало каждого сообщения; отдельные элементы данных не маркируются. NDR использует компилятор для генерации заглушек. Этот компилятор принимает описание программы, написанное на языке Interface Definition Language (IDL), и генерирует необходимые заглушки. IDL выглядит примерно как C и, таким образом, поддерживает систему типов языка C.



Рисунок 7.10. Архитектурная метка NDR.

На рис. 7.10 показан 4-байтовый архитектурный тег, который включен в начало каждого сообщения, закодированного в NDR. Первый байт содержит два 4-битных поля. Первое поле, `IntegrRep`, определяет формат всех целых чисел, содержащихся в сообщении. Ноль в этом поле указывает на целые числа в формате `big-endian`, а единица указывает на целые числа в формате `little-endian`. Поле `CharRep` указывает, какой формат символов используется: 0 означает ASCII (американский стандартный код для обмена информацией), а 1 означает EBCDIC (старую альтернативу ASCII, определенную IBM). Далее байт `FloatRep` определяет, какое представление плавающей точки используется: 0 означает IEEE 754, 1 означает VAX, 2 означает Cray и 3 означает IBM. Последние 2 байта зарезервированы для будущего использования. Обратите внимание, что в простых случаях, таких как массивы целых чисел, NDR выполняет ту же работу, что и XDR, и поэтому может достигать той же производительности.

ProtoBufs

Protocol Buffers (сокращенно Protobufs) — это язык-независимый и платформа-независимый способ сериализации структурированных данных, часто используемый с gRPC. Он использует стратегию маркирования с канонической промежуточной формой, где заглашка с обеих сторон генерируется из общего `.proto` файла. Эта спецификация использует простой синтаксис, похожий на C, как показывает следующий пример:

```
message Person {
    required string name = 1;
    required int32 id = 2;
    optional string email = 3;

    enum PhoneType {
        MOBILE = 0;
        HOME = 1;
        WORK = 2;
    }

    message PhoneNumber {
        required string number = 1;
        optional PhoneType type = 2 [default = HOME];
    }

    required PhoneNumber phone = 4;
}
```

где `message` можно примерно интерпретировать как эквивалент `typedef struct` в C. Остальная часть примера интуитивно понятна, за исключением того, что каждому полю присваивается числовой идентификатор, чтобы обеспечить уникальность в случае изменения спецификации с течением времени, и каждое поле может быть аннотировано как обязательное (`required`) или необязательное (`optional`).

Способ кодирования целых чисел в Protobufs является новаторским. Он использует технику, называемую *varints* (переменные длины целых чисел), в которой каждый 8-битный байт использует самый значимый бит для указания, есть ли еще байты в числе, а нижние семь бит кодируют представление в дополнительном коде следующих семи бит в значении. Наименее значимая группа идет первой в сериализации.

Это означает, что небольшое число (меньше 128) может быть закодировано в одном байте (например, число 2 кодируется как 0000 0010), в то время как для числа больше 128 требуется больше байтов. Например, число 365 будет закодировано как:

```
1110 1101 0000 0010
```

Чтобы увидеть это, сначала отбросьте самый значимый бит каждого байта, так как он нужен для указания конца числа. В этом примере 1 в самом значимом бите первого байта указывает, что в `varint` больше одного байта:

```
1110 1101 0000 0010
110 1101 000 0010
```

Поскольку `varints` хранят числа с наименее значимой группой первой, вы затем переворачиваете две группы из семи бит. Затем вы объединяете их, чтобы получить окончательное значение:

```
000 0010 110 1101
000 0010 || 110 1101
101101101
256 + 64 + 32 + 8 + 4 + 1 = 365
```

Для более крупной спецификации сообщения вы можете рассматривать сериализованный поток байтов как набор пар «ключ/значение», где ключ (то есть метка) имеет две подчасти: уникальный идентификатор для поля (то есть те самые дополнительные числа в примере файла `.proto`) и тип провода значения (например, `Varint` — это один из примеров типа провода, который мы видели до сих пор). Другие поддерживаемые типы провода включают 32-битный и 64-битный (для целых чисел фиксированной длины) и ограниченный по длине (для строк и вложенных сообщений). Последний указывает, сколько байтов содержит вложенное сообщение (структура), но это другая спецификация сообщения в файле `.proto`, которая указывает, как интерпретировать эти байты.

Глава 7.1.3. Языки разметки (XML)

Хотя мы обсуждали проблему форматирования представления с точки зрения RPC — то есть как кодировать примитивные типы данных и составные структуры данных, чтобы их можно было отправить из клиентской программы в серверную программу, та же самая основная проблема возникает и в других ситуациях. Например, как веб-сервер описывает веб-страницу, чтобы любое количество разных браузеров знало, что отображать на экране? В этом конкретном случае ответом является язык гипертекстовой разметки (HTML), который указывает, что определенные строковые символы должны отображаться жирным или курсивом, какой тип и размер шрифта следует использовать и где должны быть размещены изображения.

Наличие всевозможных веб-приложений и данных также создало ситуацию, в которой различные веб-приложения нуждаются в общении друг с другом и понимании данных друг друга. Например, сайт электронной коммерции может нуждаться в общении с сайтом транспортной компании, чтобы позволить клиенту отслеживать посылку, не покидая сайта электронной коммерции. Это быстро начинает напоминать RPC, и подход, принятый сегодня в Интернете для обеспечения такого общения между веб-серверами, основан на *расширяемом языке разметки* (XML) — способе описания данных, которыми обмениваются веб-приложения.

Языки разметки, примерами которых являются HTML и XML, используют подход тегированных данных. Данные представляются в виде текста, и текстовые теги, известные как разметка, перекрещиваются с текстом данных, чтобы выразить информацию о данных. В случае HTML разметка указывает, как должен быть отображен текст; другие языки разметки, такие как XML, могут выражать тип и структуру данных.

XML на самом деле является фреймворком для определения различных языков разметки для различных видов данных. Например, XML был использован для определения языка разметки, который примерно эквивалентен HTML, называемого *расширяемым языком гипертекстовой разметки* (XHTML). XML определяет базовый синтаксис

для смешивания разметки с текстом данных, но разработчик конкретного языка разметки должен назвать и определить свою разметку. Это обычная практика — называть отдельные языки, основанные на XML, просто как XML, но мы подчеркнем различие в этом вводном материале.

Синтаксис XML выглядит очень похожим на HTML. Например, запись о сотруднике в гипотетическом языке разметки на основе XML может выглядеть как следующий XML-документ, который может быть сохранен в файле под названием *employee.xml*. Первая строка указывает версию XML, которая используется, а оставшиеся строки представляют четыре поля, составляющие запись о сотруднике, последнее из которых (*hiredate*) содержит три подполя. Другими словами, синтаксис XML предусматривает вложенную структуру пар «тег/значение», что эквивалентно древовидной структуре представленных данных (с *employee* в качестве корня). Это похоже на способность XDR, ASN.1 и NDR представлять составные типы, но в формате, который может быть как обработан программами, так и прочитан человеком. Более важно, что программы, такие как парсеры, могут использоваться для различных языков на основе XML, потому что определения этих языков сами выражены в виде машинно-читаемых данных, которые могут быть введены в программы.

```
<?xml version="1.0"?>
<employee>
  <name>John Doe</name>
  <title>Head Bottle Washer</title>
  <id>123456789</id>
  <hiredate>
    <day>5</day>
    <month>June</month>
    <year>1986</year>
  </hiredate>
</employee>
```

Хотя разметка и данные в этом документе очень наглядны для прочтения человеком, именно определение языка записи сотрудника фактически определяет, какие теги допустимы, что они означают и какие типы данных они подразумевают. Без какого-либо формального определения тегов человек (или компьютер) не может сказать, является ли 1986 в поле *year*, например, строкой, целым числом, беззнаковым целым числом или числом с плавающей запятой.

Определение конкретного языка на основе XML дается схемой, которая является термином из базы данных для спецификации, как интерпретировать коллекцию данных. Для XML были определены несколько языков схем; мы сосредоточимся здесь на ведущем стандарте, известном под не слишком удивительным названием XML Schema. Индивидуальная схема, определенная с использованием XML Schema, известна как XML Schema Document (XSD). Следующее — это спецификация XSD для примера; другими словами, она определяет язык, которому соответствует пример документа. Он может быть сохранен в файле под названием *employee.xsd*.

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="employee">
    <complexType>
      <sequence>
        <element name="name" type="string"/>
        <element name="title" type="string"/>
        <element name="id" type="string"/>
        <element name="hiredate">
```

```

    <complexType>
      <sequence>
        <element name="day" type="integer" />
        <element name="month" type="string" />
        <element name="year" type="integer" />
      </sequence>
    </complexType>
  </element>
</sequence>
</complexType>
</element>
</schema>

```

Этот XSD выглядит внешне похожим на наш пример документа `employee.xml`, и на то есть причина: XML Schema сам по себе является языком на основе XML. Существует очевидная связь между этим XSD и документом, определенным выше. Например,

```
<element name="title" type="string" />
```

указывает, что значение, заключенное в разметку `title`, должно интерпретироваться как строка. Последовательность и вложенность этой строки в XSD указывают, что поле `title` должно быть вторым элементом в записи о сотруднике.

В отличие от некоторых языков схем, XML Schema предоставляет типы данных, такие как строка, целое число, десятичное число и логический тип. Он позволяет комбинировать типы данных в последовательности или вложенные структуры, как в `employee.xsd`, чтобы создавать составные типы данных. Таким образом, XSD определяет не только синтаксис; он определяет свою собственную абстрактную модель данных. Документ, который соответствует XSD, представляет собой коллекцию данных, соответствующую модели данных.

Значимость того, что XSD определяет абстрактную модель данных, а не просто синтаксис, заключается в том, что могут существовать и другие способы представления данных, соответствующих модели, кроме XML. В конце концов, у XML есть некоторые недостатки как представления «на проволоке»: он не так компактен, как другие представления данных, и его относительно медленно анализировать. В использовании находятся несколько альтернативных представлений, описываемых как бинарные. Международная организация по стандартизации (ISO) опубликовала одно из них, называемое *Fast Infoset*, в то время как Всемирный консорциум Всемирной паутины (W3C) разработал предложение под названием *Efficient XML Interchange* (EXI). Бинарные представления жертвуют читаемостью для человека ради большей компактности и более быстрого анализа.

Пространства имен XML

XML должен решать распространенную проблему конфликтов имен. Проблема возникает, потому что языки схем, такие как XML Schema, поддерживают модульность в том смысле, что схема может быть повторно использована как часть другой схемы. Предположим, что два XSD определены независимо и оба случайно определяют разметку с именем `idNumber`. Возможно, один XSD использует это имя для идентификации сотрудников компании, а другой XSD использует его для идентификации ноутбуков, принадлежащих компании. Мы могли бы захотеть повторно использовать эти две XSD в третьем XSD для описания, какие активы связаны с какими сотрудниками, но для этого нам нужен механизм для различения `idNumber` сотрудников от `idNumber` ноутбуков.

Решением XML для этой проблемы являются *пространства имен XML*. Пространство имен — это коллекция имен. Каждое пространство имен XML идентифицируется универсальным идентификатором ресурса (URI). URI будут подробно описаны в одной из следующих глав; пока все, что вам действительно нужно знать, это то, что URI — это форма гло-

бально уникального идентификатора. (HTTP URL — это определенный тип URI.) Простое имя разметки, такое как *idNumber*, может быть добавлено в пространство имен, если оно уникально в этом пространстве имен. Поскольку пространство имен глобально уникально, а простое имя уникально в пространстве имен, их комбинация представляет собой глобально уникальное *квалифицированное имя*, которое не может конфликтовать.

Обычно XSD указывает *целевое пространство имен* строкой вида:

```
targetNamespace="http://www.example.com/employee"
```

Это универсальный идентификатор ресурса, идентифицирующий вымышленное пространство имен. Вся новая разметка, определенная в этом XSD, будет принадлежать этому пространству имен.

Теперь, если XSD хочет сослаться на имена, которые были определены в других XSD, он может сделать это, квалифицируя эти имена с префиксом пространства имен. Этот префикс является коротким сокращением для полного URI, который на самом деле идентифицирует пространство имен. Например, следующая строка назначает *emp* в качестве префикса пространства имен для пространства имен сотрудников:

```
xmlns:emp="http://www.example.com/employee"
```

Любая разметка из этого пространства имен будет квалифицирована путем добавления префикса *emp*:, как это сделано с *title* в следующей строке:

```
<emp:title> Head Bottle Washer</emp:title>
```

Другими словами, *emp:title* — это квалифицированное имя, которое не будет конфликтовать с именем *title* из другого пространства имен.

Примечательно, насколько широко XML теперь используется в приложениях, которые варьируются от коммуникаций в стиле RPC между веб-службами до офисных программных инструментов и служб мгновенных сообщений. Это, безусловно, один из основных протоколов, на которых сейчас базируются верхние уровни Интернета.

Глава 7.2. Мультимедийные данные

Мультимедийные данные, включающие аудио, видео и неподвижные изображения, сейчас составляют большую часть трафика в Интернете. Часть того, что сделало возможной широкомасштабную передачу мультимедиа через сети, — это достижения в технологии сжатия. Поскольку мультимедийные данные в основном потребляются людьми, использующими свои органы чувств (зрение и слух) и обрабатываются человеческим мозгом, существуют уникальные задачи по их сжатию. Необходимо постараться сохранить информацию, наиболее важную для человека, при этом избавляясь от всего, что не улучшает его восприятие визуального или аудиального опыта. Таким образом, одновременно используются как компьютерные науки, так и изучение человеческого восприятия. В этой главе мы рассмотрим некоторые из основных усилий по представлению и сжатию мультимедийных данных.

Использование сжатия, конечно, не ограничивается мультимедийными данными — например, вы могли использовать утилиты, такие как *zip* или *compress*, чтобы сжать файлы перед их отправкой по сети или распаковать файл данных после загрузки. Оказывается, что методы, используемые для сжатия данных, которые обычно являются *без потерь*, потому что большинство людей не любит терять данные из файла, также являются частью решения для сжатия мультимедиа. В отличие от этого, *сжатие с потерями*, обычно используемое для мультимедийных данных, не обещает, что полученные данные будут точно такими же, как и отправленные данные. Как отмечалось выше, это связано с тем, что мультимедийные данные часто содержат информацию, малополезную для человека, который их получает. Наши органы чувств и мозг могут воспринимать только

определенное количество деталей. Они также очень хорошо заполняют пропущенные части и даже исправляют некоторые ошибки в том, что мы видим или слышим. И алгоритмы с потерями обычно достигают гораздо лучших коэффициентов сжатия, чем их аналоги без потерь; они могут быть на порядок лучше или даже больше.

Чтобы понять, насколько важным было сжатие для распространения мультимедийных сетей, рассмотрим следующий пример. Экран телевизора высокой четкости имеет разрешение около 1080×1920 пикселей, каждый из которых несет 24 бита цветовой информации, так что каждый кадр составляет

$$1080 \times 1920 \times 24 = 50 \text{ Mb.}$$

Поэтому если вы хотите отправлять 24 кадра в секунду, это будет более 1 Гбит/с. Это больше, чем скорость доступа большинства интернет-пользователей. Напротив, современные методы сжатия могут уменьшить сигнал HDTV до разумно высокого качества в диапазоне 10 Мбит/с, что на два порядка меньше и вполне доступно для большинства пользователей широкополосного интернета. Подобные достижения в сжатии применяются к видео более низкого качества, например, к клипам на YouTube — веб-видео никогда не достигло бы своей нынешней популярности без сжатия, которое позволяет всем этим развлекательным видео укладываться в пропускную способность современных сетей.

Методы сжатия, применяемые к мультимедиа, были областью больших инноваций, особенно сжатие с потерями. Однако методы без потерь также играют важную роль. На самом деле большинство алгоритмов с потерями включают некоторые шаги, которые являются без потерь, поэтому мы начнем наше обсуждение с обзора методов сжатия без потерь.

Глава 7.2.1. Методы сжатия без потерь

Во многих отношениях сжатие неразрывно связано с кодированием данных. Когда мы думаем о том, как закодировать фрагмент данных в наборе битов, мы можем также подумать о том, как закодировать данные в наименьшем возможном наборе битов. Например, если у вас есть блок данных, состоящий из 26 символов от A до Z, и если все эти символы имеют равные шансы появления в блоке данных, который вы кодируете, то кодирование каждого символа в 5 битов — это лучшее, что вы можете сделать (поскольку $2^5 = 32$ является наименьшей степенью 2 выше 26). Однако если символ R встречается в 50% случаев, то было бы разумно использовать меньше битов для кодирования R, чем для любого другого символа. В общем, если вы знаете относительную вероятность появления каждого символа в данных, то вы можете назначить разное количество битов каждому возможному символу таким образом, чтобы минимизировать количество битов, необходимых для кодирования данного блока данных. Это основная идея *кодов Хаффмана*, одной из важных ранних разработок в области сжатия данных.

Кодирование длин серий (RLE)

Кодирование длин серий (RLE) — это метод сжатия с простотой на уровне брутальной силы. Идея заключается в замене последовательных вхождений данного символа одним экземпляром символа плюс счетчиком, показывающим, сколько раз этот символ встречается — отсюда и название «длина серий». Например, строка AAABBCDDDD будет закодирована как 3A2B1C4D.

RLE оказывается полезным для сжатия некоторых классов изображений. Он может быть использован в контексте сравнения смежных значений пикселей и кодирования только изменений. Для изображений с большими однородными областями эта техника достаточно эффективна. Например, не редкость, что RLE может достигать коэффициентов сжатия порядка 8 к 1 для отсканированных текстовых изображений. RLE хорошо работает на таких файлах, потому что они часто содержат большое количество пробелов, которые

можно удалить. Для тех, кто помнит технологии того времени, RLE был ключевым алгоритмом сжатия, используемым для передачи факсимильных сообщений. Однако для изображений даже с небольшой степенью локальной вариации нередко сжатие может фактически увеличить размер изображения в байтах, поскольку на каждый символ, который не повторяется, требуется 2 байта.

Дифференциальная пульсовая кодовая модуляция (DPCM)

Другой простой алгоритм без потерь — дифференциальная пульсовая кодовая модуляция (DPCM). Идея заключается в том, чтобы сначала вывести опорный символ, а затем для каждого символа в данных вывести разницу между этим символом и опорным символом. Например, используя символ А в качестве опорного символа, строка AAABBCDDDD будет закодирована как A0001123333, потому что А такой же, как и опорный символ, В имеет разницу 1 от опорного символа и так далее. Следует отметить, что этот простой пример не иллюстрирует реальной выгоды от DPCM, которая заключается в том, что при малых различиях можно закодировать их меньшим количеством бит, чем сам символ. В этом примере диапазон различий от 0 до 3 может быть представлен 2 битами каждое, вместо 7 или 8 бит, требуемых для полного символа. Как только разница становится слишком большой, выбирается новый опорный символ.

DPCM работает лучше, чем RLE, для большинства цифровых изображений, потому что он использует тот факт, что смежные пиксели обычно похожи. Благодаря этой корреляции динамический диапазон различий между значениями смежных пикселей может быть значительно меньше, чем динамический диапазон исходного изображения, и поэтому этот диапазон можно представить с помощью меньшего количества бит. Используя DPCM, мы достигли коэффициентов сжатия 1,5 к 1 для цифровых изображений. DPCM также работает с аудио, потому что смежные примеры аудиоформы чаще всего близки по значению.

Немного другой подход, называемый *дельта-кодированием*, просто закодирует символ как разницу от предыдущего. Таким образом, например, AAABBCDDDD будет представлено как A001011000. Следует отметить, что дельта-кодирование, вероятно, хорошо работает для кодирования изображений, где смежные пиксели похожи. Также возможно выполнить RLE после дельта-кодирования, поскольку можно обнаружить длинные строки нулей, если рядом много похожих символов.

Методы на основе словаря

Последний метод сжатия без потерь, который мы рассматриваем, — это метод на основе словаря, из которых наиболее известен алгоритм сжатия Лемпеля-Зива (LZ). Команды Unix `compress` и `gzip` используют варианты алгоритма LZ.

Идея алгоритма сжатия на основе словаря заключается в построении словаря (таблицы) переменной длины строк (рассматривайте их как общие фразы), которые вы ожидаете найти в данных, и замене каждой из этих строк при ее появлении в данных на соответствующий индекс в словаре. Например, вместо работы с отдельными символами в текстовых данных можно рассматривать каждое слово как строку и выводить индекс в словаре для этого слова. Для дополнительного пояснения этого примера слово «сжатие» имеет индекс 4978 в определенном словаре; оно является 4978-м словом в этом словаре. Для сжатия текстового тела каждый раз, когда строка «сжатие» появляется, она будет заменена на 4978. Поскольку этот конкретный словарь содержит чуть более 25 000 слов, для кодирования индекса потребуется 15 бит, а это означает, что строка «сжатие» может быть представлена в 15 битах вместо 77 бит, требуемых для 7-битного ASCII. Это коэффициент сжатия 5 к 1! В другом случае мы смогли достичь коэффициента сжатия 2 к 1 при применении команды `compress` к исходному коду протоколов, описанных в этой книге.

Конечно, возникает вопрос, откуда берется словарь. Один из вариантов — определить статический словарь, предпочтительно подходящий для сжимаемых данных. Более общее решение, используемое в сжатии LZ, заключается в адаптивном определении словаря на основе содержимого сжимаемых данных. В этом случае словарь, созданный во время сжатия, должен быть отправлен вместе с данными, чтобы половина алгоритма декомпрессии могла выполнить свою работу. Точные методы построения адаптивного словаря были предметом обширных исследований.

Глава 7.2.2. Представление и сжатие изображений (GIF, JPEG)

Учитывая всеобщее использование цифровых изображений — это использование возникло благодаря изобретению графических дисплеев, а не высокоскоростным сетям — становятся необходимыми стандартное представление форматов и алгоритмы сжатия для цифровых изображений. В ответ на эту потребность ISO определила цифровой формат изображения, известный как JPEG, названный в честь группы экспертов по фотографии (Joint Photographic Experts Group), разработавших его. (Слово «Joint» в JPEG означает совместное усилие ISO/ITU.) JPEG является наиболее распространенным форматом для статических изображений, используемым в настоящее время. В основе определения формата лежит алгоритм сжатия, который мы опишем ниже. Многие техники, используемые в JPEG, также применяются в MPEG — наборе стандартов для сжатия и передачи видео, созданном группой экспертов по движущимся изображениям (Moving Picture Experts Group).

Прежде чем мы погрузимся в детали JPEG, стоит отметить, что есть несколько шагов, которые необходимо выполнить, чтобы перейти от цифрового изображения к сжатому представлению этого изображения, которое может быть передано, декомпрессировано и корректно отображено получателем. Вероятно, вы знаете, что цифровые изображения состоят из пикселей (отсюда мегапиксели, упоминаемые в рекламе смартфонов с камерами). Каждый пиксель представляет собой одну позицию в двумерной сетке, составляющей изображение, и для цветных изображений каждый пиксель имеет числовое значение, представляющее цвет. Существует множество способов представления цвета, называемых *цветовыми пространствами*; наиболее известным из них является RGB (красный, зеленый, синий). Можно представить цвет как трехмерное количество — можно создать любой цвет, используя красный, зеленый и синий свет в различных количествах. В трехмерном пространстве есть много различных, допустимых способов описания данной точки (рассмотрим, например, декартовы и полярные координаты). Аналогично существуют различные способы описания цвета с использованием трех величин, и наиболее распространенной альтернативой RGB является YUV. Y представляет собой яркость, приблизительно общую яркость пикселя, а U и V содержат цветность или информацию о цвете. Путаница в том, что существует несколько различных вариантов цветового пространства YUV. Больше об этом чуть позже.

Важность этого обсуждения заключается в том, что кодирование и передача цветных изображений (статических или движущихся) требует согласования между двумя концами по поводу цветового пространства. В противном случае конечный получатель будет отображать разные цвета, отличающиеся от того, что отправитель задумал. Поэтому согласование определения цветового пространства (и, возможно, способа передачи информации об используемом конкретном пространстве) является частью определения любого формата изображения или видео.

Давайте рассмотрим пример формата графического обмена (GIF). GIF использует цветовое пространство RGB и изначально использует 8 бит для представления каждого из трех измерений цвета, что в сумме составляет 24 бита. Однако вместо отправки этих 24 бит на пиксель GIF сначала сокращает 24-битные цветные изображения до 8-битных цветных изображений. Это достигается путем определения цветов, используемых на картинке, из которых, как правило, будет значительно меньше, чем 2^6 в степени 24, и затем выбором 256 цветов, наиболее близких к используемым цветам на изображении. Тем

не менее может быть больше 256 цветов, поэтому трюк заключается в том, чтобы не искажать цвета слишком сильно, выбирая 256 цветов таким образом, чтобы ни у одного пикселя не происходило слишком сильного изменения цвета.

256 цветов хранятся в таблице, которую можно индексировать 8-битным числом, и значение для каждого пикселя заменяется соответствующим индексом. Следует отметить, что это пример потерь при сжатии для любого изображения с более чем 256 цветами. Затем GIF применяет вариант алгоритма LZ к результату, обрабатывая общие последовательности пикселей как строки, составляющие словарь — операция без потерь. Используя этот подход, GIF иногда может достигать коэффициентов сжатия в районе 10:1, но только если изображение состоит из относительно небольшого числа дискретных цветов. Например, графические логотипы хорошо обрабатываются GIF. Изображения природных сцен, включающие часто непрерывный спектр цветов, не могут быть сжаты на такой коэффициент с помощью GIF. В некоторых случаях человеческий глаз также может легко заметить искажения, вызванные потерями при сжатии цвета в GIF.

Формат JPEG значительно лучше подходит для фотографических изображений, чего можно было ожидать, учитывая название создавшей его группы. JPEG не сокращает количество цветов, как GIF. Вместо этого JPEG начинает с преобразования цветов RGB (которые обычно получаются из цифровой камеры) в цветовое пространство YUV. Причина в том, как глаз воспринимает изображения. В глазу есть рецепторы для яркости и отдельные рецепторы для цвета. Поскольку мы очень чувствительны к изменениям в яркости, имеет смысл тратить больше битов на передачу информации о яркости. Поскольку компонент Y в YUV, грубо говоря, представляет собой яркость пикселя, мы можем сжимать этот компонент отдельно и менее агрессивно, чем два остальных (компоненты хроминанса).

Как указано выше, YUV и RGB — альтернативные способы описания точки в трехмерном пространстве, и возможно преобразование из одного цветового пространства в другое с помощью линейных уравнений. Для одного из часто используемых цветовых пространств YUV уравнения такие:

$$Y = 0,299R + 0,587G + 0,114B \quad U = (B - Y) \times 0,565$$

$$V = (R - Y) \times 0,713$$

Точные значения констант здесь не важны, до тех пор пока кодер и декодер согласованы по их значениям. (Декодер должен применить обратные преобразования для восстановления компонентов RGB, необходимых для вывода на дисплей.) Однако эти константы тщательно выбираются с учетом человеческого восприятия цвета. Можно заметить, что Y, яркость, представляет собой сумму компонентов красного, зеленого и синего, в то время как U и V — компоненты разности цвета. U представляет разницу между яркостью и синим, а V — разницу между яркостью и красным. Вы можете заметить, что установка R, G и B в их максимальные значения (которые будут равны 255 для 8-битных представлений) также даст значение Y=255, тогда как U и V в этом случае будут равны нулю. То есть полностью белый пиксель (255,255,255) в цветовом пространстве RGB и (255,0,0) в цветовом пространстве YUV.

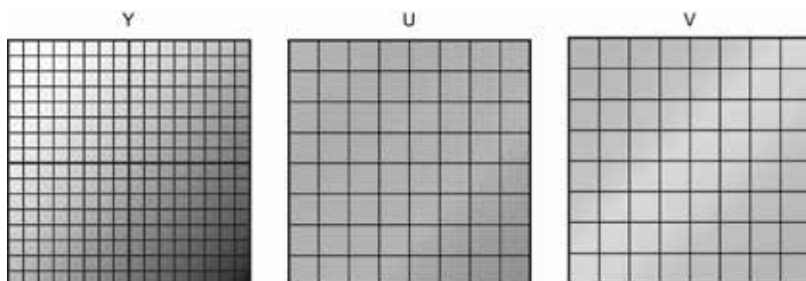


Рисунок 7.11. Поддискретизация U- и V-компоненты изображения.

После того как изображение было преобразовано в цветовое пространство YUV, мы можем продолжить сжимать каждый из трех компонентов отдельно. Мы хотим более агрессивно сжимать компоненты U и V, на которые глаз человека менее реагирует. Один из способов сжатия компонентов U и V — их *поддискретизация*. Основная идея поддискретизации заключается в том, чтобы взять несколько смежных пикселей, вычислить среднее значение U или V для этой группы пикселей и передавать его вместо значения для каждого пикселя. Рис. 7.11 иллюстрирует эту идею. Компонент яркости (Y) не поддискретизируется, поэтому значение Y для всех пикселей будет передано, как показано сеткой 16×16 пикселей слева. В случае компонентов U и V мы рассматриваем каждую группу из четырех смежных пикселей как группу, вычисляем среднее значение U или V для этой группы и передаем его. Таким образом, в этом примере для каждого четырех пикселей мы передаем шесть значений (четыре Y и по одному U и V) вместо исходных 12 значений (по четыре для всех трех компонентов), что дает сокращение информации на 50%.

Стоит отметить, что можно быть как более, так и менее агрессивным в поддискретизации, что соответственно увеличивает сжатие и снижает качество. Поддискретизация, показанная здесь, при которой хроминанс поддискретизируется в два раза как по горизонтали, так и по вертикали (и которая обозначается как 4:2:0), случайно соответствует наиболее распространенному подходу, используемому как в JPEG, так и в MPEG.

После завершения поддискретизации у нас теперь есть три сетки пикселей, и каждая из них обрабатывается отдельно. Сжатие JPEG каждого компонента происходит в три фазы, как показано на рис. 7.12. На этапе сжатия изображение передается через эти три фазы по одному блоку 8×8 за раз. Первая фаза применяет дискретное косинусное преобразование (DCT) к блоку. Если думать об изображении как о сигнале в пространственной области, то DCT преобразует этот сигнал в эквивалентный сигнал в *частотной области*. Это операция без потерь, но необходимая предварительная стадия для следующего, потерянного шага. После DCT вторая фаза применяет квантование к полученному сигналу и тем самым теряет наименее значимую информацию, содержащуюся в этом сигнале. Третья фаза кодирует конечный результат, но при этом также добавляет элемент без потерь в сжатие с потерями, достигнутое первыми двумя фазами. Распаковка следует за этими же тремя фазами, но в обратном порядке.

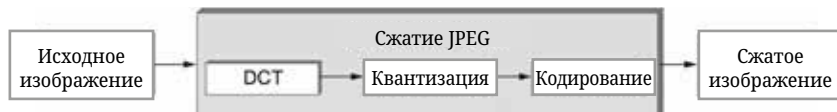


Рисунок 7.12. Блок-схема сжатия JPEG.

Фаза DCT

DCT — это преобразование, тесно связанное с быстрым преобразованием Фурье (FFT). Оно принимает матрицу 8×8 пиксельных значений на входе и выдает матрицу 8×8 частотных коэффициентов. Можно представлять себе входную матрицу как 64-точечный сигнал, определенный в двух пространственных измерениях (x и y); DCT разбивает этот сигнал на 64 пространственных частоты. Чтобы получить интуитивное понимание пространственной частоты, представьте себе движение по изображению в направлении x. Вы увидите, как значение каждого пикселя изменяется как функция от x. Если это значение медленно изменяется с увеличением x, то у него низкая пространственная частота; если оно быстро меняется, то у него высокая пространственная частота. Таким образом, низкие частоты соответствуют общим чертам изображения, а высокие частоты — мелким деталям. Идея DCT заключается в отделении общих черт, которые важны для восприятия изображения, от мелких деталей, которые менее важны и в некоторых случаях могут быть едва заметны для глаза.

DCT, а также его обратное преобразование, которое восстанавливает исходные пиксели во время распаковки, определяются следующей формулой:

$$DCT(i, j) = \frac{1}{\sqrt{2N}} C(i) C(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} pixel(x, y) \cos \left[\frac{(2x+1)i\pi}{2N} \right] \cos \left[\frac{(2y+1)j\pi}{2N} \right]$$

где $C(x) = 1/\sqrt{2}$ при $x = 0$ и 1 при $x > 0$, а $pixel(x, y)$ — это оттенок серого пикселя в позиции (x, y) в 8×8 блоке, подлежащем сжатию; $N = 8$ в данном случае.

Первый коэффициент частоты, находящийся в месте $(0,0)$ в выходной матрице, называется *коэффициентом DC*. Интуитивно можно понять, что коэффициент DC — это мера среднего значения 64 входных пикселей. Другие 63 элемента выходной матрицы называются *коэффициентами AC*. Они добавляют информацию о более высоких пространственных частотах к этому среднему значению. Таким образом, двигаясь от первого частотного коэффициента к 64-му, мы переходим от низкочастотной информации к высокочастотной, от общих черт изображения к все более тонким деталям. Эти коэффициенты более высоких частот становятся все менее важными для восприятия качества изображения. Именно вторая фаза JPEG решает, какую часть из каких коэффициентов нужно отбросить.

Фаза квантования

Вторая фаза JPEG — это та стадия, на которой компрессия выполняется с потерями. Дискретное косинусное преобразование (DCT) само по себе не теряет информацию; оно просто преобразует изображение в форму, которая упрощает понимание того, какую информацию следует удалить. (Хотя DCT и не является по своей сути с потерями, конечно, существует некоторая потеря точности из-за использования фиксированной арифметики.) Квантование легко понять — это просто вопрос отбрасывания незначительных битов коэффициентов частоты.

Чтобы понять, как работает фаза квантования, представьте, что вы хотите сжать некоторые целые числа меньше 100, такие как 45, 98, 23, 66 и 7. Если вы решили, что для ваших целей достаточно знать эти числа, округленные до ближайшего множителя 10, то вы можете разделить каждое число на квант 10, используя целочисленную арифметику, получая 4, 9, 2, 6 и 0. Эти числа можно кодировать в 4 бита вместо 7 бит, необходимых для кодирования исходных чисел.

Таблица 7.1.
Пример таблицы квантования JPEG.

Квант							
3	5	7	9	11	13	15	17
5	7	9	11	13	15	17	19
7	9	11	13	15	17	19	21
9	11	13	15	17	19	21	23
11	13	15	17	19	21	23	25
13	15	17	19	21	23	25	27
15	17	19	21	23	25	27	29
17	19	21	23	25	27	29	31

Вместо использования одного и того же кванта для всех 64 коэффициентов JPEG использует таблицу квантования, которая задает квант для каждого из коэффициентов, как указано в приведенной ниже формуле. Можно рассматривать эту таблицу (Квант)

как параметр, который можно настроить для контроля степени потери информации и, соответственно, степени сжатия. На практике стандарт JPEG задает набор таблиц квантования, которые оказались эффективными для сжатия цифровых изображений; пример таблицы квантования приведен в табл. 7.1. В таблицах, подобных этой, низкие коэффициенты имеют квант, близкий к 1 (что означает, что теряется мало информации низкой частоты), а высокие коэффициенты имеют большие значения (что означает, что теряется больше информации высокой частоты). Обратите внимание, что в результате таких таблиц квантования многие коэффициенты высокой частоты становятся равными 0 после квантования, что делает их подходящими для дальнейшего сжатия на третьей фазе.

Основное уравнение квантования имеет вид

$$\text{QuantizedValue}(i, j) = \text{IntegerRound}(\text{DCT}(i, j), \text{Quantum}(i, j))$$

Где

$$\begin{aligned} \text{IntegerRound}(x) &= \\ \text{Floor}(x + 0.5) &\text{ if } x \geq 0 \\ \text{Floor}(x - 0.5) &\text{ if } x < 0 \end{aligned}$$

Тогда декомпрессия определяется просто как

$$\text{DCT}(i, j) = \text{QuantizedValue}(i, j) \times \text{Quantum}(i, j)$$

Например, если коэффициент DC (то есть DCT(0,0)) для определенного блока равен 25, то квантование этого значения с помощью таблицы 7.1 приведет к следующему результату:

$$\text{Floor}(25/3+0.5) = 8$$

При декомпрессии этот коэффициент будет восстановлен как $8 \times 3 = 24$.

Фаза кодирования

Заключительная фаза JPEG кодирует квантованные частотные коэффициенты в компактной форме. Это приводит к дополнительному сжатию, но это сжатие без потерь. Начиная с коэффициента DC в позиции (0,0) коэффициенты обрабатываются в зигзагообразной последовательности, показанной на рис. 7.13. По этому зигзагу применяется форма кодирования длин серий (RLE) — RLE применяется только к нулевым коэффициентам, что важно, поскольку многие из последующих коэффициентов равны 0. Индивидуальные значения коэффициентов затем кодируются с использованием кода Хаффмана. (Стандарт JPEG позволяет реализатору использовать арифметическое кодирование вместо кода Хаффмана.)

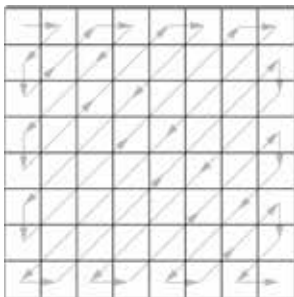


Рисунок 7.13. Зигзагообразный обход квантованных частотных коэффициентов.

Кроме того, поскольку коэффициент DC содержит значительную часть информации о блоке 8×8 из исходного изображения, и изображения обычно изменяются медленно от блока к блоку, каждый коэффициент DC кодируется как разность от предыдущего коэффициента DC. Это метод дельта-кодирования, описанный в следующей главе.

JPEG включает множество вариаций, которые контролируют степень сжатия в зависимости от точности изображения. Это можно сделать, например, используя различные таблицы квантования. Эти вариации, а также тот факт, что разные изображения имеют разные характеристики, делают невозможным точное определение того, каких коэффициентов сжатия можно достичь с помощью JPEG. Коэффициенты 30:1 являются обычными, и, конечно, возможны более высокие коэффициенты, но *артефакты* (заметные искажения из-за сжатия) становятся более серьезными при более высоких коэффициентах.

Глава 7.2.3. Сжатие видео (MPEG)

Теперь мы обратим внимание на формат MPEG, названный в честь группы экспертов по движущимся изображениям, которая его определила. В первом рассмотрении движущееся изображение (то есть видео) представляет собой просто последовательность неподвижных изображений — также называемых *кадрами* (или *фреймами*) либо картинками, — *отображаемых* с некоторой скоростью видео. Каждый из этих кадров может быть сжат с использованием той же техники, основанной на DCT, которая используется в JPEG. Однако остановка на этом этапе была бы ошибкой, поскольку она не устраняет межкадровую избыточность, присутствующую в видеопоследовательности. Например, два последовательных кадра видео будут содержать почти идентичную информацию, если в сцене не происходит активного движения, поэтому нет необходимости отправлять одну и ту же информацию дважды. Даже когда есть движение, может быть много избыточности, так как движущийся объект может не изменяться от одного кадра к следующему; в некоторых случаях меняется только его положение. MPEG учитывает эту межкадровую избыточность. MPEG также определяет механизм кодирования аудиосигнала вместе с видео, но в этой главе мы рассматриваем только видеосоставляющую MPEG.

Типы фреймов (кадров)

MPEG принимает последовательность видеок кадров в качестве входных данных и сжимает их в три типа фреймов: I-фреймы (внутрикадровые), P-фреймы (прогнозируемые кадры) и B-фреймы (двусторонне прогнозируемые кадры). Каждый входной кадр сжимается в один из этих трех типов фреймов. I-фреймы можно считать опорными кадрами; они самодостаточны и не зависят ни от предыдущих, ни от последующих кадров. В первом приближении I-фрейм — это просто JPEG-сжатая версия соответствующего кадра в виде источника. P- и B-фреймы не являются самодостаточными; они указывают относительные различия по отношению к некоторому опорному кадру. Если говорить более конкретно, P-фрейм указывает различия по сравнению с предыдущим I-фреймом, тогда как B-фрейм дает интерполяцию между предыдущим и последующим I- или P-фреймами.

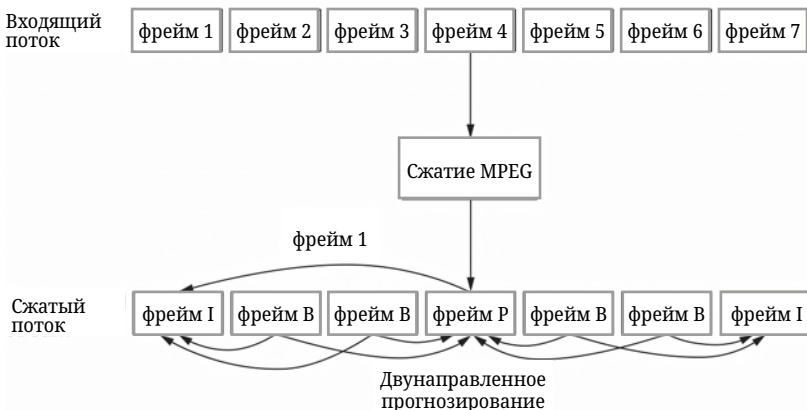


Рисунок 7.14. Последовательность кадров I, P и B, сгенерированных MPEG.

Рис. 7.14 иллюстрирует последовательность из семи видеокадров, которые после сжатия MPEG превращаются в последовательность I, P и B фреймов. Два I-фрейма являются автономными; каждый из них может быть декомпрессирован на приемной стороне независимо от других кадров. P-фрейм зависит от предыдущего I-фрейма; он может быть декомпрессирован на приемной стороне только при наличии предыдущего I-фрейма. Каждый из B-фреймов зависит как от предыдущего I- или P-фрейма, так и от последующего I- или P-фрейма. Оба этих опорных кадра должны прибыть на приемную сторону до того, как MPEG сможет декомпрессировать B-фрейм для восстановления исходного видеокадра.

Обратите внимание, что, поскольку каждый B-фрейм зависит от более позднего кадра в последовательности, сжатые кадры не передаются в последовательном порядке. Вместо этого последовательность I B B P B B I, показанная на рис. 7.14, передается как I P B B I B B. Кроме того, MPEG не определяет соотношение I-фреймов к P- и B-фреймам; это соотношение может варьироваться в зависимости от требуемой степени сжатия и качества изображения. Например, допустимо передавать только I-фреймы. Это было бы похоже на использование JPEG для сжатия видео.

В отличие от предыдущего обсуждения JPEG, далее основное внимание уделяется декодированию MPEG-потока. Описание декодирования несколько проще, и эта операция чаще реализуется в современных сетевых системах, поскольку кодирование MPEG настолько дорогостоящее, что его часто выполняют офлайн (то есть не в реальном времени). Например, в системе видео по запросу видео будет закодировано и сохранено на диске заранее. Когда зритель захочет посмотреть видео, MPEG-поток будет передан на устройство зрителя, которое будет декодировать и отображать поток в реальном времени.

Рассмотрим более подробно три типа фреймов. Как упоминалось выше, I-фреймы примерно равны JPEG-сжатой версии исходного кадра. Основное различие состоит в том, что MPEG работает с блоками размером 16×16 . Для цветного видео, представленного в YUV, компоненты U и V в каждом макроблоке субдискретизируются в блок 8×8 , как мы обсуждали выше в контексте JPEG. Каждый 2×2 подблок в макроблоке представлен одним значением U и одним значением V — средним из четырех значений пикселей. Подблок все еще имеет четыре значения Y. Соотношение между кадром и соответствующими макроблоками показано на рис. 7.15.

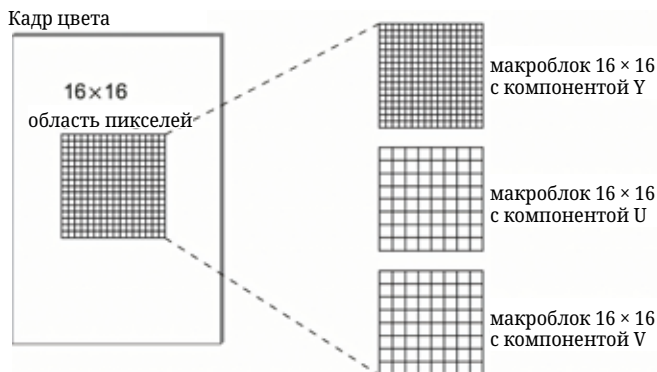


Рисунок 7.15. Каждый кадр как набор макроблоков.

P- и B-фреймы также обрабатываются в единицах макроблоков. Интуитивно понятно, что информация, которую они несут для каждого макроблока, отражает движение в видео; то есть показывает, в каком направлении и насколько далеко макроблок переместился относительно опорного кадра(ов). Далее описывается, как используется B-фрейм для восстановления кадра во время декомпрессии; P-фреймы обрабатываются аналогичным образом, за исключением того, что они зависят только от одного опорного кадра, а не от двух.

Прежде чем перейти к подробностям декомпрессии В-фрейма, отметим, что каждый макроблок в В-фрейме не обязательно определяется относительно как предыдущего, так и последующего кадров, как это предполагалось выше, а может быть определен только относительно одного из них. Фактически данный макроблок в В-фрейме может использовать ту же внутрикадровую кодировку, что и в I-фрейме. Эта гибкость существует потому, что если движущееся изображение меняется слишком быстро, то иногда имеет смысл использовать внутрикадровую кодировку, а не прогнозируемую кодировку вперед или назад. Таким образом, каждый макроблок в В-фрейме включает поле типа, которое указывает, какая кодировка используется для этого макроблока. Однако в следующем обсуждении мы рассмотрим общий случай, когда макроблок использует двустороннюю прогнозируемую кодировку.

В таком случае каждый макроблок в В-фрейме представлен четырьмя элементами: (1) координата макроблока в кадре, (2) вектор движения относительно предыдущего опорного кадра, (3) вектор движения относительно последующего опорного кадра и (4) дельта (δ) для каждого пикселя в макроблоке (то есть насколько изменился каждый пиксель относительно двух опорных пикселей). Для каждого пикселя в макроблоке первая задача — найти соответствующий опорный пиксель в прошлых и будущих опорных кадрах. Это делается с помощью двух векторов движения, связанных с макроблоком. Затем дельта для пикселя добавляется к среднему значению этих двух опорных пикселей. Точнее говоря, если обозначить F_p и F_f как прошлый и будущий опорные кадры соответственно, а векторы движения прошлого и будущего заданы как (x_p, y_p) и (x_f, y_f) , тогда пиксель с координатами (x, y) в текущем кадре (обозначаемом как F_c) вычисляется следующим образом:

$$F_c(x, y) = \frac{F_p(x + x_p, y + y_p) + F_f(x + x_f, y + y_f)}{2 + \delta(x, y)}$$

где δ — это дельта для пикселя, как указано в В-фрейме. Эти дельты кодируются так же, как пиксели в I-фреймах; то есть они проходят через DCT, а затем квантируются. Поскольку дельты обычно малы, большинство коэффициентов DCT равны 0 после квантования; следовательно, их можно эффективно сжать.

Из предыдущего обсуждения должно быть довольно ясно, как осуществляется кодирование, за одним исключением. При генерации В- или Р-фрейма во время сжатия MPEG должен решить, где разместить макроблоки. Напомним, что каждый макроблок в Р-фрейме, например, определяется относительно макроблока в I-фрейме, но макроблок в Р-фрейме не обязательно должен находиться в той же части кадра, что и соответствующий макроблок в I-фрейме — разница в положении задается вектором движения. Вам нужно выбрать вектор движения, который сделает макроблок в Р-фрейме как можно более похожим на соответствующий макроблок в I-фрейме, чтобы дельты для этого макроблока были как можно меньше. Это означает, что нужно выяснить, куда переместились объекты на изображении от одного кадра к следующему. Это задача *оценки движения*, и для ее решения известно несколько методов (эвристик). (Мы обсуждаем статьи, рассматривающие эту проблему, в конце данного раздела.) Сложность этой задачи — одна из причин, по которой кодирование MPEG занимает больше времени, чем декодирование на эквивалентном оборудовании. MPEG не определяет никакой конкретной техники; он только определяет формат для кодирования этой информации в В- и Р-фреймах и алгоритм для восстановления пикселя во время декомпрессии, как указано выше.

Эффективность и производительность

MPEG обычно достигает коэффициента сжатия 90:1, хотя коэффициенты до 150:1 тоже не редкость. Что касается отдельных типов кадров, можно ожидать коэффициент сжатия примерно 30:1 для I-фреймов (это соответствует коэффициентам, достигаемым с использованием JPEG, когда 24-битный цвет сначала уменьшается до 8-битного цвета), тог-

да как коэффициенты сжатия для Р- и В-фреймов обычно в три-пять раз меньше, чем для I-фреймов. Без предварительного уменьшения 24-битного цвета до 8-битного достижимое сжатие с MPEG обычно составляет от 30:1 до 50:1.

MPEG требует значительных вычислительных ресурсов. На стороне сжатия оно обычно выполняется офлайн, что не является проблемой при подготовке фильмов для службы видео по запросу. Сегодня видео может сжиматься в реальном времени с использованием аппаратных средств, но программные реализации быстро сокращают этот разрыв. На стороне декомпрессии доступны недорогие MPEG-видеоплаты, но они делают не намного больше, чем преобразование цвета YUV, что, к счастью, является самым затратным шагом. Большая часть фактической декомпрессии MPEG выполняется программно. В последние годы процессоры стали достаточно мощными, чтобы справиться с видео на скорости 30 кадров в секунду при декодировании потоков MPEG исключительно программным способом — современные процессоры могут даже декодировать потоки MPEG для видео высокой четкости (HDTV).

Стандарты кодирования видео

Завершая данную тему, отметим, что MPEG является развивающимся стандартом значительной сложности. Эта сложность возникает из-за стремления предоставить алгоритму кодирования максимальную свободу в том, как он кодирует данный видеопоток, что приводит к различным скоростям передачи видео. Она также возникает из-за эволюции стандарта с течением времени, при этом группа экспертов по движущимся изображениям усердно работает над сохранением обратной совместимости (например, MPEG-1, MPEG-2, MPEG-4). В этой книге описаны основные идеи, лежащие в основе сжатия на основе MPEG, но, конечно, не все тонкости международного стандарта.

Более того, MPEG не является единственным доступным стандартом для кодирования видео. Например, ITU-T также определила *серию H* для кодирования мультимедийных данных в реальном времени. В общем, серия H включает стандарты для видео, аудио, управления и мультиплексирования (например, смешивание аудио, видео и данных в один битовый поток). В рамках серии H.261 и H.263 были первыми и вторыми поколениями стандартов видеокodирования. В принципе, как H.261, так и H.263 выглядят очень похоже на MPEG: они используют DCT, квантование и межкадровое сжатие. Различия между H.261/H.263 и MPEG заключаются в деталях.

Сегодня партнерство между ITU-T и группой MPEG привело к совместному стандарту H.264/MPEG-4, который используется как для Blu-ray дисков, так и для многих популярных потоковых источников (например, YouTube, Vimeo).

Глава 7.2.4. Передача MPEG по сети

Как мы уже отмечали, MPEG и JPEG — это не только стандарты сжатия, но и определения форматов видео и изображений соответственно. Сосредоточившись на MPEG, первым делом нужно помнить, что он определяет формат *видеопотока*; он не указывает, как этот поток разбивается на сетевые пакеты. Таким образом, MPEG может использоваться как для видео, хранящегося на диске, так и для видео, передаваемого по сетевому соединению, ориентированному на поток, как в случае TCP.

То, что мы описываем ниже, называется *основным профилем* MPEG-видеопотока, который передается по сети. Вы можете думать о профиле MPEG как о «версии», за исключением того, что профиль не указывается явно в заголовке MPEG; приемник должен определить (deduce) профиль из комбинации полей заголовка, которые он видит.

Основной профиль MPEG-потока имеет вложенную структуру, как показано на рис. 7.16. (Имейте в виду, что этот рисунок скрывает множество мелких деталей.) На самом внешнем уровне видео содержит последовательность групп изображений (GOP), разделенных SeqHdr. Последовательность завершается SeqEndCode (0xb7). SeqHdr, предшествующий каждому GOP, указывает, среди прочего, размер каждого изображения (кадра) в GOP (из-

меряемый как в пикселях, так и в макроблоках), интервал между кадрами (измеряемый в микросекундах) и две квантующие матрицы для макроблоков внутри этого GOP: одну для внутрикодированных макроблоков (I-блоки) и одну для межкодированных макроблоков (B- и P-блоки). Поскольку эта информация предоставляется для каждого GOP, а не один раз для всего видеопотока, как можно было бы ожидать, возможна смена квантующей таблицы и частоты кадров на границах GOP на протяжении всего видео. Это позволяет адаптировать видеопоток со временем, как мы обсудим далее.

Каждый GOP определяется GOPHdr, за которым следует набор изображений, составляющих GOP. GOPHdr указывает количество изображений в GOP, а также информацию о синхронизации для GOP (то есть когда GOP должен воспроизводиться относительно начала видео). Каждое изображение, в свою очередь, определяется PictureHdr и набором срезов, составляющих изображение. (Срез — это область изображения, такая как одна горизонтальная линия.) PictureHdr идентифицирует тип изображения (I, B или P) и определяет конкретную для изображения квантующую таблицу. SliceHdr указывает вертикальное положение данного среза, а также дает еще одну возможность изменить квантующую таблицу — на этот раз с помощью постоянного коэффициента масштабирования, а не предоставляя целую новую таблицу. Далее за SliceHdr следует последовательность макроблоков. Наконец, каждый макроблок включает заголовок, который указывает адрес блока внутри изображения, а также данные для шести блоков внутри макроблока: одного для компонента U, одного для компонента V и четырех для компонента Y. (Напомним, что компонент Y имеет размер 16×16 , в то время как компоненты U и V — 8×8 .)

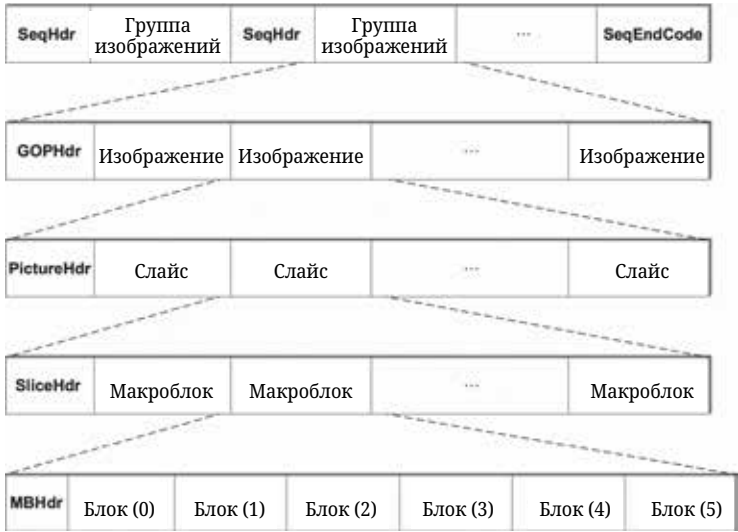


Рисунок 7.16. Формат MPEG-сжатого видеопотока.

Очевидно, что одной из сильных сторон формата MPEG является то, что он предоставляет кодеру возможность изменять кодирование со временем. Можно изменять частоту кадров, разрешение, сочетание типов кадров, которые определяют GOP, квантующую таблицу и кодирование, используемое для отдельных макроблоков. В результате можно адаптировать скорость передачи видео по сети, меняя качество изображения в зависимости от пропускной способности сети. То, как именно сетевой протокол может использовать эту адаптивность, до сих пор исследуется.

Еще один интересный аспект передачи MPEG-потока по сети заключается в том, как именно поток разбивается на пакеты. Если передача идет по соединению TCP, паке-

тирование не является проблемой; TCP решает, когда у него достаточно байтов, чтобы отправить следующую IP-дейтаграмму. Однако при использовании видео в интерактивном режиме его редко передают по TCP, поскольку у TCP есть несколько функций, которые плохо подходят для приложений с высокой чувствительностью к задержкам (например, резкие изменения скорости после потери пакета и повторная передача потерянных пакетов). Если мы передаем видео с использованием UDP, то имеет смысл разрывать поток в тщательно выбранных точках, таких как границы макроблоков. Это потому, что мы хотели бы ограничить влияние потерянного пакета на один макроблок, а не повреждать несколько макроблоков одной потерей. Это пример кадрирования на уровне приложений, о котором говорилось в предыдущей главе.

Пакетирование потока — это только первая проблема при передаче видео, сжатого с помощью MPEG, по сети. Следующая сложность заключается в обработке потерь пакетов. С одной стороны, если сеть потеряет В-фрейм, можно просто воспроизвести предыдущий кадр без серьезного ущерба для видео; потеря 1 кадра из 30 не является большой проблемой. С другой стороны, потеря I-фрейма имеет серьезные последствия — ни один из последующих В- и Р-фреймов не может быть обработан без него. Таким образом, потеря I-фрейма приведет к потере нескольких кадров видео. Хотя можно передать заново потерянный I-фрейм, задержка, скорее всего, будет неприемлемой в режиме реального времени, например, на видеоконференции. Одним из решений этой проблемы является использование методов дифференцированных сервисов, описанных в предыдущей главе, чтобы пометить пакеты, содержащие I-фреймы, с меньшей вероятностью их отбрасывания по сравнению с другими пакетами.

Последнее замечание заключается в том, что выбор способа кодирования видео зависит не только от доступной пропускной способности сети, но и от требований к задержке приложения. Опять же интерактивные приложения, такие как видеоконференции, требуют малых задержек. Критическим фактором является комбинация I-, Р- и В-фреймов в GOP. Рассмотрим следующий GOP:

I B B B P B B B I

Проблема, которую этот GOP вызывает для приложения видеоконференции, заключается в том, что отправитель должен задерживать передачу четырех В-фреймов до тех пор, пока не станет доступным следующий Р- или I-фрейм. Это потому, что каждый В-фрейм зависит от последующего Р- или I-фрейма. Если видео воспроизводится со скоростью 15 кадров в секунду (то есть один кадр каждые 67 мс), это означает, что первый В-фрейм задерживается на 4×67 мс, что составляет более четверти секунды. Эта задержка добавляется к любой задержке распространения, вызванной сетью. Четверть секунды — это гораздо больше, чем порог в 100 мс, который могут воспринимать люди. Именно по этой причине многие приложения видеоконференций кодируют видео с использованием JPEG, который часто называют motion-JPEG. (Motion-JPEG также решает проблему отбрасывания опорного кадра, поскольку все кадры могут быть самостоятельными) Однако заметьте, что межкадровое кодирование, зависящее только от предыдущих кадров, а не от последующих, не является проблемой. Таким образом, GOP вида

I P P P P I

прекрасно подойдет для проведения интерактивных видеоконференций.

Адаптивная потоковая передача

Поскольку схемы кодирования, такие как MPEG, позволяют балансировать между потребляемой пропускной способностью и качеством изображения, возникает возможность адаптировать видеопоток в соответствии с доступной пропускной способностью сети. Это фактически то, что сегодня делают такие сервисы, как Netflix.

Для начала предположим, что у нас есть способ измерять количество свободной емкости и уровень перегрузки на пути, например, наблюдая за скоростью, с которой пакеты успешно доходят до пункта назначения. По мере колебания доступной пропускной способности мы можем передавать эту информацию обратно на кодек, чтобы он настраивал свои параметры кодирования, уменьшая нагрузку во время перегрузок и посылая данные более агрессивно (с более высоким качеством изображения), когда сеть свободна. Это аналогично поведению TCP, за исключением того, что в случае видео мы фактически изменяем общий объем отправляемых данных, а не время, необходимое для отправки фиксированного объема данных, поскольку мы не хотим вводить задержку в видеоприложение.

В случае сервисов видео по запросу, таких как Netflix, мы не адаптируем кодирование на лету, а вместо этого заранее кодируем видео на нескольких уровнях качества и сохраняем их в файлах с соответствующими именами. Получатель просто изменяет имя запрашиваемого файла в соответствии с качеством, которое, по его измерениям, сеть сможет доставить. Получатель следит за своей очередью воспроизведения и запрашивает кодирование более высокого качества, когда очередь становится слишком заполненной, и кодирование более низкого качества, когда очередь становится слишком пустой.

Как этот подход определяет, куда в фильме нужно перейти при изменении запрашиваемого качества? По сути, получатель никогда не просит отправителя передавать весь фильм, вместо этого он запрашивает последовательность коротких сегментов фильма, обычно длительностью в несколько секунд (и всегда на границе GOP). Каждый сегмент предоставляет возможность изменить уровень качества в соответствии с возможностями сети. (Оказывается, запрос кусочков фильма также упрощает реализацию различных режимов воспроизведения, таких как прыжки по фильму из одного места в другое.) Другими словами, фильм обычно хранится как набор из $N \times M$ кусочков (файлов): N уровней качества для каждого из M сегментов.

Отметим один последний нюанс. Поскольку получатель фактически запрашивает последовательность отдельных видеокусков по имени, наиболее распространенный способ делать эти запросы — использовать HTTP. Каждый кусочек — это отдельный HTTP GET запрос с URL, идентифицирующим конкретный кусочек, который хочет получить получатель. Когда вы начинаете скачивать фильм, ваш видеоплеер сначала загружает файл *манифеста*, содержащий только URL для $N \times M$ кусочков в фильме, а затем он посылает последовательность HTTP-запросов, используя соответствующий URL для данной ситуации. Этот общий подход называется *адаптивным потоковым видео по HTTP*, хотя он был стандартизирован по-разному различными организациями, наиболее известными из которых являются DASH (*Dynamic Adaptive Streaming over HTTP*) от MPEG и HLS (*HTTP Live Streaming*) от Apple.

Глава 7.2.5. Сжатие аудиоданных (MP3)

MPEG определяет не только то, как сжимается видео, но также и стандарт для сжатия аудио. Этот стандарт можно использовать для сжатия аудиочасти фильма (в этом случае стандарт MPEG определяет, как сжатое аудио интерливируется со сжатым видео в один поток MPEG), или для сжатия отдельно стоящего аудио (например, аудио CD).

Чтобы понять сжатие аудио, нужно начать с данных. Аудио CD-качества, которое является *фактическим* стандартом для высококачественного аудио, оцифровывается с частотой 44.1 кГц (то есть образец собирается примерно каждые 23 мкс). Каждый образец имеет 16 бит, а это означает, что стерео (2-канальный) аудиопоток имеет битрейт

$$2 \times 44,1 \times 1000 \times 16 = 1,41 \text{ Мбит/с}$$

Для сравнения, голос телефонного качества оцифровывается с частотой 8 кГц, с 8-битными образцами, что дает битрейт 64 кбит/с.

Очевидно, что потребуется некоторое сжатие, чтобы передать аудио CD-качества по, скажем, 128-килобитной линии ISDN. Ситуация усугубляется тем, что синхронизация и исправление ошибок требуют использования 49 бит для кодирования каждого 16-битного образца, что дает реальный битрейт

$$49/16 \times 1,41 \text{ М бps} = 4,32 \text{ Мбит/с}$$

MPEG решает эту задачу, определяя три уровня сжатия, перечисленные в табл. 7.2. Из них наиболее широко используется Слой III, более известный как MP3.

Таблица 7.2.
Степени сжатия MP3.

Кодирование	Скорость передачи битов	Коэффициент сжатия
Слой I	384 кбит/с	14
Слой II	192 кбит/с	18
Слой III	128 кбит/с	12

Для достижения этих коэффициентов сжатия MP3 использует техники, схожие с теми, которые MPEG применяет для сжатия видео. Во-первых, аудиопоток делится на несколько полос частот, что аналогично обработке компонент Y, U и V в видеопотоке MPEG отдельно. Во-вторых, каждая полоса делится на последовательность блоков, которые схожи с макроблоками MPEG, за исключением того, что их длина может варьироваться от 64 до 1024 образцов. (Алгоритм кодирования может изменять размер блока в зависимости от определенных искажений, выходящих за рамки нашего обсуждения.) Наконец, каждый блок преобразуется с использованием модифицированного алгоритма DCT, квантуется и кодируется методом Хаффмана, как и в видео MPEG.

Секрет MP3 заключается в количестве полос, которые он выбирает для использования, и количестве бит, выделяемых каждой полосе, учитывая, что он пытается создать аудио наивысшего качества для заданного битрейта. Точное распределение определяется психоакустическими моделями, выходящими за рамки данной книги, но для иллюстрации можно сказать, что логично выделять больше бит низкочастотным полосам при сжатии мужского голоса и больше бит высокочастотным полосам при сжатии женского голоса. Операционно MP3 динамически изменяет таблицы квантования, используемые для каждой полосы, чтобы достичь желаемого эффекта.

После сжатия полосы упаковываются в блоки фиксированного размера, и к ним добавляется заголовок. Этот заголовок включает информацию о синхронизации, а также информацию о распределении бит, необходимую декодеру для определения количества бит, использованных для кодирования каждой полосы. Как упоминалось выше, эти аудио-блоки затем могут быть интерлированы с видеокадрами для формирования полного MPEG-потока. Один интересный побочный момент заключается в том, что хотя можно потерять В-фреймы в сети в случае перегрузки, опыт показывает, что не стоит терять аудио-блоки, поскольку пользователи лучше переносят плохое видео, чем плохое аудио.

Перспектива: большой объем данных и аналитика

Этот раздел посвящен данным, и поскольку ни одна тема в информатике не привлекает столько внимания, как *большой объем данных* (или, альтернативно, *аналитика данных*), естественно возникает вопрос, какое отношение может иметь большой объем данных к компьютерным сетям. Хотя этот термин часто используется неформально в популярных СМИ, рабочее определение довольно простое: данные с сенсоров собираются путем мониторинга какой-либо физической или созданной человеком системы, а затем

анализируются для получения информации с использованием статистических методов машинного обучения. Поскольку объем собранных необработанных данных часто огромен, применяется прилагательное «большой». Итак, есть ли какие-то последствия для сетей?

На первый взгляд, сети специально спроектированы таким образом, чтобы быть независимыми от данных. Если вы их собираете и хотите отправить куда-то для анализа, сеть с радостью выполнит это для вас. Вы можете сжать данные, чтобы уменьшить требуемую для передачи полосу пропускания, но в остальном большой объем данных ничем не отличается от обычного объема данных. Но данное высказывание игнорирует два важных фактора.

Первый — это то, что, хотя сети не волнует смысл данных (то есть что представляют собой биты), они беспокоятся об объеме данных. Это особенно влияет на доступную сеть, которая была разработана с приоритетом на скорости загрузки над скоростями выгрузки. Такой уклон имеет смысл, когда доминирующий случай использования — это видео, которое передается к конечным пользователям, но в мире, где ваш автомобиль, каждое устройство в вашем доме и дроны, летающие над вашим городом, все отправляют данные обратно в сеть (загружаются в облако), ситуация меняется. На самом деле объем данных, генерируемых автономными транспортными средствами и Интернетом вещей (IoT), потенциально ошеломляющий.

Хотя можно представить решение этой проблемы с использованием одного из алгоритмов сжатия, описанных в главе 7.2, люди вместо этого думают нестандартно и разрабатывают новые приложения, которые работают на границе сети. Эти приложения, работающие на краю сети, обеспечивают как менее чем миллисекундную задержку, так и значительно уменьшают объем данных, которые в конечном итоге нужно загружать в облако. Это уменьшение объема данных можно рассматривать как специфическое для приложения сжатие, но точнее сказать, что приложение на границе сети должно записывать в облако только сводки данных, а не необработанные данные.

Мы уже представили технологии облачных вычислений на границе сети, необходимые для поддержки таких приложений, в конце раздела 2, но, возможно, более интересно рассмотреть примеры этих приложений. Один из таких примеров — предприятия в автомобильной, производственной и складской областях все чаще хотят развертывать частные сети 5G для различных случаев использования *физической автоматизации*. Это может быть гараж, где удаленный парковщик паркует ваш автомобиль, или производственная линия, использующая автоматизированных роботов. Общая тема — высокая пропускная способность и низкая задержка подключения роботов к интеллектуальной системе, расположенной поблизости в облаке на границе сети. Это снижает затраты на роботов (не нужно оснащать каждого из них мощными вычислительными ресурсами) и позволяет масштабируемо координировать работу роя роботов.

Еще один наглядный пример — *носимые устройства для когнитивной помощи*. Идея в том, чтобы обобщить то, что делает для нас навигационное ПО: оно использует один датчик (GPS), дает пошаговые инструкции по сложной задаче (передвижение по незнакомому городу), быстро выявляет наши ошибки и помогает нам их исправить. Можно ли обобщить эту метафору? Мог бы человек, носящий устройство (например, Google Glass, Microsoft Hololens), получать пошаговые инструкции по сложной задаче, возможно, выполняемой впервые? Система будет действовать как «ангел на плече». Все датчики на устройстве (например, видео, аудио, акселерометр, гироскоп) передаются по беспроводной связи (возможно, после некоторой предварительной обработки устройством) на ближайший облачный сервер на границе сети, который выполняет основную обработку. Это метафора «человек в петле», с «внешним видом дополненной реальности», но реализованная с помощью алгоритмов искусственного интеллекта (например, компьютерное зрение, распознавание естественного языка).

Второй фактор заключается в том, что поскольку сеть подобна многим другим созданным человеком системам, можно собирать данные о ее поведении (например, производительность, сбои, модели трафика), применять аналитические программы к этим дан-

ным и использовать полученные инсайды для улучшения сети. Неудивительно, что это активная область исследований, целью которой является построение замкнутого контура управления. Оставив в стороне саму аналитику, которая выходит за рамки этой книги, отметим, что интересные вопросы заключаются в следующем: (1) какие полезные данные мы можем собрать и (2) какие аспекты сети наиболее перспективны для управления? Рассмотрим два перспективных ответа.

Один из них — это сотовые сети 5G, которые по своей сути комплексны. Они включают в себя несколько уровней виртуальных функций, виртуальные и физические активы RAN, использование спектра и, как мы уже обсуждали, узлы вычислений на периферии. Ожидается, что аналитика сети будет необходима для построения гибкой сети 5G. Это будет включать сетевое планирование, которое должно будет решать, где масштабировать конкретные сетевые функции и прикладные сервисы на основе алгоритмов машинного обучения, анализирующих использование сети и шаблоны данных трафика.

Второй — это *внутриполосная телеметрия сети* (In-band Network Telemetry, INT), структура для сбора и передачи состояния сети непосредственно в канальном уровне. Это контрастирует с традиционной отчетностью, выполняемой на уровне управления сетью, как это описано в примерах систем, приведенных в главе 9.3. В архитектуре INT пакеты содержат поля заголовка, которые интерпретируются сетевыми устройствами как «телеметрические инструкции». Эти инструкции говорят устройству с поддержкой INT, какое состояние собирать и записывать в пакет, когда он проходит через сеть. *Источники трафика* INT (например, приложения, сетевые стеки конечных хостов, гипервизоры VM) могут встраивать инструкции либо в обычные данные пакетов, либо в специальные пробные пакеты. Аналогично *приемники трафика* INT извлекают (и при необходимости сообщают) собранные результаты этих инструкций, что позволяет им отслеживать точное состояние канального уровня, которое пакеты «наблюдали» во время пересылки. INT находится на ранней стадии и использует программируемые конвейеры, описанные в главе 3.5, но имеет потенциал для предоставления более глубоких инсайдов в шаблоны трафика и первопричины сбоев сети.

Раздел 8.

Безопасность сети

Истинное величие состоит в том, чтобы обладать человеческой хрупкостью и божественной безопасностью.

Сенека

Проблема: атаки на систему безопасности

Компьютерные сети, как правило, являются общим ресурсом, используемым многими приложениями, представляющими разные интересы. Интернет особенно широко используется, его используют конкурирующие бизнесы, враждующие правительства и преступники. Если не принимать меры безопасности, сетевая беседа или распределенное приложение могут быть скомпрометированы злоумышленником.

Рассмотрим, например, некоторые угрозы безопасному использованию сети. Предположим, вы клиент, использующий кредитную карту для заказа товара с веб-сайта. Очевидная угроза заключается в том, что злоумышленник будет подслушивать ваше сетевое общение, читая ваши сообщения, чтобы получить информацию о вашей кредитной карте. Как может быть осуществлено это подслушивание? Это тривиально в широкоэвещательной сети, такой как Ethernet или Wi-Fi, где любой узел может быть настроен на прием всего сетевого трафика. Более сложные подходы включают перехват и установку шпионского программного обеспечения на любой из цепочек узлов, участвующих в передаче данных. Только в самых экстремальных случаях (например, национальная безопасность) принимаются серьезные меры для предотвращения такого мониторинга, и Интернет не является одним из таких случаев. Однако возможно и практически зашифровать сообщения, чтобы предотвратить понимание их содержимого злоумышленником. Протокол, который это делает, считается обеспечивающим *конфиденциальность*. Сделав шаг дальше, сокрытие количества или назначения общения называется *конфиденциальностью трафика* — потому что знание лишь того, сколько общения происходит и куда, может быть полезным для злоумышленника в некоторых ситуациях.

Даже при обеспечении конфиденциальности остаются угрозы для клиента веб-сайта. Злоумышленник, не способный прочесть содержимое вашего зашифрованного сообщения, все еще может изменить несколько битов в нем, что приведет к действительному заказу, например, совершенно другого товара или, возможно, 1000 единиц товара. Существуют техники для обнаружения, если не предотвращения, такого вмешательства. Протокол, который обнаруживает такое вмешательство в сообщения, считается обеспечивающим *целостность*.

Еще одной угрозой для клиента является незнание того, что он направляется на поддельный веб-сайт. Это может быть результатом атаки на систему доменных имен (DNS), при которой в DNS-сервер или кеш сервиса имен компьютера клиента вводится ложная информация. Это приводит к преобразованию правильного URL в неправильный IP-адрес — адрес поддельного веб-сайта. Протокол, который гарантирует, что вы действительно общаетесь с тем, с кем думаете, что общаетесь, считается обеспечивающим *аутентификацию*. Аутентификация предполагает целостность, так как бессмысленно говорить, что сообщение пришло от определенного участника, если оно больше не является тем же самым сообщением.

Владелец веб-сайта также может подвергаться атакам. Некоторые веб-сайты были испорчены; к файлам, составляющим содержимое веб-сайта, был получен удаленный доступ, и их содержание было изменено без разрешения. Это проблема *управления доступом*: обеспечение выполнения правил относительно того, кто и что имеет право делать. Веб-сайты также подвергались атакам отказа в обслуживании (DoS), в результате которых потенциальные клиенты не могут получить доступ к веб-сайту, так как он перегружен ложными запросами. Обеспечение определенной степени доступа называется *доступностью*.

Кроме этих проблем Интернет зачастую использовался как средство для развертывания вредоносного кода, обычно называемого *вредоносным ПО* (malware), которое использует уязвимости в конечных системах. Черви, части самовоспроизводящегося кода, распространяющиеся по сетям, известны уже несколько десятилетий и продолжают создавать проблемы, как и их родственники, вирусы, которые распространяются через передачу зараженных файлов. Зараженные машины могут затем объединяться в *бот-неты*, которые могут использоваться для причинения дальнейшего вреда, например, для запуска DoS-атак.

Глава 8.1. Доверие и угрозы

Прежде чем мы перейдем к рассмотрению того, как и почему нужно строить безопасные сети, важно установить одну простую истину: мы неизбежно потерпим неудачу. Это потому, что безопасность в конечном итоге является упражнением в делании предположений о доверии, оценке угроз и снижении рисков. Не существует такой вещи, как идеальная безопасность.

Доверие и угрозы — это две стороны одной медали. Угроза — это потенциальный сценарий сбоя, которого вы пытаетесь избежать при проектировании своей системы, а доверие — это предположение, которое вы делаете о том, как будут вести себя внешние участники и внутренние компоненты, с использованием которых вы строите сеть. Например, если вы передаете сообщение по Wi-Fi на открытом кампусе, вы, вероятно, идентифицируете подслушивающего, который может перехватить сообщение, как угрозу (и примените один из методов, обсуждаемых в этой главе, как контрмеру), но если вы передаете сообщение по оптоволоконной линии между двумя машинами в запертом центре обработки данных, вы можете быть уверены, что этот канал безопасен, и не предпринимать дополнительных шагов.

Вы можете утверждать, что, поскольку у вас уже есть способ защиты передачи данных по Wi-Fi, вы также можете использовать его для защиты волоконного канала, но это предполагает результат анализа затрат и выгод. Предположим, что защита любого сообщения, будь то отправленного по Wi-Fi или оптоволокну, замедляет передачу на 10 % из-за накладных расходов на шифрование. Если вам нужно выжать каждую последнюю унцию производительности из научных вычислений (например, вы пытаетесь смоделировать ураган), и вероятность того, что кто-то проникнет в центр обработки данных, составляет один на миллион (и даже если это произойдет, передаваемые данные имеют малую ценность), тогда у вас будут все основания не защищать оптоволоконный канал передачи данных.

Подобные расчеты происходят постоянно, хотя часто они подразумеваются и не озвучиваются. Например, вы можете использовать самый безопасный в мире алгоритм шифрования для сообщения перед его передачей, но вы по умолчанию доверяете, что сервер, на котором вы работаете, добросовестно выполняет этот алгоритм и не передает копию вашего незашифрованного сообщения злоумышленнику. Рассматриваете ли вы это как угрозу или доверяете, что сервер не ведет себя ненадлежащим образом? В конце концов, лучшее, что вы можете сделать, это снизить риск: выявить те угрозы, которые можно устранить экономически эффективно, и явно обозначить, какие предположения о доверии вы делаете, чтобы не быть застигнутым врасплох изменяющимися обстоятельствами, такими как все более решительный или изощренный злоумышленник.

В данном конкретном примере угроза компрометации сервера злоумышленником стала вполне реальной, так как все больше наших вычислений перемещается с локальных серверов в облако, и поэтому сейчас ведутся исследования по созданию базы *доверенных вычислений* (Trusted Computing Base, TCB), интересной темы, но относящейся к области компьютерной архитектуры, а не компьютерных сетей. Для целей данного раздела наша рекомендация — обращать внимание на слова «*доверие*» и «*угроза*» (или «*злоумышленник*»), так как они являются ключевыми для понимания контекста, в котором делаются заявления о безопасности.

Есть одна историческая заметка, которая помогает подготовить почву для этого обсуждения. Интернет (и ARPANET до него) финансировался Министерством обороны США, организацией, которая, безусловно, понимает анализ угроз. Первоначальная оценка была сосредоточена на проблемах выживания сети в условиях отказов (или уничтожения) маршрутизаторов и сетей, что объясняет, почему алгоритмы маршрутизации децентрализованы, без единой точки отказа. С другой стороны, первоначальный дизайн предполагал, что все участники внутри сети доверяемы, и поэтому мало или совсем не уделялось внимания тому, что сегодня мы называем кибербезопасностью (атаки со стороны злоумышленников, которые могут подключиться к сети). Это означает, что многие из инструментов, описанных в этом разделе, можно считать «заплатками». Они прочно основаны на криптографии, но все же являются «дополнениями». Если бы происходил всеобъемлющий редизайн Интернета, интеграция безопасности, вероятно, стала бы главной движущей силой.

Глава 8.2. Криптографические строительные блоки

Мы вводим концепции безопасности на основе криптографии шаг за шагом. Первый шаг — это криптографические алгоритмы — шифры и криптографические хеши, которые вводятся в этом разделе. Сами по себе они не являются решением, а скорее «строительными блоками», из которых можно построить решение. Криптографические алгоритмы параметризуются ключами, и в следующей главе рассматривается проблема распределения ключей. На следующем шаге мы описываем, как включить криптографические строительные блоки в протоколы, которые обеспечивают безопасную связь между участниками, обладающими правильными ключами. В последней главе рассматривается несколько полных протоколов и систем безопасности, используемых в настоящее время.

Глава 8.2.1. Принципы шифров

Шифрование преобразует сообщение таким образом, что оно становится непонятным для любой стороны, не имеющей секрета о том, как обратить это преобразование. Отправитель применяет функцию *шифрования* к исходному открытому тексту сообщения, в результате чего получается *шифротекст*, который отправляется по сети, как показано на рис. 8.1. Получатель применяет секретную функцию *дешифрования* — обратную функцию шифрования, — чтобы восстановить исходный открытый текст. Шифротекст, передаваемый по сети, непонятен для любого подслушивающего, если подслушивающий не знает функцию дешифрования. Преобразование, представленное функцией шифрования и соответствующей ей функцией дешифрования, называется *шифром*.

Криптографы пришли к принципу, впервые заявленному в 1883 году, что функции шифрования и дешифрования должны быть параметризованы *ключом*, и кроме того, что функции должны считаться общедоступными знаниями — только ключ должен быть *секретом*. Таким образом, шифротекст, созданный для данного сообщения на некотором языке, зависит как от функции шифрования, так и от ключа. Одна из причин этого принципа заключается в том, что если вы рассчитываете на то, что шифр остается секретным, то вам придется выводить шифр из эксплуатации (а не только ключи), когда вы поняли, что он больше не секретен. Это означает потенциально частые изменения шифра, что проблематично, так как разработка нового шифра требует много работы. Кроме того, один из лучших способов узнать, что шифр безопасен — это использовать его долгое время: если никто не взломал его, вероятно, он безопасен. Таким образом, существуют значительные затраты и риски при внедрении нового шифра. Наконец, параметризация шифра ключами дает нам, по сути, очень большую семью шифров; переключаясь между ключами, мы фактически переключаем шифры, тем самым ограничивая количество данных, которые *криптоаналитик* (взломщик кода) может использовать для попытки взлома нашего ключа/шифра и количество информации, которое он может прочитать, если ему это удастся.

Основное требование к алгоритму шифрования заключается в том, чтобы он преобразовывал открытый текст в шифротекст таким образом, чтобы только предполагаемый получатель — обладатель ключа дешифрования — мог восстановить открытый текст. Это означает, что зашифрованные сообщения не могут быть прочитаны людьми, не имеющими ключа.

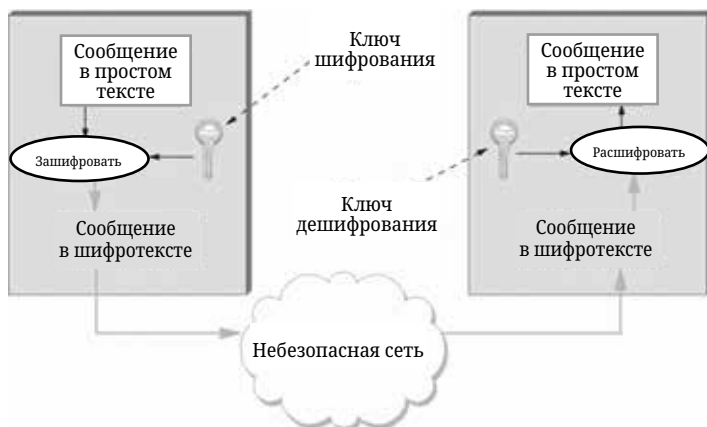


Рисунок 8.1. Шифрование и дешифрование с использованием секретного ключа.

Важно понимать, что когда потенциальный злоумышленник получает часть шифротекста, в его распоряжении может быть больше информации, чем просто сам шифротекст. Например, он может знать, что открытый текст был написан на английском языке, а это значит, что буква *e* встречается в нем чаще, чем любая другая буква; также можно предсказать частоту встречаемости многих других букв и распространенных буквосочетаний. Эта информация может значительно упростить задачу поиска ключа. Аналогично он может знать что-то о вероятном содержании сообщения; например, слово «login», скорее всего, будет встречаться в начале сеанса удаленного входа в систему. Это может позволить провести атаку с *известным открытым текстом*, которая имеет гораздо больше шансов на успех, чем атака *только с шифротекстом*. Еще лучше атака с *выбором открытого текста*, которая может быть осуществлена путем передачи отправителю некоторой информации, которую, как вы знаете, отправитель, скорее всего, передаст — такое случилось, например, в военное время.

Поэтому лучшие криптографические алгоритмы могут предотвратить определение ключа атакующим, даже если он знает и открытый текст, и шифротекст. Это оставляет атакующему только один выход — попробовать все возможные ключи — исчерпывающий, «грубой силы» поиск. Если у ключей n бит, то существует 2^n возможных значений ключа (каждый из n бит может быть либо нулем, либо единицей). Атакующему может повезти и он попробует правильное значение сразу же, или же ему может не повезти и он попробует все неправильные значения, прежде чем наконец-то подберет правильное значение ключа, попробовав все 2^n возможных значений; среднее число догадок для нахождения правильного значения находится между этими крайностями, $2^{n/2}$. Это можно сделать вычислительно непрактичным, выбрав достаточно большое пространство ключей и сделав операцию проверки ключа достаточно затратной. Что усложняет задачу, так это то, что вычислительные скорости продолжают расти, делая ранее неосуществимые вычисления осуществимыми. Кроме того, хотя мы делаем акцент на безопасности данных при их передаче по сети — то есть данные иногда уязвимы только в течение короткого времени, — в общем, специалисты по безопасности должны учитывать уязвимость данных, которые необходимо хранить в архивах в течение десятилетий. Это

аргумент в пользу большого размера ключа. С другой стороны, большие ключи делают шифрование и дешифрование более медленными.

Большинство шифров являются *блочными* шифрами; они определяются как принимающие на вход блок открытого текста фиксированного размера, обычно от 64 до 128 бит. Использование блочного шифра для шифрования каждого блока независимо, известное как режим шифрования *электронной кодовой книги* (Electronic Codebook, ECB), имеет уязвимость: одинаковое значение блока открытого текста всегда приведет к одинаковому блоку шифротекста. Таким образом, повторяющиеся значения блоков в открытом тексте будут распознаваемы как таковые в шифротексте, что значительно упрощает криптоанализу задачу взлома шифра.

Чтобы предотвратить это, блочные шифры всегда дополняются, чтобы сделать шифротекст блока варьирующимся в зависимости от контекста. Способы, которыми можно дополнить блочный шифр, называются режимами работы. Один из распространенных режимов работы — *цепочное шифрование блоков* (Cipher Block Chaining, CBC), при котором каждый блок открытого текста подвергается операции XOR с шифротекстом предыдущего блока перед шифрованием. В результате шифротекст каждого блока частично зависит от предыдущих блоков (то есть от его контекста). Поскольку первый блок открытого текста не имеет предыдущего блока, он подвергается операции XOR со случайным числом. Это случайное число, называемое вектором инициализации (Initialization Vector, IV), включается в серию блоков шифротекста, чтобы можно было расшифровать первый блок шифротекста. Этот режим показан на рис. 8.2. Другой режим работы — *счетный режим*, при котором в шифрование последовательных блоков открытого текста включаются последовательные значения счетчика (например, 1, 2, 3 и т.д.).

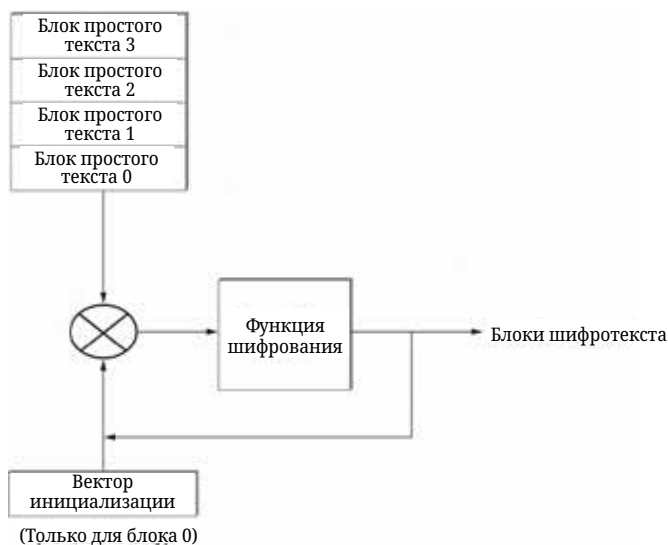


Рисунок 8.2. Цепочка блоков шифра.

Глава 8.2.2. Шифры с секретными ключами

В шифре с секретным ключом оба участника коммуникации используют один и тот же ключ.¹ Другими словами, если сообщение зашифровано с использованием определенного ключа, то для расшифровки сообщения требуется тот же самый ключ. Если бы шифр,

¹ Мы используем термин «участник» для обозначения сторон, участвующих в безопасном обмене данными, поскольку именно этот термин мы использовали на протяжении всей книги для обозначения двух конечных точек канала. В мире безопасности их обычно называют *принципалами*.

показанный на рис. 8.1, был шифром с секретным ключом, то ключи шифрования и дешифрования были бы идентичны. Шифры с секретным ключом также известны как симметричные шифры, поскольку секрет разделяется между обоими участниками. Вскоре мы рассмотрим альтернативу — шифры с открытым ключом. (Шифры с открытым ключом также известны как асимметричные шифры, поскольку, как мы скоро увидим, оба участника используют разные ключи.)

Национальный институт стандартов и технологий США (NIST) выпустил стандарты для ряда шифров с секретным ключом. Первым был *стандарт шифрования данных* (Data Encryption Standard, DES), который выдержал проверку временем, поскольку не было обнаружено криптоаналитической атаки, лучшей, чем перебор. Однако перебор стал быстрее. Ключи DES (56 независимых бит) теперь слишком малы, учитывая текущие скорости процессоров. Ключи DES имеют 56 независимых бит (хотя всего они имеют 64 бита; последний бит каждого байта — это бит четности). Как отмечалось выше, в среднем нужно было бы перебирать половину пространства из 256 возможных ключей, чтобы найти правильный, что дает $255 = 3,6 \times 10^{16}$ ключей. Это может показаться большим числом, но такой поиск легко параллелизуется, так что можно использовать столько компьютеров, сколько можно достать — а в наши дни легко арендовать тысячи компьютеров. (Amazon сдает их в аренду за несколько центов в час.) К концу 1990-х годов стало возможно восстановить ключ DES за несколько часов. Следовательно, NIST обновил стандарт DES в 1999 году, указав, что DES следует использовать только для устаревших систем.

NIST также стандартизировал шифр Triple DES (3DES), который использует криптоаналитическую стойкость DES, увеличивая размер ключа. Ключ 3DES имеет 168 ($= 3 \times 56$) независимых бит и используется как три ключа DES; назовем их DES-ключ1, DES-ключ2 и DES-ключ3. Шифрование 3DES блока выполняется сначала шифрованием DES с использованием DES-ключ1, затем дешифрованием результата с использованием DES-ключ2 и, наконец, шифрованием этого результата с использованием DES-ключ3. Дешифрование включает дешифрование с использованием DES-ключ3, затем шифрование с использованием DES-ключ2, затем дешифрование с использованием DES-ключ1.

Причина, по которой шифрование 3DES использует дешифрование с DES-ключ2, заключается в том, чтобы быть совместимым с устаревшими системами DES. Если устаревшая система DES использует один ключ, то система 3DES может выполнить ту же функцию шифрования, используя этот ключ для каждого из DES-ключ1, DES-ключ2 и DES-ключ3; на первых двух шагах мы шифруем и затем дешифруем с одним и тем же ключом, получая исходный открытый текст, который затем шифруем снова.

Хотя 3DES решает проблему длины ключа DES, он наследует некоторые другие недостатки. Программные реализации DES/3DES медленны, поскольку DES изначально разрабатывался IBM для реализации в аппаратном обеспечении. Кроме того, DES/3DES использует размер блока в 64 бита; больший размер блока более эффективен и безопасен.

3DES сейчас вытесняется стандартом *Advanced Encryption Standard* (AES), выпущенным NIST. Шифр, лежащий в основе AES (с небольшими модификациями), изначально назывался Rijndael (произносится примерно как «Райн дал») по именам его создателей, Деймэна и Реймена. AES поддерживает длину ключей в 128, 192 или 256 бит, а длина блока составляет 128 бит. AES допускает быструю реализацию как в программном, так и в аппаратном обеспечении. Он не требует много памяти, что делает его подходящим для небольших мобильных устройств. AES обладает некоторыми математически доказанными свойствами безопасности и на момент написания книги не подвергался значительным успешным атакам.

Глава 8.2.3. Шифры с открытым ключом

Альтернативой шифрам с секретным ключом являются шифры с открытым ключом. Вместо одного ключа, разделяемого двумя участниками, шифр с открытым ключом использует пару связанных ключей, один для шифрования и другой для дешифрования.

Пара ключей «принадлежит» только одному участнику. Владелец держит ключ дешифрования в секрете, чтобы только он мог расшифровывать сообщения; этот ключ называется *закрытым ключом*. Владелец делает *ключ шифрования открытым*, так что любой может шифровать сообщения для владельца; этот ключ называется открытым. Очевидно, что для работы такой схемы необходимо, чтобы из открытого ключа нельзя было вывести закрытый. Следовательно, любой участник может получить открытый ключ и отправить зашифрованное сообщение владельцу ключей, и только владелец обладает закрытым ключом, необходимым для его расшифровки. Этот сценарий изображен на рис. 8.3.

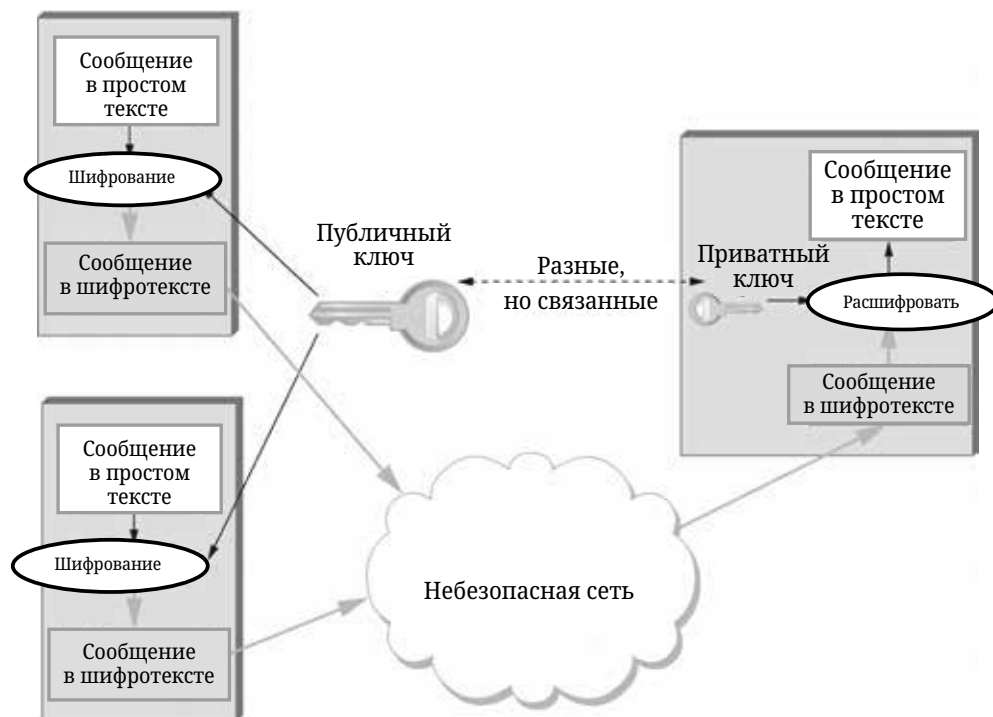


Рисунок 8.3. Шифрование с открытым ключом.

Поскольку это несколько неочевидно, мы подчеркиваем, что публичный ключ шифрования бесполезен для расшифровки сообщения — вы не сможете расшифровать сообщение, которое только что зашифровали, если у вас нет приватного ключа расшифровки. Если рассматривать ключи как определяющие канал связи между участниками, то еще одно отличие шифров с открытым ключом и с секретным ключом заключается в топологии каналов. Ключ для шифра с секретным ключом обеспечивает двусторонний канал между двумя участниками — каждый участник имеет один и тот же (симметричный) ключ, который может использоваться для шифрования или расшифровки сообщений в любом направлении. Пара публичного и приватного ключей, напротив, обеспечивает канал, который является односторонним и многоточечным: от всех, у кого есть публичный ключ, к единственному владельцу приватного ключа, как показано на рис. 8.3.

Важное дополнительное свойство шифров с открытым ключом заключается в том, что приватный «дешифровочный» ключ можно использовать с алгоритмом шифрования для шифрования сообщений, так что их можно расшифровать только с помощью публич-

ного «шифровочного» ключа. Это свойство явно не полезно для конфиденциальности, поскольку любой с публичным ключом сможет расшифровать такое сообщение. (Действительно, для двусторонней конфиденциальности между двумя участниками каждому из них нужна своя пара ключей, и каждый шифрует сообщения, используя публичный ключ другого.) Это свойство, однако, полезно для аутентификации, поскольку оно говорит получателю такого сообщения, что оно могло быть создано только владельцем ключей (при условии выполнения определенных предположений, которые мы рассмотрим позже). Это иллюстрируется на рис. 8.4. Из рисунка ясно, что любой с публичным ключом может расшифровать зашифрованное сообщение и, предполагая, что результат расшифровки совпадает с ожидаемым, можно сделать вывод, что для шифрования использовался приватный ключ. Как именно эта операция используется для обеспечения аутентификации, будет рассмотрено в следующей главе. Как мы увидим, шифры с открытым ключом используются в основном для аутентификации и для конфиденциальной передачи секретных (симметричных) ключей, оставляя остальную часть конфиденциальности шифрам с секретным ключом.

Немного интересной истории: концепция шифров с открытым ключом была впервые опубликована в 1976 году Диффи и Хеллманом. Однако впоследствии стали известны документы, доказывающие, что британская Служба коммуникационной электронной безопасности (Communications-Electronics Security Group) открыла шифры с открытым ключом к 1970 году, а Агентство национальной безопасности США (NSA) утверждает, что открыло их в середине 1960-х годов.

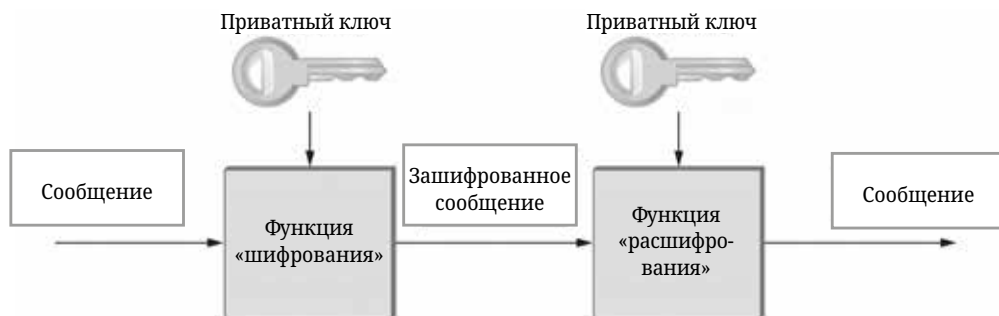


Рисунок 8.4. Аутентификация с использованием открытых ключей.

Самый известный шифр с открытым ключом — это RSA, названный в честь своих изобретателей: Ривеста, Шамира и Адлемана. RSA опирается на высокую вычислительную сложность факторизации больших чисел. Проблема нахождения эффективного способа факторизации чисел является такой, над которой математики безуспешно работали задолго до появления RSA в 1978 году, а последующая стойкость RSA к криптоанализу еще больше укрепила уверенность в его безопасности. К сожалению, RSA требует относительно больших ключей, по крайней мере 1024 бит, чтобы быть безопасным. Это больше, чем ключи для шифров с секретным ключом, потому что взлом приватного ключа RSA путем факторизации большого числа, на котором основана пара ключей, быстрее, чем исчерпывающий поиск по пространству ключей.

Другой шифр с открытым ключом — это Эль-Гамаль. Как и RSA, он опирается на математическую проблему — задачу дискретного логарифма, для которой не найдено эффективного решения, и требует ключей размером как минимум 1024 бита. Существует вариант задачи дискретного логарифма, возникающий при использовании эллиптической кривой, который считается еще более трудным для вычисления; криптографические схемы, основанные на этой проблеме, называются *криптографией на эллиптических кривых*.

Шифры с открытым ключом, к сожалению, на несколько порядков медленнее, чем шифры с секретным ключом. Следовательно, шифры с секретным ключом используются для подавляющего большинства шифрования, в то время как шифры с открытым ключом зарезервированы для использования в аутентификации и установлении сеансовых ключей.

Глава 8.2.4. Аутентификаторы

Шифрование само по себе не обеспечивает целостность данных. Например, случайное изменение зашифрованного сообщения может привести к его дешифровке в корректно выглядящий текст, и тогда вмешательство останется незамеченным для получателя. Также шифрование само по себе не обеспечивает аутентификацию. Нет смысла утверждать, что сообщение пришло от определенного участника, если его содержимое было изменено после того, как участник его создал. В некотором смысле целостность и аутентификация неразрывно связаны.

Аутентификатор — это значение, включаемое в передаваемое сообщение, которое может быть использовано для одновременной проверки подлинности и целостности данных сообщения. Мы рассмотрим, как аутентификаторы могут использоваться в протоколах. Сейчас мы сосредоточимся на алгоритмах, которые производят аутентификаторы.

Возможно, вы помните, что контрольные суммы и циклические избыточные проверки (CRC) — это информация, добавляемая к сообщению, чтобы получатель мог обнаружить, если сообщение было случайно изменено из-за ошибок бит. Похожая концепция применяется к аутентификаторам, с добавлением вызова, что повреждение сообщения вероятнее всего будет намеренно совершено кем-то, кто хочет, чтобы повреждение осталось незамеченным. Для поддержки аутентификации аутентификатор включает доказательство того, что создатель аутентификатора знает секрет, известный только предполагаемому отправителю сообщения; например, секретом может быть ключ, а доказательством — значение, зашифрованное с использованием ключа. Существует взаимная зависимость между формой избыточной информации и формой доказательства знания секрета. Мы рассмотрим несколько рабочих комбинаций.

Изначально предполагается, что исходное сообщение не должно быть конфиденциальным — передаваемое сообщение будет состоять из открытого текста исходного сообщения плюс аутентификатор. Позже мы рассмотрим случай, когда требуется конфиденциальность.

Один вид аутентификатора комбинирует шифрование и *криптографическую хеш-функцию*. Криптографические хеш-алгоритмы рассматриваются как общедоступные знания, как и алгоритмы шифрования. Криптографическая хеш-функция (также известная как *криптографическая контрольная сумма*) — это функция, которая выводит достаточно избыточную информацию о сообщении, чтобы выявить любое вмешательство. Как и контрольная сумма или CRC, криптографическая контрольная сумма предназначена для выявления намеренной порчи сообщений злоумышленником. Значение, которое она выводит, называется *дайджестом сообщения* и, как обычная контрольная сумма, добавляется к сообщению. Все дайджесты сообщений, производимые данным хешем, имеют одинаковое количество битов, независимо от длины исходного сообщения. Поскольку пространство возможных входных сообщений больше пространства возможных дайджестов сообщений, будут разные входные сообщения, которые производят одинаковые дайджесты сообщений, как коллизии в хеш-таблице.

Аутентификатор может быть создан путем шифрования дайджеста сообщения. Получатель вычисляет дайджест части сообщения в открытом тексте и сравнивает его с расшифрованным дайджестом сообщения. Если они равны, то получатель может заключить, что сообщение действительно от предполагаемого отправителя (поскольку оно должно

было быть зашифровано правильным ключом) и не было подделано. Ни один злоумышленник не сможет отправить поддельное сообщение с соответствующим поддельным дайджестом, поскольку у него не будет ключа для правильного шифрования поддельного дайджеста. Однако злоумышленник может получить исходное сообщение в открытом тексте и его зашифрованный дайджест путем подслушивания. Затем злоумышленник может (поскольку хеш-функция является общедоступной) вычислить дайджест исходного сообщения и создать альтернативные сообщения, пытаясь найти такое, у которого будет такой же дайджест сообщения. Если он найдет такое, он сможет незаметно отправить новое сообщение со старым аутентификатором. Поэтому для обеспечения безопасности хеш-функция должна иметь *однаправленное* свойство: для злоумышленника должно быть вычислительно невозможно найти любое сообщение в открытом тексте, имеющее тот же дайджест, что и исходное сообщение.

Для того чтобы хеш-функция соответствовала этому требованию, ее выходные значения должны быть случайно распределены. Например, если длина дайджестов 128 бит и они случайно распределены, то в среднем нужно будет попробовать 2^{127} сообщений, прежде чем будет найдено второе сообщение с таким же дайджестом, как у данного сообщения. Если выходные значения не случайно распределены — то есть некоторые выходные значения более вероятны, чем другие, — то для некоторых сообщений можно будет намного легче найти другое сообщение с таким же дайджестом, что снизит безопасность алгоритма. Если же вы просто пытаетесь найти любую *коллизия* — любые два сообщения, которые производят одинаковый дайджест, — то в среднем нужно будет вычислить дайджесты только 2^{64} сообщений.

Существовало несколько распространенных криптографических хеш-алгоритмов на протяжении многих лет, включая Message Digest 5 (MD5) и семейство Secure Hash Algorithm (SHA). Уязвимости MD5 и более ранних версий SHA были известны уже давно, что привело к рекомендации NIST использовать SHA-3 в 2015 году. При создании зашифрованного дайджеста сообщения для его шифрования можно использовать либо симметричный шифр, либо асимметричный шифр. Если используется асимметричный шифр, дайджест будет зашифрован с помощью приватного ключа отправителя (того, который обычно используется для расшифровки), и получатель — или любой другой — сможет расшифровать дайджест с помощью публичного ключа отправителя.

Дайджест, зашифрованный с использованием асимметричного алгоритма, но с использованием приватного ключа, называется *цифровой подписью*, потому что она обеспечивает неопровержимость, как и письменная подпись. Получатель сообщения с цифровой подписью может доказать любой третьей стороне, что отправитель действительно отправил это сообщение, поскольку третья сторона может использовать публичный ключ отправителя для самостоятельной проверки. (Симметричное шифрование дайджеста не имеет этого свойства, поскольку только два участника знают ключ; более того, поскольку оба участника знают ключ, предполагаемый получатель мог бы сам создать сообщение.) Любой асимметричный шифр может быть использован для цифровых подписей. *Стандарт цифровой подписи* (DSS) — это формат цифровой подписи, стандартизированный NIST. DSS-подписи могут использовать любой из трех асимметричных шифров: один на основе RSA, другой на основе ElGamal и третий, называемый *алгоритмом цифровой подписи на эллиптических кривых*.

Другой вид аутентификатора похож, но вместо шифрования хеша использует хеш-подобную функцию, которая принимает секретное значение (известное только отправителю и получателю) в качестве параметра, как показано на рис. 8.5. Такая функция выводит аутентификатор, называемый *кодом аутентификации сообщения* (MAC). Отправитель добавляет MAC к своему открытому тексту сообщения. Получатель пересчитывает MAC, используя открытый текст и секретное значение, и сравнивает пересчитанный MAC с полученным MAC.

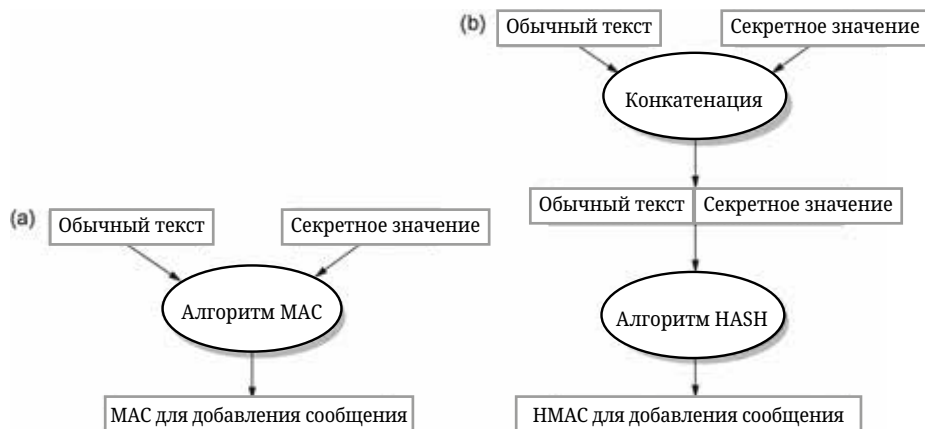


Рисунок 8.5. Вычисление MAC (a) в сравнении с вычислением HMAC (b).

Распространенным вариантом MAC является применение криптографического хеша (например, MD5 или SHA-1) к объединению открытого текста сообщения и секретного значения, как показано на рис. 8.5. Полученный дайджест называется *хешированным кодом аутентификации сообщения* (HMAC), так как по сути это MAC. HMAC, но не секретное значение, добавляется к открытому тексту. Только получатель, который знает секретное значение, может вычислить правильный HMAC для сравнения с полученным HMAC. Если бы не однонаправленное свойство хеша, злоумышленник мог бы найти входные данные, которые сгенерировали HMAC, и сравнить их с открытым текстом сообщения, чтобы определить секретное значение.

До этого момента мы предполагали, что сообщение не является конфиденциальным, поэтому исходное сообщение может передаваться в открытом виде. Чтобы добавить конфиденциальность к сообщению с аутентификатором, достаточно зашифровать объединение всего сообщения, включая его аутентификатор — MAC, HMAC или зашифрованный дайджест. Помните, что на практике конфиденциальность реализуется с использованием симметричных шифров, поскольку они гораздо быстрее, чем асимметричные шифры. Кроме того, включение аутентификатора в шифрование практически ничего не стоит и увеличивает безопасность. Распространенным упрощением является шифрование сообщения с его (сырым) дайджестом, так что дайджест шифруется только один раз; в этом случае все зашифрованное сообщение считается аутентификатором.

Хотя аутентификаторы могут показаться решением проблемы аутентификации, мы увидим в следующей главе, что они лишь основа для решения. Однако сначала мы рассмотрим вопрос о том, как участники изначально получают ключи.

Глава 8.3. Предраспределение ключей

Чтобы использовать шифры и аутентификаторы, участники общения должны знать, какие ключи использовать. В случае симметричного шифра как пара участников получает общий ключ? В случае асимметричного шифра как участники узнают, какой публичный ключ принадлежит определенному участнику? Ответ зависит от того, являются ли ключи краткосрочными *сеансовыми ключами* или более долговечными *предраспределенными ключами*.

Сеансовый (или сессионный) ключ — это ключ, используемый для защиты одного, относительно короткого эпизода связи: сессии (сеанса). Каждая отдельная сессия между парой участников использует новый сеансовый ключ, который всегда является симметричным для скорости. Участники определяют, какой сеансовый ключ использовать, с по-

мощью протокола — протокола установления сеансового ключа. Протокол установления сеансового ключа требует собственной безопасности (чтобы, например, злоумышленник не мог узнать новый сеансовый ключ); эта безопасность основана на более долговечных предраспределенных ключах.

Существует два основных предлога для такого «разделения труда» между сеансовыми ключами и предраспределенными ключами:

- Ограничение времени использования ключа приводит к меньшему времени для вычислительно сложных атак, меньшему количеству шифротекста для криптоанализа и меньшему объему информации, подвергающейся опасности в случае взлома ключа.
- Асимметричные шифры обычно превосходят для аутентификации и установления сеансовых ключей, но слишком медленны для шифрования целых сообщений для обеспечения конфиденциальности.

В следующей главе объясняется, как распределяются предраспределенные ключи, а далее будет объяснено, как затем устанавливаются сеансовые ключи. В дальнейшем мы будем использовать «Алиса» и «Боб» для обозначения участников, как это принято в литературе по криптографии. Имейте в виду, что хотя мы склонны говорить об участниках в антропоморфных терминах, чаще всего нас интересует связь между программными или аппаратными сущностями, такими как клиенты и серверы, которые часто не имеют прямого отношения к какому-либо конкретному человеку.

Глава 8.3.1. Предраспределение публичных ключей

Алгоритмы для генерации пары совпадающих публичного и частного ключей общедоступны, и программное обеспечение, которое это делает, широко доступно. Таким образом, если Алиса захочет использовать асимметричный шифр, она может сгенерировать свою собственную пару публичного и частного ключей, держать частный ключ в секрете и обнародовать публичный ключ. Но как она может обнародовать свой публичный ключ — утверждать, что он принадлежит ей — таким образом, чтобы другие участники могли быть уверены, что он действительно принадлежит ей? Не через электронную почту или веб, потому что злоумышленник мог бы подделать не менее правдоподобное утверждение, что ключ x принадлежит Алисе, тогда как x на самом деле принадлежит злоумышленнику.

Полная схема для сертифицикации связей между публичными ключами и идентичностями — какой ключ кому принадлежит — называется *инфраструктурой публичных ключей* (Public Key Infrastructure, PKI). PKI начинает свои полномочия с возможности проверки идентичности и связывания их с ключами вне полосы. Под термином «вне полосы» мы подразумеваем что-то за пределами сети и компьютеров, которые ее составляют, например, следующее: Если Алиса и Боб знают друг друга, они могут встретиться в одной комнате, и Алиса может передать свой публичный ключ Бобу напрямую, возможно, на визитной карточке. Если Боб представляет организацию, Алиса может предъявить обычное удостоверение личности, возможно, с фотографией или отпечатками пальцев. Если Алиса и Боб — компьютеры, принадлежащие одной компании, то системный администратор может настроить Боба на использование публичного ключа Алисы.

Установление ключей вне полосы звучит как не масштабируемое решение, но этого достаточно, чтобы инициализировать PKI. Знание Бобом, что ключ Алисы — x , может быть широко и масштабируемо распространено с использованием комбинации цифровых подписей и концепции доверия. Например, предположим, что вы получили публичный ключ Боба вне полосы и достаточно знаете о Бобе, чтобы доверять ему в вопросах ключей и идентичностей. Тогда Боб мог бы отправить вам сообщение, утверждающее, что ключ Алисы — x , и — так как вы уже знаете публичный ключ Боба — вы могли бы аутентифицировать сообщение как пришедшее от Боба. (Помните, что для цифровой подписи утверждения Боб добавил бы к нему криптографический хеш, зашифрованный его

частным ключом.) Так как вы доверяете Бобу, вы теперь знаете, что ключ Алисы — х, даже если вы никогда не встречались с ней или не обменивались ни единым сообщением. Используя цифровые подписи, Бобу даже не нужно было бы отправлять вам сообщение; он мог бы просто создать и опубликовать цифровое подписанное заявление, что ключ Алисы — х. Такое цифровое подписанное заявление о привязке публичного ключа называется *сертификатом публичного ключа*, или просто сертификатом. Боб мог бы отправить Алисе копию сертификата или разместить его на веб-сайте. Если когда-нибудь кому-то нужно будет проверить публичный ключ Алисы, он сможет сделать это, получив копию сертификата, возможно, напрямую от Алисы — если он доверяет Бобу и знает его публичный ключ. Вы можете видеть, как, начиная с очень небольшого количества ключей (в данном случае только ключа Боба), можно со временем создать большой набор доверенных ключей. В данном случае Боб играет роль, часто называемую *центром сертификации* (Certification Authority, CA), и большая часть современной интернет-безопасности зависит от удостоверяющих центров. VeriSign — один из известных коммерческих удостоверяющих центров. Мы вернемся к этой теме позже.

Одним из основных стандартов для сертификатов является X.509. Этот стандарт оставляет много деталей открытыми, но определяет базовую структуру. Сертификат должен включать:

- Идентичность сущности, которую сертифицируют
- Публичный ключ сущности, которую сертифицируют
- Идентичность подписанта
- Цифровую подпись
- Идентификатор алгоритма цифровой подписи (криптографический хеш и шифр)

Необязательным компонентом является время истечения срока действия сертификата. Мы увидим конкретное использование этой функции ниже.

Поскольку сертификат создает привязку между идентичностью и публичным ключом, нам следует более внимательно рассмотреть, что мы понимаем под «идентичностью». Например, сертификат, который говорит: «Этот публичный ключ принадлежит Джону Смитту», может оказаться не очень полезным, если вы не можете определить, какой из тысяч Джонов Смитов идентифицируется. Таким образом, сертификаты должны использовать хорошо определенное пространство имен для идентичностей, которые сертифицируются; например, сертификаты часто выдаются для адресов электронной почты и доменов DNS.

Существуют разные способы, которыми PKI может формализовать понятие доверия. Мы обсудим два основных подхода.

Центры сертификации

В этой модели доверие бинарно; вы либо полностью доверяете кому-то, либо не доверяете вообще. Вместе с сертификатами это позволяет строить *цепочки доверия*. Если X сертифицирует, что определенный публичный ключ принадлежит Y, а затем Y сертифицирует, что другой публичный ключ принадлежит Z, то существует цепочка сертификатов от X до Z, даже если X и Z никогда не встречались. Если вы знаете ключ X и доверяете X и Y, то вы можете верить сертификату, который указывает на ключ Z. Другими словами, все, что вам нужно, это цепочка сертификатов, все подписанные сущностями, которым вы доверяете, до тех пор, пока она ведет к сущности, ключ которой вам уже известен.

Удостоверяющий центр (или центр сертификации) (Certification Authority, CA) — это сущность, которая считается (кем-то) доверенной для проверки идентичностей и выдачи сертификатов публичных ключей. Существуют коммерческие удостоверяющие центры, правительственные удостоверяющие центры и даже бесплатные удостоверяющие центры. Чтобы использовать удостоверяющий центр, вы должны знать его собственный ключ. Вы можете узнать ключ удостоверяющего центра, если сможете получить цепочку сертификатов, подписанных удостоверяющим центром, которая начинается с удостове-

ряющего центра, ключ которого вы уже знаете. Тогда вы сможете доверять любому сертификату, подписанному этим новым удостоверяющим центром.

Обычный способ построения таких цепочек — организовать их в древовидную иерархию, как показано на рис. 8.6. Если у всех есть публичный ключ корневого удостоверяющего центра, то любой участник может предоставить цепочку сертификатов другому участнику и быть уверенным, что этого будет достаточно для создания цепочки доверия для этого участника.

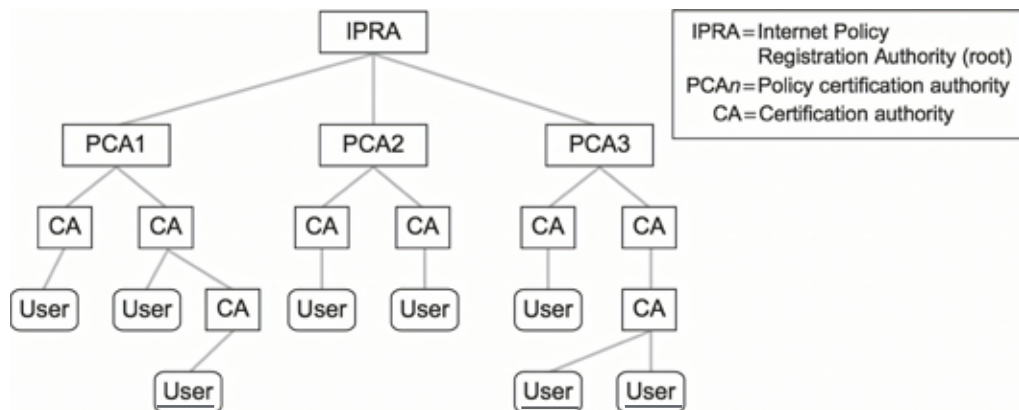


Рисунок 8.6. Древовидная структура иерархии сертификационных центров.

Существует несколько значительных проблем с построением цепочек доверия. Наиболее важно то, что даже если вы уверены, что у вас есть публичный ключ корневого удостоверяющего центра, вы должны быть уверены, что каждый удостоверяющий центр от корня и далее выполняет свою работу должным образом. Если хотя бы один удостоверяющий центр в цепочке готов выдавать сертификаты сущностям без проверки их идентичностей, то цепочка сертификатов, которая выглядит как действительная, становится бессмысленной. Например, корневой удостоверяющий центр может выдать сертификат удостоверяющему центру второго уровня и тщательно проверить, что имя в сертификате соответствует названию удостоверяющего центра, но этот удостоверяющий центр второго уровня может быть готов продать сертификаты любому, кто попросит, без проверки их идентичности. Эта проблема усугубляется с увеличением длины цепочки доверия. Сертификаты X.509 предоставляют возможность ограничивать набор сущностей, которые объект сертификата, в свою очередь, доверен сертифицировать.

В дереве сертификации может быть более одного корня, и это распространено при обеспечении безопасности веб-транзакций сегодня, например. Веб-браузеры, такие как Firefox и Internet Explorer, поставляются с сертификатами для набора удостоверяющих центров; по сути, производитель браузера решил, что этим удостоверяющим центрам и их ключам можно доверять. Пользователь также может добавить удостоверяющие центры к тем, которые его браузер признает доверенными. Эти сертификаты принимаются протоколом Secure Socket Layer (SSL)/Transport Layer Security (TLS), который чаще всего используется для обеспечения безопасности веб-транзакций, о котором мы поговорим в дальнейшем. (Если вам интересно, вы можете покопаться в настройках предпочтений вашего браузера и найти опцию «просмотреть сертификаты», чтобы увидеть, какому количеству удостоверяющих центров ваш браузер настроен доверять.)

Сеть доверия

Альтернативная модель доверия — это *сеть доверия* (или веб доверия), представленная системой Pretty Good Privacy (PGP), которая обсуждается далее. PGP — это систе-

ма безопасности для электронной почты, поэтому адреса электронной почты являются идентичностями, к которым привязаны ключи и которыми подписаны сертификаты. В соответствии с корнями PGP как защиты от правительственного вмешательства здесь нет удостоверяющих центров. Вместо этого каждый человек сам решает, кому он доверяет и насколько ему доверяет — в этой модели доверие является вопросом степени. Кроме того, сертификат публичного ключа может включать уровень уверенности, указывающий, насколько уверен подписант в привязке ключа, заявленной в сертификате, поэтому конкретному пользователю может потребоваться несколько сертификатов, подтверждающих одну и ту же привязку ключа, прежде чем он будет готов доверять ей.

Например, предположим, что у вас есть сертификат для Боба, предоставленный Алисой; вы можете присвоить этому сертификату умеренный уровень доверия. Однако если у вас есть дополнительные сертификаты для Боба, предоставленные С и D, каждый из которых также умеренно надежен, это может значительно увеличить вашу уверенность в том, что публичный ключ, который у вас есть для Боба, является действительным. Короче говоря, PGP признает, что проблема установления доверия является довольно личным вопросом и предоставляет пользователям исходные данные для принятия собственных решений, вместо того чтобы предполагать, что все готовы доверять единой иерархической структуре удостоверяющих центров. По словам Фила Циммермана, разработчика PGP, «PGP предназначен для людей, которые предпочитают паковать свои парашюты сами».

PGP стал довольно популярным в сетевом сообществе, и мероприятия по подписанию ключей PGP являются регулярным элементом различных сетевых мероприятий, таких как встречи IETF. На этих собраниях человек может:

- Собрать публичные ключи от других людей, чью личность он знает.
- Предоставить свой публичный ключ другим.
- Получить свой публичный ключ, подписанный другими, таким образом собирая сертификаты, которые будут убедительны для все большего числа людей.
- Подписать публичный ключ других людей, таким образом помогая им собирать их набор сертификатов, которые они могут использовать для распространения своих публичных ключей.
- Собрать сертификаты от других людей, которым он доверяет достаточно, чтобы подписывать ключи.

Таким образом, со временем пользователь соберет набор сертификатов с различными уровнями доверия.

Отзыв сертификата

Одна из проблем, которая возникает с сертификатами, — как отозвать или аннулировать сертификат. Почему это важно? Предположим, вы подозреваете, что кто-то обнаружил ваш приватный ключ. Может быть любое количество сертификатов, которые утверждают, что вы являетесь владельцем публичного ключа, соответствующего этому приватному ключу. Человек, который обнаружил ваш приватный ключ, таким образом, имеет все, что нужно для вашей подмены: действительные сертификаты и ваш приватный ключ. Чтобы решить эту проблему, было бы полезно иметь возможность отозвать сертификаты, которые связывают ваш старый, скомпрометированный ключ с вашей идентичностью, чтобы подменяющий злоумышленник больше не мог убеждать других людей, что он — это вы.

Основное решение этой проблемы достаточно простое. Каждый удостоверяющий центр может выпустить *список отозванных сертификатов* (Certificate Revocation List, CRL), который представляет собой подписанный цифровой подписью список отозванных сертификатов. CRL периодически обновляется и становится общедоступным. Поскольку он подписан цифровой подписью, его можно просто разместить на веб-сайте. Теперь, когда Алиса получает сертификат для Боба, который она хочет проверить, она сначала обратится к последнему CRL, выпущенному удостоверяющим центром. Пока

сертификат не отозван, он действителен. Обратите внимание, что если все сертификаты имеют неограниченный срок действия, CRL будет постоянно увеличиваться, поскольку сертификат нельзя будет убрать из CRL из-за опасения, что какая-то копия отозванного сертификата может быть использована. По этой причине обычно при выдаче сертификата к нему прикрепляется дата истечения срока действия. Таким образом, мы можем ограничить время, в течение которого отозванный сертификат должен оставаться в CRL. Как только истекает его первоначальная дата истечения, его можно удалить из CRL.

Глава 8.3.2. Предварительное распределение секретных ключей

Если Алиса хочет использовать симметричный шифр для общения с Бобом, она не может просто выбрать ключ и отправить его ему, потому что, не имея ключа, они не могут зашифровать этот ключ, чтобы сохранить его конфиденциальность, и не могут аутентифицировать друг друга. Как и с публичными ключами, требуется схема предраспределения. Предраспределение для секретных ключей сложнее, чем для публичных ключей, по двум очевидным причинам:

- В то время как один публичный ключ на сущность достаточен для аутентификации и конфиденциальности, должен быть секретный ключ для каждой пары сущностей, которые хотят общаться. Если есть N сущностей, то будет $N(N-1)/2$ ключей.
- В отличие от публичных ключей, секретные ключи должны оставаться в секрете.

В итоге необходимо распределить гораздо больше ключей, и нельзя использовать сертификаты, которые могут читать все.

Наиболее распространенное решение — использовать *Центр распределения ключей* (Key Distribution Center, KDC). KDC — это доверенный субъект, который делится секретным ключом с каждой другой сущностью. Это сокращает количество ключей до более управляемого числа $N-1$, что достаточно для установления вне сети в некоторых приложениях. Когда Алиса хочет общаться с Бобом, это общение не проходит через KDC. Вместо этого KDC участвует в протоколе, который аутентифицирует Алису и Боба, используя ключи, которые KDC уже делит с каждым из них, и генерирует новый сеансовый ключ для их использования. Затем Алиса и Боб общаются напрямую, используя свой сеансовый ключ. Kerberos — широко используемая система, основанная на этом подходе. Мы опишем Kerberos (который также обеспечивает аутентификацию) в следующей главе. В следующей подглаве описан мощный альтернативный метод.

Глава 8.3.3. Обмен ключами Диффи-Хеллмана

Другой подход к установлению общего секретного ключа — использовать протокол обмена ключами Диффи-Хеллмана, который работает без использования заранее распределенных ключей. Сообщения, обмен которыми происходит между Алисой и Бобом, могут быть прочитаны любым, кто сможет подслушать, однако подслушивающий не узнает секретного ключа, который в итоге получают Алиса и Боб.

Диффи-Хеллман не аутентифицирует участников. Поскольку редко полезно общаться безопасно, не будучи уверенным, с кем вы общаетесь, протокол Диффи-Хеллмана обычно дополняется каким-то способом для обеспечения аутентификации. Один из основных способов использования Диффи-Хеллмана — в протоколе обмена ключами в Интернете (Internet Key Exchange, IKE), который является центральной частью архитектуры IP Security (IPsec).

Протокол Диффи-Хеллмана имеет два параметра, p и g , оба из которых являются открытыми и могут использоваться всеми пользователями в конкретной системе. Параметр p должен быть простым числом. Целые числа $\bmod p$ (сокращение от modulo p) — это числа от 0 до $p-1$, поскольку $x \bmod p$ — это остаток после деления x на p , и образуют то, что математики называют группой при умножении. Параметр g (обычно называемый генератором) должен быть примитивным корнем из p : для каждого числа n от 1 до $p-1$

должно существовать некоторое значение k , такое, что $n = gk \bmod p$. Например, если p — простое число 5 (в реальной системе используется гораздо большее число), то в качестве генератора g мы можем выбрать 2, так как:

$$\begin{aligned}1 &= 2^0 \bmod p \\2 &= 2^1 \bmod p \\3 &= 2^3 \bmod p \\4 &= 2^2 \bmod p\end{aligned}$$

Предположим, что Алиса и Боб хотят согласовать общий секретный ключ. Алиса и Боб, и все остальные уже знают значения p и g . Алиса генерирует случайное приватное значение a , а Боб генерирует случайное приватное значение b . Оба a и b берутся из множества целых $\{1, \dots, p-1\}$. Алиса и Боб выводят свои соответствующие публичные значения — значения, которые они будут отправлять друг другу незашифрованными, — следующим образом. Публичное значение Алисы

$$g^a \bmod p$$

а публичное значение Боба

$$g^b \bmod p$$

Затем они обмениваются своими публичными значениями. Наконец, Алиса вычисляет

$$g^{ab} \bmod p = (g^b \bmod p)^a \bmod p$$

а Боб вычисляет

$$g^{ba} \bmod p = (g^a \bmod p)^b \bmod p$$

Теперь у Алисы и Боба есть $g^{ab} \bmod p$ (что равно $g^{ba} \bmod p$) как их общий секретный ключ.

Любой подслушивающий знал бы p , g и два публичных значения $g^a \bmod p$ и $g^b \bmod p$. Если бы подслушивающий мог определить a или b , он легко мог бы вычислить результирующий ключ. Определение a или b из этой информации, однако, вычислительно невозможно для достаточно больших p , a и b ; это известно как проблема *дискретного логарифма*.

Например, используя $p = 5$ и $g = 2$ из вышеупомянутого примера, предположим, что Алиса выбирает случайное число $a = 3$, а Боб выбирает случайное число $b = 4$. Тогда Алиса отправляет Бобу публичное значение

$$2^3 \bmod 5 = 3$$

а Боб отправляет Алисе публичное значение

$$2^4 \bmod 5 = 1$$

Алиса затем может вычислить

$$g^{ab} \bmod p = (2^b \bmod 5)^3 \bmod 5 = (1)^3 \bmod 5 = 1$$

подставив публичное значение Боба для $(2^b \bmod 5)$. Аналогично Боб может вычислить

$$g^{ba} \bmod p = (g^a \bmod 5)^4 \bmod 5 = (3)^4 \bmod 5 = 1$$

подставив публичное значение Алисы для $(2^a \bmod 5)$. Оба, Алиса и Боб, теперь согласны, что секретный ключ — это 1.

Проблема заключается в отсутствии аутентификации у Диффи-Хеллмана. Одна из атак, которая может воспользоваться этим, — это *атака посредника* (man-in-the-middle attack). Предположим, что Мэллори — это злоумышленник, способный перехватывать сообщения.

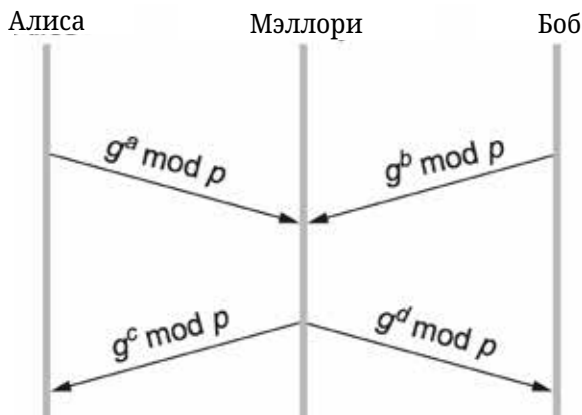


Рисунок 8.7. Атака посредника.

Мэллори уже знает p и g , так как они публичные, и она генерирует случайные приватные значения c и d для использования с Алисой и Бобом соответственно. Когда Алиса и Боб отправляют свои публичные значения друг другу, Мэллори перехватывает их и отправляет свои собственные публичные значения, как показано на рис. 8.7. В результате Алиса и Боб, сами того не зная, в итоге делят ключ с Мэллори вместо друг друга.

Вариант Диффи-Хеллмана, иногда называемый *фиксированным Диффи-Хеллманом*, поддерживает аутентификацию одного или обоих участников. Он опирается на сертификаты, которые аналогичны сертификатам открытых ключей, но вместо этого сертифицируют публичные параметры Диффи-Хеллмана субъекта. Например, такой сертификат будет указывать, что параметры Диффи-Хеллмана Алисы — это p , g и g^a (заметьте, что значение a по-прежнему будет известно только Алисе). Такой сертификат убедит Боба, что другой участник Диффи-Хеллмана — это Алиса, или же другой участник не сможет вычислить секретный ключ, так как он не знает a . Если оба участника имеют сертификаты для своих параметров Диффи-Хеллмана, они могут аутентифицировать друг друга. Если сертификат есть только у одного, то только этот один может быть аутентифицирован. Это полезно в некоторых ситуациях; например, когда один участник — это веб-сервер, а другой — произвольный клиент, который может аутентифицировать веб-сервер и установить секретный ключ для конфиденциальности перед отправкой номера кредитной карты на веб-сервер.

Глава 8.4. Протоколы аутентификации

До сих пор мы описывали, как шифровать сообщения, создавать аутентификаторы, предварительно распределять необходимые ключи. Может показаться, что все, что нам нужно сделать, чтобы сделать протокол безопасным, это добавить аутентификатор к каждому сообщению и, если мы хотим конфиденциальности, зашифровать сообщение.

Есть две основные причины, почему это не так просто. Во-первых, существует проблема *атаки повторного воспроизведения* (replay attack): противник может повторно передать копию сообщения, которое было отправлено ранее. Если это сообщение было заказом, который вы разместили на сайте, то воспроизведенное сообщение будет выглядеть для сайта, как будто вы заказали, больше чем было раньше. Хотя это не было оригинальным воплощением сообщения, его аутентификатор все еще будет действительным; в конце концов, сообщение было создано вами и не было изменено. Очевидно, нам нужно решение, которое гарантирует *оригинальность*.

В варианте этой атаки, называемом *атакой подавленного воспроизведения* (suppress-replay attack), злоумышленник может просто задержать ваше сообщение (перехватив

и позже воспроизведя его), так что оно будет получено в неподходящее время. Например, злоумышленник может задержать ваш заказ на покупку акций с благоприятного времени на время, когда вы не хотели бы покупать. Хотя это сообщение в некотором смысле будет оригинальным, оно не будет своевременным. Таким образом, нам также нужно обеспечить *своевременность*. Оригинальность и своевременность можно считать аспектами целостности. Обеспечение их в большинстве случаев потребует нетривиального, двухстороннего протокола.

Вторая проблема, которую мы еще не решили, это как установить сеансовый ключ. Сеансовый ключ — это ключ симметричного шифрования, сгенерированный на лету и используемый только для одной сессии. Это также требует нетривиального протокола.

Что объединяет эти две проблемы, так это аутентификация. Если сообщение не является оригинальным и своевременным, то с практической точки зрения мы считаем его не аутентичным, то есть не исходящим от того, кем оно заявлено. Очевидно, что при установлении нового сеансового ключа вы хотите быть уверены, что делитесь им с правильным человеком. Обычно протоколы аутентификации одновременно устанавливают сеансовый ключ, чтобы в конце протокола Алиса и Боб аутентифицировали друг друга и получили новый секретный ключ для использования. Без нового сеансового ключа протокол только аутентифицировал бы Алису и Боба в один момент времени; сеансовый ключ позволяет им эффективно аутентифицировать последующие сообщения. В общем, протоколы установления сеансовых ключей выполняют аутентификацию. Замечательное исключение — протокол Диффи-Хеллмана, как описано ниже, поэтому термины «*протокол аутентификации*» и «*протокол установления сеансового ключа*» почти синонимичны.

Существует основной набор методов, используемых для обеспечения оригинальности и своевременности в протоколах аутентификации. Мы опишем эти методы, прежде чем переходить к конкретным протоколам.

Глава 8.4.1. Техники обеспечения оригинальности и своевременности

Мы видели, что одних аутентификаторов недостаточно, чтобы обнаружить сообщения, которые не являются оригинальными или своевременными. Один из подходов — включить временную метку в сообщение. Очевидно, что сама временная метка должна быть защищена от подделки, поэтому она должна быть покрыта аутентификатором. Основной недостаток временных меток заключается в том, что они требуют синхронизации распределенных часов. Поскольку наша система будет зависеть от синхронизации, сама синхронизация часов должна быть защищена от угроз безопасности, помимо обычных вызовов синхронизации часов. Еще одной проблемой является то, что распределенные часы синхронизируются только до определенной степени, что создает некоторую погрешность. Таким образом, временная целостность, обеспечиваемая временными метками, ограничена степенью синхронизации.

Другой подход заключается во включении *одноразового случайного числа* (nonce) в сообщение. Участники могут обнаруживать атаки воспроизведения, проверяя, использовался ли ранее nonce. К сожалению, это требует отслеживания прошлых nonce, которых может накопиться очень много. Одно из решений — комбинировать использование временных меток и nonce, чтобы nonce должно было быть уникальным только в определенный промежуток времени. Это делает обеспечение уникальности nonce управляемым, требуя лишь слабой синхронизации часов.

Еще одно решение недостатков временных меток и nonce — использование одного или обоих в протоколе *challenge-response* (вызов-ответ). Допустим, мы используем временную метку. В протоколе challenge-response Алиса отправляет Бобу временную метку, бросая вызов Бобу зашифровать ее в ответном сообщении (если они делят секретный ключ) или подписать ее в ответном сообщении (если у Боба есть открытый ключ, как на рис. 8.8).

Зашифрованная временная метка подобна аутентификатору, который дополнительно доказывает своевременность. Алиса может легко проверить своевременность временной метки в ответе от Боба, так как эта временная метка исходит от часов Алисы — никакая синхронизация распределенных часов не требуется. Предположим, что протокол использует попсе. Тогда Алисе нужно лишь отслеживать те попсе, для которых ответы все еще ожидаются и не ожидались слишком долго; любой предполагаемый ответ с нераспознанным попсе должен быть поддельным.

Удобство challenge-response, который иначе может показаться чрезмерно сложным, заключается в том, что он сочетает своевременность и аутентификацию; в конце концов, только Боб (и, возможно, Алиса, если это шифр с секретным ключом) знает ключ, необходимый для шифрования не зафиксированной ранее временной метки или попсе. Временные метки или попсе используются в большинстве следующих протоколов аутентификации.

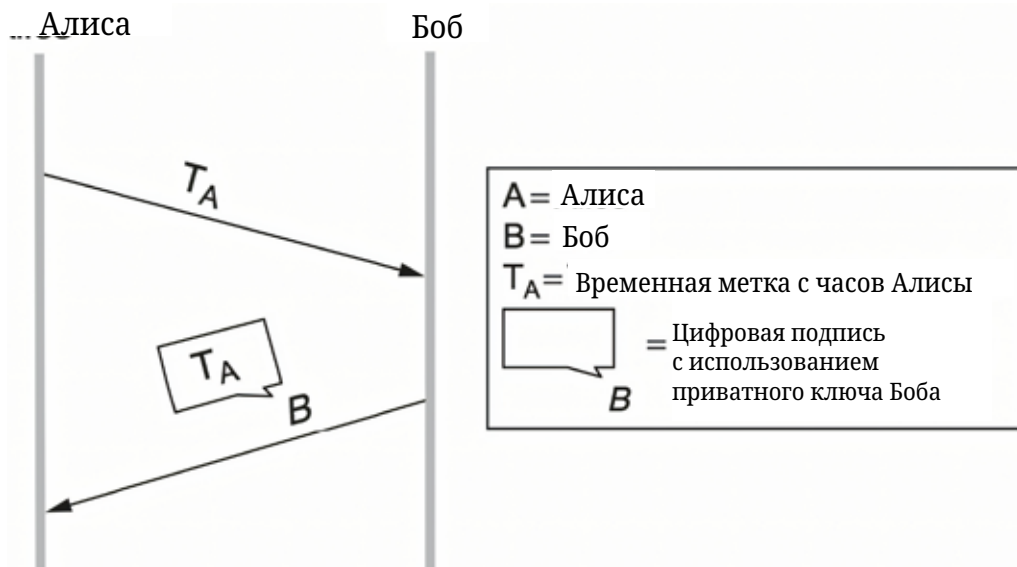


Рисунок 8.8. Протокол «challenge-response».

Глава 8.4.2. Протоколы аутентификации с открытым ключом

В следующих обсуждениях мы предполагаем, что открытые ключи Алисы и Боба были предварительно распределены друг другу с помощью таких средств, как инфраструктура открытых ключей (PKI). Мы подразумеваем случай, когда Алиса включает свой сертификат в свое первое сообщение Бобу, а также случай, когда Боб ищет сертификат об Алисе, получив ее первое сообщение.

Первый протокол (рис. 8.9) зависит от синхронизации часов Алисы и Боба. Алиса отправляет Бобу сообщение с временной меткой и своей идентификацией в открытом виде плюс свою цифровую подпись. Боб использует цифровую подпись для аутентификации сообщения и временную метку для проверки его свежести. Боб отправляет обратно сообщение с временной меткой и своей идентификацией в открытом виде, а также новый сеансовый ключ, зашифрованный (для конфиденциальности) с использованием открытого ключа Алисы, все это подписано цифровой подписью. Алиса может проверить аутентичность и свежесть сообщения, чтобы знать, что может доверять новому сеансовому ключу. Чтобы справиться с несовершенной синхронизацией часов, временные метки могут быть дополнены одноразовыми числами (nonces).

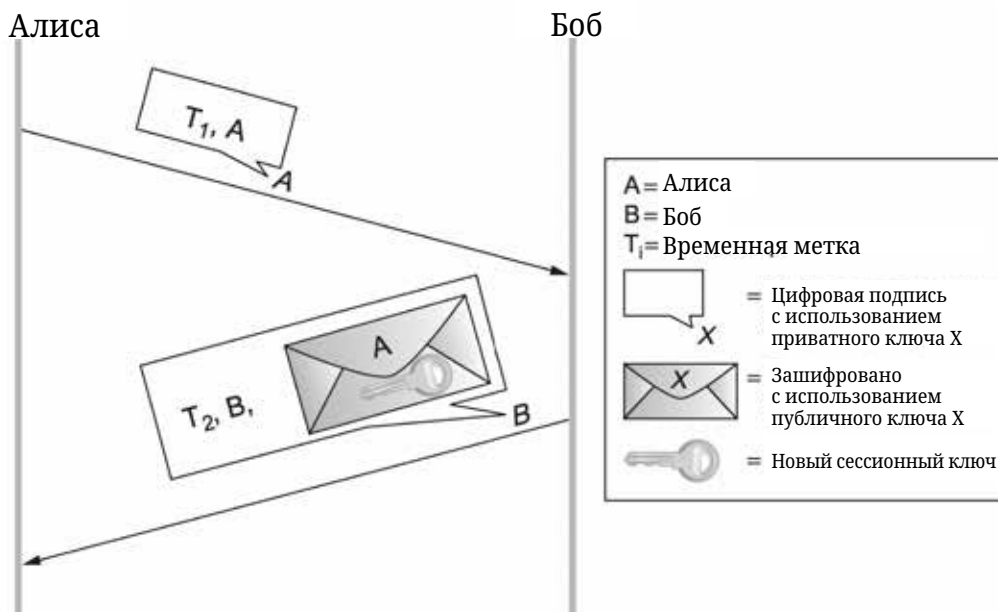


Рисунок 8.9. Протокол аутентификации с публичным ключом, зависящий от синхронизации.

Глава 8.4.3. Протоколы аутентификации с секретными ключами

Только в довольно небольших системах целесообразно предварительно распределять секретные ключи каждой паре сущностей. Мы сосредоточимся здесь на более крупных системах, где каждая сущность будет иметь свой *мастер-ключ*, общий только с Центром распределения ключей (KDC). В этом случае протоколы аутентификации на основе секретного ключа включают три стороны: Алису, Боба и KDC. Конечный продукт протокола аутентификации — сеансовый ключ, общий для Алисы и Боба, который они будут использовать для прямого общения, не вовлекая KDC.

Протокол аутентификации Нидхэма-Шредера иллюстрируется на рис. 8.11. Обратите внимание, что KDC фактически не аутентифицирует первоначальное сообщение Алисы и не общается с Бобом вообще. Вместо этого KDC использует свои знания мастер-ключей Алисы и Боба для создания ответа, который был бы бесполезен для кого-либо, кроме Алисы (поскольку только Алиса может его расшифровать) и содержит необходимые элементы, чтобы Алиса и Боб могли выполнить остальную часть протокола аутентификации самостоятельно.

Одноразовое число (nonce) в первых двух сообщениях предназначено для того, чтобы убедить Алису, что ответ KDC свежий. Второе и третье сообщения включают новый сеансовый ключ и идентификатор Алисы, зашифрованные вместе с использованием мастер-ключа Боба. Это своего рода версия сертификата с использованием секретного ключа; это фактически подписанное заявление KDC (поскольку KDC является единственной сущностью, кроме Боба, которая знает мастер-ключ Боба), что вложенный сеансовый ключ принадлежит Алисе и Бобу.

Хотя одноразовое число в последних двух сообщениях предназначено для того, чтобы убедить Боба в том, что третье сообщение было свежим, в этом рассуждении есть недостаток.

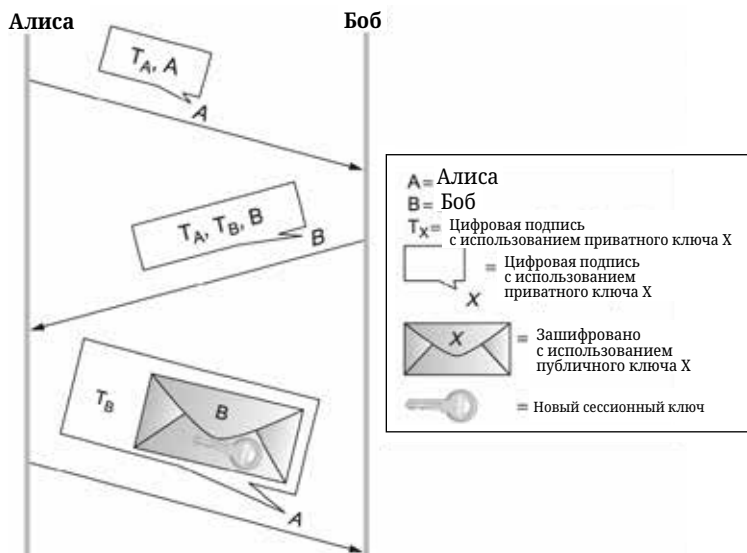


Рисунок 8.10. Протокол аутентификации с публичным ключом, не зависящий от синхронизации. Алиса сверяет свою временную метку с собственными часами, и то же самое делает Боб.

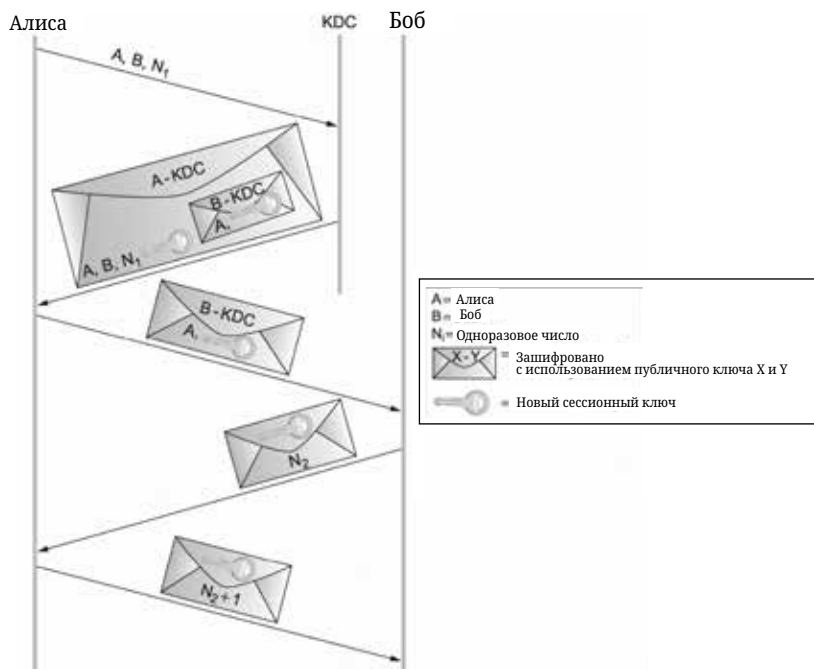


Рисунок 8.11. Протокол аутентификации Нидхэма-Шредера.

Kerberos

Kerberos — это система аутентификации, основанная на протоколе Нидхэма-Шредера и специализированная для клиент/серверных сред. Изначально разработанная в MIT, она была стандартизирована IETF и доступна как в виде открытого исходного кода, так

и в виде коммерческих продуктов. Мы сосредоточимся на некоторых интересных нововведениях Kerberos.

Клиенты Kerberos обычно являются людьми-пользователями, и пользователи аутентифицируются с помощью паролей. Главный ключ Алисы, общий с KDC, выводится из ее пароля — если вы знаете пароль, вы можете вычислить ключ. Kerberos предполагает, что любой может физически получить доступ к любому клиентскому компьютеру; поэтому важно минимизировать риск раскрытия пароля или главного ключа Алисы не только в сети, но и на любом компьютере, где она входит в систему. Kerberos использует протокол Нидхэма-Шредера для достижения этой цели. В протоколе Нидхэма-Шредера единственное время, когда Алиса должна использовать свой пароль, это когда она расшифровывает ответ от KDC. Клиентское программное обеспечение Kerberos ждет, пока не поступит ответ от KDC, затем запрашивает у Алисы ввод пароля, вычисляет главный ключ и расшифровывает ответ KDC, а затем стирает всю информацию о пароле и главном ключе, чтобы минимизировать их раскрытие. Также следует отметить, что единственным признаком для пользователя, что используется Kerberos, является запрос на ввод пароля.

В протоколе Нидхэма-Шредера ответ KDC для Алисы выполняет две роли: он дает ей возможность доказать свою личность (только Алиса может расшифровать ответ), и он дает ей своего рода сертификат с секретным ключом или «билет», который она может предъявить Бобу — сеансовый ключ и идентификатор Алисы, зашифрованные с использованием мастер-ключа Боба. В Kerberos эти две функции — и сам KDC, по сути, — разделены (рис. 8.12). Доверенный сервер, называемый сервером аутентификации (AS), выполняет первую роль KDC, предоставляя Алисе что-то, что она может использовать для доказательства своей личности — не Бобу, а второму доверенному серверу, называемому сервером выдачи билетов (TGS). TGS выполняет вторую роль KDC, отвечая Алисе билетом, который она может предъявить Бобу. Привлекательность этой схемы в том, что если Алисе нужно общаться с несколькими серверами, а не только с Бобом, то она может получить билеты для каждого из них от TGS, не возвращаясь к AS.

В домене клиент/серверных приложений, для которых предназначен Kerberos, можно допустить некоторую степень синхронизации часов. Это позволяет Kerberos использовать временные метки и продолжительности вместо одноразовых чисел (nonces) в протоколе Нидхэма-Шредера, тем самым устраняя его уязвимость. Kerberos поддерживает выбор хеш-функций и шифров с секретным ключом, что позволяет ему идти в ногу с последними достижениями в криптографических алгоритмах.

Глава 8.5. Примеры систем

Теперь мы рассмотрели многие компоненты, необходимые для обеспечения одного или двух аспектов безопасности. Эти компоненты включают криптографические алгоритмы, механизмы предраспределения ключей и протоколы аутентификации. В этой главе мы рассмотрим несколько завершенных систем, использующих эти компоненты.

Эти системы можно грубо классифицировать по уровню протокола, на котором они работают. Системы, работающие на уровне приложений, включают Pretty Good Privacy (PGP), который обеспечивает безопасность электронной почты, и Secure Shell (SSH), безопасный удаленный доступ. На транспортном уровне существует стандарт Transport Layer Security (TLS) от IETF и более старый протокол, из которого он исходит, Secure Socket Layer (SSL). Протоколы IPsec (IP Security), как следует из их названия, работают на уровне IP (сетевой уровень). 802.11i обеспечивает безопасность на канальном уровне беспроводных сетей. В этой главе описываются основные особенности каждого из этих подходов.

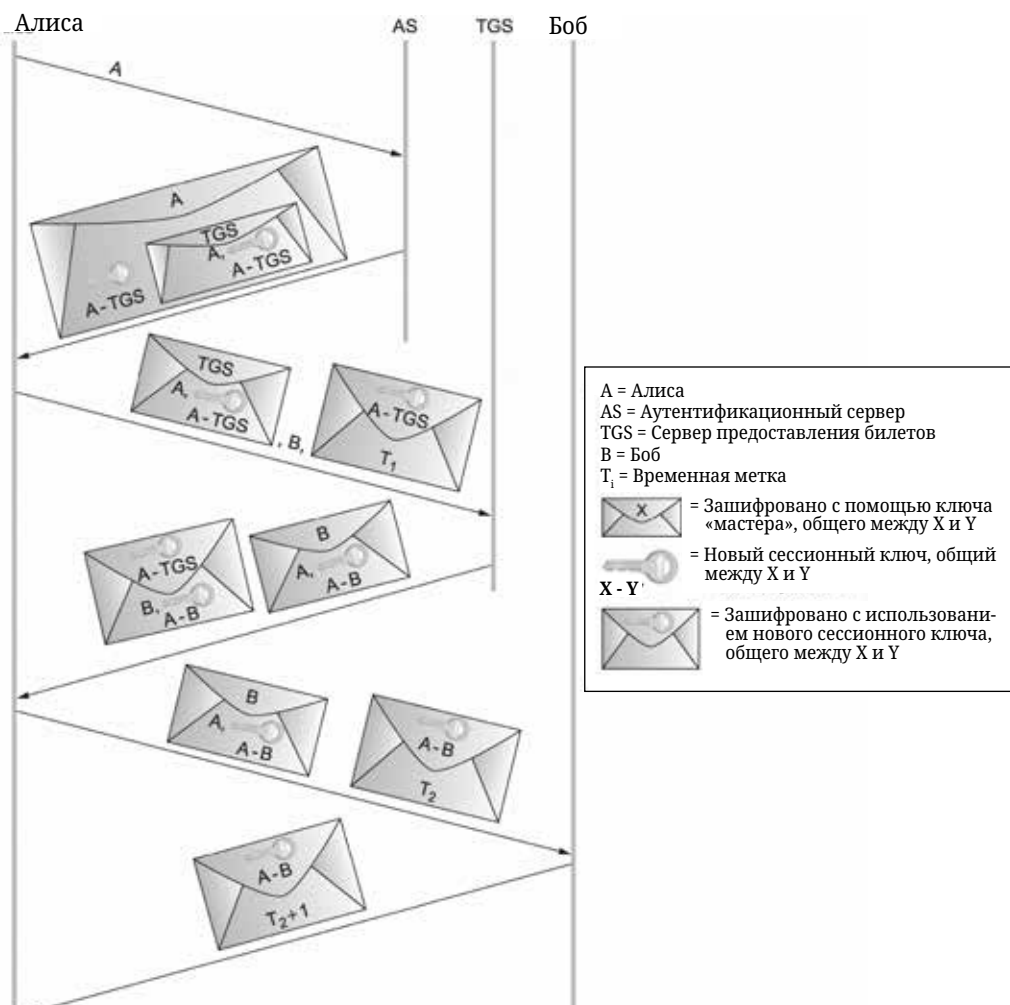


Рисунок 8.12. Аутентификация Kerberos.

Вы можете обоснованно задаться вопросом, почему безопасность должна быть обеспечена на стольких различных уровнях. Одна из причин заключается в том, что различные угрозы требуют разных мер защиты, что часто переводится в обеспечение безопасности на разных уровнях протоколов. Например, если вас больше всего беспокоит, что кто-то в соседнем здании подслушивает ваш трафик, когда он передается между вашим ноутбуком и точкой доступа 802.11, то, вероятно, вам нужна безопасность на канальном уровне.

Однако если вы хотите быть действительно уверены, что подключены к веб-сайту вашего банка и предотвращаете чтение всех данных, которые вы отправляете в банк, любопытными сотрудниками какого-то интернет-провайдера, тогда что-то, что охватывает весь путь от вашего устройства до сервера банка — например, транспортный уровень — может быть правильным местом для обеспечения безопасности трафика. Как часто бывает, нет универсального решения.

Описанные ниже системы безопасности имеют возможность изменять используемые криптографические алгоритмы. Идея сделать систему безопасности независимой от алгоритмов является очень хорошей, потому что вы никогда не знаете, когда ваш люби-

мый криптографический алгоритм может оказаться недостаточно сильным для ваших целей. Было бы хорошо, если бы вы могли быстро перейти на новый алгоритм без необходимости изменять спецификацию или реализацию протокола. Обратите внимание на аналогию с возможностью изменять ключи без изменения алгоритма; если один из ваших криптографических алгоритмов окажется с изъянами, было бы здорово, если бы вся ваша архитектура безопасности не нуждалась в немедленной переработке.

Глава 8.5.1. Pretty Good Privacy (PGP)

Pretty Good Privacy (PGP) — широко используемый метод обеспечения безопасности для электронной почты. Он обеспечивает аутентификацию, конфиденциальность, целостность данных и неотказуемость. Первоначально разработанный Филом Циммерманом, он эволюционировал в стандарт IETF, известный как OpenPGP. Как мы видели в предыдущей главе, PGP примечателен использованием модели «веб доверия» для распределения ключей, а не иерархической структуры.

Конфиденциальность и аутентификация получателя в PGP зависят от того, что у получателя электронного письма есть открытый ключ, известный отправителю. Для обеспечения аутентификации отправителя и неотказуемости отправитель должен иметь открытый ключ, известный получателю. Эти открытые ключи предраспределяются с использованием сертификатов и PKI «веба доверия». PGP поддерживает RSA и DSS для сертификатов открытых ключей. Эти сертификаты могут также указывать, какие криптографические алгоритмы поддерживаются или предпочитаются владельцем ключа. Сертификаты обеспечивают связь между адресами электронной почты и открытыми ключами.

Рассмотрим следующий пример использования PGP для обеспечения аутентификации отправителя и конфиденциальности. Предположим, что у Алисы есть сообщение, которое она хочет отправить Бобу по электронной почте. Приложение PGP Алисы выполняет шаги, иллюстрированные на рис. 8.13. Во-первых, сообщение подписывается цифровой подписью Алисой; MD5, SHA-1 и семейство SHA-2 входят в число хешей, которые могут использоваться в цифровой подписи. Затем ее приложение PGP генерирует новый сеансовый ключ только для этого сообщения; среди поддерживаемых симметричных шифров — AES и 3DES. Сообщение, подписанное цифровой подписью, шифруется с использованием сеансового ключа, затем сам сеансовый ключ шифруется с использованием открытого ключа Боба и добавляется к сообщению. Приложение PGP Алисы напоминает ей об уровне доверия, который она ранее назначила открытому ключу Боба, на основе количества сертификатов, которые у нее есть для Боба, и надежности лиц, подписавших сертификаты. Наконец, не ради безопасности, а потому что электронные письма должны быть отправлены в формате ASCII, к сообщению применяется кодирование base64, чтобы преобразовать его в ASCII-совместимое представление.

Получив сообщение PGP по электронной почте, приложение PGP Боба выполняет этот процесс в обратном порядке, шаг за шагом, чтобы получить оригинальное сообщение в открытом тексте и подтвердить цифровую подпись Алисы, и напоминает Бобу об уровне доверия, который он имеет к открытому ключу Алисы.

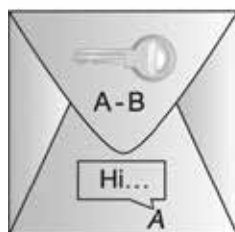
Электронная почта обладает особыми характеристиками, позволяющими PGP встроить адекватный протокол аутентификации в этот односообщенческий протокол передачи данных, избегая необходимости в каком-либо предварительном обмене сообщениями (и обходя некоторые сложности, описанные в предыдущей главе). Цифровая подпись Алисы достаточна для ее аутентификации. Хотя нет доказательств того, что сообщение своевременно, законная электронная почта также не гарантирует своевременности. Также нет доказательств того, что сообщение оригинально,

но Боб является пользователем электронной почты и, вероятно, достаточно толерантен к ошибкам, чтобы справиться с дублирующимися сообщениями (которые, опять же, не исключены при нормальной работе). Алиса может быть уверена, что только Боб мог прочитать сообщение, потому что сеансовый ключ был зашифрован с использованием его открытого ключа. Хотя этот протокол не доказывает Алисе, что Боб действительно получил электронное письмо, аутентифицированное письмо от Боба обратно к Алисе могло бы это сделать.

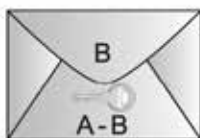
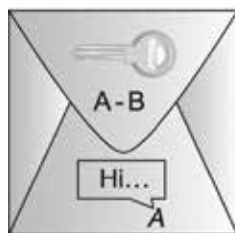
Hi... = обычный текст



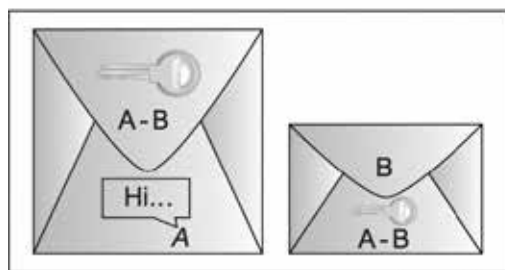
- 1) Цифровая подпись с использованием
приватного ключа Алисы



- 2) Зашифровано с использованием вновь
созданного одноразового сессионного
ключа



- 3) Зашифровать сессионный ключ,
используя публичный ключ Боба,
и потом добавить это



- 4) Использовать кодировку base64,
чтобы заполучить
ASCII-совместимую презентацию

Рисунок 8.13. Шаги PGP по подготовке сообщения для отправки
по электронной почте от Алисы к Бобу.

Предыдущее обсуждение дает хороший пример того, почему механизмы безопасности на уровне приложения могут быть полезны. Только обладая полным знанием о том, как работает приложение, вы можете сделать правильный выбор, против каких атак следует защищаться (например, поддельная электронная почта) и какие можно игнорировать (например, задержанная или повторная электронная почта).

Глава 8.5.2. Безопасная оболочка (Secure Shell, SSH)

Протокол Secure Shell (SSH) используется для обеспечения удаленного входа в систему, заменяя менее безопасный Telnet, использовавшийся в первые дни Интернета. (SSH также может использоваться для удаленного выполнения команд и передачи файлов, но сначала мы сосредоточимся на том, как SSH поддерживает удаленный вход в систему.) SSH чаще всего используется для обеспечения сильной аутентификации «клиент/сервер» и целостности сообщений — когда клиент SSH работает на настольной машине пользователя, а сервер SSH работает на удаленной машине, в которую пользователь хочет войти — но он также поддерживает конфиденциальность. Telnet не предоставляет ни одной из этих возможностей. Обратите внимание, что термин «SSH» часто используется для обозначения как протокола SSH, так и приложений, его использующих; из контекста нужно понимать, о чем идет речь.

Чтобы лучше оценить важность SSH в сегодняшнем Интернете, рассмотрим пару сценариев, где он используется. Например, телеработники (работающие удаленно) часто подписываются на услуги интернет-провайдеров (ISP), которые предлагают высокоскоростной доступ к интернету по оптоволокну до дома, и используют этих провайдеров (плюс еще несколько других провайдеров) для подключения к машинам, управляемым их работодателем. Это означает, что когда телеработники входят в систему внутри центра обработки данных своего работодателя, как пароли, так и все данные, отправленные или полученные, потенциально проходят через множество недоверенных сетей. SSH предоставляет способ шифровать данные, отправляемые по этим соединениям, и улучшает механизм аутентификации, используемый для входа в систему. (Аналогичная ситуация возникает, когда сотрудник подключается к работе, используя публичный Wi-Fi в Starbucks.) Другое использование SSH — удаленный вход в маршрутизатор, возможно, для изменения его конфигурации или чтения его журналов; очевидно, сетевой администратор хочет быть уверен, что он может безопасно войти в маршрутизатор и что неавторизованные лица не смогут войти в систему или перехватить команды, отправленные маршрутизатору, или вывод, отправленный обратно администратору.

Последняя версия SSH, версия 2, состоит из трех протоколов:

- SSH-TRANS, протокол транспортного уровня
- SSH-AUTH, протокол аутентификации
- SSH-CONN, протокол соединения

Мы сосредоточимся на первых двух, которые участвуют в удаленном входе в систему. В конце главы мы кратко обсудим назначение SSH-CONN.

SSH-TRANS предоставляет зашифрованный канал между клиентской и серверной машинами. Он работает поверх TCP-соединения. Каждый раз, когда пользователь использует SSH-приложение для входа в удаленную машину, первый шаг — это настройка канала SSH-TRANS между этими двумя машинами. Эти машины устанавливают безопасный канал, сначала аутентифицируя сервер с использованием RSA. После аутентификации клиент и сервер устанавливают сеансовый ключ, который они будут использовать для шифрования любых данных, передаваемых по каналу. Это высокоуровневое описание опускает несколько деталей, включая тот факт, что протокол SSH-TRANS включает согласование алгоритма шифрования, который будут использовать обе стороны. Например, часто выбирается AES. Также SSH-TRANS включает проверку целостности сообщения для всех данных, обмененных по каналу.

Один вопрос, который мы не можем оставить без внимания, заключается в том, как клиент получил открытый ключ сервера, необходимый для аутентификации сервера. Как это ни странно, сервер сообщает клиенту свой открытый ключ во время подключения. В первый раз, когда клиент подключается к определенному серверу, прило-

жение SSH предупреждает пользователя, что оно никогда раньше не общалось с этой машиной, и спрашивает, хочет ли пользователь продолжить. Хотя это рискованное дело, потому что SSH фактически не может аутентифицировать сервер, пользователи часто отвечают «да» на этот вопрос. Приложение SSH затем запоминает открытый ключ сервера, и в следующий раз, когда пользователь подключается к той же машине, оно сравнивает этот сохраненный ключ с тем, которым сервер отвечает. Если они совпадают, SSH аутентифицирует сервер. Однако если они разные, приложение SSH снова предупреждает пользователя, что что-то не так, и пользователь получает возможность прервать соединение. В качестве альтернативы предусмотрительный пользователь может узнать открытый ключ сервера через какой-либо внешний механизм, сохранить его на клиентской машине и таким образом никогда не подвергаться риску «первого раза».

Как только канал SSH-TRANS установлен, следующим шагом является фактический вход пользователя на удаленную машину, или, более конкретно, аутентификация пользователя на сервере. SSH позволяет использовать три различных механизма для этого. Во-первых, поскольку две машины общаются по безопасному каналу, пользователю можно просто отправить свой пароль на сервер. Это небезопасно при использовании Telnet, так как пароль будет передан в открытом виде, но в случае SSH пароль шифруется в канале SSH-TRANS. Второй механизм использует шифрование с открытым ключом. Это требует, чтобы пользователь заранее разместил свой открытый ключ на сервере. Третий механизм, называемый *аутентификацией на основе хоста*, заключается в том, что любой пользователь, утверждающий, что он тот или иной человек с определенного набора доверенных хостов, автоматически считается этим пользователем на сервере. Аутентификация на основе хоста требует, чтобы клиентский хост аутентифицировал себя на сервере при первом подключении; стандартный SSH-TRANS по умолчанию аутентифицирует только сервер.

Главное, что нужно усвоить из этого обсуждения, это то, что SSH является довольно простым применением протоколов и алгоритмов, которые мы видели на протяжении всего этого раздела. Однако что иногда делает SSH сложным для понимания, так это все ключи, которые пользователь должен создать и управлять ими, где точный интерфейс зависит от операционной системы. Например, пакет OpenSSH, который работает на большинстве Unix-машин, поддерживает команду, которая может быть использована для создания пар открытого/закрытого ключей. Эти ключи затем хранятся в различных файлах в домашнем каталоге пользователя. Например, файл `~/.ssh/known_hosts` записывает ключи всех хостов, в которые пользователь входил, файл `~/.ssh/authorized_keys` содержит открытые ключи, необходимые для аутентификации пользователя при входе в эту машину (то есть они используются на стороне сервера), и файл содержит закрытые ключи, необходимые для аутентификации пользователя на удаленных машинах (то есть они используются на стороне клиента).

Наконец, SSH оказался настолько полезен в качестве системы защиты удаленного входа в систему, что его расширили для поддержки других приложений, таких как отправка и получение электронной почты. Идея заключается в том, чтобы запускать эти приложения через защищенный «туннель SSH». Эта возможность называется *пробросом портов* и использует протокол SSH-CONN. Идея проиллюстрирована на рис. 8.14, где показано, как клиент на хосте А косвенно взаимодействует с сервером на хосте В, перенаправляя свой график через SSH-соединение. Механизм называется *переадресацией портов*, поскольку, когда сообщения поступают на известный порт SSH на сервере, SSH сначала расшифровывает их содержимое, а затем «переадресует» данные на реальный порт, на котором прослушивает сервер. Это просто еще один вид туннеля, который в данном случае обеспечивает конфиденциальность и аутентификацию. Таким образом можно создать виртуальную частную сеть (VPN), используя туннели SSH.



Рисунок 8.14. Использование перенаправления портов SSH для защиты других приложений на базе TCP.

Глава 8.5.3. Безопасность транспортного уровня (TLS, SSL, HTTPS)

Чтобы понять цели и требования к стандарту защиты транспортного уровня (Transport Layer Security, TLS) и слоя защищенных сокетов (Secure Socket Layer, SSL), на котором основан TLS, полезно рассмотреть одну из основных проблем, которые они предназначены решить. С распространением Всемирной паутины и началом интереса к ней со стороны коммерческих предприятий стало очевидно, что для транзакций в интернете необходим определенный уровень безопасности. Каноническим примером этого является покупка с использованием кредитной карты. Возникает несколько вопросов, связанных с отправкой вашей кредитной информации на компьютер в интернете. Во-первых, вы можете беспокоиться, что информация будет перехвачена в процессе передачи и впоследствии использована для несанкционированных покупок. Вы также можете беспокоиться о том, что детали транзакции будут изменены, например, сумма покупки. И, конечно, вы хотели бы быть уверены, что компьютер, на который вы отправляете информацию о вашей кредитной карте, действительно принадлежит соответствующему продавцу, а не какой-либо другой стороне. Таким образом, мы сразу видим необходимость в конфиденциальности, целостности и аутентификации в интернет-транзакциях. Первым широко используемым решением этой проблемы был SSL, первоначально разработанный компанией Netscape и впоследствии ставший основой для стандарта TLS, разработанного IETF.

Разработчики SSL и TLS осознали, что эти проблемы не являются специфичными для интернет-транзакций (т. е. тех, которые используют HTTP), и вместо этого создали универсальный протокол, который находится между прикладным протоколом, таким как HTTP, и транспортным протоколом, таким как TCP. Причина, по которой это называется «защитой транспортного уровня», заключается в том, что с точки зрения приложения этот протокольный уровень выглядит как обычный транспортный протокол, за исключением того, что он является защищенным. То есть отправитель может устанавливать соединения и передавать байты для передачи, а защищенный транспортный уровень доставит их получателю с необходимыми конфиденциальностью, целостностью и аутентификацией. Путем запуска защищенного транспортного уровня поверх TCP все обычные функции TCP (надежность, управление потоком, контроль перегрузок и т. д.) также предоставляются приложению. Эта организация протокольных уровней показана на рис. 8.15.

Когда HTTP используется таким образом, он называется HTTPS (Secure HTTP). Фактически сам HTTP не изменяется. Он просто передает данные и принимает данные от уровня SSL/TLS, а не TCP. Для удобства по умолчанию назначен TCP-порт для HTTPS (443). То есть если вы попытаетесь подключиться к серверу на TCP-порту 443, вы, вероятно, обнаружите, что общаетесь с протоколом SSL/TLS, который передаст ваши данные через HTTP, при условии успешного прохождения аутентификации и расшифровки. Хотя доступны автономные реализации SSL/TLS, чаще всего реализация включается в состав приложений, которые в этом нуждаются, в первую очередь веб-браузеров.

В оставшейся части нашего обсуждения защиты транспортного уровня мы сосредоточимся на TLS. Хотя SSL и TLS, к сожалению, несовместимы, они различаются лишь незначительно, поэтому почти все, что описывается в отношении TLS, применимо и к SSL.

Приложение (напр. HTTP)
Безопасный транспортный уровень
TCP
IP
Подсеть

Рисунок 8.15. Защищенный транспортный уровень, вставленный между прикладным и TCP-уровнями.

Протокол рукопожатия

Пара участников TLS на этапе выполнения договаривается о том, какую криптографию использовать. Участники договариваются о выборе:

- Хеша для обеспечения целостности данных (MD5, SHA-1 и т. д.), используемого для реализации HMAC
- Симметричного шифра для обеспечения конфиденциальности (среди возможных вариантов — DES, 3DES и AES)
- Подхода к установлению сеансового ключа (среди возможных вариантов — Диффи-Хеллман и протоколы аутентификации с открытым ключом, использующие DSS)

Интересно, что участники могут также договориться об использовании алгоритма сжатия, не потому что это предоставляет какие-либо преимущества в области безопасности, а потому что это легко сделать, когда вы уже согласились на выполнение других дорогих операций на каждый байт данных.

В TLS шифр для конфиденциальности использует два ключа, один для каждого направления, а также два вектора инициализации. Аналогично HMACs тоже используют разные ключи для двух участников. Таким образом, независимо от выбора шифра и хеша, сеанс TLS требует в общей сложности шести ключей. TLS выводит их всех из одного общего *мастер-ключа*. Мастер-ключ представляет собой значение длиной 384 бита (48 байт), которое, в свою очередь, частично выводится из «сеансового ключа», полученного в результате протокола установления сеансового ключа в TLS.

Часть TLS, которая согласовывает выборы и устанавливает общий мастер-ключ, называется *протоколом рукопожатия*. (Фактическая передача данных осуществляется *протоколом записи TLS*.) Протокол рукопожатия по своей сути является протоколом установления сеансового ключа, с мастер-ключом вместо сеансового ключа. Поскольку TLS поддерживает выбор подходов к установлению сеансового ключа, это требует соответствующих вариантов протоколов. Кроме того, протокол рукопожатия поддерживает выбор между взаимной аутентификацией обоих участников, аутентификацией только одного участника (это самый распространенный случай, например, аутентификация веб-сайта, но не пользователя) или отсутствием аутентификации вообще (анонимный Диффи-Хеллман). Таким образом, протокол рукопожатия объединяет несколько протоколов установления сеансового ключа в единый протокол.

Рис. 8.16 показывает протокол рукопожатия на высоком уровне. Клиент первоначально отправляет список комбинаций криптографических алгоритмов, которые он поддерживает, в порядке убывания предпочтений. Сервер отвечает, предоставляя единственную комбинацию криптографических алгоритмов, выбранную из тех, что были указаны клиентом. Эти сообщения также содержат соответственно *клиентский одноразовый номер*

(nonce) и серверный одноразовый номер, которые будут использованы при генерации мастер-ключа позже.

На этом этапе фаза переговоров завершена. Теперь сервер отправляет дополнительные сообщения на основе согласованного протокола установления сеансового ключа. Это может включать отправку сертификата с открытым ключом или набора параметров Диффи-Хеллмана. Если сервер требует аутентификации клиента, он отправляет отдельное сообщение, указывающее на это. Затем клиент отвечает своей частью согласованного протокола обмена ключами.

Теперь клиент и сервер имеют всю необходимую информацию для генерации мастер-ключа. «Сеансовый ключ», который они обменяли, на самом деле не является ключом, а тем, что TLS называет *предварительным мастер-ключом*. Мастер-ключ вычисляется (с использованием опубликованного алгоритма) из этого предварительного мастер-ключа, клиентского одноразового номера и серверного одноразового номера. Используя ключи, полученные из мастер-ключа, клиент затем отправляет сообщение, включающее хеш всех предшествующих сообщений рукопожатия, на которое сервер отвечает аналогичным сообщением. Это позволяет им обнаружить любые несоответствия между отправленными и полученными сообщениями рукопожатия, например, если человек посередине изменил первоначальное незашифрованное сообщение клиента, чтобы ослабить его выборы криптографических алгоритмов.

Протокол записи

В рамках сеанса, установленного протоколом рукопожатия, протокол записи TLS добавляет конфиденциальность и целостность к базовой транспортной службе. Сообщения, поступающие из прикладного уровня, проходят следующие шаги:

- Фрагментируются или объединяются в блоки удобного размера для последующих шагов.
- При необходимости сжимаются.
- Защищаются с помощью HMAC.
- Шифруются с использованием симметричного шифра.
- Передаются транспортному уровню (обычно TCP) для отправки.

Протокол записи использует HMAC в качестве аутентификатора. HMAC использует тот хеш-алгоритм (MD5, SHA-1 и т. д.), который был согласован участниками. Клиент и сервер имеют разные ключи для вычисления HMAC, что делает их еще более сложными для взлома. Более того, каждому сообщению протокола записи присваивается порядковый номер, который включается при вычислении HMAC, даже если порядковый номер неявно указан в сообщении. Этот неявный порядковый номер предотвращает повторные или переставленные сообщения. Это необходимо, потому что, хотя TCP может доставлять последовательные, не дублированные сообщения на верхний уровень в обычных условиях, эти условия не включают наличие злоумышленника, который может перехватывать TCP-сообщения, изменять их или отправлять поддельные. С другой стороны, именно гарантии доставки TCP позволяют TLS полагаться на то, что у законного сообщения TLS будет следующий неявный порядковый номер.

Еще одной интересной особенностью протокола TLS является возможность возобновления сеанса. Чтобы понять первоначальную мотивацию для этого, полезно понять, как HTTP изначально использовал TCP-соединения. (Подробности HTTP представлены в следующем разделе.) Каждая операция HTTP, такая как получение страницы с сервера, требовала открытия нового TCP-соединения. Получение одной страницы с несколькими встроенными графическими объектами могло потребовать множество TCP-соединений. Открытие TCP-соединения требует выполнения трехстороннего рукопожатия до того, как начнется передача данных. После того как TCP-соединение готово принять данные, клиент должен был начать протокол рукопожатия TLS, что занимает как минимум еще два времени кругового пути (и потребляет некоторое количество ресурсов обработки

и сетевой пропускной способности), прежде чем можно будет отправить фактические данные приложения. Возможность возобновления сеанса TLS была разработана, чтобы решить эту проблему.

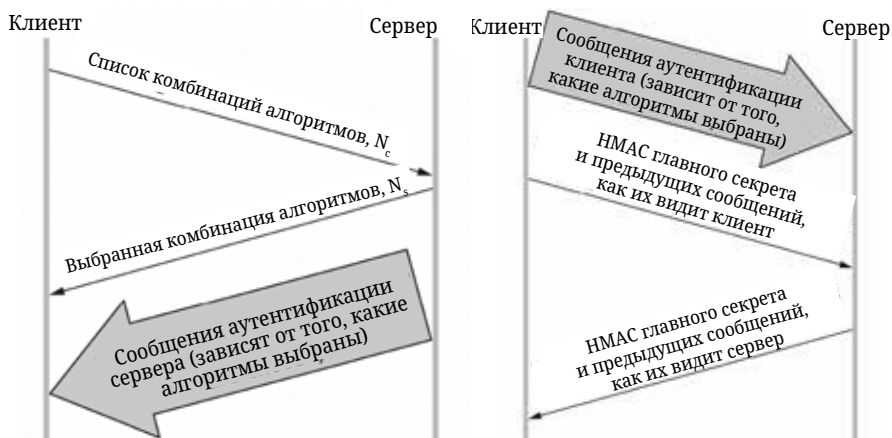


Рисунок 8.16. Протокол квитирования для установления сессии TLS.

Идея возобновления сеанса заключается в оптимизации рукопожатия в тех случаях, когда клиент и сервер уже установили общее состояние в прошлом. Клиент просто включает идентификатор сеанса из ранее установленного сеанса в свое начальное сообщение рукопожатия. Если сервер обнаруживает, что у него все еще есть состояние для этого сеанса и опция возобновления была согласована при первоначальном создании сеанса, то сервер может ответить клиенту с указанием на успех, и передача данных может начаться с использованием ранее согласованных алгоритмов и параметров. Если идентификатор сеанса не совпадает с каким-либо состоянием сеанса, кешированным на сервере, или если возобновление не было разрешено для сеанса, то сервер вернется к обычному процессу рукопожатия.

Причина, по которой в предыдущем обсуждении подчеркивалась первоначальная мотивация, заключается в том, что необходимость выполнения TCP-рукопожатия для каждого встроенного объекта на веб-странице приводила к столь большим накладным расходам, независимо от TLS, что HTTP в конечном итоге был оптимизирован для поддержки постоянных соединений (также обсуждается в следующем разделе). Поскольку оптимизация HTTP уменьшила ценность возобновления сеанса в TLS (плюс осознание того, что повторное использование одних и тех же идентификаторов сеансов и мастер-ключей в серии возобновленных сеансов представляет собой риск для безопасности), TLS изменил свой подход к возобновлению в последней версии (1.3).

В TLS 1.3 клиент отправляет серверу непрозрачный, зашифрованный сервером билет сеанса при возобновлении. Этот билет содержит всю информацию, необходимую для возобновления сеанса. Один и тот же мастер-ключ используется в нескольких рукопожатиях, но поведение по умолчанию заключается в выполнении обмена сеансовыми ключами при возобновлении.

Основные выводы

Мы обращаем внимание на это изменение в TLS, потому что оно иллюстрирует сложность определения, какой уровень должен решать ту или иную проблему. В изоляции возобновление сеанса, реализованное в более ранней версии TLS, кажется хорошей идеей, но его нужно рассматривать в контексте доминирующего случая использования, которым является HTTP. Как только накладные расходы

на выполнение нескольких TCP-соединений были устранены с помощью HTTP, изменилась формула того, как возобновление должно быть реализовано в TLS. Главный урок заключается в том, что нам нужно избегать жесткого мышления о правильном уровне для реализации данной функции — ответ меняется со временем по мере развития сети — где требуется целостный/кросс-уровневый анализ для правильного проектирования.

Глава 8.5.4. IP-безопасность (IPsec)

Вероятно, самым амбициозным из всех усилий по интеграции безопасности в Интернет является работа на уровне IP. Поддержка IPsec, как называется эта архитектура, являясь опциональной в IPv4, но обязательной в IPv6.

IPsec на самом деле является фреймворком (в отличие от одного протокола или системы) для предоставления всех услуг безопасности, обсуждаемых в этом разделе. IPsec предоставляет три степени свободы. Во-первых, он модульный и позволяет пользователям (или, скорее, системным администраторам) выбирать из множества криптографических алгоритмов и специализированных протоколов безопасности. Во-вторых, IPsec позволяет пользователям выбирать из большого меню свойств безопасности, включая контроль доступа, целостность, аутентификацию, оригинальность и конфиденциальность. В-третьих, IPsec может использоваться для защиты узких потоков (например, пакетов, принадлежащих конкретному TCP-соединению, отправляемых между парой хостов) или широких потоков (например, всех пакетов, передаваемых между парой маршрутизаторов).

На высоком уровне IPsec состоит из двух частей. Первая часть — это пара протоколов, реализующих доступные услуги безопасности. Это *заголовок аутентификации* (Authentication Header, AH), который предоставляет контроль доступа, целостность сообщений без подключения, аутентификацию и защиту от повторных атак, и *инкапсулирующая безопасная нагрузка* (Encapsulating Security Payload, ESP), которая поддерживает те же самые службы, плюс конфиденциальность. AH редко используется, поэтому здесь мы сосредоточимся на ESP. Вторая часть — это поддержка управления ключами, которая осуществляется под зонтичным протоколом, известным как *протокол ассоциации безопасности и управления ключами в Интернете* (Internet Security Association and Key Management Protocol, ISAKMP).

Абстракцией, связывающей эти две части вместе, является *ассоциация безопасности* (SA). SA — это одностороннее (simplex) соединение с одним или несколькими доступными свойствами безопасности. Для обеспечения двусторонней связи между парой хостов, соответствующей TCP-соединению, например, требуются две SA, по одной в каждом направлении. Хотя IP является протоколом без соединения, безопасность зависит от состояния соединения, такого как ключи и порядковые номера. При создании SA присваивается идентификационный номер, называемый *индексом параметров безопасности* (SPI), принимающей машиной. Комбинация этого SPI и IP-адресов назначения уникально идентифицирует SA. Заголовок ESP включает SPI, чтобы принимающий хост мог определить, к какой SA принадлежит входящий пакет и, следовательно, какие алгоритмы и ключи применить к пакету.

SA создаются, согласовываются, модифицируются и удаляются с использованием ISAKMP. Он определяет форматы пакетов для обмена данными генерации ключей и аутентификации. Эти форматы не особо интересны, поскольку они предоставляют только фреймворк — точная форма ключей и данных аутентификации зависит от техники генерации ключей, шифра и механизма аутентификации, который используется. Более того, ISAKMP не специфицирует конкретный протокол обмена ключами, хотя он предлагает *интернет-обмен ключами* (IKE) как одну из возможностей, и на практике используется IKE v2.

ESP — это протокол, используемый для безопасной передачи данных по установленной SA. В IPv4 заголовок ESP следует за заголовком IP; в IPv6 это расширяющий заголовок. Его формат использует как заголовок, так и трейлер, как показано на рис. 8.17. Поле SPI по-

звolyет принимающему хосту идентифицировать ассоциацию безопасности, к которой принадлежит пакет. Поле SeqNum защищает от повторных атак. Поле PayloadData пакета содержит данные, описанные полем NextHdr. Если выбрана конфиденциальность, данные шифруются с использованием того шифра, который был связан с SA. Поле PadLength указывает, сколько добавлено заполнения к данным; заполнение иногда необходимо, поскольку, например, шифр требует, чтобы открытый текст был кратен определенному количеству байтов или чтобы зашифрованный текст завершился на границе в 4 байта. Наконец, поле AuthenticationData содержит аутентификатор.

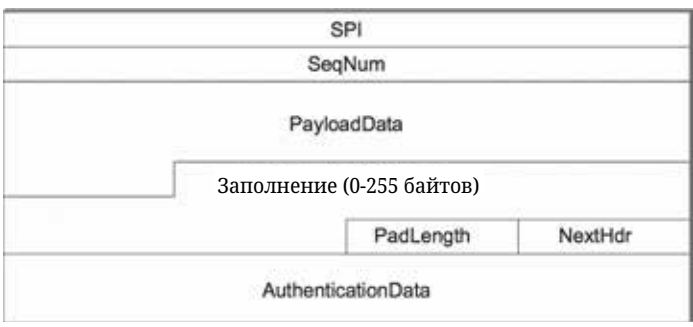


Рисунок 8.17. Формат ESP в IPsec.

IPsec поддерживает *туннельный режим*, а также более прямолинейный *транспортный режим*. Каждая SA работает в одном из этих режимов. В транспортном режиме SA данные полезной нагрузки ESP — это просто сообщение для более высокого уровня, такого как UDP или TCP. В этом режиме IPsec действует как промежуточный уровень протокола, аналогично тому, как SSL/TLS работает между TCP и более высоким уровнем. Когда сообщение ESP принимается, его полезная нагрузка передается на более высокий уровень протокола.

Однако в туннельном режиме SA данные полезной нагрузки ESP являются IP-пакетом, как показано на рис. 8.18. Источник и назначение этого внутреннего IP-пакета могут отличаться от источника и назначения внешнего IP-пакета. Когда сообщение ESP принимается, его полезная нагрузка передается как обычный IP-пакет. Самый распространенный способ использования ESP — это создание «IPsec туннеля» между двумя маршрутизаторами, обычно межсетевыми экранами. Например, корпорация, желающая связать два сайта с использованием Интернета, могла бы открыть пару SA в туннельном режиме между маршрутизатором на одном сайте и маршрутизатором на другом сайте. Исходящий IP-пакет с одного сайта на выходном маршрутизаторе станет полезной нагрузкой сообщения ESP, отправленного на маршрутизатор другого сайта. Принимающий маршрутизатор распакует внутренний IP-пакет и передаст его на его истинное место назначения.

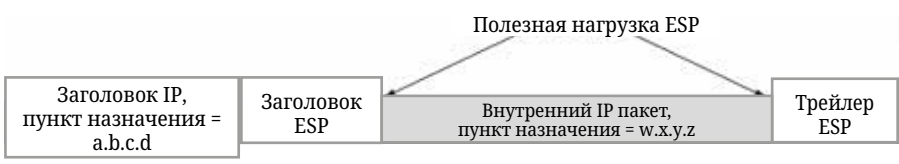


Рисунок 8.18. IP-пакет с вложенным IP-пакетом, инкапсулированным с помощью ESP в туннельном режиме. Обратите внимание, что внутренний и внешний пакеты имеют разные адреса.

Эти туннели также могут быть настроены для использования ESP с конфиденциальностью и аутентификацией, предотвращая таким образом несанкционированный доступ к данным, проходящим через эту виртуальную связь, и обеспечивая, что на другом конце туннеля не будет получено никаких ложных данных. Кроме того, туннели могут обеспечивать конфиденциальность трафика, поскольку мультиплексирование нескольких потоков через один туннель скрывает информацию о том, сколько трафика проходит между конкретными конечными точками. Сеть таких туннелей может использоваться для создания целой виртуальной частной сети. Хосты, взаимодействующие через VPN, даже не обязательно должны осознавать, что она существует.

Глава 8.5.5. Безопасность беспроводных сетей (802.11i)

Беспроводные соединения особенно подвержены угрозам безопасности из-за отсутствия какой-либо физической безопасности среды. Несмотря на то, что удобство 802.11 способствовало повсеместному принятию этой технологии, отсутствие безопасности стало постоянной проблемой. Например, сотруднику компании слишком легко подключить точку доступа 802.11 к корпоративной сети. Поскольку радиоволны проходят через большинство стен, если точка доступа не имеет надлежащих мер безопасности, атакующий теперь может получить доступ к корпоративной сети из-за пределов здания. Точно так же компьютер с беспроводным сетевым адаптером внутри здания может подключиться к точке доступа за пределами здания, потенциально подвергая себя атаке, не говоря уже о всей корпоративной сети, если этот же компьютер, скажем, также имеет Ethernet-соединение.

В результате было проведено значительное количество работ по обеспечению безопасности Wi-Fi соединений. Довольно удивительно, что одна из ранних технологий безопасности, разработанных для 802.11, известная как Wired Equivalent Privacy (WEP), оказалась серьезно уязвимой и довольно легко взламываемой.

Стандарт IEEE 802.11i обеспечивает аутентификацию, целостность сообщений и конфиденциальность для 802.11 (Wi-Fi) на канальном уровне. WPA3 (Wi-Fi Protected Access 3) часто используется как синоним 802.11i, хотя технически это торговая марка Wi-Fi Alliance, которая сертифицирует соответствие продуктов стандарту 802.11i.

Для обеспечения обратной совместимости 802.11i включает определения алгоритмов безопасности первого поколения, включая WEP, которые теперь известны своими серьезными недостатками в области безопасности. Здесь мы сосредоточимся на новых, более сильных алгоритмах 802.11i.

802.11i поддерживает два режима аутентификации. В любом из режимов результатом успешной аутентификации является общий *Pairwise Master Key* (парный мастер-ключ) (PMK). Персональный режим, также известный как режим *Pre-Shared Key* (предварительный общий ключ) (PSK), обеспечивает более слабую безопасность, но является более удобным и экономичным для ситуаций, таких как домашняя сеть 802.11. Беспроводное устройство и точка доступа (AP) предварительно настраиваются с помощью общей *парольной фразы* — по сути, очень длинного пароля, из которого криптографически выводится Pairwise Master Key.

Более сильный режим аутентификации 802.11i основан на структуре IEEE 802.1X для контроля доступа к локальной сети (LAN), которая использует сервер аутентификации (AS), как показано на рис. 8.19. AS и AP должны быть подключены через защищенный канал и могут даже быть реализованы в одном устройстве, но логически они разделены. AP передает сообщения аутентификации между беспроводным устройством и AS. Протокол, используемый для аутентификации, называется *Extensible Authentication Protocol* (EAP). EAP разработан для поддержки множества методов аутентификации — смарт-карт, Kerberos, одноразовых паролей, аутентификации с открытым ключом и так далее — а также односторонней и взаимной аутентификации. Таким образом, EAP лучше рассматривать как структуру аутентификации, чем как протокол. Конкретные протоколы, соответствующие EAP, называются методами EAP. Например, EAP-TLS — это метод EAP, основанный на аутентификации TLS.

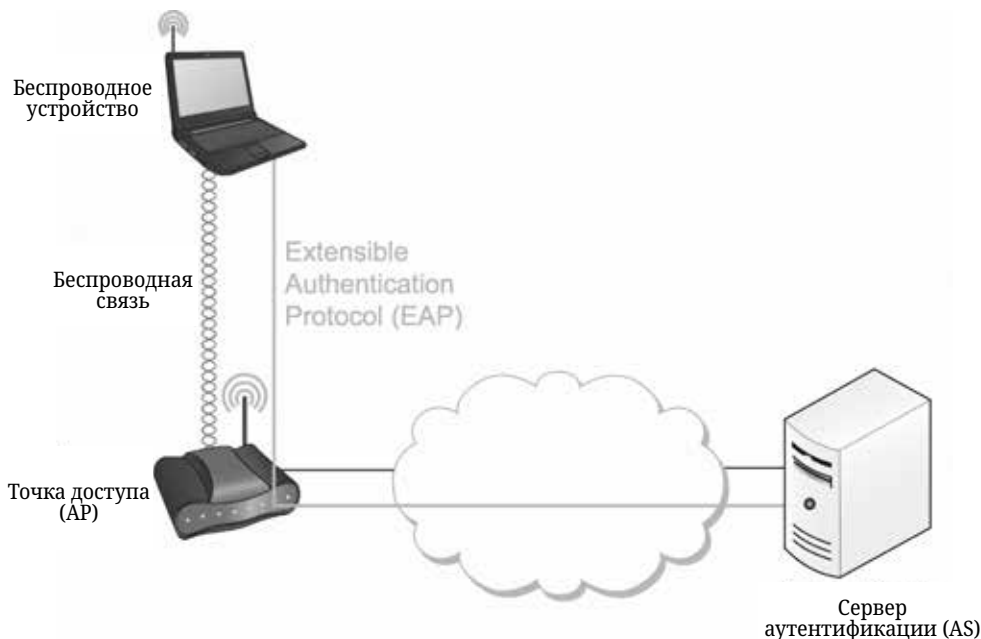


Рисунок 8.19. Использование сервера аутентификации в 802.11i.

802.11i не накладывает ограничений на то, что может использоваться в качестве основы для аутентификации в методе EAP. Однако он требует метода EAP, который выполняет взаимную аутентификацию, потому что мы хотим не только предотвратить доступ злоумышленника к сети через нашу точку доступа, но и предотвратить обман наших беспроводных устройств с помощью ложной, вредоносной точки доступа. Результатом успешной аутентификации является Pairwise Master Key, общий между беспроводным устройством и AS, который затем передается AS точке доступа.

Одним из основных различий между более сильным режимом с AS и более слабым персональным режимом является то, что первый легко поддерживает уникальный ключ для каждого клиента. Это в свою очередь упрощает изменение набора клиентов, которые могут аутентифицироваться (например, отзыв доступа у одного клиента) без необходимости изменять секрет, хранящийся у каждого клиента.

Имея Pairwise Master Key, беспроводное устройство и точка доступа выполняют протокол установления ключа сеанса, называемый 4-сторонним рукопожатием, для установления Pairwise Transient Key. Этот Pairwise Transient Key на самом деле представляет собой набор ключей, включающий ключ сеанса, называемый *Temporal Key* (временный ключ). Этот ключ сеанса используется протоколом, называемым CCMP, который обеспечивает конфиденциальность и целостность данных 802.11i.

CCMP означает CTR (режим счетчика) с CBC-MAC (шифрование с цепочкой блоков с кодом аутентификации сообщений). CCMP использует AES в режиме счетчика для шифрования конфиденциальности. Напомним, что в режиме счетчика шифрования последовательные значения счетчика включаются в шифрование последовательных блоков открытого текста.

CCMP использует код аутентификации сообщений (MAC) в качестве аутентификатора. Алгоритм MAC основан на CBC, хотя CCMP не использует CBC в шифровании конфиденциальности. Фактически CBC выполняется без передачи каких-либо зашифрованных блоков CBC, исключительно для того, чтобы последний зашифрованный блок CBC можно было использовать в качестве MAC (только первые 8 байт фактически используются). Роль вектора инициализации выполняет специально сконструированный первый блок,

включающий 48-битный номер пакета — порядковый номер. (Номер пакета также включается в шифрование конфиденциальности и служит для обнаружения атак с повторением.) MAC затем шифруется вместе с открытым текстом, чтобы предотвратить атаки на основе совпадения.

Глава 8.5.6. Брандмауэры

В то время как большая часть данного раздела сосредоточена на использовании криптографии для обеспечения таких функций безопасности, как аутентификация и конфиденциальность, существует целый ряд вопросов безопасности, которые не могут быть легко решены криптографическими средствами. Например, черви и вирусы распространяются, используя ошибки в операционных системах и прикладных программах (а иногда и человеческую доверчивость), и никакая криптография не поможет, если ваша машина имеет уязвимости. Поэтому часто используются другие подходы для предотвращения различных форм потенциально вредного трафика. Одним из наиболее распространенных способов сделать это являются межсетевые экраны (или брандмауэры) (firewalls).

Брандмауэр — это система, которая обычно находится в точке соединения между защищаемым сайтом и остальной сетью, как показано на рис. 8.20. Обычно он реализуется как «устройство» или часть маршрутизатора, хотя «персональный брандмауэр» может быть реализован на конечной пользовательской машине. Безопасность на основе межсетевых экранов зависит от того, что межсетевой экран является единственной точкой подключения к сайту извне; не должно быть возможности обойти межсетевой экран через другие шлюзы, беспроводные соединения или модемные подключения.

Метафора стены несколько вводит в заблуждение в контексте сетей, так как через межсетевой экран проходит большое количество трафика. Один из способов представить себе принцип работы межсетевого экрана заключается в том, что по умолчанию он блокирует трафик, если этот трафик не разрешен для прохождения. Например, он может фильтровать все входящие сообщения, кроме тех, которые адресованы определенному набору IP-адресов или определенным номерам портов TCP.

По сути, брандмауэр (firewall) делит сеть на более доверенную зону внутри межсетевого экрана и менее доверенную зону вне межсетевого экрана. Это полезно, если вы не хотите, чтобы внешние пользователи имели доступ к определенному хосту или службе внутри вашего сайта. Большая часть сложности заключается в том, что вы хотите предоставить разные виды доступа разным внешним пользователям, начиная от широкой общественности и заканчивая деловыми партнерами и удаленно расположенными членами вашей организации. Брандмауэр также может накладывать ограничения на исходящий трафик, чтобы предотвратить определенные атаки и ограничить потери, если злоумышленнику удастся получить доступ внутрь межсетевого экрана.

Расположение брандмауэра часто совпадает с линией раздела между глобально адресуемыми регионами и теми, которые используют локальные адреса. Поэтому функциональность трансляции сетевых адресов (NAT) и функциональность брандмауэра часто встречаются в одном устройстве, хотя логически они разделены.

Брандмауэры могут использоваться для создания нескольких *зон доверия*, таких как иерархия зон с увеличивающимся уровнем доверия. Распространенная схема включает три зоны доверия: внутреннюю сеть, DMZ (демилитаризованную зону) и остальную часть Интернета. DMZ используется для размещения служб, таких как DNS и почтовые серверы, которые должны быть доступны снаружи. Внутренняя сеть и внешний мир могут получать доступ к DMZ, но хосты в DMZ не могут получать доступ к внутренней сети; таким образом, злоумышленник, которому удалось скомпрометировать хост в открытой DMZ, все равно не сможет получить доступ к внутренней сети. DMZ можно периодически восстанавливать до чистого состояния.

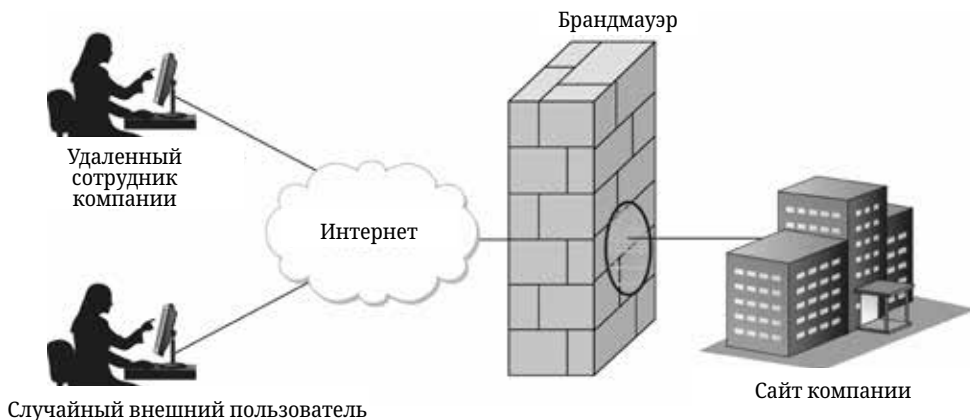


Рисунок 8.20. Брандмауэр фильтрует пакеты, проходящие между сайтом и остальной частью Интернета.

Брандмауэры фильтруют информацию на основе данных IP, TCP и UDP, среди прочего. Они настраиваются с таблицей адресов, характеризующей пакеты, которые они будут и не будут пересылать. Под адресами мы подразумеваем не только IP-адрес назначения, хотя это и является одной из возможностей. Как правило, каждая запись в таблице представляет собой 4-кортеж: она указывает IP-адрес и номер порта TCP (или UDP) для источника и места назначения.

Например, межсетевой экран может быть настроен на фильтрацию (не пересылку) всех пакетов, которые соответствуют следующему описанию:

(192.12.13.14, 1234, 128.7.6.5, 80)

Этот шаблон указывает отбрасывать все пакеты с порта 1234 на хосте 192.12.13.14, адресованные к порту 80 на хосте 128.7.6.5. (Порт 80 является хорошо известным TCP-портом для HTTP.) Конечно, зачастую непрактично называть каждый исходный хост, пакеты которого вы хотите фильтровать, поэтому шаблоны могут включать подстановочные знаки. Например,

(*, *, 128.7.6.5, 80)

указывает фильтровать все пакеты, адресованные к порту 80 на 128.7.6.5, независимо от того, какой исходный хост или порт отправил пакет. Обратите внимание, что такие адресные шаблоны требуют от межсетевого экрана принимать решения о пересылке/фильтрации на основе номеров портов уровня 4, а также адресов хостов уровня 3. По этой причине межсетевые экраны сетевого уровня иногда называют *коммутаторами уровня 4*.

В предыдущем обсуждении брандмауэр пересылает все, за исключением случаев, когда ему специально указано фильтровать определенные виды пакетов. Брандмауэр также может фильтровать все, если явно не указано разрешить пересылку, или использовать смешанную стратегию. Например, вместо блокировки доступа к порту 80 на хосте 128.7.6.5 межсетевой экран может быть настроен только на разрешение доступа к порту 25 (SMTP-порт почты) на определенном почтовом сервере, таком как

(*, *, 128.19.20.21, 25)

и блокировать весь остальной трафик. Опыт показывает, что межсетевые экраны часто настраиваются неправильно, что позволяет осуществлять небезопасный доступ. Часть проблемы заключается в том, что правила фильтрации могут пересекаться сложными способами, что затрудняет системному администратору правильное выражение

предполагаемого фильтра. Принцип проектирования, максимизирующий безопасность, заключается в настройке межсетевого экрана на отбрасывание всех пакетов, кроме тех, которые явно разрешены. Конечно, это означает, что некоторые действительные приложения могут быть случайно отключены; предположительно пользователи этих приложений в конечном итоге заметят это и попросят системного администратора внести соответствующие изменения.

Многие клиент-серверные приложения динамически назначают порт клиенту. Если клиент внутри брандмауэра инициирует доступ к внешнему серверу, ответ сервера будет адресован на динамически назначенный порт. Это создает проблему: как настроить брандмауэр так, чтобы разрешить ответный пакет от произвольного сервера, но заблокировать аналогичный пакет, для которого не было запроса от клиента? Это невозможно с помощью *брандмауэра без состояния*, который оценивает каждый пакет в изоляции. Это требует *брандмауэра с отслеживанием состояния*, который отслеживает состояние каждого соединения. Входящий пакет, адресованный на динамически назначенный порт, будет разрешен только в том случае, если это допустимый ответ в текущем состоянии соединения на данном порту.

Современные брандмауэры также понимают и фильтруют на основе многих конкретных протоколов прикладного уровня, таких как HTTP, Telnet или FTP. Они используют специфическую для этого протокола информацию, такую как URL в случае HTTP, чтобы решить, нужно ли отбрасывать сообщение.

Сильные и слабые стороны брандмауэров

В лучшем случае брандмауэр защищает сеть от нежелательного доступа из остальной части Интернета; он не может обеспечить безопасность для легитимной коммуникации между внутренней и внешней сторонами брандмауэра. В отличие от этого, механизмы безопасности, основанные на криптографии, описанные в данном разделе, способны обеспечить безопасную связь между любыми участниками где угодно. Почему же брандмауэры так широко распространены? Одна из причин заключается в том, что брандмауэры могут быть развернуты в одностороннем порядке с использованием зрелых коммерческих продуктов, в то время как для криптографически основанной безопасности требуется поддержка на обоих концах связи. Более фундаментальная причина доминирования брандмауэров заключается в том, что они концентрируют безопасность в одном централизованном месте, фактически выделяя безопасность из остальной части сети. Системный администратор может управлять брандмауэром для обеспечения безопасности, освобождая пользователей и приложения внутри брандмауэра от вопросов безопасности — по крайней мере, от некоторых видов угроз.

К сожалению, у брандмауэров есть серьезные ограничения. Поскольку брандмауэр не ограничивает связь между хостами внутри брандмауэра, злоумышленник, которому удастся запустить код внутри сети, может получить доступ ко всем локальным хостам. Как злоумышленник может проникнуть внутрь брандмауэра? Злоумышленником может быть недовольный сотрудник с легитимным доступом, или программное обеспечение злоумышленника может быть скрыто в каком-либо программном обеспечении, установленном с компакт-диска или загруженном из Интернета. Возможно также обойти брандмауэр, используя беспроводную связь или модемные соединения.

Еще одна проблема заключается в том, что любые стороны, которым предоставлен доступ через ваш брандмауэр, такие как деловые партнеры или внешние сотрудники, становятся уязвимостью безопасности. Если их безопасность не так хороша, как у вас, то злоумышленник может проникнуть в вашу систему, проникнув в их систему.

Одна из самых серьезных проблем для брандмауэров заключается в их уязвимости к эксплуатации ошибок в машинах внутри брандмауэра. Такие ошибки регулярно обнаруживаются, поэтому системный администратор должен постоянно отслеживать объявления о них. Администраторы часто не справляются с этим, поскольку нарушения безопасности через брандмауэры регулярно эксплуатируют давно известные уязвимости, для которых существуют простые решения.

Вредоносное ПО (malware) — это термин для обозначения программного обеспечения, которое предназначено для работы на компьютере скрытным образом и против воли пользователя. *Вирусы*, черви и шпионские программы являются распространенными типами *вредоносного ПО*. (Вирус иногда используется как синоним вредоносного ПО, но мы будем использовать его в более узком смысле, относящем только к определенному типу вредоносного ПО.) Вредоносный код не обязательно должен быть выполняемым объектным кодом; это может быть интерпретируемый код, такой как скрипт или выполняемая макрос-команда, например, используемая в Microsoft Word.

Вирусы и черви характеризуются способностью создавать и распространять свои копии; разница между ними заключается в том, что червь — это полная программа, которая самостоятельно размножается, в то время как вирус — это фрагмент кода, который вставляется (и вставляет свои копии) в другую программу или файл, так что он выполняется как часть выполнения этой программы или в результате открытия файла. Вирусы и черви обычно вызывают такие проблемы, как потребление пропускной способности сети, являясь побочными эффектами попыток распространения своих копий. Еще хуже, они могут намеренно повредить систему или подорвать ее безопасность различными способами. Например, они могут установить «*черный ход*» — *программное обеспечение*, позволяющее осуществлять удаленный доступ к системе без нормальной аутентификации. Это может привести к тому, что брандмауэр будет подвергаться риску, предоставляя доступ к услуге, которая должна была бы использовать собственные процедуры аутентификации, но была подорвана «черным ходом».

Шпионское ПО (spyware) — это программное обеспечение, которое без разрешения собирает и передает частную информацию о компьютерной системе или ее пользователях. Обычно шпионское ПО скрыто в полезной программе и распространяется путем намеренной установки пользователями копий этой программы. Проблема для брандмауэров заключается в том, что передача частной информации выглядит как легитимная коммуникация.

Естественный вопрос заключается в том, могут ли брандмауэры (или криптографическая безопасность) предотвратить проникновение вредоносного ПО в систему с самого начала. Большинство вредоносного ПО действительно передается через сети, хотя оно также может передаваться через портативные устройства хранения данных, такие как компакт-диски и флеш-накопители. Это является одним из аргументов в пользу подхода «блокировать все, что не разрешено явно», используемого многими администраторами в настройках их брандмауэров.

Один из подходов, который используется для обнаружения вредоносного ПО, заключается в поиске сегментов кода от известных вредоносных программ, иногда называемых сигнатурами. Этот подход имеет свои собственные трудности, так как умело разработанное вредоносное ПО может изменять свою структуру различными способами. Также возможен отрицательный эффект на производительность сети из-за столь детального анализа данных, поступающих в сеть. Криптографическая безопасность также не может полностью устранить проблему, хотя она предоставляет средства для аутентификации отправителя программного обеспечения и обнаружения любых изменений, таких как вставка вируса.

Системы, связанные с брандмауэрами, известны как *системы обнаружения вторжений* (IDS) и *системы предотвращения вторжений* (IPS). Эти системы пытаются обнаружить аномальную активность, такую как необычно большой объем трафика, направленный на определенный хост или номер порта, и генерируют сигналы тревоги для сетевых администраторов или даже принимают прямые меры для ограничения возможной атаки. Хотя сегодня существуют коммерческие продукты в этой области, это все еще развивающаяся сфера.

Перспектива: блокчейн и децентрализованный интернет

Скорее всего, без особых раздумий пользователи возложили огромное доверие на приложения, которые они используют, особенно такие, как Facebook и Google, которые не только хранят их личные фотографии и видео, но и управляют их идентичностью (например,

предоставляют единую регистрацию для других веб-приложений). Это беспокоит многих людей, что вызвало интерес к *децентрализованным платформам* — системам, для которых пользователи не должны доверять третьей стороне. Такие системы часто строятся на основе криптовалюты, такой как Bitcoin, не из-за его денежной ценности, а потому, что криптовалюта сама по себе основана на децентрализованной технологии (называемой *блокчейном*), которую не контролирует ни одна организация. Легко отвлечься на всю шумиху, но блокчейн по сути является децентрализованным журналом (реестром), в который любой может записать «факт» и позже доказать миру, что этот факт был записан.

Blockstack — это реализация с открытым исходным кодом децентрализованной платформы, включающей блокчейн, но, что более интересно, она использовалась для реализации службы самоидентификации для интернет-приложений. Служба самоидентификации — это тип службы идентификации, которая *административно децентрализована*: у нее нет определенного оператора службы, и ни один субъект не может контролировать, кто может создавать идентичность, а кто не может. Blockstack использует общедоступный блокчейн для создания реплицируемого журнала базы данных идентификаций. Когда этот журнал базы данных воспроизводится узлом Blockstack, он создает такое же представление всех идентификаций в системе, как и любой другой узел Blockstack, считающий то же представление базового блокчейна. Любой может зарегистрировать идентификацию в Blockstack, добавив запись в блокчейн.

Вместо того чтобы требовать от пользователей доверия к определенному набору поставщиков идентификаций, протокол идентификации Blockstack просит пользователей доверять тому, что большинство узлов принятия решений в блокчейне (называемых *майнерами*) будут поддерживать порядок записей (*транзакций*). Базовый блокчейн предоставляет криптовалюту для стимулирования майнеров к честному участию. В нормальных условиях майнеры получают наибольшее количество криптовалюты, участвуя честно. Это позволяет журналу базы данных Blockstack оставаться защищенным от вмешательства без конкретного оператора службы. Злоумышленник, желающий подделать журнал, должен конкурировать с большинством майнеров, чтобы создать альтернативную историю транзакций в базовом блокчейне, которую сеть узлов блокчейна примет как каноническую историю записей.

Протокол для чтения и добавления в журнал базы данных идентификаций Blockstack работает на логическом уровне выше блокчейна. Транзакции блокчейна являются кадрами данных для записей журнала базы данных идентификаций. Клиент добавляет запись в журнал базы данных идентификаций, отправляя транзакцию блокчейна, которая включает запись журнала базы данных, а клиент считывает журнал, извлекая записи журнала из транзакций блокчейна в порядке блокчейна. Это позволяет реализовать журнал базы данных «поверх» любого блокчейна.

Идентификаторы в Blockstack отличают выбранное пользователем имя. Протокол идентификации Blockstack связывает имя с открытым ключом и некоторым состоянием *маршрутизации* (описано ниже). Он гарантирует, что имена уникальны на глобальном уровне, назначая их по принципу «первым пришел, первым обслужен».

Имена регистрируются в два этапа: сначала связывается открытый ключ клиента с «посоленной» хеш-суммой имени, а затем раскрывается само имя. Этот двухэтапный процесс необходим для предотвращения перехвата — только клиент, подписавший хеш-сумму имени, может раскрыть имя, и только клиент, вычисливший хеш-сумму, может раскрыть исходное значение. После регистрации имени только владелец закрытого ключа имени может передать или отозвать имя либо обновить его состояние маршрутизации.

Каждое имя в Blockstack связано с частью состояния маршрутизации, которое содержит один или несколько URL-адресов, указывающих на место, где можно найти информацию об идентификации пользователя в интернете. Эти данные слишком большие и дорогостоящие для хранения непосредственно в блокчейне, поэтому Blockstack реализует слой косвенного обращения: хеш состояния маршрутизации записывается в журнал базы данных идентификации, а узлы Blockstack реализуют сеть обмена для распространения

и аутентификации состояния маршрутизации. Каждый узел поддерживает полную копию состояния маршрутизации.

Объединив все вместе, рис. 8.21 показывает, как работает разрешение имени в его соответствующее состояние идентификации. Получив имя, клиент сначала запрашивает у узла Blockstack соответствующий открытый ключ и состояние маршрутизации (Шаг 1). Получив состояние маршрутизации, клиент получает данные идентификации, разрешая URL-адрес(а), содержащиеся в нем, и аутентифицирует информацию об идентификации, проверяя, что она подписана открытым ключом имени (Шаг 2).

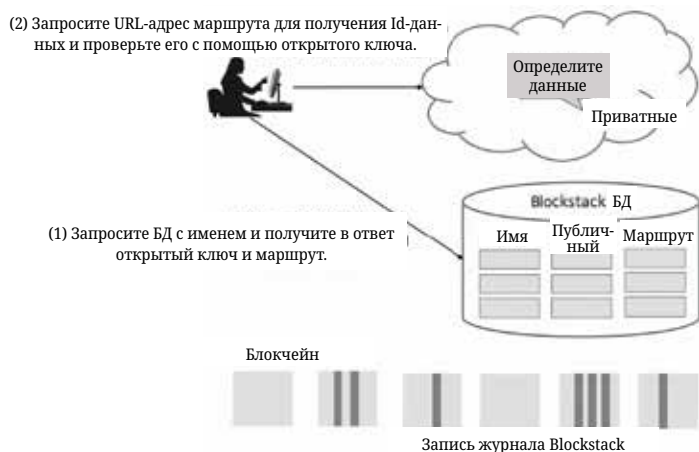


Рисунок 8.21. Децентрализованное управление идентификацией, построенное на основе блокчейна.

Раздел 9.

Приложения

Это еще не конец.
Это даже не начало конца.
Но, возможно, это конец начала.
Уинстон Черчилль

Проблема: приложениям нужны собственные протоколы

Мы начали эту книгу с разговора о приложениях — от веб-браузеров до инструментов для видеоконференций, — которые люди хотят запускать по компьютерным сетям. В предыдущих разделах мы рассмотрели, шаг за шагом, сетевую инфраструктуру, необходимую для реализации таких приложений. Теперь мы возвращаемся к сетевым приложениям. Эти приложения частично состоят из сетевого протокола (в смысле обмена сообщениями с их аналогами на других машинах) и частично из традиционной программы (в смысле взаимодействия с оконной системой, файловой системой и, в конечном итоге, с пользователем). В этом разделе рассматриваются некоторые популярные сетевые приложения, доступные сегодня.

Рассмотрение приложений подчеркивает *системный подход*, на который мы обращали внимание на протяжении всей книги. То есть лучший способ создать эффективные сетевые приложения — понять строительные блоки, которые может предоставить сеть, и как эти блоки могут взаимодействовать друг с другом. Таким образом, например, конкретному сетевому приложению может понадобиться надежный транспортный протокол, механизмы аутентификации и конфиденциальности, а также возможности выделения ресурсов базовой сети. Приложения часто работают лучше, когда разработчик приложения знает, как лучше всего использовать эти возможности (и существует множество примеров приложений, которые плохо используют доступные сетевые возможности). Приложениям также часто нужны собственные протоколы, в большинстве случаев использующие те же принципы, которые мы рассмотрели при изучении протоколов нижних уровней. Таким образом, наша цель в этом разделе — объединить идеи и техники, уже описанные ранее, для создания эффективных сетевых приложений. Иными словами, если вы когда-либо возьметесь за написание сетевого приложения, то вы по определению также станете разработчиком (и реализатором) протокола.

Мы продолжаем, изучая разнообразие известных и менее известных сетевых приложений. Они варьируются от обмена электронной почтой и серфинга в интернете до интеграции приложений между предприятиями, мультимедийных приложений, таких как видеоконференции, управления набором сетевых элементов и новых одноранговых и сетей распространения контента. Этот список далеко не исчерпывающий, но он служит для иллюстрации многих ключевых принципов проектирования и создания приложений. Приложения должны выбирать и использовать подходящие строительные блоки, доступные на других уровнях либо внутри сети, либо в протоколах хостов, а затем дополнять эти базовые сервисы, чтобы обеспечить необходимое для приложения точное общение.

Глава 9.1. Традиционные приложения

Мы начинаем наше обсуждение приложений, сосредотачиваясь на двух самых популярных — Всемирной паутине и электронной почте. В широком смысле оба эти приложения используют парадигму «запрос/ответ» — пользователи отправляют запросы серверам, которые затем соответствующим образом отвечают. Мы называем их «традиционными» приложениями, потому что они являются типичными для приложений,

существовавших с ранних дней компьютерных сетей (хотя веб гораздо новее электронной почты, но его корни уходят в файловые передачи, которые предшествовали ему). В отличие от этого, в следующих главах рассматривается класс приложений, которые стали популярны относительно недавно: потоковые приложения (например, мультимедийные приложения, такие как видео и аудио) и различные приложения на основе наложенных сетей. (Заметьте, что здесь есть некоторое размывание классов, так как вы, конечно, можете получить доступ к потоковым мультимедийным данным через веб, но сейчас мы сосредоточимся на общем использовании веба для запроса страниц, изображений и т. д.).

Прежде чем внимательно рассмотреть каждое из этих приложений, нужно сделать три общих замечания. Во-первых, важно различать *программы* приложений и *протоколы* приложений. Например, протокол передачи гипертекста (HTTP) — это протокол приложения, который используется для получения веб-страниц с удаленных серверов. Многие различные программы приложений — то есть веб-клиенты, такие как Internet Explorer, Chrome, Firefox и Safari — предоставляют пользователям различный внешний вид и ощущения, но все они используют один и тот же протокол HTTP для общения с веб-серверами через интернет. Действительно, именно благодаря тому, что протокол опубликован и стандартизирован, программы приложений, разработанные разными компаниями и отдельными людьми, могут взаимодействовать друг с другом. Это позволяет множеству браузеров взаимодействовать со всеми веб-серверами (которых также существует много разновидностей).

В этой главе рассматриваются два очень широко используемых стандартизированных протокола приложений:

- Простой протокол передачи почты (SMTP) используется для обмена электронной почтой.
- Протокол передачи гипертекста (HTTP) используется для общения между веб-браузерами и веб-серверами.

Во-вторых, мы замечаем, что многие протоколы прикладного уровня, включая HTTP и SMTP, имеют сопутствующий протокол, который определяет формат данных, которые могут быть обменены. Это одна из причин, почему эти протоколы относительно просты: большая часть сложности управляется в этом сопутствующем стандарте. Например, SMTP — это протокол для обмена электронными письмами, но RFC 822 и MIME (Multipurpose Internet Mail Extensions) определяют формат электронных писем. Аналогично HTTP — это протокол для получения веб-страниц, но HTML (HyperText Markup Language) — это сопутствующая спецификация, которая определяет базовую форму этих страниц.

Наконец, поскольку описанные в этом разделе протоколы приложений следуют одной и той же модели общения «запрос/ответ», можно было бы ожидать, что они будут построены на транспортном протоколе удаленного вызова процедур (RPC). Однако это не так, поскольку они реализованы поверх TCP. По сути, каждый протокол заново изобретает простой механизм, подобный RPC, поверх надежного транспортного протокола (TCP). Мы говорим «простой», потому что каждый протокол не предназначен для поддержки произвольных удаленных вызовов процедур, как обсуждалось в одном из предыдущих разделов, а предназначен для отправки и ответа на конкретный набор запросов. Интересно, что подход, принятый в HTTP, оказался весьма мощным, что привело к его широкому принятию *архитектурой веб-сервисов*, где общие механизмы RPC построены *поверх HTTP*, а не наоборот. Об этом подробнее ниже.

Глава 9.1.1. Электронная почта (SMTP, MIME, IMAP)

Электронная почта — одно из самых старых сетевых приложений. В конце концов, что может быть естественнее, чем желание отправить сообщение пользователю на другом конце страны, с которым вы только что установили связь? Это вер-

сия XX века фразы «Мистер Ватсон, идите сюда... Я хочу вас видеть». Удивительно, но пионеры ARPANET не рассматривали электронную почту как ключевое приложение при создании сети — удаленный доступ к вычислительным ресурсам был основной целью проектирования, — но в итоге она оказалась оригинальным «убийственным приложением» Интернета.

Как упоминалось выше, важно (1) различать пользовательский интерфейс (т.е. ваш почтовый клиент) и основные протоколы передачи сообщений (такие как SMTP или IMAP), и (2) различать этот протокол передачи и сопутствующий стандарт (RFC 822 и MIME), который определяет формат передаваемых сообщений. Начнем с рассмотрения формата сообщения.

Формат сообщения

RFC 822 определяет сообщения, состоящие из двух частей: *заголовка* и *тела*. Обе части представлены в ASCII-тексте. Первоначально предполагалось, что тело будет простым текстом. Это по-прежнему так, хотя RFC 822 был дополнен MIME, чтобы позволить телу сообщения нести различные данные. Эти данные по-прежнему представлены в виде ASCII-текста, но поскольку это может быть закодированная версия, скажем, изображения в формате JPEG, они не обязательно читаемы пользователем. Подробнее о MIME чуть позже.

Заголовок сообщения — это серия строк, заканчивающихся <CRLF> (возврат каретки и перевод строки — это пара управляющих символов ASCII, часто используемых для обозначения конца строки текста). Заголовок отделяется от тела сообщения пустой строкой. Каждая строка заголовка содержит тип и значение, разделенные двоеточием. Многие из этих строк заголовка знакомы пользователям, так как их просят заполнить при составлении электронного письма; например, заголовок указывает получателя сообщения, и заголовок говорит что-то о цели сообщения. Другие заголовки заполняются системой доставки почты. Примеры включают (когда сообщение было отправлено), (какой пользователь отправил сообщение) и (каждый почтовый сервер, который обрабатывал это сообщение). Конечно, есть много других строк заголовка; заинтересованный читатель может обратиться к RFC 822.

RFC 822 был расширен в 1993 году (и обновлялся несколько раз с тех пор), чтобы позволить электронным сообщениям нести различные типы данных: аудио, видео, изображения, PDF-документы и так далее. MIME состоит из трех основных частей. Первая часть — это набор строк заголовка, дополняющих оригинальный набор, определенный в RFC 822. Эти строки заголовка в различных аспектах описывают данные, содержащиеся в теле сообщения. Они включают (версию MIME, которая используется), (человекочитаемое описание содержимого сообщения, аналогичное строке темы), (тип данных, содержащихся в сообщении), и (как данные в теле сообщения закодированы).

Вторая часть — это определения для набора типов содержимого (и подтипов). Например, MIME определяет два разных типа неподвижных изображений, обозначенных `'image/jpeg'` и `'image/gif'`, каждый с очевидным значением. Еще один пример — `'text/plain'` относится к простому тексту, который можно найти в стандартном сообщении в стиле 822, тогда как `'text/html'` обозначает сообщение, содержащее «размеченный» текст (текст, использующий специальные шрифты, курсив и т.д.). Еще один пример — MIME определяет тип `'application'`, где подтипы соответствуют выводу разных прикладных программ (например, `'application/postscript'` и `'application/msword'`).

MIME также определяет тип `'multipart'`, который указывает, как структурировано сообщение, содержащее более одного типа данных. Это похоже на язык программирования, который определяет как базовые типы (например, целые числа и числа с плавающей точкой), так и составные типы (например, структуры и массивы). Один из возможных подтипов `'multipart'` — `'mixed'`, который указывает, что сообщение содержит набор не-

зависимых фрагментов данных в определенном порядке. Каждый фрагмент имеет свою собственную строку заголовка, описывающую тип этого фрагмента.

Третья часть — это способ кодирования различных типов данных, чтобы их можно было отправить в ASCII-сообщении электронной почты. Проблема в том, что для некоторых типов данных (например, изображения JPEG) любой 8-битный байт в изображении может содержать одно из 256 разных значений. Только подмножество этих значений является допустимыми символами ASCII. Важно, чтобы сообщения электронной почты содержали только ASCII, поскольку они могут пройти через несколько промежуточных систем (шлюзов, как описано ниже), которые предполагают, что вся электронная почта в формате ASCII, и испортят сообщение, если оно содержит не-ASCII символы. Чтобы решить эту проблему, MIME использует простой способ кодирования двоичных данных в набор символов ASCII. Это кодирование называется `base64`. Идея заключается в том, чтобы сопоставить каждые три байта исходных двоичных данных четырем символам ASCII. Это делается путем группировки двоичных данных в 24-битные единицы и разделения каждой такой единицы на четыре 6-битных части. Каждая 6-битная часть сопоставляется одному из 64 допустимых символов ASCII; например, 0 соответствует A, 1 соответствует B и так далее. Если вы посмотрите на сообщение, закодированное с использованием схемы кодирования base64, вы заметите только 52 прописные и строчные буквы, 10 цифр от 0 до 9 и специальные символы + и /. Это первые 64 значения в наборе символов ASCII.

Как дополнение, чтобы сделать чтение почты максимально безболезненным для тех, кто по-прежнему использует только текстовые почтовые клиенты, сообщение MIME, состоящее только из обычного текста, может быть закодировано с использованием 7-битного ASCII. Также существует читаемое кодирование для данных, в основном представленных в ASCII.

Соединяя все вместе, сообщение, содержащее простой текст, изображение JPEG и файл PostScript, будет выглядеть примерно так:

```
MIME-Version: 1.0
Content-Type: multipart/mixed;
    boundary=«-----417CA6E2DE4ABCAFBC5»
From: Alice Smith <Alice@cisco.com>
To: Bob@cs.Princeton.edu
Subject: promised material
Date: Mon, 07 Sep 1998 19:45:19 -0400

-----417CA6E2DE4ABCAFBC5
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit

Bob,
Here is the jpeg image and draft report I promised.
--Alice
-----417CA6E2DE4ABCAFBC5
Content-Type: image/jpeg
Content-Transfer-Encoding: base64
... (здесь будет закодированное изображение в формате base64)
-----417CA6E2DE4ABCAFBC5
Content-Type: application/postscript; name=«draft.ps»
Content-Transfer-Encoding: 7bit
... (здесь будет кодировка PostScript-документа)
```

Каждый фрагмент сообщения отделяется строкой символов, которая не встречается в самих данных. Каждый фрагмент имеет свои собственные строки.

Передача сообщений

На протяжении многих лет основная часть электронной почты перемещалась с хоста на хост, используя только SMTP. Хотя SMTP продолжает играть центральную роль, он теперь является лишь одним из нескольких протоколов электронной почты; Internet Message Access Protocol (IMAP) и Post Office Protocol (POP) — два других важных протокола для получения почтовых сообщений. Мы начнем наше обсуждение с SMTP, а затем перейдем к IMAP.

Чтобы поставить SMTP в нужный контекст, нам нужно определить ключевых участников. Во-первых, пользователи взаимодействуют с *почтовым клиентом*, когда составляют, хранят, ищут и читают свою электронную почту. Существует бесчисленное множество почтовых клиентов, так же как и множество веб-браузеров. В ранние дни Интернета пользователи обычно входили в систему, на которой находился их *почтовый ящик*, и почтовый клиент, который они запускали, был локальной прикладной программой, извлекающей сообщения из файловой системы. Сегодня, конечно, пользователи удаленно получают доступ к своему почтовому ящику со своего ноутбука или смартфона; они не сначала входят на хост, где хранится их почта (почтовый сервер). Второй протокол передачи почты, такой как POP или IMAP, используется для удаленной загрузки электронной почты с почтового сервера на устройство пользователя.

Во-вторых, на каждом хосте, который держит *почтовый ящик*, работает *почтовый демон* (или процесс, *daemon*). Этот процесс, также называемый агентом *передачи сообщений* (MTA), выполняет роль почтового отделения: пользователи (или их почтовые клиенты) передают демону (*daemon*) сообщения, которые они хотят отправить другим пользователям, демон (*daemon*) использует SMTP поверх TCP для передачи сообщения демону, работающему на другой машине, и демон помещает входящие сообщения в почтовый ящик пользователя (где почтовый клиент этого пользователя позже может их найти). Поскольку SMTP — это протокол, который может реализовать любой желающий, теоретически может существовать множество разных реализаций почтового демона. Оказывается, однако, что существует всего несколько популярных реализаций, среди которых наиболее широко распространены старая программа *sendmail* из Berkeley Unix и *postfix*.

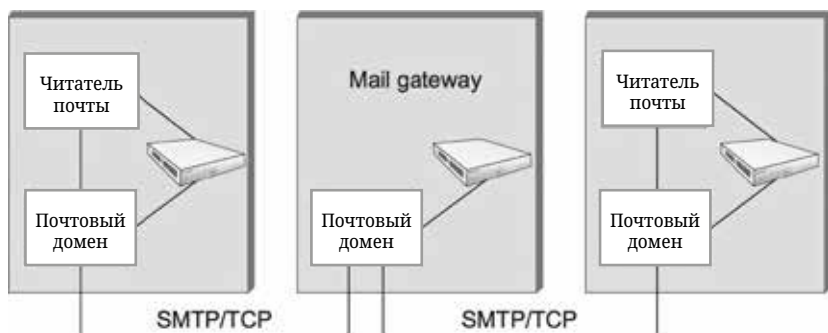


Рисунок 9.1. Последовательность хранения и пересылки почтовых сообщений почтовыми шлюзами.

Хотя вполне возможно, что MTA на машине отправителя устанавливает соединение SMTP/TCP с MTA на почтовом сервере получателя, в большинстве случаев почта проходит через один или несколько *почтовых шлюзов* на пути от хоста отправителя к хосту получателя. Как и конечные хосты, эти шлюзы также запускают процесс агента передачи сообщений. Не случайно, что эти промежуточные узлы называются *шлюзами*, поскольку их задача — хранить и пересылать сообщения электронной почты, как «IP-шлюз» (который мы называем *маршрутизатором*) хранит и пересылает IP-дейтаграммы. Единственное от-

личие заключается в том, что почтовый шлюз обычно буферизует сообщения на диске и готов повторять попытки их передачи на следующую машину в течение нескольких дней, тогда как IP-маршрутизатор буферизует дейтаграммы в памяти и готов повторять попытки их передачи только в течение долей секунды. Рис. 9.1 иллюстрирует двухшаговый путь от отправителя к получателю.

Вы можете спросить: почему необходимы почтовые шлюзы? Почему хост отправителя не может отправить сообщение непосредственно хосту получателя? Одна из причин в том, что получатель не хочет включать конкретный хост, на котором он читает почту, в свой адрес. Другая причина — масштабируемость: в больших организациях часто несколько различных машин содержат *почтовые ящики* организации. Например, почта, отправленная на `bob@cs.princeton.edu`, сначала отправляется на почтовый шлюз в CS Department в Принстоне (то есть на хост с именем `cs.princeton.edu`), а затем перенаправляется — с использованием второго соединения — на конкретную машину, на которой находится почтовый ящик Боба. Шлюз перенаправления поддерживает базу данных, которая сопоставляет пользователей с машиной, на которой находится их почтовый ящик; отправителю не нужно знать это конкретное имя. (Список строк заголовков в сообщении поможет вам отследить почтовые шлюзы, через которые прошло данное сообщение.) Еще одна причина, особенно актуальная в ранние дни электронной почты, заключается в том, что машина, на которой размещен почтовый ящик любого пользователя, может не всегда быть доступна, в этом случае почтовый шлюз удерживает сообщение до тех пор, пока его не удастся доставить.

Независимо от того, сколько почтовых шлюзов находится на пути, независимое соединение SMTP используется между каждым хостом для перемещения сообщения ближе к получателю. Каждая сессия SMTP включает диалог между двумя почтовыми демонами, один из которых действует как клиент, а другой — как сервер. В течение одной сессии может быть передано несколько сообщений между двумя хостами. Поскольку RFC 822 определяет сообщения с использованием ASCII как основной формы представления, неудивительно, что SMTP также основан на ASCII. Это означает, что человек с клавиатурой может притвориться программой SMTP-клиента.

SMTP лучше всего понимается на простом примере. Следующий обмен происходит между отправляющим хостом `cs.princeton.edu` и принимающим хостом `cisco.com`. В этом случае пользователь Боб из Принстона пытается отправить письмо пользователям Алисе и Тому в Cisco. Добавлены дополнительные пустые строки, чтобы диалог был более читаемым.

```
HELO cs.princeton.edu
250 Hello daemon@mail.cs.princeton.edu [128.12.169.24]
MAIL FROM:<Bob@cs.princeton.edu>
250 OK
RCPT TO:<Alice@cisco.com>
250 OK
RCPT TO:<Tom@cisco.com>
550 No such user here
DATA
354 Start mail input; end with <CRLF>.<CRLF>
Blah blah blah...
...etc. etc. etc.
<CRLF>.<CRLF>
250 OK
QUIT
221 Closing connection
```

Как видно, SMTP включает последовательность обменов между клиентом и сервером. В каждом обмене клиент отправляет команду (например, QUIT), а сервер отвечает кодом (например, 221). Сервер также возвращает человекочитаемое объяснение кода (напри-

мер, «Прощание»). В этом конкретном примере клиент сначала идентифицирует себя серверу с помощью команды HELO. Он передает свое доменное имя в качестве аргумента. Сервер проверяет, соответствует ли это имя IP-адресу, используемому соединением TCP; вы заметите, что сервер сообщает этот IP-адрес клиенту. Затем клиент спрашивает сервер, готов ли он принять почту для двух разных пользователей; сервер отвечает «да» одному и «нет» другому. Затем клиент отправляет сообщение, которое завершается строкой с одной точкой («.»). Наконец, клиент завершает соединение.

Существуют, конечно, и другие команды и коды ответа. Например, сервер может ответить на команду RCPT клиента кодом 251, что указывает на то, что у пользователя нет почтового ящика на этом хосте, но сервер обещает перенаправить сообщение на другой почтовый демон. Иными словами, хост выполняет функцию почтового шлюза. В другом примере клиент может выдать операцию VRFY для проверки адреса электронной почты пользователя, но без фактической отправки сообщения пользователю.

Единственный интересный момент — это аргументы для операций MAIL и RCPT; например, FROM:<Bob@cs.princeton.edu> и TO:<Alice@cisco.com> соответственно. Они очень похожи на поля заголовка 822, и в некотором смысле так и есть. На самом деле почтовый демон анализирует сообщение, чтобы извлечь информацию, необходимую для выполнения SMTP. Извлеченная информация формирует так называемый конверт для сообщения. SMTP-клиент использует этот конверт для параметризации своего обмена с SMTP-сервером. Один исторический факт: причиной популярности sendmail стало то, что никто не хотел заново реализовывать эту функцию анализа сообщений. Хотя сегодняшние адреса электронной почты выглядят довольно просто (например, Bob@cs.princeton.edu), это не всегда было так. В те времена, когда не все были подключены к Интернету, можно было увидеть адреса электронной почты вида user %host@site!neighbor.

Почтовый клиент

Последний шаг — это фактическое получение пользователем своих сообщений из почтового ящика, их чтение, ответ на них и, возможно, сохранение копии для будущего использования. Пользователь выполняет все эти действия, взаимодействуя с почтовым клиентом. Как упоминалось ранее, этот клиент изначально был просто программой, работающей на той же машине, что и почтовый ящик пользователя, и в этом случае он мог просто читать и записывать файл, который реализует почтовый ящик. Это было обычным случаем в эпоху до ноутбуков. Сегодня чаще всего пользователь получает доступ к своему почтовому ящику с удаленной машины, используя еще один протокол, такой как POP или IMAP. Обсуждение аспектов пользовательского интерфейса почтового клиента выходит за рамки этой книги, но в нашу компетенцию определенно входит обсуждение протокола доступа. Мы рассмотрим IMAP в частности.

IMAP во многом похож на SMTP. Это клиент-серверный протокол, работающий поверх TCP, где клиент (работающий на настольной машине пользователя) выдает команды в виде строк ASCII, заканчивающихся <CRLF>, и почтовый сервер (работающий на машине, которая поддерживает почтовый ящик пользователя) отвечает аналогично. Обмен начинается с аутентификации клиента и идентификации почтового ящика, к которому он хочет получить доступ. Это можно представить с помощью простой диаграммы переходов состояний, показанной на рис. 9.2. На этой диаграмме LOGIN и LOGOUT — это примеры команд, которые может выдать клиент, а OK — один из возможных ответов сервера. Другие распространенные команды включают и EXPUNGE с очевидными значениями. Дополнительные ответы сервера включают NO (клиент не имеет разрешения на выполнение этой операции) и BAD (команда неверно сформирована).

Когда пользователь запрашивает получение (FETCH) сообщения, сервер возвращает его в формате MIME, а почтовый клиент декодирует его. В дополнение к самому сообщению IMAP также определяет набор *атрибутов* сообщений, которые обмениваются в рамках других команд, независимо от передачи самого сообщения. Атрибуты сообщений включают информацию о размере сообщения и, что более интересно, различные флаги,

связанные с сообщением (например, и Recent). Эти флаги используются для синхронизации клиента и сервера; то есть когда пользователь удаляет сообщение в почтовом клиенте, клиент должен сообщить об этом почтовому серверу. Позже, если пользователь решит удалить все удаленные сообщения, клиент выдает команду EXPUNGE серверу, который знает, что нужно фактически удалить все ранее удаленные из почтового ящика сообщения.



- (1) Соединение без предварительной аутентификации (приветствие OK)
- (2) Соединение с предварительной аутентификацией (приветствие PREAUTH)
- (3) Отклоненное соединение (приветствие BYE)
- (4) Успешная команда LOGIN или AUTHENTICATE
- (5) Успешная команда SELECT или EXAMINE
- (6) Команда CLOSE, или неудачная команда SELECT или EXAMINE
- (7) Команда LOGOUT, выключение сервера или закрытие соединения

Рисунок 9.2. Диаграмма переходов состояний IMAP.

Наконец, обратите внимание, что когда пользователь отвечает на сообщение или отправляет новое сообщение, программа чтения почты не пересылает его с клиента на почтовый сервер по протоколу IMAP, а использует SMTP. Это означает, что почтовый сервер пользователя фактически является первым почтовым шлюзом на пути от рабочего стола к почтовому ящику получателя.

Глава 9.1.2. Всемирная паутина (HTTP)

Всемирная паутина (Web) оказалась настолько успешной и сделала Интернет доступным для стольких людей, что иногда кажется, будто она синонимична Интернету. На самом деле разработка системы, ставшей паутиной, началась около 1989 года, гораздо позже того, как Интернет стал широко распространенной системой. Первоначальная цель паутины за-

ключалась в поиске способов организации и извлечения информации, основываясь на идеях гипертекста — взаимосвязанных документов, которые существовали с 1960-х годов.¹ Основная идея гипертекста заключается в том, что один документ может ссылаться на другой документ, а протокол (HTTP) и язык разметки документов (HTML) были разработаны для достижения этой цели.

Можно представить паутину как набор взаимодействующих клиентов и серверов, все из которых говорят на одном языке: HTTP. Большинство людей сталкиваются с паутиной через графические клиентские программы или веб-браузеры, такие как Safari, Chrome, Firefox или Internet Explorer. На рис. 9.3 показан браузер Safari, отображающий страницу с информацией Принстонского университета.

Очевидно, что если вы хотите организовать информацию в систему связанных документов или объектов, вам нужно иметь возможность получить один документ, чтобы начать. Поэтому любой веб-браузер имеет функцию, позволяющую пользователю получить объект, открыв URL. Универсальные указатели ресурсов (URLs) настолько знакомы большинству из нас, что легко забыть, что они существуют не так давно. Они предоставляют информацию, которая позволяет найти объекты в сети, и выглядят следующим образом:

`http://www.cs.princeton.edu/index.html`

Если вы откроете этот конкретный URL, ваш веб-браузер установит TCP-соединение с веб-сервером на машине под названием `www.cs.princeton.edu` и сразу же получит и отобразит файл под названием `index.html`. Большинство файлов в сети содержат изображения и текст, а многие имеют другие объекты, такие как аудио- и видеоклипы, фрагменты кода и т. д. Они также часто включают URL-адреса, указывающие на другие файлы, которые могут находиться на других машинах, что является основой части «гипертекст» в HTTP и HTML. Веб-браузер имеет способ распознавать URL-адреса (часто выделяя или подчеркивая текст), и вы можете попросить браузер открыть их. Эти встроенные URL-адреса называются *гипертекстовыми ссылками*. Когда вы попросите свой веб-браузер открыть одну из этих встроенных URL-адресов (например, щелкнув на нее мышью), он откроет новое соединение, получит и отобразит новый файл. Это называется *переходом по ссылке*. Таким образом, становится очень легко переходить с одной машины на другую по сети, следуя ссылкам на всевозможную информацию. Как только у вас появляется возможность встроить ссылку в документ и позволить пользователю перейти по этой ссылке, чтобы получить другой документ, у вас есть основа гипертекстовой системы.



Рисунок 9.3. Веб-браузер Safari.

Когда вы просите свой браузер просмотреть страницу, ваш браузер (клиент) получает страницу с сервера, используя HTTP поверх TCP. Как и SMTP, HTTP — это текстоориенти-

¹ Краткая история Сети, представленная консорциумом World Wide Web, ведет свое начало от статьи 1945 года, описывающей связи между документами на микрофишах.

рованный протокол. В своей основе HTTP является протоколом «запрос/ответ», где каждое сообщение имеет общую форму:

```
START_LINE <CRLF> MESSAGE_HEADER <CRLF>
<CRLF> MESSAGE_BODY <CRLF>
```

где, как и раньше, <CRLF> означает возврат каретки + перевод строки. Первая строка (START_LINE) указывает, является ли это запросом или ответом. По сути, она определяет «удаленную процедуру», которая должна быть выполнена (в случае сообщения-запроса), или статус запроса (в случае сообщения-ответа). Следующий набор строк указывает коллекцию опций и параметров, которые уточняют запрос или ответ. Таких строк-заголовков (MESSAGE_HEADER) может быть ноль или больше, и этот набор завершается пустой строкой; каждая из этих строк выглядит как строка заголовка в электронном письме. HTTP определяет множество возможных типов заголовков, некоторые из которых относятся к сообщениям-запросам, некоторые к сообщениям-ответам, а некоторые к данным, содержащимся в теле сообщения. Вместо полного списка возможных типов заголовков мы приводим лишь несколько примеров. Наконец, после пустой строки следует содержимое запрашиваемого сообщения (MESSAGE_BODY); в этой части сообщения сервер помещает запрашиваемую страницу при ответе на запрос, и эта часть обычно пуста для сообщений-запросов.

Почему HTTP работает поверх TCP? Разработчики не обязательно должны были делать это так, но TCP довольно хорошо соответствует требованиям HTTP, особенно предоставляя надежную доставку (кто хочет веб-страницу с отсутствующими данными?), управление потоком и управление перегрузкой. Однако, как мы увидим далее, есть несколько проблем, которые могут возникнуть при построении протокола «запрос/ответ» поверх TCP, особенно если игнорировать тонкости взаимодействия между протоколами прикладного и транспортного уровней.

Запрос сообщений

Первая строка сообщения запроса HTTP указывает три вещи: операцию, которую нужно выполнить, веб-страницу, над которой должна быть выполнена операция, и версию HTTP, которая используется. Хотя HTTP определяет множество возможных операций запроса — включая операции записи, которые позволяют разместить веб-страницу на сервере, — две самые распространенные операции это GET (получить указанную веб-страницу) и HEAD (получить информацию о статусе указанной веб-страницы). Первая, очевидно, используется, когда ваш браузер хочет получить и отобразить веб-страницу. Вторая используется для проверки валидности гипертекстовой ссылки или для того, чтобы увидеть, была ли конкретная страница изменена с тех пор, как браузер последний раз ее запрашивал. Полный набор операций представлен в табл. 9.1. Хотя это звучит безобидно, команда POST позволяет совершать множество злоупотреблений (включая спам) в Интернете.

Таблица 9.1.
Операции HTTP-запроса.

Операция	Описание
OPTION	Запрос информации о доступных вариантах
GET	Получение документа, указанного в URL
HEAD	Получение метаинформации о документе, указанном в URL
POST	Передача информации(например, аннотации) на сервер
PUT	Сохранение документа по указанному URL
DELETE	Удаление указанного URL
TRACE	Сообщение о запросе шлейфа
CONNECT	Для использования прокси-серверами

Например, `START_LINE`

```
GET http://www.cs.princeton.edu/index.html
HTTP/1.1
```

говорит о том, что клиент хочет, чтобы сервер на хосте вернул страницу с именем. Этот конкретный пример использует *абсолютный* URL. Также возможно использовать *относительный* идентификатор и указать имя хоста в одной из строк `MESSAGE_HEADER`; например,

```
GET index.html HTTP/1.1
Host: www.cs.princeton.edu
```

Здесь `Host` является одним из возможных полей `MESSAGE_HEADER`, что дает клиенту возможность условно запросить веб-страницу — сервер возвращает страницу только в том случае, если она была изменена с указанного времени в этой строке заголовка.

Сообщения ответа

Как и сообщения запроса, сообщения ответа начинаются с одной `START_LINE`. В этом случае строка указывает используемую версию HTTP, трехзначный код, указывающий, был ли запрос успешным, и текстовую строку, дающую причину ответа. Например, `START_LINE`

```
HTTP/1.1 202 Accepted
```

указывает на то, что сервер смог удовлетворить запрос, в то время как

```
HTTP/1.1 404 Not Found
```

указывает, что он не смог удовлетворить запрос, так как страница не была найдена. Существует пять общих типов кодов ответа, причем первая цифра кода указывает его тип. Табл. 9.2 суммирует пять типов кодов.

Таблица 9.2.
Пять типов кодов результатов HTTP.

Код	Тип	Примеры причин использования
1xx	Информация	запрос получен, процесс продолжается
2xx	Успех	действие успешно получено, понято и принято
3xx	Перенаправление	необходимо предпринять дополнительные действия для завершения запроса
4xx	Ошибка клиента	запрос содержит неправильный синтаксис или не может быть выполнен
5xx	Ошибка сервера	серверу не удалось выполнить, казалось бы, правильный запрос

Как и в случае неожиданных последствий сообщения запроса `POST`, иногда удивительно, как различные сообщения ответа используются на практике. Например, перенаправление запроса (конкретно код `302`) оказывается мощным механизмом, который

играет большую роль в сетях доставки контента (CDN), перенаправляя запросы к ближайшему кешу.

Так же, как и сообщения запроса, сообщения ответа могут содержать одну или несколько строк MESSAGE_HEADER. Эти строки передают дополнительную информацию обратно клиенту. Например, строка заголовка Location указывает, что запрашиваемый URL доступен по другому адресу. Таким образом, если веб-страница кафедры информатики Принстонского университета переместилась по оригинальному адресу, сервер может ответить с

```
HTTP/1.1 301 Moved Permanently
Location: http://www.princeton.edu/cs/index.html
```

В обычном случае сообщение ответа также будет содержать запрашиваемую страницу. Эта страница представляет собой HTML-документ, но поскольку она может содержать нетекстовые данные (например, изображение GIF), она кодируется с использованием MIME (см. предыдущую главу). Некоторые строки MESSAGE_HEADER указывают атрибуты содержимого страницы, включая (количество байтов в содержимом), Expires (время, когда содержимое считается устаревшим), и (время, когда содержимое было в последний раз изменено на сервере).

Унифицированные идентификаторы ресурсов (Uniform Resource Identifiers, URI)

URL-адреса, которые HTTP использует в качестве адресов, являются одним из типов унифицированных *идентификаторов ресурсов* (URI). URI — это строка символов, которая идентифицирует ресурс, где ресурсом может быть что угодно, что имеет идентичность, например, документ, изображение или услуга.

Формат URI позволяет различным более специализированным типам идентификаторов ресурсов быть включенными в пространство URI. Первая часть URI — это схема, которая указывает определенный способ идентификации определенного типа ресурса, например, mailto для адресов электронной почты или file для имен файлов. Вторая часть URI, отделенная от первой части двоеточием, является *схемо-специфичной частью*. Это идентификатор ресурса, соответствующий схеме в первой части, как в URI mailto:santa@northpole.org и file:///C:/foo.html.

Ресурс не обязательно должен быть извлекаемым или доступным. Мы видели пример этого в предыдущем разделе — пространства имен расширяемого языка разметки (XML) идентифицируются с помощью URI, которые выглядят очень похоже на URL, но, строго говоря, они не являются *локаторами*, потому что они не указывают, как найти что-либо; они просто предоставляют глобально уникальный идентификатор для пространства имен. Нет требования, чтобы вы могли извлечь что-либо по URI, указанному как целевое пространство имен XML-документа. Мы увидим еще один пример URI, который не является URL, в следующей главе.

TCP-соединения

Оригинальная версия HTTP (1.0) устанавливала отдельное TCP-соединение для каждого элемента данных, извлекаемого с сервера. Несложно увидеть, насколько этот механизм был неэффективным: сообщения установки и завершения соединения должны были обмениваться между клиентом и сервером, даже если все, что клиент хотел сделать, это проверить, имеет ли он самую последнюю копию страницы. Таким образом, получение страницы, включающей какой-то текст и дюжину иконок или других мелких графических элементов, приводило к установлению и закрытию 13 отдельных TCP-соединений. Рисунок 9.4 показывает последовательность

событий для получения страницы, которая имеет только один встроенный объект. Светлые линии обозначают TCP-сообщения, в то время как черные линии обозначают HTTP-запросы и ответы. (Некоторые TCP-ACKs не показаны, чтобы избежать перегруженности рисунка.) Вы можете видеть, что два времени кругового обхода тратятся на установку TCP-соединений, в то время как еще два (как минимум) тратятся на получение страницы и изображения. Кроме латентности существует также процессорная нагрузка на сервер для обработки дополнительного установления и завершения TCP-соединения.

Чтобы исправить эту ситуацию, версия HTTP 1.1 ввела постоянные соединения — клиент и сервер могут обмениваться несколькими сообщениями запроса/ответа через одно и то же TCP-соединение. Постоянные соединения имеют много преимуществ. Во-первых, они очевидно устраняют накладные расходы на установку соединения, тем самым уменьшая нагрузку на сервер, нагрузку на сеть, вызванную дополнительными TCP-пакетами, и задержку, воспринимаемую пользователем. Во-вторых, поскольку клиент может отправлять несколько сообщений запроса через одно TCP-соединение, механизм окна перегрузки TCP может работать более эффективно. Это потому, что нет необходимости проходить фазу медленного старта для каждой страницы. Рис. 9.5 показывает транзакцию с рис. 9.4, используя постоянное соединение в случае, когда соединение уже открыто (предположительно из-за некоторого предыдущего доступа к тому же серверу).

Однако постоянные соединения не обходятся без цены. Проблема в том, что ни клиент, ни сервер не знают, как долго держать открытым конкретное TCP-соединение. Это особенно критично на сервере, который запрашивается на удержание соединений открытыми от имени тысяч клиентов. Решение состоит в том, что сервер должен отслеживать и закрывать соединение, если он не получил запросов на соединение в течение определенного времени. Также и клиент, и сервер должны следить, не решил ли другой конец закрыть соединение, и они должны использовать эту информацию как сигнал, что они также должны закрыть свою сторону соединения. (Напомним, что обе стороны должны закрыть TCP-соединение, прежде чем оно будет полностью завершено.) Озабоченность по поводу этой добавленной сложности может быть одной из причин, почему постоянные соединения не использовались с самого начала, но сегодня широко признано, что преимущества постоянных соединений более чем компенсируют недостатки.

Хотя версия 1.1 все еще широко поддерживается, новая версия (2.0) была формально утверждена IETF в 2015 году. Известная как HTTP/2, новая версия обратно совместима с 1.1 (то есть она использует тот же синтаксис для строк заголовка, кодов состояния и URI), но добавляет две новые функции.

Первое нововведение HTTP/2 заключается в том, чтобы упростить для веб-серверов *минимизацию* информации, которую они отправляют обратно веб-браузерам. Если вы внимательно посмотрите на структуру HTML на типичной веб-странице, вы обнаружите множество ссылок на различные мелкие элементы (например, изображения, скрипты, файлы стилей), которые браузеру нужны для отображения страницы. Вместо того чтобы заставлять клиента запрашивать эти мелкие элементы (технически известные как *ресурсы*) в последующих запросах, HTTP/2 предоставляет средства для сервера, чтобы объединить необходимые ресурсы и проактивно отправить их клиенту без задержки на круговой обход, связанной с запросом этих ресурсов. Эта функция сочетается с механизмом сжатия, уменьшающим количество байтов, которые нужно передать. Основная цель состоит в том, чтобы минимизировать задержку, которую испытывает конечный пользователь с момента клика по гиперссылке до полного отображения выбранной страницы.

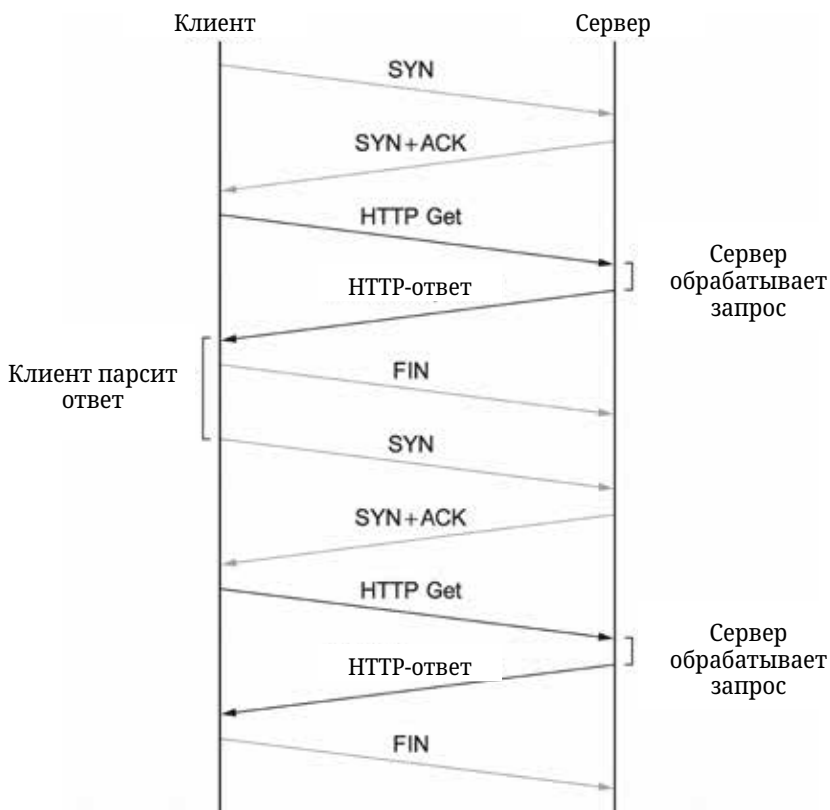


Рисунок 9.4. Поведение HTTP 1.0.

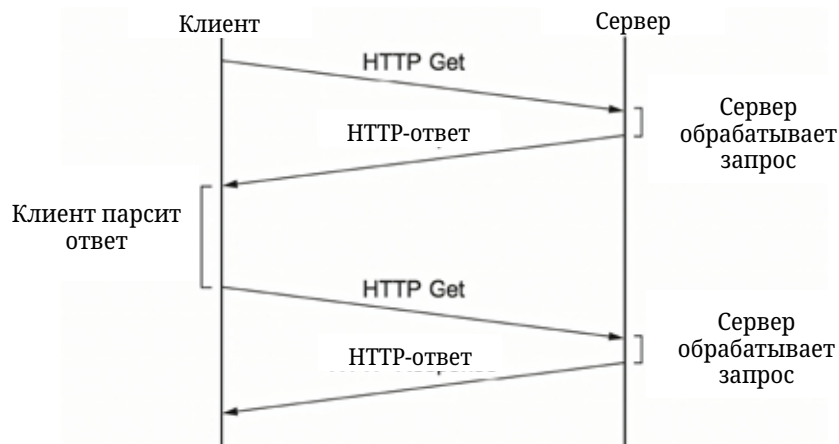


Рисунок 9.5. Поведение HTTP 1.1 с постоянными соединениями.

Второе большое нововведение HTTP/2 заключается в мультиплексировании нескольких запросов на одном TCP-соединении. Это выходит за рамки возможностей версии 1.1, позволяющей *последовательности* запросов повторно использовать TCP-соединение и разрешая этим запросам перекрываться друг с другом. Способ, которым HTTP/2 это делает, должен показаться знакомым: он определяет абстракцию канала

(технически каналы называются *потоками*), разрешает множеству одновременных потоков быть активными в любое время (каждый помечен уникальным *идентификатором потока*) и ограничивает каждый поток одной активной операцией «запрос/ответ» в одно время.

Кеширование

Важная стратегия реализации, делающая веб более удобным, заключается в кешировании веб-страниц. Кеширование имеет много преимуществ. С точки зрения клиента страница, которая может быть извлечена из ближайшего кеша, может быть отображена намного быстрее, чем если бы ее пришлось запрашивать из другого конца мира. С точки зрения сервера наличие кеша, который перехватывает и удовлетворяет запрос, уменьшает нагрузку на сервер.

Кеширование может быть реализовано в различных местах. Например, браузер пользователя может кешировать недавно посещенные страницы и просто отображать кешированную копию, если пользователь снова посещает ту же страницу. В другом примере сайт может поддерживать единый кеш на уровне всего сайта. Это позволяет пользователям воспользоваться страницами, ранее загруженными другими пользователями. Ближе к середине Интернета интернет-провайдеры (ISP)¹ могут кешировать страницы. В этом случае пользователи внутри сайта, скорее всего, знают, какая машина кеширует страницы от имени сайта, и настраивают свои браузеры для прямого подключения к этому кешированному хосту. Этот узел иногда называют прокси. В отличие от этого, сайты, подключенные к ISP, вероятно, не знают, что ISP кеширует страницы. Просто так получается, что HTTP-запросы, выходящие из различных сайтов, проходят через общий маршрутизатор ISP. Этот маршрутизатор может заглянуть внутрь сообщения запроса и посмотреть на URL запрашиваемой страницы. Если у него есть эта страница в кеше, он ее возвращает. Если нет, он перенаправляет запрос на сервер и следит за ответом, летящим в обратном направлении. Когда это происходит, маршрутизатор сохраняет копию в надежде, что он сможет использовать ее для удовлетворения будущего запроса.

Неважно, где кешируются страницы, возможность кеширования веб-страниц настолько важна, что HTTP был разработан, чтобы облегчить эту задачу. Секрет в том, что кеш должен убедиться, что он не отвечает устаревшей версией страницы. Например, сервер назначает дату истечения срока действия (поле заголовка *Expires*) каждой странице, которую он отправляет обратно клиенту (или кешу между сервером и клиентом). Кеш запоминает эту дату и знает, что не нужно повторно проверять страницу каждый раз, когда она запрашивается, до тех пор, пока не пройдет эта дата истечения срока. После этого времени (или если это поле заголовка не установлено) кеш может использовать операцию HEAD или условный GET (GET с заголовочной строкой), чтобы убедиться, что у него есть самая последняя копия страницы. Говоря обобщенно, существует набор *директив кеша*, которые должны соблюдаться всеми механизмами кеширования по цепочке «запрос/ответ». Эти директивы указывают, можно ли кешировать документ, как долго его можно кешировать, насколько свежим должен быть документ и так далее. Мы рассмотрим связанную проблему сетей доставки контента (CDN), которые фактически являются распределенными кешами, в следующей главе.

Глава 9.1.3. Веб-сервисы

До сих пор мы сосредотачивались на взаимодействиях между человеком и веб-сервером. Например, человек использует веб-браузер для взаимодействия с сервером, и взаимодействие происходит в ответ на действие пользователя (например, при клике

¹ Существует довольно много проблем с подобным кешированием, начиная от технических и заканчивая нормативными. Одним из примеров технической проблемы является эффект *асимметричных путей*, когда запрос к серверу и ответ клиенту не следуют в одной и той же последовательности маршрутных переходов.

по ссылкам). Однако существует возрастающий спрос на прямое взаимодействие между компьютерами. И, как и описанные выше приложения, такие приложения, которые взаимодействуют напрямую друг с другом, также нуждаются в протоколах. Мы завершаем эту главу, рассмотрев проблемы построения большого количества протоколов для взаимодействия приложений и некоторые предложенные решения.

Первоначальная причина для прямого взаимодействия приложений часто исходит из бизнес-сферы. Исторически взаимодействие между предприятиями — бизнесами или другими организациями — включало некоторые ручные шаги, такие как заполнение заказа или звонок, чтобы узнать, есть ли продукт в наличии. Даже внутри одного предприятия часто существуют ручные шаги между программными системами, которые не могут взаимодействовать напрямую, потому что они были разработаны независимо. Все чаще такие ручные взаимодействия заменяются на прямое взаимодействие приложений. Приложение для заказов на предприятии А отправляет сообщение приложению для выполнения заказов на предприятии Б, которое немедленно отвечает, указывая, можно ли выполнить заказ. Возможно, если заказ не может быть выполнен в Б, приложение в А немедленно закажет у другого поставщика или запросит предложения от группы поставщиков.

Вот простой пример того, о чем идет речь. Предположим, вы покупаете книгу в интернет-магазине, таком как Amazon. После того как ваша книга будет отправлена, Amazon может отправить вам номер для отслеживания в электронном письме, и затем вы можете зайти на веб-сайт транспортной компании — например, <http://www.fedex.com> — и отследить посылку. Однако вы также можете отследить вашу посылку напрямую с веб-сайта Amazon.com. Чтобы это произошло, Amazon должен иметь возможность отправить запрос FedEx в формате, который FedEx понимает, интерпретировать результат и отобразить его на веб-странице, которая, возможно, содержит другую информацию о вашем заказе. Основой пользовательского опыта получения всей информации о заказе сразу на веб-странице Amazon.com является тот факт, что Amazon и FedEx должны иметь протокол для обмена информацией, необходимой для отслеживания посылок — назовем его протоколом отслеживания посылок. Очевидно, что существует так много потенциальных протоколов подобного типа, что нам понадобятся инструменты для упрощения задачи их спецификации и построения.

Сетевые приложения, даже те, которые пересекают границы организаций, не новы — электронная почта и веб-серфинг пересекают такие границы. Что нового в этой задаче, так это масштаб. Не масштаб сети, а масштаб в количестве различных типов сетевых приложений. Как правило, спецификации протоколов и реализации этих протоколов для традиционных приложений, таких как электронная почта и передача файлов, разрабатывались небольшой группой экспертов по сетям. Чтобы обеспечить быстрое развитие огромного количества потенциальных сетевых приложений, необходимо было придумать технологии, которые упрощают и автоматизируют задачу проектирования и реализации протоколов приложений.

Были предложены две архитектуры для решения этой проблемы. Обе архитектуры называются *веб-сервисами*, заимствуя свое название от термина для отдельных приложений, которые предлагают удаленно доступный сервис клиентским приложениям, чтобы формировать сетевые приложения. Термины, используемые как неформальные сокращения для различения двух архитектур веб-сервисов, — *SOAP* и *REST*. Мы обсудим технические значения этих терминов чуть позже.

Подход архитектуры SOAP к решению проблемы заключается в том, чтобы сделать возможным, по крайней мере теоретически, создание протоколов, настроенных для каждого сетевого приложения. Ключевые элементы подхода включают в себя каркас для спецификации протоколов, программные инструменты для автоматического создания реализаций протоколов из спецификаций и модульные частичные спецификации, которые могут быть повторно использованы в разных протоколах.

Подход архитектуры REST к решению проблемы заключается в том, чтобы рассматривать отдельные веб-сервисы как ресурсы Всемирной паутины — идентифицированные с помощью URI и доступные через HTTP. По сути, архитектура REST — это просто архитектура Веба. Преимущества архитектуры Веба включают стабильность и доказанную масштабируемость (в смысле размера сети). Можно считать недостатком то, что HTTP не очень подходит для обычного процедурного или операционно-ориентированного стиля вызова удаленного сервиса. Однако сторонники REST утверждают, что богатые сервисы все же могут быть предоставлены, используя более ориентированный на данные или документальный стиль, для которого HTTP подходит лучше.

Пользовательские протоколы приложений (WSDL, SOAP)

Архитектура, неформально называемая SOAP, основана на языке *описания веб-сервисов* (WSDL) и протоколе SOAP.¹ Оба этих стандарта выпускаются Консорциумом Всемирной паутины (W3C). Это та архитектура, которую обычно имеют в виду, когда используется термин «веб-сервисы» без каких-либо дополнительных уточнений. Поскольку эти стандарты продолжают развиваться, наше обсуждение является своего рода мгновенным снимком.

WSDL и SOAP представляют собой структуры для спецификации и реализации протоколов приложений и транспортных протоколов соответственно. Они обычно используются вместе, хотя это и не является строгим требованием. WSDL используется для определения специфичных для приложения деталей, таких как поддерживаемые операции, форматы данных приложения для вызова или ответа на эти операции, и то, предполагает ли операция ответ. Роль SOAP заключается в том, чтобы облегчить определение транспортного протокола с необходимыми семантиками относительно таких характеристик протокола, как надежность и безопасность.

Оба стандарта, WSDL и SOAP, состоят в основном из языка спецификации протокола. Оба языка основаны на XML, с акцентом на доступность спецификаций для программных инструментов, таких как компиляторы заглушек и службы каталогов. В мире множества пользовательских протоколов поддержка автоматизации генерации реализаций имеет решающее значение, чтобы избежать трудозатрат на ручную реализацию каждого протокола. Программная поддержка обычно принимает форму наборов инструментов и серверов приложений, разработанных сторонними поставщиками, что позволяет разработчикам отдельных веб-сервисов сосредоточиться на бизнес-задаче, которую они должны решить (например, отслеживание посылки, купленной клиентом).

Определение протоколов приложений

WSDL выбрал модель *процедурных операций* для протоколов приложений. Абстрактный интерфейс веб-сервиса состоит из набора именованных операций, каждая из которых представляет собой простое взаимодействие между клиентом и веб-сервисом. Операция аналогична процедуре удаленного вызова в системе RPC. Пример из руководства W3C по WSDL — это веб-сервис бронирования отеля с двумя операциями: CheckAvailability (проверить доступность) и MakeReservation (осуществить бронирование).

Каждая операция определяет *шаблон обмена сообщениями* (MEP), задающий последовательность, в которой сообщения должны передаваться, включая сообщения об ошибках, которые отправляются при нарушении потока сообщений. Несколько MEP предопределены, и могут быть определены новые пользовательские MEP, но на практике используются только два MEP: In-Only (одно сообщение от клиента к сервису) и In-Out (запрос от клиента и соответствующий ответ от сервиса). Эти шаблоны должны быть хорошо знакомы, и предполагается, что затраты на поддержку гибкости MEP могут перевешивать преимущества.

¹ Хотя название SOAP возникло как аббревиатура, официально она никак не расшифровывается.

MEP являются шаблонами, которые имеют заполнитель вместо конкретных типов или форматов сообщений, поэтому часть определения операции включает указание, какие форматы сообщений будут сопоставлены с заполнителями в шаблоне. Форматы сообщений не определяются на уровне битов, что типично для протоколов, которые мы обсуждали. Вместо этого они определяются как абстрактная модель данных, используя XML. XML Schema предоставляет набор примитивных типов данных и способы определения составных типов данных. Данные, соответствующие формату, определенному XML Schema — его абстрактной модели данных, — могут быть конкретно представлены с использованием XML или могут использовать другое представление, такое как «бинарное» представление Fast Infoset.

WSDL отделяет части протокола, которые могут быть абстрактно определены — операции, MEP, абстрактные форматы сообщений — от тех частей, которые должны быть конкретными. Конкретная часть WSDL определяет подлежащий протокол, а также то, как MEP сопоставляются с ним и какое битовое представление используется для сообщений в сети. Эта часть спецификации известна как привязка, хотя ее лучше описывать как реализацию или сопоставление с реализацией. WSDL имеет предопределенные привязки для протоколов на основе HTTP и SOAP, с параметрами, которые позволяют разработчику протокола точно настроить сопоставление с этими протоколами. Существует структура для определения новых привязок, но протоколы на основе SOAP доминируют.

Ключевой аспект того, как WSDL смягчает проблему определения большого количества протоколов, заключается в повторном использовании модулей спецификации. Спецификация WSDL веб-сервиса может быть составлена из нескольких документов WSDL, и отдельные документы WSDL также могут использоваться в других спецификациях веб-сервисов. Эта модульность упрощает разработку спецификации и помогает убедиться, что если две спецификации должны иметь некоторые одинаковые элементы (например, чтобы их можно было поддерживать одним и тем же инструментом), то эти элементы действительно идентичны. Эта модульность, вместе с правилами по умолчанию WSDL, также помогает избежать чрезмерной многословности спецификаций для разработчиков протоколов.

Модульность WSDL должна быть знакома каждому, кто разрабатывал достаточно большие программные системы. Документ WSDL не обязательно должен быть полной спецификацией; он может, например, определять только один формат сообщения. Частичные спецификации уникально идентифицируются с помощью XML-пространств имен; каждый документ WSDL указывает URI *целевого пространства имен*, и любые новые определения в документе именуются в контексте этого пространства имен. Один документ WSDL может *включать* компоненты другого, если оба имеют одинаковое целевое пространство имен, или импортировать его, если целевые пространства имен различаются.

Определение транспортных протоколов

Хотя SOAP иногда называют протоколом, его лучше рассматривать как структуру для определения протоколов. Как объясняется в спецификации SOAP 1.2, «SOAP предоставляет простую структуру обмена сообщениями, основная функция которой заключается в обеспечении расширяемости». SOAP использует многие из тех же стратегий, что и WSDL, включая форматы сообщений, определенные с помощью XML Schema, привязки к основным протоколам, шаблоны обмена сообщениями (MEP) и повторно используемые элементы спецификации, идентифицированные с помощью XML-пространств имен.

SOAP используется для определения транспортных протоколов с точно нужными функциями для поддержки определенного прикладного протокола. Цель SOAP — сделать возможным определение многих таких протоколов с применением повторно используемых компонентов. Каждый компонент включает информацию заголовка и логику, необходимые для реализации определенной функции. Чтобы определить протокол с определенным набором функций, достаточно составить соответствующие компоненты. Давайте рассмотрим этот аспект SOAP более подробно.

SOAP 1.2 ввел абстракцию функции, которую спецификация описывает следующим образом:

«Функция SOAP — это расширение структуры обмена сообщениями SOAP. Хотя SOAP не накладывает ограничений на потенциальный охват таких функций, примеры функций могут включать “надежность”, “безопасность”, “корреляцию”, “маршрутизацию” и шаблоны обмена сообщениями (MEP), такие как запрос/ответ, однонаправленные и равноправные беседы.»

Спецификация функции SOAP должна включать:

- URI, который идентифицирует функцию
- Информацию о состоянии и обработке, абстрактно описанные, которые требуются на каждом узле SOAP для реализации функции
- Информацию, которую нужно передать следующему узлу
- (Если функция является MEP) Жизненный цикл и временные/причинно-следственные отношения передаваемых сообщений, например, ответы следуют за запросами и отправляются инициатору запроса

Обратите внимание, что эта формализация концепции функции протокола находится на довольно низком уровне; это почти дизайн.

Имея набор функций, можно выделить две стратегии для определения протокола SOAP, который их реализует. Первая — это создание уровня: связывание SOAP с основным протоколом таким образом, чтобы извлечь функции. Например, мы можем получить протокол «запрос/ответ», связывая SOAP с HTTP, с запросом SOAP в HTTP-запросе и ответом SOAP в HTTP-ответе. Поскольку это широко распространенный пример, у SOAP есть предопределенная привязка к HTTP; новые привязки можно определить с помощью структуры привязки протокола SOAP.

Второй и более гибкий способ реализации функций включает *блоки заголовков*. Сообщение SOAP состоит из оболочки (Envelope), которая содержит заголовок (Header) с блоками заголовков, и тела (Body), которое содержит полезную нагрузку, предназначенную для конечного получателя. Эта структура сообщения иллюстрируется на рис. 9.6.

Эта идея вам должна быть уже знакома: определенная информация в заголовке соответствует конкретным функциям. Цифровая подпись используется для реализации аутентификации, порядковый номер — для надежности, а контрольная сумма — для обнаружения повреждения сообщений. Блок заголовка SOAP предназначен для инкапсуляции информации заголовка, которая соответствует конкретной функции. Соответствие не всегда однозначно, так как несколько блоков заголовков могут быть задействованы в одной функции, или один блок заголовка может использоваться в нескольких функциях. *Модуль SOAP* — это спецификация синтаксиса и семантики одного или нескольких блоков заголовков. Каждый модуль предназначен для предоставления одной или нескольких функций и должен объявлять функции, которые он реализует.



Рисунок 9.6. Структура сообщения SOAP.

Цель модулей SOAP — возможность составить протокол с набором функций, просто включив соответствующие спецификации модулей. Если ваш протокол требует семантики «не более одного раза» и аутентификации, включите соответствующие модули в вашу спецификацию. Это представляет собой новый подход к модульному проектированию услуг протокола, альтернативу слоистой архитектуре протоколов, рассмотренной в этой книге. Это похоже на уплощение серии слоев протоколов в один протокол, но структурированным способом. Остается выяснить, насколько хорошо функции и модули SOAP, введенные в версии 1.2, будут работать на практике. Основной слабостью этой схемы является то, что модули могут мешать друг другу. Спецификация модуля должна указывать любые известные взаимодействия с другими модулями SOAP, но очевидно, что это мало что делает для решения проблемы. С другой стороны, основной набор функций и модулей, обеспечивающий наиболее важные свойства, может быть достаточно мал, чтобы быть хорошо известным и понятным.

Стандартизация протоколов веб-служб

Как мы уже говорили, WSDL и SOAP не являются протоколами; это стандарты для спецификации протоколов. Для того чтобы разные предприятия могли реализовать веб-службы, которые взаимодействуют друг с другом, недостаточно согласовать использование WSDL и SOAP для определения их протоколов; они должны договориться о стандартизации конкретных протоколов. Например, можно представить, что онлайн-ритейлеры и транспортные компании могут захотеть стандартизировать протокол для обмена информацией, подобно простому примеру отслеживания посылки в начале этой главы. Эта стандартизация важна как для поддержки инструментов, так и для взаимодействия. И все же различные сетевые приложения в этой архитектуре должны обязательно различаться хотя бы форматами сообщений и операциями, которые они используют.

Это напряжение между стандартизацией и кастомизацией решается путем создания частичных стандартов, называемых *профилями*. Профиль — это набор руководящих принципов, которые сужают или ограничивают выбор, доступный в WSDL, SOAP и других стандартах, которые могут быть упомянуты при определении протокола. Они также могут решать неоднозначности или пробелы в этих стандартах. На практике профиль часто формализует возникающий *де-факто* стандарт.

Самый широкий и наиболее принятый профиль известен как *базовый профиль WS-I*. Он был предложен Организацией Интероперабельности Веб-Сервисов (WS-I), промышленным консорциумом, тогда как WSDL и SOAP были специфицированы Консорциумом Всемирной Паутины (W3C). Базовый профиль разрешает некоторые из самых основных выборов, с которыми сталкиваются при определении веб-службы. В частности, он требует, чтобы WSDL был связан исключительно с SOAP, а SOAP был связан исключительно с HTTP и использовал метод HTTP POST. Он также указывает, какие версии WSDL и SOAP должны использоваться.

Базовый профиль безопасности WS-I добавляет ограничения безопасности к базовому профилю, указывая, как должен использоваться уровень SSL/TLS и требуя соответствия *WS-Security* (безопасность веб-сервисов). *WS-Security* определяет, как использовать различные существующие техники, такие как сертификаты открытого ключа X.509 и Kerberos, для предоставления функций безопасности в протоколах SOAP.

WS-Security — это только первый из растущего набора стандартов уровня SOAP, установленных промышленным консорциумом OASIS (Организация по продвижению стандартов структурированной информации). Стандарты, известные как *WS**, включают *WS-Reliability*, *WS-ReliableMessaging*, *WS-Coordination* и *WS-AtomicTransaction*.

Общий протокол приложений (REST)

Архитектура веб-сервисов на основе WSDL/SOAP исходит из предположения, что лучший способ интеграции приложений через сети — это использование протоколов, кастомизированных для каждого приложения. Эта архитектура разработана так, чтобы сделать

практичными спецификацию и реализацию всех этих протоколов. В отличие от этого, архитектура веб-сервисов REST основана на предположении, что лучший способ интеграции приложений через сети — это повторное применение модели, лежащей в основе архитектуры Всемирной паутины. Эта модель, сформулированная архитектором веба Роем Филдингом, известна как *Representational State Transfer* (REST). Нет необходимости в новой REST-архитектуре для веб-сервисов — существующая веб-архитектура достаточно хороша, хотя, возможно, потребуются некоторые расширения. В веб-архитектуре отдельные веб-сервисы рассматриваются как ресурсы, идентифицированные с помощью URI и доступные через HTTP — единственный универсальный протокол приложений с единой системой адресации.

В то время как в WSDL есть пользовательские операции, REST использует небольшой набор доступных методов HTTP, таких как GET и POST (см. табл. 9.1). Так как же эти простые методы могут обеспечить интерфейс для богатого веб-сервиса? Путем использования модели REST, в которой сложность смещена с протокола на полезную нагрузку. Полезная нагрузка — это представление абстрактного состояния ресурса. Например, GET может вернуть представление текущего состояния ресурса, а POST может отправить представление желаемого состояния ресурса.

Представление состояния ресурса абстрактно; оно не обязательно должно походить на то, как ресурс действительно реализован конкретным экземпляром веб-сервиса. Нет необходимости передавать полное состояние ресурса в каждом сообщении. Размер сообщений можно уменьшить, передавая только те части состояния, которые представляют интерес (например, только те части, которые изменяются). И, поскольку веб-сервисы используют единый протокол и пространство адресов с другими веб-ресурсами, части состояний могут передаваться по ссылке — через URI, даже если это другие веб-сервисы.

Этот подход лучше всего описывается как стиль, ориентированный на данные или передачу документов, в отличие от процедурного стиля. Определение протокола приложения в этой архитектуре состоит в определении структуры документа (т.е. представления состояния). XML и более легковесный JavaScript Object Notation (JSON) являются наиболее часто используемыми языками представления для этого состояния. Совместимость зависит от соглашения между веб-сервисом и его клиентами о представлении состояния. Конечно, то же самое верно и в архитектуре SOAP; веб-сервис и его клиент должны согласовать формат полезной нагрузки. Разница в том, что в архитектуре SOAP совместимость также зависит от соглашения о протоколе; в архитектуре REST протокол всегда HTTP, поэтому данный источник проблем совместимости устраняется.

Одним из преимуществ REST является то, что он использует инфраструктуру, развернутую для поддержки веба. Например, веб-прокси могут обеспечивать безопасность или кешировать информацию. Существующие сети доставки контента (CDN) могут быть использованы для поддержки RESTful приложений.

В отличие от WSDL/SOAP, веб-архитектура имела время для стабилизации стандартов и демонстрации отличной масштабируемости. Она также включает в себя некоторые средства безопасности в виде Secure Socket Layer (SSL)/Transport Layer Security (TLS). Веб и REST также могут иметь преимущество в плане эволюционируемости. Хотя WSDL и SOAP обладают высокой гибкостью в отношении того, какие новые функции и привязки могут быть включены в определение протокола, эта гибкость становится неактуальной после определения протокола. Стандартизированные протоколы, такие как HTTP, разработаны с возможностью расширения в обратной совместимости. Расширяемость HTTP принимает форму заголовков, новых методов и новых типов контента. Дизайнерам протоколов, использующим WSDL/SOAP, необходимо закладывать такую расширяемость в каждый из своих пользовательских протоколов. Конечно, дизайнеры представлений состояния в REST-архитектуре также должны проектировать с учетом возможности эволюционирования.

Область, в которой WSDL/SOAP может иметь преимущество, — это адаптация или обертка ранее написанных, «наследуемых» приложений для соответствия веб-сервисам. Это важный момент, так как большинство веб-сервисов в ближайшем будущем будет основываться на наследуемых приложениях. Эти приложения обычно имеют процедурный интерфейс, который легче сопоставить с операциями WSDL, чем с состояниями REST. Конкуренция между REST и WSDL/SOAP может зависеть от того, насколько легко или сложно окажется разрабатывать интерфейсы в стиле REST для отдельных веб-сервисов. Мы можем обнаружить, что некоторые веб-сервисы лучше обслуживаются с помощью WSDL/SOAP, а другие — с помощью REST.

Интернет-ритейлер Amazon, как оказалось, был одним из первых (2002) пользователей веб-сервисов. Интересно, что Amazon сделал свои системы общедоступными через обе архитектуры веб-сервисов, и, по некоторым сообщениям, большинство разработчиков использует интерфейс REST. Конечно, это лишь одна точка данных и может отражать факторы, специфичные для Amazon.

От веб-сервисов к облачным сервисам

Если веб-сервисы — это то, как мы называем ситуацию, когда веб-сервер, реализующий мое приложение, отправляет запрос веб-серверу, реализующему ваше приложение, то как мы будем называть это, когда оба наших приложения находятся в облаке, чтобы поддерживать масштабируемые рабочие нагрузки? Мы можем называть их *облачными сервисами*, если хотим, но есть ли в этом разница? Это зависит от обстоятельств.

Перенос серверного процесса с физической машины в моей серверной в виртуальную машину в дата-центре облачного провайдера перекладывает ответственность за поддержание работы машины с моего системного администратора на операционную команду облачного провайдера, но приложение все еще спроектировано в соответствии с архитектурой веб-сервисов. С другой стороны, если приложение спроектировано с нуля для работы на масштабируемой облачной платформе, например, следуя *архитектуре микросервисов*, тогда мы называем это приложение *облачно-нативным*. Таким образом, важное различие заключается в том, облачно-нативное приложение это или старое веб-приложение, развернутое в облаке.

Мы уже кратко рассматривали архитектуру микросервисов в разделе 5 при описании gRPC, и хотя трудно однозначно заявить, что микросервисы превосходят веб-сервисы, текущий тренд в индустрии почти наверняка отдает предпочтение первым. Возможно, более интересным является продолжающаяся дискуссия о том, что предпочтительнее в качестве механизма RPC для реализации микросервисов: REST+Json или gRPC+Protobufs. Учитывая, что оба работают поверх HTTP, оставим это как упражнение для читателя — выбрать сторону и защитить ее.

Глава 9.2. Мультимедийные приложения

Как и традиционные приложения, описанные в предыдущей главе, мультимедийные приложения, такие как телефония и видеоконференции, нуждаются в своих собственных протоколах. Большая часть начального опыта в разработке протоколов для мультимедийных приложений была получена благодаря инструментам MBone — приложениям, таким как *vat* и *vic*, которые были разработаны для использования на MBone, накладной сети, поддерживающей IP-мультитрансляцию для многопользовательских конференций. (Подробнее о накладных сетях, включая MBone, в следующей главе.) Изначально каждое приложение реализовывало свой собственный протокол (или протоколы), но стало очевидно, что многие мультимедийные приложения имеют общие требования. Это в конечном итоге привело к разработке ряда универсальных протоколов для использования мультимедийными приложениями.

Мы уже видели ряд протоколов, которые используют мультимедийные приложения. Протокол транспортировки в реальном времени (RTP) обеспечивает многие функции, общие для мультимедийных приложений, такие как передача информации о времени и идентификация схем кодирования и типов медиа.

Протокол резервирования ресурсов (RSVP) может быть использован для запроса выделения ресурсов в сети, чтобы обеспечить необходимое качество обслуживания (QoS) для приложения. Мы увидим, как распределение ресурсов взаимодействует с другими аспектами мультимедийных приложений позже в этой главе.

Помимо этих протоколов для мультимедийной транспортировки и распределения ресурсов многим мультимедийным приложениям также нужен протокол сигнализации или *управления сессией*. Например, предположим, что мы хотим иметь возможность совершать телефонные звонки через Интернет (Voice over IP, или VoIP). Нам нужен будет какой-то механизм для уведомления предполагаемого получателя такого звонка о том, что мы хотим с ним поговорить, например, отправив сообщение на какое-либо мультимедийное устройство, которое издаст звонок. Мы также хотели бы поддерживать такие функции, как переадресация вызовов, трехсторонние звонки и т. д. Протокол инициации сеанса (SIP) и H.323 — примеры протоколов, решающих проблемы управления сессиями; начнем обсуждение мультимедийных приложений с рассмотрения этих протоколов.

Глава 9.2.1. Управление сеансами и контроль вызовов (SDP, SIP, H.323)

Чтобы понять некоторые вопросы управления сессиями, рассмотрим следующую проблему. Допустим, вы хотите провести видеоконференцию в определенное время и сделать ее доступной для большого числа участников. Возможно, вы решили закодировать видеопоток с использованием стандарта MPEG-2, использовать мультикаст-адрес IP 224.1.1.1 для передачи данных и отправлять его с использованием RTP по UDP-порту номер 4000. Как вы предоставите всю эту информацию предполагаемым участникам? Один из способов — поместить всю эту информацию в электронное письмо и разослать его, но в идеале должен быть стандартный формат и протокол для распространения такого рода информации. IETF определила протоколы для этой цели. Протоколы, которые были определены, включают:

- Протокол описания сессии (SDP)
- Протокол объявления сессии (SAP)
- Протокол инициации сессии (SIP)
- Простой протокол управления конференцией (SCCP)

Вы можете подумать, что это много протоколов для, казалось бы, простой задачи, но есть множество аспектов проблемы и несколько различных ситуаций, в которых она должна быть решена. Например, есть разница между объявлением о том, что определенная сессия конференции будет доступна на Mbone (что будет сделано с использованием SDP и SAP), и попыткой совершить интернет-звонок определенному пользователю в определенное время (что может быть сделано с использованием SDP и SIP).

В первом случае можно считать задачу выполненной, как только вы отправили всю информацию о сессии в стандартном формате на известный мультикаст-адрес.

Во втором случае вам нужно будет найти одного или нескольких пользователей, отправить им сообщение с уведомлением о вашем желании поговорить (аналогично звонку на их телефон) и, возможно, договориться о подходящем аудиокодировании между всеми сторонами. Сначала рассмотрим SDP, который является общим для многих приложений, затем SIP, который широко используется для ряда интерактивных приложений, таких как интернет-телефония.

Протокол описания сессии (SDP)

Протокол описания сессии (или сеанса) (SDP) — это довольно общий протокол, который может использоваться в различных ситуациях и обычно применяется вместе с одним или несколькими другими протоколами (например, SIP). Он передает следующую информацию:

- Имя и цель сессии
- Время начала и окончания сессии
- Типы медиа (например, аудио, видео), которые составляют сессию
- Подробная информация, необходимая для получения сессии (например, мультикаст-адрес, на который будут отправляться данные, транспортный протокол, который будет использоваться, номера портов, схема кодирования)

SDP предоставляет эту информацию, отформатированную в ASCII с использованием последовательности строк текста, каждая из которых имеет форму «тип=значение». Пример сообщения SDP иллюстрирует основные моменты.

```
v=0
o=larry 2890844526 2890842807 IN IP4 128.112.136.10
s=Networking 101
i=A class on computer networking
u=http://www.cs.princeton.edu/
e=larry@cs.princeton.edu
c=IN IP4 224.2.17.12/127
t=2873397496 2873404696
m=audio 49170 RTP/AVP 0
m=video 51372 RTP/AVP 31
m=application 32416 udp wb
```

Следует отметить, что SDP, как и HTML, довольно легко читается человеком, но имеет строгие правила форматирования, которые делают возможным однозначное интерпретирование данных машинами. Например, спецификация SDP определяет все возможные типы информации, которые могут появляться, порядок, в котором они должны появляться, и формат и зарезервированные слова для каждого типа, который определен.

Первое, что следует заметить, это то, что каждый тип информации идентифицируется одной буквой. Например, строка «v=0» говорит нам, что «версия» имеет значение ноль; то есть это сообщение отформатировано в соответствии с нулевой версией SDP. Следующая строка предоставляет «источник» сессии, который содержит достаточно информации, чтобы уникально идентифицировать сессию. «larry» — это имя пользователя создателя сессии, а «198.51.100.1» — это IP-адрес его компьютера. Число, следующее за «larry», — это идентификатор сессии, который выбран уникальным для этой машины. Это сопровождается номером «версии» для объявления SDP; если информация о сессии была обновлена более поздним сообщением, номер версии будет увеличен.

Следующие три строки (i, s и u) предоставляют имя сессии, описание сессии и унифицированный идентификатор ресурса сессии (URI, как описано ранее в этой главе) — информацию, которая может быть полезна пользователю при решении, участвовать ли в этой сессии (или сеансе). Такая информация может быть отображена в пользовательском интерфейсе инструмента каталога сессий, который показывает текущие и предстоящие события, объявленные с использованием SDP. Следующая строка (e=...) содержит адрес электронной почты лица, с которым можно связаться по поводу сессии. На рис. 9.7 показан скриншот (сейчас уже устаревшего) инструмента каталога сессий, называемого sdr, вместе с описаниями нескольких сессий, которые были объявлены на момент съемки.



Рисунок 9.7. Инструмент каталога сессий отображает информацию, извлеченную из сообщений SDP.

Далее мы переходим к техническим деталям, которые позволят программе приложения участвовать в сессии. Строка, начинающаяся с `c = . . .`, предоставляет IP многоадресную (multicast) рассылку, на которую будут отправляться данные для этой сессии; пользователь должен будет присоединиться к этой многоадресной группе, чтобы получить доступ к сессии. Далее мы видим время начала и окончания сессии (кодированное в виде целых чисел в соответствии с Протоколом сетевого времени). Наконец, мы переходим к информации о медиа для этой сессии. В этой сессии доступны три типа медиа — аудио, видео и приложение для совместной работы с электронной доской, известное как «wb». Для каждого типа медиа есть одна строка информации, отформатированная следующим образом:

```
m=<media> <port> <transport> <format>
```

Типы медиа объясняются сами по себе, а номера портов в каждом случае — это порты UDP. Когда мы смотрим на поле «transport», мы видим, что приложение wb работает непосредственно через UDP, тогда как аудио и видео передаются с использованием «RTP/AVP». Это означает, что они работают через RTP и используют *профиль приложения*, известный как AVP. Этот профиль приложения определяет несколько различных схем кодирования для аудио и видео; мы видим, что в этом случае аудио использует кодировку 0 (которая представляет собой кодирование с частотой дискретизации 8 кГц и 8 бит на выборку), а видео использует кодировку 31, что представляет собой схему кодирования H.261. Эти «магические числа» для схем кодирования определены в RFC, который определяет профиль AVP; также возможно описать нестандартные схемы кодирования в SDP.

Наконец, мы видим описание медиа типа «wb». Вся информация о кодировании для этих данных специфична для приложения wb, поэтому достаточно просто указать имя приложения в поле «format». Это аналогично указанию `application/wb` в MIME сообщении.

Теперь, когда мы знаем, как описывать сеансы, мы можем рассмотреть, как они могут быть инициированы. Один из способов использования SDP — это объявление мультимедийных конференций путем отправки сообщений SDP на хорошо известный многоадресный адрес. Инструмент каталога сеансов, показанный на рис. 9.7, будет функционировать, присоединяясь к этой многоадресной группе и отображая информацию, которую он извлекает из полученных сообщений SDP. SDP также используется при доставке развлекательного видео через IP (часто называемого IPTV), чтобы предоставить информацию о видеоконтенте на каждом телевизионном канале.

SDP также играет важную роль в сочетании с *протоколом инициации сеансов* (SIP). С широким распространением *голоса через IP* (т.е. поддержкой приложений, подобных телефонным, через IP сети) и IP-видеоконференций, SIP теперь является одним из более важных элементов набора Интернет-протоколов.

SIP

SIP — это протокол уровня приложений, который в некоторой степени напоминает HTTP, так как основан на аналогичной модели «запрос/ответ». Однако он предназначен для совсем других типов приложений и поэтому предоставляет совершенно иные возможности, чем HTTP. Возможности, предоставляемые SIP, можно сгруппировать в пять категорий:

- Определение местоположения пользователя — определение правильного устройства для связи с конкретным пользователем
- Доступность пользователя — определение, готов ли пользователь или способен ли он принять участие в конкретной коммуникационной сессии (сеансе)
- Возможности пользователя — определение таких параметров, как выбор медиа и схемы кодирования
- Настройка сеанса (сессии) — установление параметров сеанса, таких как номера портов, которые будут использоваться сторонами, участвующими в коммуникации
- Управление сеансом — ряд функций, включая передачу сеансов (например, для реализации «переадресации вызовов»), и изменение параметров сеанса

Большинство этих функций достаточно легко понять, но вопрос определения местоположения требует дополнительного обсуждения. Одно важное отличие между SIP и, скажем, HTTP заключается в том, что SIP в первую очередь используется для общения «человек-человек». Таким образом, важно иметь возможность определить местоположение конкретных *пользователей*, а не только машин. И, в отличие от электронной почты, недостаточно просто найти сервер, который пользователь проверит позже, и оставить там сообщение — нам нужно знать, где пользователь находится прямо сейчас, если мы хотим связаться с ним в реальном времени. Это усложняется тем, что пользователь может выбирать для общения различные устройства, такие как настольный ПК в офисе и портативное устройство в поездке. Несколько устройств могут быть активны одновременно и иметь сильно различающиеся возможности (например, буквенно-цифровой пейджер и видеотелефон на базе ПК). В идеале другие пользователи должны иметь возможность находить и общаться с подходящим устройством в любое время. Кроме того, пользователь должен иметь возможность контролировать, когда, где и от кого он получает вызовы.

Чтобы дать пользователю возможность осуществлять необходимый уровень контроля над своими вызовами, SIP вводит понятие прокси. SIP прокси можно рассматривать как точку контакта для пользователя, к которой направляются начальные запросы на связь с ним. Прокси также выполняют функции от имени вызывающих абонентов.

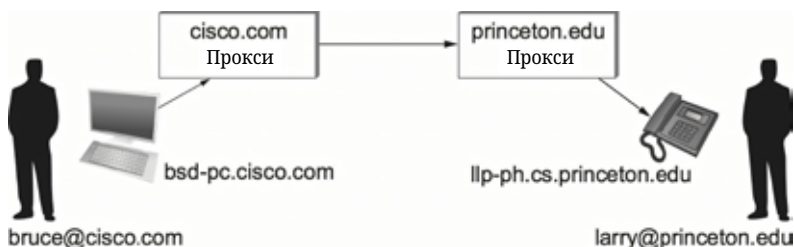


Рисунок 9.8. Установление связи через SIP-прокси.

Давайте посмотрим, как работают прокси. Рассмотрим двух пользователей на рис. 9.8. Первое, что стоит заметить, это то, что каждый пользователь имеет имя в формате `user@domain`, очень похожее на адрес электронной почты. Когда пользователь Брюс хочет инициировать сеанс с Ларри, он отправляет свое начальное SIP-сообщение на локальный прокси для своего домена, `cisco.com`. Среди прочего, это начальное сообщение содержит *SIP URI* — это форма унифицированного идентификатора ресурса, которая выглядит так:

```
SIP:larry@princeton.edu.
```

SIP URI предоставляет полную идентификацию пользователя, но (в отличие от URL) не предоставляет его местоположение, так как оно может изменяться со временем. Мы скоро увидим, как можно определить местоположение пользователя.

Получив начальное сообщение от Брюса, прокси смотрит на SIP URI и делает вывод, что это сообщение должно быть отправлено на прокси. Сейчас предположим, что прокси имеет доступ к некоторой базе данных, которая позволяет ему получить сопоставление имени с IP-адресом одного или нескольких устройств, на которые Ларри в данный момент хочет получать сообщения. Таким образом, прокси может переслать сообщение на выбранное(ые) Ларри устройство(а). Отправка сообщения на несколько устройств называется *форкингом* (forking) и может выполняться параллельно или последовательно (например, отправка на мобильный телефон, если он не ответит на телефон на рабочем столе).

Начальное сообщение от Брюса к Ларри, вероятно, будет SIP-приглашением (*invite*), которое выглядит примерно так:

```
INVITE sip:larry@princeton.edu SIP/2.0
Via: SIP/2.0/UDP bsd-pc.cisco.com;branch=z9hG4bK433yte4
To: Larry <sip:larry@princeton.edu>
From: Bruce <sip:bruce@cisco.com>;tag=55123
Call-ID: xy745jj210re3@bsd-pc.cisco.com
CSeq: 271828 INVITE
Contact: <sip:bruce@bsd-pc.cisco.com>
Content-Type: application/sdp
Content-Length: 142
```

Первая строка идентифицирует тип выполняемой функции (*invite*); ресурс, на котором ее нужно выполнить (`sip:larry@princeton.edu`); и версию протокола (2.0). Последующие строки заголовка, вероятно, выглядят знакомо из-за их сходства с заголовками в электронном письме. SIP определяет большое количество полей заголовка, только некоторые из которых мы опишем здесь. Обратите внимание, что заголовок в этом примере идентифицирует устройство, с которого было отправлено сообщение. Поля *Content-Type* и *Content-Length* описывают содержимое сообщения после заголовка, как в MIME-кодированном электронном письме. В данном случае содержимое — это сообщение SDP. Это сообщение описывало бы такие вещи, как тип медиа (аудио, видео и т.д.), которым Брюс хотел бы обмениваться с Ларри, и другие свойства сеанса, такие как поддерживаемые типы кодеков. Обратите внимание, что поле *Contact* в SIP предоставляет возможность использовать любой протокол для этой цели, хотя SDP является наиболее распространенным.

Вернемся к примеру: когда сообщение приглашения (*invite*) приходит на прокси, прокси не только пересылает сообщение в сторону `princeton.edu`, но и отвечает отправителю приглашения. Как и в HTTP, все ответы имеют код ответа, и организация кодов аналогична той, что используется в HTTP. На рис. 9.9 мы можем увидеть последовательность сообщений и ответов SIP.

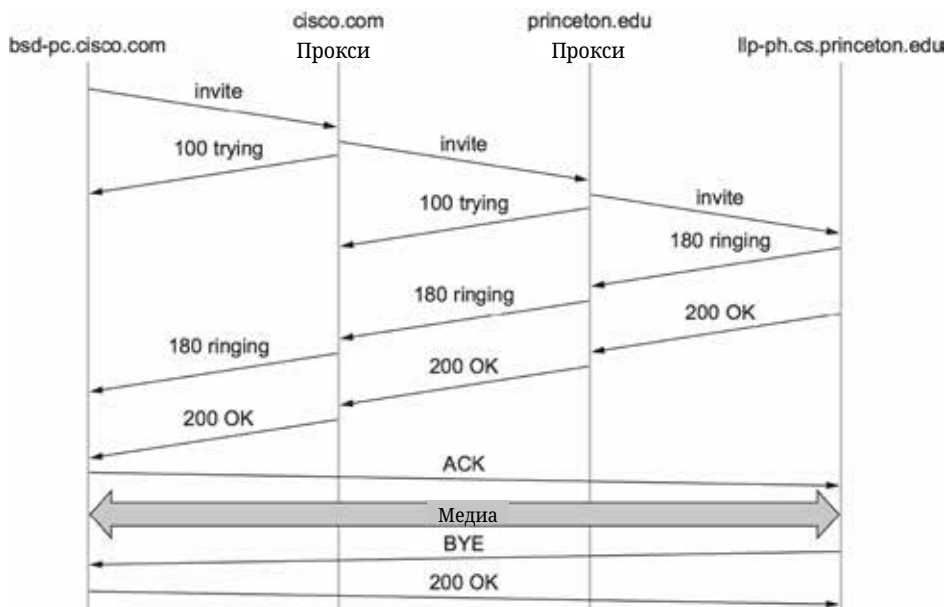


Рисунок 9.9. Поток сообщений для базового сеанса SIP.

Первое сообщение ответа на этом рисунке — это предварительный ответ 100 Trying, который указывает на то, что сообщение было получено без ошибок прокси вызывающего абонента. Как только приглашение доставляется на телефон Ларри, оно уведомляет Ларри и отвечает сообщением 180 Ringing. Прибытие этого сообщения на компьютер Брюса является сигналом, что можно сгенерировать «рингтон». Если Ларри готов и способен общаться с Брюсом, он может поднять трубку, что вызовет отправку сообщения 200 OK. Компьютер Брюса отвечает подтверждением (ACK), и медиа (например, аудиопоток, инкапсулированный в RTP) могут начать течь между двумя сторонами. Обратите внимание, что на этом этапе стороны знают адреса друг друга, поэтому подтверждение может быть отправлено напрямую, минуя прокси. Прокси теперь больше не участвуют в звонке. Заметьте, что медиа, таким образом, обычно проходят по другой траектории через сеть, чем исходные сигнальные сообщения. Более того, даже если один или оба прокси выйдут из строя в этот момент, звонок может продолжаться нормально. Наконец, когда одна из сторон желает завершить сеанс, она отправляет сообщение BYE, которое при нормальных обстоятельствах вызывает ответ 200 OK.

Есть несколько деталей, которые мы опустили. Одна из них — это согласование характеристик сеанса. Возможно, Брюс хотел бы общаться с использованием как аудио, так и видео, но телефон Ларри поддерживает только аудио. Таким образом, телефон Ларри отправит сообщение SDP в своем 200 OK, описывающее свойства сеанса, которые будут приемлемы для Ларри и устройства, учитывая варианты, предложенные в приглашении Брюса. Таким образом, взаимоприемлемые параметры сеанса согласовываются до начала потока медиа.

Еще одна большая проблема, которую мы опустили, это определение правильно устройства для Ларри. Сначала компьютер Брюса должен был отправить свое приглашение на прокси `cisco.com`. Это могло быть настроенной информацией на компьютере или могло быть получено через DHCP. Затем прокси `cisco.com` должен был найти прокси `princeton.edu`. Это можно сделать с помощью специального вида поиска DNS, который возвращает IP-адрес SIP прокси для домена. (Мы обсудим, как DNS может это сделать, в следующей главе.) Наконец, прокси `princeton.edu` должен был найти устройство, на котором можно связаться с Ларри. Обычно прокси-сервер имеет доступ к базе данных местоположе-

ний, которая может заполняться несколькими способами. Ручная настройка — один из вариантов, но более гибкий вариант — использование *регистрационных* возможностей SIP.

Пользователь может зарегистрироваться в службе местоположений, отправив сообщение SIP регистрации (*register*) регистратору своего домена. Это сообщение создает связь между «адресом записи» и «контактным адресом». «Адрес записи», вероятно, будет SIP URI, который является общеизвестным адресом для пользователя (например, `sip:larry@princeton.edu`), а «контактный адрес» будет адресом, по которому пользователь в данный момент может быть найден (например, `sip:larry@llp-ph.cs.princeton.edu`). Именно такая связь была необходима прокси `princeton.edu` в нашем примере.

Заметьте, что пользователь может зарегистрироваться в нескольких местах, и несколько пользователей могут зарегистрироваться на одном устройстве. Например, можно представить группу людей, входящих в конференц-зал, оборудованный IP-телефоном, и всех их, регистрирующихся на нем, чтобы они могли получать вызовы на этот телефон.

SIP — это очень богатый и гибкий протокол, который может поддерживать широкий спектр сложных сценариев вызовов, а также приложения, которые мало или совсем не связаны с телефонией. Например, SIP поддерживает операции, позволяющие перенаправлять вызов на сервер с музыкой на удержании или на сервер голосовой почты. Также легко увидеть, как он может быть использован для приложений, таких как мгновенные сообщения, и стандартизация расширений SIP для таких целей продолжается.

H.323

Международный союз электросвязи (ITU) также был очень активен в области управления вызовами, что неудивительно, учитывая его актуальность для телефонии, традиционной сферы этой организации. К счастью, в данном случае было значительное сотрудничество между IETF и ITU, поэтому различные протоколы в некоторой степени совместимы. Основная рекомендация ITU для мультимедийной связи по пакетным сетям известна как H.323, которая объединяет многие другие рекомендации, включая H.225 для управления вызовами. Полный набор рекомендаций, охваченных H.323, занимает много сотен страниц, и протокол известен своей сложностью, поэтому здесь можно дать только краткий обзор.

H.323 популярен как протокол для интернет-телефонии, включая видеозвонки, и мы рассматриваем этот класс приложений здесь. Устройство, которое инициирует или завершает вызовы, называется терминалом H.323; это может быть рабочая станция, на которой работает приложение интернет-телефонии, или это может быть специально разработанное «устройство» — телефоноподобное устройство с сетевым программным обеспечением и портом Ethernet, например. Терминалы H.323 могут напрямую общаться друг с другом, но вызовы часто обрабатываются устройством, известным как *страж*. Стражи выполняют ряд функций, таких как перевод различных форматов адресов, используемых для телефонных вызовов, и контроль за количеством вызовов, которые могут быть сделаны в данный момент, чтобы ограничить полосу пропускания, используемую приложениями H.323. H.323 также включает в себя концепцию *шлюза*, который подключает сеть H.323 к другим типам сетей. Наиболее распространенное использование шлюза — подключение сети H.323 к общественной коммутируемой телефонной сети (PSTN), как показано на рис. 9.10. Это позволяет пользователю, использующему приложение H.323 на компьютере, разговаривать с человеком, использующим обычный телефон в общественной телефонной сети. Одна из полезных функций, выполняемых шлюзом, — это помощь терминалу в нахождении шлюза, возможно, выбирая среди нескольких вариантов, чтобы найти тот, который относительно близок к конечному пункту назначения вызова. Это явно полезно в мире, где обычные телефоны значительно превосходят по количеству телефоны на базе ПК. Когда терминал H.323 совершает вызов на конечную точку, которая является обычным телефоном, шлюз становится фактической конечной точкой для вызова H.323 и отвечает за выполнение соответствующего перевода как сигнальной информации, так и медиапотока, который необходимо передавать по телефонной сети.

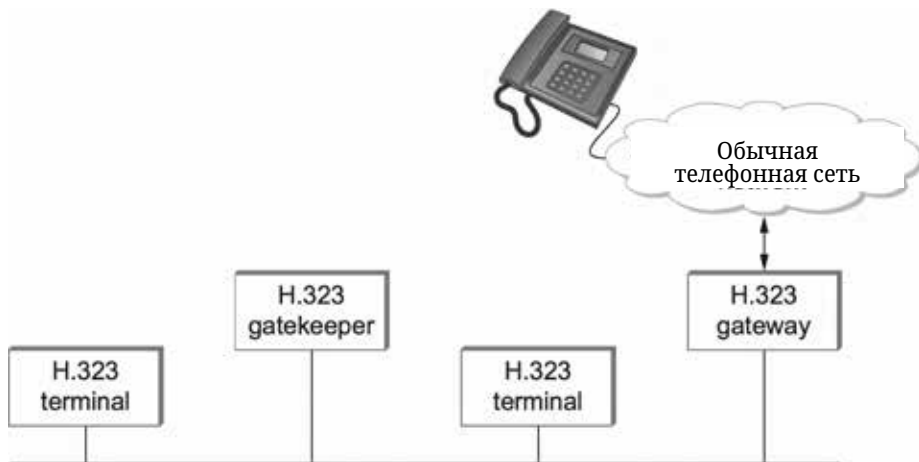


Рисунок 9.10. Устройства в сети H.323.

Важной частью H.323 является протокол H.245, который используется для согласования свойств вызова, аналогично использованию SDP, описанному выше. Сообщения H.245 могут перечислять несколько различных стандартов аудиокодеков, которые он может поддерживать; дальняя конечная точка вызова ответит своим списком поддерживаемых кодеков, и две стороны смогут выбрать стандарт кодирования, который будет удовлетворительным для обеих. H.245 также может использоваться для передачи номеров портов UDP, которые будут использоваться RTP и протоколом управления реальным временем (RTCP) для медиапотока (или потоков — вызов может включать как аудио, так и видео, например) для этого вызова. Как только это будет выполнено, вызов может продолжаться, при этом RTP используется для передачи медиапотоков, а RTCP — для передачи соответствующей управляющей информации.

Глава 9.2.2. Распределение ресурсов для мультимедийных приложений

Как мы только что видели, протоколы управления сеансами, такие как SIP и H.323, могут использоваться для инициирования и управления коммуникацией в мультимедийных приложениях, тогда как RTP обеспечивает функции уровня транспортировки для потоков данных приложений. Последний элемент головоломки в работе мультимедийных приложений заключается в обеспечении того, чтобы подходящие ресурсы были выделены внутри сети, чтобы удовлетворить потребности приложений в качестве обслуживания. Мы представили ряд методов распределения ресурсов в предыдущем разделе. Мотивацией для разработки этих технологий в значительной степени была поддержка мультимедийных приложений. Так как же приложения используют возможности сети по распределению ресурсов?

Стоит отметить, что многие мультимедийные приложения успешно работают в сетях с «лучшим усилием», таких как публичный Интернет. Широкий спектр коммерческих услуг VOIP (таких как Skype) свидетельствует о том, что о выделении ресурсов нужно беспокоиться только тогда, когда ресурсы ограничены, а во многих частях современного Интернета избыток ресурсов является нормой.

Протокол, такой как RTCP, может помочь приложениям в сетях, работающих по принципу лучшего усилия, предоставляя приложению подробную информацию о качестве обслуживания, которое обеспечивает сеть. Напомним, что RTCP передает информацию о скорости потерь и характеристиках задержки между участниками мультимедийного приложения. Приложение может использовать эту информацию для изменения своей

схемы кодирования — например, перейти на кодек с более низкой скоростью передачи данных, когда пропускная способность ограничена. Заметим, что хотя может возникнуть соблазн перейти на кодек, отправляющий дополнительную, избыточную информацию при высоких уровнях потерь, это не приветствуется; это аналогично увеличению размера окна TCP при наличии потерь, что является противоположностью необходимому для избежания коллапса перегрузки.

Как обсуждалось в предыдущем разделе, Differentiated Services (DiffServ) можно использовать для обеспечения довольно базового и масштабируемого распределения ресурсов для приложений. Мультимедийное приложение может установить кодовую точку дифференцированных услуг (DSCP) в заголовке IP-пакетов, которые оно генерирует, для обеспечения того, чтобы как медиа, так и управляющие пакеты получали соответствующее качество обслуживания. Например, обычно медиапакеты голосовой связи помечаются как «EF» (ускоренная пересылка), чтобы их помещали в очередь с низкой задержкой или приоритетную очередь в маршрутизаторах по пути, в то время как пакеты управления вызовами (например, пакеты SIP) часто помечаются как «AF» (гарантированная пересылка), чтобы их можно было помещать в отдельную очередь от трафика с лучшим усилием и таким образом уменьшить риск их потери.

Конечно, имеет смысл маркировать пакеты внутри отправляющего хоста или устройства только в том случае, если сетевые устройства, такие как маршрутизаторы, обращают внимание на DSCP. В общем маршрутизаторы в публичном Интернете игнорируют DSCP, предоставляя услуги с лучшим усилием всем пакетам. Однако корпоративные сети имеют возможность использовать DiffServ для своего внутреннего мультимедийного трафика и часто делают это. Также даже домашние пользователи Интернета могут часто улучшить качество VOIP или других мультимедийных приложений, просто используя DiffServ на исходящем направлении своих интернет-соединений, как показано на рис. 9.11. Это эффективно из-за асимметрии многих широкополосных интернет-соединений: если исходящая линия значительно медленнее (т.е. имеет меньше ресурсов), чем входящая, то распределение ресурсов с использованием DiffServ на этой линии может быть достаточным, чтобы значительно улучшить качество для приложений, чувствительных к задержкам и потерям.



Рисунок 9.11. Дифференцированные услуги, применяемые к приложению VOIP.
Очередь DiffServ применяется только на восходящем канале
от маршрутизатора клиента к провайдеру.

Хотя DiffServ привлекателен своей простотой, ясно, что он не может удовлетворить потребности приложений при любых условиях. Например, предположим, что пропускная способность восходящего канала на рис. 9.11 составляет всего 100 кбит/с, и клиент пыта-

ется совершить два VOIP-вызова, каждый с кодеком на 64 кбит/с. Очевидно, что восходящий канал теперь более чем на 100% загружен, что приведет к большим задержкам в очереди и потере пакетов. Никакое количество умных очередей в маршрутизаторе клиента не сможет это исправить.

Характеристики многих мультимедийных приложений таковы, что, вместо того чтобы пытаться впихнуть слишком много вызовов в слишком узкий канал, лучше заблокировать один вызов, позволив другому продолжиться. То есть лучше, чтобы один человек успешно разговаривал, пока другой слышит сигнал «занято», чем чтобы оба абонента испытывали неприемлемое качество звука одновременно. Мы иногда называем такие приложения имеющими *крутые кривые полезности*, что означает, что полезность приложения быстро падает по мере ухудшения качества обслуживания, предоставляемого сетью. Мультимедийные приложения часто обладают этим свойством, тогда как многие традиционные приложения — нет. Электронная почта, например, продолжает работать довольно хорошо, даже если задержки достигают часов.

Приложения с крутыми кривыми полезности часто хорошо подходят для некоторых форм контроля доступа. Если вы не можете быть уверены, что достаточные ресурсы всегда будут доступны для поддержки предлагаемой нагрузки приложений, тогда контроль доступа предоставляет способ сказать «нет» некоторым приложениям, позволяя другим получить необходимые им ресурсы.

Мы видели один способ осуществления контроля доступа с использованием RSVP в предыдущем разделе и вскоре вернемся к этому, но мультимедийные приложения, которые используют протоколы управления сеансами, предоставляют некоторые другие варианты контроля доступа. Основной момент, который нужно здесь отметить, заключается в том, что протоколы управления сеансами, такие как SIP или H.323, часто включают в себя некоторый обмен сообщениями между конечной точкой и другой сущностью (SIP-прокси или стражем H.323) в начале вызова или сеанса. Это может обеспечить удобный способ сказать «нет» новому вызову, для которого недостаточно ресурсов.

Рассмотрим сеть, изображенную на рис. 9.12. Допустим, что широкополосное соединение из филиала в головной офис имеет достаточную пропускную способность для одновременного обслуживания трех вызовов VOIP, использующих кодеки с 64 кбит/с. Каждый телефон уже должен связываться с локальным SIP-прокси или H.323-шлюзом, когда начинает вызов, поэтому прокси/шлюз может легко отправить сообщение, которое скажет IP-телефону воспроизводить сигнал занятости, если это соединение уже полностью загружено. Прокси или шлюз также может учитывать возможность того, что конкретный IP-телефон может совершать несколько вызовов одновременно и что могут использоваться различные скорости кодеков. Однако эта схема будет работать только в том случае, если ни одно другое устройство не сможет перегрузить канал, не связавшись сначала с шлюзом или прокси. Очереди DiffServ можно использовать для обеспечения того, чтобы, например, ПК, выполняющий передачу файлов, не мешал вызовам VOIP. Но предположим, что в удаленном офисе включено какое-то VOIP-приложение, которое сначала не связывается со шлюзом или прокси. Такое приложение, если оно может правильно пометить свои пакеты и поместить их в ту же очередь, что и существующий VOIP-трафик, может явно перегрузить канал без обратной связи с прокси или шлюзом.

Еще одна проблема описанного подхода заключается в том, что он зависит от того, что шлюз или прокси знают о пути, который будет использовать каждое приложение. В простой топологии, как на рис. 9.12, это не является большой проблемой, но в более сложных сетях это может быстро стать неуправляемым. Достаточно представить случай, когда у удаленного офиса есть два разных соединения с внешним миром, чтобы понять, что мы просим прокси или шлюз понять не только SIP или H.323, но и маршрутизацию, сбой в каналах и текущие условия сети.

Мы называем описанный контроль доступа «вне пути» (off-path), в том смысле, что устройство, принимающее решения о контроле доступа, не находится на пути данных, где нужно выделять ресурсы. Очевидной альтернативой является контроль доступа «на пути» (on-path), и стандартным примером протокола, который выполняет контроль

доступа «на пути» в IP-сетях, является протокол резервирования ресурсов (RSVP). Мы видели в предыдущем разделе, как RSVP можно использовать для обеспечения достаточного выделения ресурсов по пути, и его легко использовать в приложениях, описанных в этом разделе. Единственной деталью, которую еще нужно уточнить, является то, как протокол контроля доступа взаимодействует с протоколом управления сеансами.

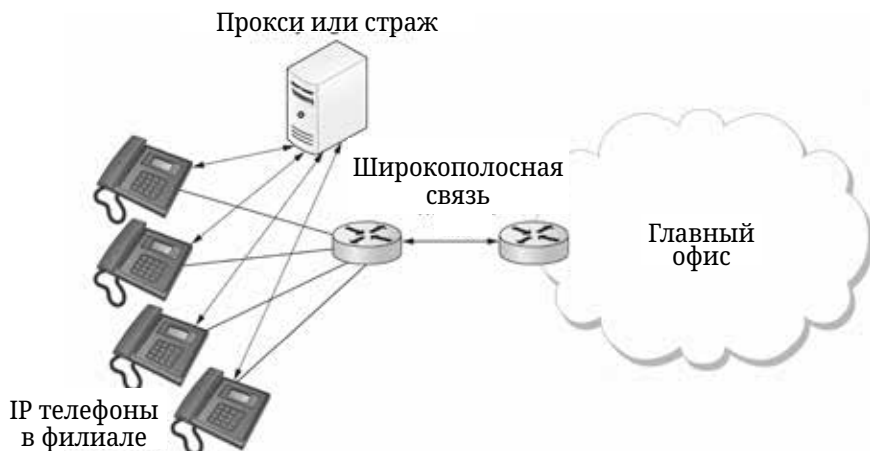


Рисунок 9.12. Контроль допуска с помощью протокола управления сеансами.

Координация действий протокола контроля доступа (или резервирования ресурсов) и протокола управления сеансами не является чем-то сложным, но требует внимания к деталям. Например, рассмотрим простой телефонный звонок между двумя сторонами. Прежде чем вы сможете сделать резервирование, вам нужно знать, сколько полосы пропускания будет использовать вызов, а это означает, что вам нужно знать, какие кодеки будут использоваться. Это подразумевает, что сначала нужно выполнить некоторую часть управления сеансом, чтобы обменяться информацией о поддерживаемых кодеках между двумя телефонами. Однако нельзя выполнить все действия по управлению сеансом заранее, потому что вы не захотите, чтобы телефон зазвонил до принятия решения о контроле доступа, в случае если контроль доступа не удастся. На рис. 9.13 показана подобная ситуация, когда SIP используется для управления сеансом, а RSVP — для принятия решения о контроле доступа (успешно в этом случае).

Основное, на что нужно обратить внимание здесь, — это чередование задач управления сеансом и распределения ресурсов. Сплошные линии представляют собой сообщения SIP, пунктирные линии — сообщения RSVP. Заметьте, что сообщения SIP передаются напрямую от телефона к телефону в этом примере (то есть мы не показали никаких SIP-прокси), тогда как сообщения RSVP также обрабатываются промежуточными маршрутизаторами, которые проверяют наличие достаточных ресурсов для допуска вызова.

Мы начинаем с начального обмена информацией о кодеках в первых двух сообщениях SIP (напомним, что SDP используется для перечисления доступных кодеков и других вещей). PRACK — это «предварительное подтверждение». После обмена этими сообщениями можно отправить сообщения RSVP PATH, которые содержат описание требуемых ресурсов, как первый шаг в резервировании ресурсов в обоих направлениях вызова. Затем можно отправить сообщения RESV для фактического резервирования ресурсов. Как только инициирующий телефон получает сообщение RESV, он может отправить обновленное сообщение SDP, сообщающее о том, что ресурсы были зарезервированы в одном направлении. Когда вызываемый телефон получит это сообщение и RESV от другого телефона, он может начать звонить и сообщить другому телефону, что ресурсы теперь зарезервированы в обоих направлениях (с помощью сообщения SDP), а также уведомить вызывающий телефон

о том, что он звонит. С этого момента продолжают нормальная сигнализация SIP и поток мультимедиа, аналогичный показанному на рис. 9.9.

Мы снова видим, как создание приложений требует понимания взаимодействия между различными строительными блоками (в данном случае SIP и RSVP). Дизайнеры SIP на самом деле внесли некоторые изменения в протокол, чтобы позволить это чередование функций между протоколами с разными задачами, поэтому мы неоднократно подчеркиваем в этой книге важность фокусирования на полных системах, а не только на одном уровне или компоненте в изоляции от других частей системы.

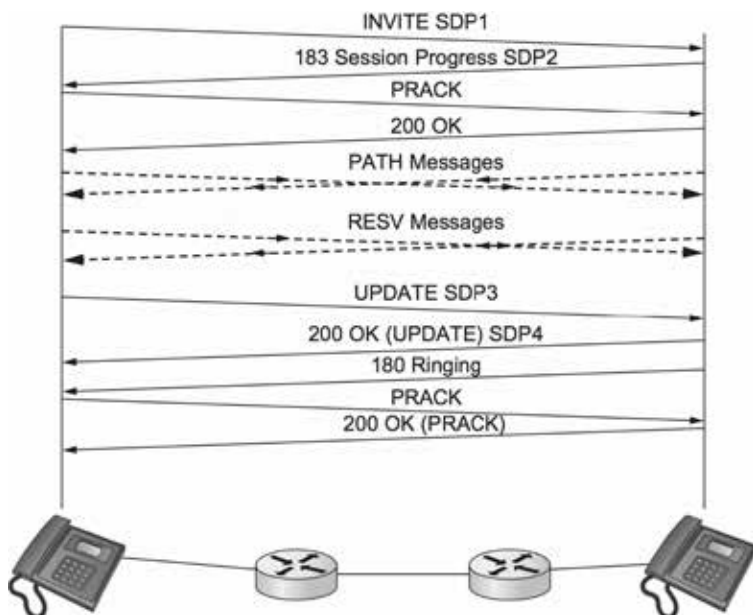


Рисунок 9.13. Координация сигнализации SIP и резервирования ресурсов.

Глава 9.3. Инфраструктурные приложения

Существуют протоколы, которые жизненно важны для бесперебойной работы Интернета, но не вписываются в строго слоистую модель. Одним из таких является система доменных имен (DNS) — это не то приложение, которое пользователи обычно вызывают напрямую, но сервис, от которого зависят почти все остальные приложения. Это потому, что служба имен используется для преобразования имен хостов в адреса хостов; наличие такого приложения позволяет пользователям других приложений ссылаться на удаленные хосты по имени, а не по адресу. Другими словами, служба имен обычно используется другими приложениями, а не людьми.

Второй критической функцией является управление сетью, которое, хотя и не так знакомо среднему пользователю, выполняется чаще всего людьми, которые управляют сетью от имени пользователей. Управление сетью повсеместно считается одной из трудных проблем сетевых технологий и продолжает оставаться объектом многочисленных инноваций. Мы рассмотрим некоторые из проблем и подходов к решению этой задачи ниже.

Глава 9.3.1. Служба имен (DNS)

В большинстве разделов этой книги для идентификации хостов мы использовали адреса. Хотя адреса идеально подходят для обработки маршрутизаторами, они не очень удоб-

ны для пользователей. Именно поэтому каждому хосту в сети обычно присваивается уникальное имя. Уже в этом разделе мы видели, каким образом такие протоколы приложений, как HTTP, используют имена, например, `www.princeton.edu`. Теперь мы опишем, как можно разработать службу генерации имен для сопоставления удобных для пользователя имен с адресами, удобными для маршрутизаторов. Службы имен иногда называют *промежуточным программным обеспечением* (middleware), потому что они заполняют разрыв между приложениями и основной сетью.

Имена хостов отличаются от адресов хостов двумя важными свойствами. Во-первых, они обычно переменной длины и мнемоничны, что делает их более удобными для запоминания людьми. (В отличие от этого, числовые адреса фиксированной длины легче обрабатывать маршрутизаторам.) Во-вторых, имена обычно не содержат информации, которая помогает сети найти хост (маршрутизировать пакеты к нему). В отличие от этого, адреса иногда содержат встроенную информацию о маршрутизации; *плоские* адреса (те, которые не делятся на составные части) являются исключением.

Прежде чем углубиться в детали того, как хосты именуются в сети, сначала введем некоторые основные термины. Во-первых, *пространство имен* определяет набор возможных имен. Пространство имен может быть либо плоским (имена не делятся на компоненты), либо *иерархическим* (очевидный пример — имена файлов в Unix). Во-вторых, система имен поддерживает коллекцию *привязок* имен к значениям. Значение может быть любым, каким мы хотим, чтобы система имен возвращала при предъявлении имени; в большинстве случаев это адрес. Наконец, *механизм разрешения* — это процедура, которая, будучи вызванной с именем, возвращает соответствующее значение. *Сервер имен* — это конкретная реализация механизма разрешения, доступная в сети, к которой можно обращаться, отправляя ей сообщение.

Из-за своего большого размера Интернет имеет особенно хорошо развитую систему имен — систему доменных имен (DNS). Поэтому мы используем DNS как основу для обсуждения проблемы именования хостов. Следует отметить, что Интернет не всегда использовал DNS. В начале своей истории, когда в Интернете было всего несколько сотен хостов, центральный орган под названием *Центр сетевой информации* (NIC) вел плоскую таблицу привязок имен к адресам; эта таблица называлась HOSTS.TXT¹. Каждый раз, когда сайт хотел добавить новый хост в Интернет, администратор сайта отправлял электронное письмо в NIC с именем/адресом нового хоста. Эта информация вручную вводилась в таблицу, измененная таблица рассылалась по различным сайтам каждые несколько дней, и системный администратор на каждом сайте устанавливал таблицу на каждом хосте на сайте. Разрешение имен тогда просто реализовывалось процедурой, которая искала имя хоста в локальной копии таблицы и возвращала соответствующий адрес.

Неудивительно, что такой подход к именованию плохо работал по мере увеличения числа хостов в Интернете. Поэтому в середине 1980-х годов была введена система *доменных имен* (DNS). DNS использует иерархическое пространство имен вместо плоского пространства имен, а «таблица» привязок, реализующая это пространство имен, разделена на непересекающиеся части и распределена по всему Интернету. Эти подтаблицы доступны на серверах имен, к которым можно обращаться через сеть.

Что происходит в Интернете: пользователь представляет имя хоста программе приложения (возможно, встроенное в составное имя, такое как адрес электронной почты или URL), и эта программа использует систему имен для преобразования этого имени в адрес хоста. Затем приложение устанавливает соединение с этим хостом, представляя некоторый транспортный протокол (например, TCP) с IP-адресом хоста. Эта ситуация проиллюстрирована (в случае отправки электронной почты) на рис. 9.14. Хотя эта схема делает задачу разрешения имен достаточно простой, на самом деле все немного сложнее, чем кажется.

¹ Хотите верить, хотите нет, но периодически издавалась бумажная книга (наподобие телефонного справочника), в которой перечислялись все машины, подключенные к Интернету, и все люди, у которых была учетная запись электронной почты в Интернете.

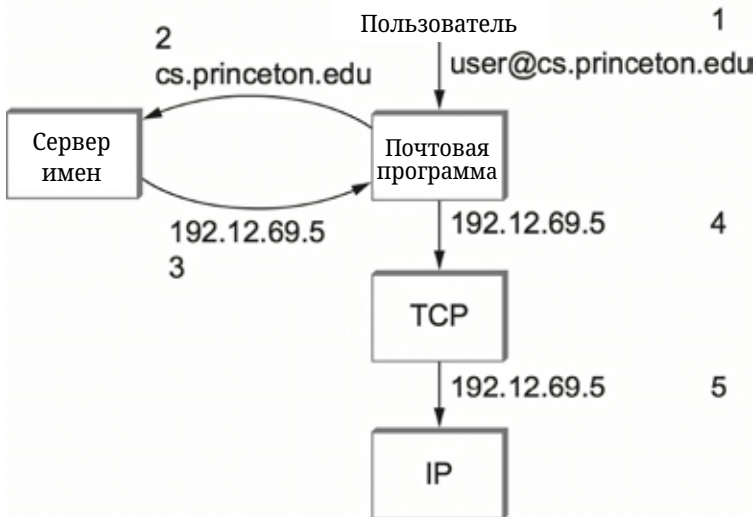


Рисунок 9.14. Имена, переведенные в адреса, где цифры от 1 до 5 показывают последовательность шагов в процессе.

Иерархия доменов

DNS реализует иерархическое пространство имен для интернет-объектов. В отличие от имен файлов в Unix, которые обрабатываются слева направо с разделением компонентов имен с помощью косых черт, имена DNS обрабатываются справа налево и используют точки в качестве разделителей. (Хотя они обрабатываются справа налево, люди все еще читают доменные имена слева направо.) Пример доменного имени для хоста: `cs.princeton.edu`.

Заметьте, мы сказали, что доменные имена используются для именования интернет-«объектов». Это означает, что DNS не строго используется для сопоставления имен хостов с адресами хостов.

Более точно будет сказать, что DNS сопоставляет доменные имена со значениями. Пока что мы предполагаем, что эти значения — IP-адреса; мы вернемся к этому вопросу позже в данной главе.

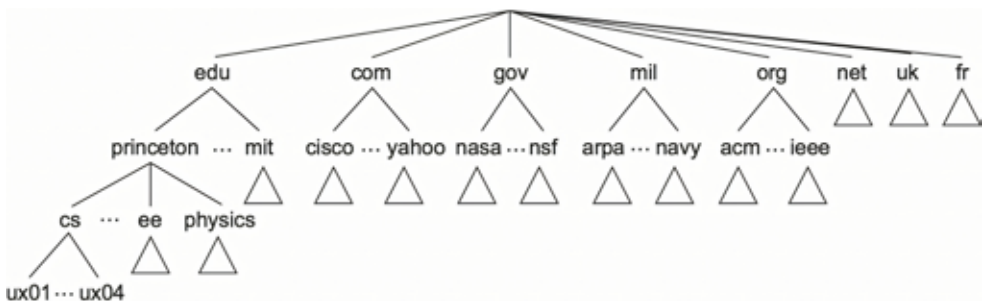


Рисунок 9.15. Пример иерархии доменов.

Как и иерархию файлов в Unix, иерархию DNS можно визуализировать как дерево, где каждый узел в дереве соответствует домену, а листья в дереве соответствуют именуемым хостам. Рис. 9.15 дает пример иерархии доменов. Обратите внимание, что мы не должны

придавать термину «домен» какого-либо значения, кроме того, что это просто контекст, в котором могут быть определены дополнительные имена.¹

Когда иерархия доменных имен только разрабатывалась, велись серьезные обсуждения о том, какие конвенции будут регулировать имена, которые будут выдаваться на вершине иерархии. Не углубляясь в подробности этих обсуждений, обратите внимание, что иерархия на первом уровне не очень широкая. Существуют домены для каждой страны, а также «большая шестерка» доменов: .edu, .com, .gov, .mil, .org и .net. Эти шесть доменов изначально базировались в Соединенных Штатах (где были изобретены Интернет и DNS); например, только аккредитованные учебные заведения США могут регистрировать доменные имена .edu. В последние годы количество доменов верхнего уровня было расширено, отчасти для удовлетворения высокого спроса на доменные имена .com. Новые домены верхнего уровня включают .biz, .coop и .info. Сейчас существует более 1200 доменов верхнего уровня.

Серверы имен

Полная иерархия доменных имен существует только в абстракции. Теперь мы обратим внимание на вопрос, как эта иерархия фактически реализована. Первый шаг — разделить иерархию на поддеревья, называемые *зонами*. Рис. 9.16 показывает, как иерархия, представленная на рис. 9.15, может быть разделена на зоны. Каждую зону можно рассматривать как соответствующую некоторому административному органу, ответственному за эту часть иерархии. Например, верхний уровень иерархии образует зону, управляемую Корпорацией интернета по присвоению имен и номеров (ICANN). Ниже находится зона, соответствующая Принстонскому университету. Внутри этой зоны некоторые департаменты не хотят брать на себя ответственность за управление иерархией (и поэтому остаются в зоне университетского уровня), в то время как другие, такие как факультет компьютерных наук, управляют своей собственной зоной уровня департамента.

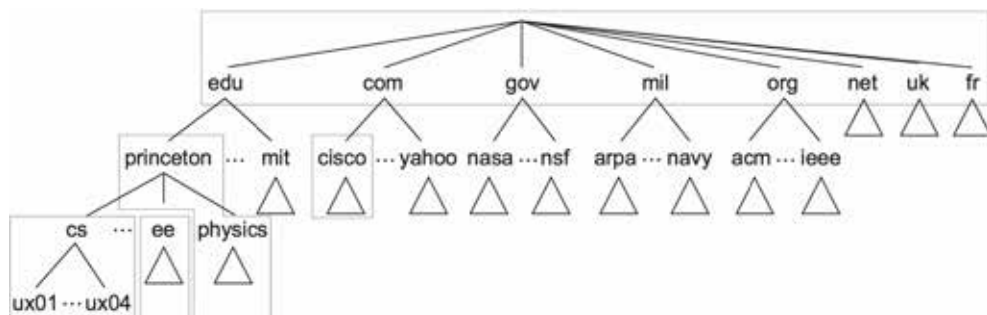


Рисунок 9.16. Иерархия домена, разделенная на зоны.

Зона соответствует фундаментальной единице реализации в DNS — серверу имен. Информация, содержащаяся в каждой зоне, реализована в двух или более серверах имен. Каждый сервер имен, в свою очередь, представляет собой программу, к которой можно получить доступ через Интернет. Клиенты отправляют запросы серверам имен, а серверы имен отвечают на запросы с запрашиваемой информацией. Иногда ответ содержит окончательный ответ, который хочет получить клиент, а иногда ответ содержит указатель на другой сервер, к которому клиент должен обратиться. Таким образом, с точки зрения реализации правильнее считать, что DNS представлен иерархией серверов имен, а не иерархией доменов, как показано на рис. 9.17.

¹ Как ни странно, слово «домен» также используется в маршрутизации Интернета, где оно означает нечто иное, чем в DNS, будучи примерно эквивалентным термину *автономная система*.

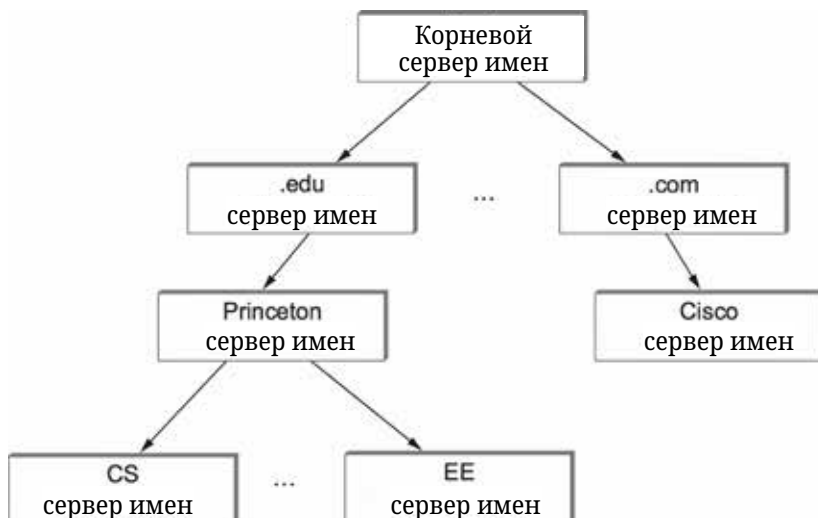


Рисунок 9.17. Иерархия серверов имен.

Каждая зона реализована в двух или более серверах имен для обеспечения избыточности; то есть информация будет доступна, даже если один сервер имен выйдет из строя. Кроме того, один сервер имен может обслуживать более одной зоны.

Каждый сервер имен реализует информацию зоны в виде набора ресурсных записей. По сути, ресурсная запись представляет собой привязку имени к значению, или, более конкретно, 5-элементный кортеж, содержащий следующие поля:

(Name, Value, Type, Class, TTL)

Поля Name (Имя) и Value (Значение) — это то, что вы ожидаете, в то время как поле Type (Тип) указывает, как следует интерпретировать значение. Например, Type=A указывает, что Value является IP-адресом. Таким образом, A-записи реализуют сопоставление имени и адреса, которое мы рассматривали ранее. Другие типы записей включают:

- NS — в поле Value указывается доменное имя хоста, на котором работает сервер имен, знающий, как разрешить имена в указанном домене.
- CNAME — в поле Value указывается каноническое имя для конкретного хоста; используется для определения псевдонимов.
- MX — в поле Value указывается доменное имя хоста, на котором работает почтовый сервер, принимающий сообщения для указанного домена.

Поле Class было включено для того, чтобы позволить другим организациям, помимо NIC, определять полезные типы записей. До настоящего времени единственным широко используемым классом является тот, который применяется в Интернете; он обозначается как IN. Наконец, поле TTL (время жизни) показывает, как долго данная ресурсная запись является действительной. Оно используется серверами, кеширующими ресурсные записи с других серверов; когда время жизни истекает, сервер должен удалить запись из своего кеша.

Чтобы лучше понять, как ресурсные записи представляют информацию в иерархии доменов, рассмотрим следующие примеры, взятые из иерархии доменов на рис. 9.15. Для упрощения примера мы игнорируем поле TTL и приводим актуальную информацию только для одного из серверов имен, реализующих каждую зону.

Во-первых, корневой сервер имен содержит запись NS для каждого сервера имен домена верхнего уровня (TLD). Это определяет сервер, который может разрешать запросы

для этой части иерархии DNS (.edu и .com в данном примере). Также имеются записи A, которые переводят эти имена в соответствующие IP-адреса. В совокупности эти две записи эффективно реализуют указатель от корневого сервера имен к одному из серверов TLD.

```
(edu, a3.nstld.com, NS, IN)
(a3.nstld.com, 192.5.6.32, A, IN)
(com, a.gtld-servers.net, NS, IN)
(a.gtld-servers.net, 192.5.6.30, A, IN)
```

...

Перемещаясь вниз по иерархии на один уровень, сервер имеет записи для доменов, таких как:

```
(princeton.edu, dns.princeton.edu, NS, IN)
(dns.princeton.edu, 128.112.129.15, A, IN)
```

...

другом уровне иерархии (например, запрос пингвинов на penguins.cs.princeton.edu).

В этом случае у нас есть запись NS и запись A для сервера имен, ответственного за часть иерархии princeton.edu. Этот сервер может непосредственно разрешить некоторые запросы (например, для email.princeton.edu), в то время как другие запросы он перенаправит на сервер на другом уровне иерархии (например, для запроса о penguins.cs.princeton.edu).

```
(email.princeton.edu, 128.112.198.35, A, IN) (penguins.cs.princeton.edu, dns1.
cs.princeton.edu, NS, IN) (dns1.cs.princeton.edu, 128.112.136.10, A, IN)
```

...

Наконец, сервер имен третьего уровня, такой как управляемый доменом cs.princeton.edu, содержит A-записи для всех своих хостов. Он также может определять набор псевдонимов (CNAME-записей) для каждого из этих хостов. Псевдонимы иногда просто являются удобными (например, более короткими) именами для машин, но они также могут использоваться для обеспечения уровня косвенности. Например, www.cs.princeton.edu является псевдонимом для хоста с именем coreweb.cs.princeton.edu. Это позволяет веб-серверу сайта перемещаться на другую машину, не влияя на удаленных пользователей; они просто продолжают использовать псевдоним, не обращая внимания на то, какая машина в данный момент запускает веб-сервер домена. Почтовые обменные записи (MX-записи) служат той же цели для почтового приложения — они позволяют администратору изменить хост, который получает почту от имени домена, без необходимости менять электронный адрес каждого пользователя.

```
(penguins.cs.princeton.edu, 128.112.155.166, A, IN)
(www.cs.princeton.edu, coreweb.cs.princeton.edu, CNAME, IN)
(coreweb.cs.princeton.edu, 128.112.136.35, A, IN)
(cs.princeton.edu, mail.cs.princeton.edu, MX, IN)
(mail.cs.princeton.edu, 128.112.136.72, A, IN)
```

...

Заметьте, что хотя ресурсные записи могут быть определены практически для любого типа объекта, DNS обычно используется для именования хостов (включая серверы) и сайтов. Он не используется для именования отдельных людей или других объектов, таких как файлы или директории; для идентификации таких объектов обычно используются другие системы именования. Например, X.500 — это система именования ISO, разработанная для упрощения идентификации людей. Она позволяет вам назвать человека, указав набор атрибутов: имя, должность, номер телефона, почтовый адрес и так далее. X.500 оказался слишком громоздким — и, в некотором смысле,

был вытеснен мощными поисковыми системами, доступными сейчас в Интернете, — но он в конечном итоге эволюционировал в Протокол легкого доступа к директориям (Lightweight Directory Access Protocol, LDAP). LDAP является подмножеством X.500, первоначально разработанным как ПК-фронтенд для X.500. Сегодня широко используется, в основном на уровне предприятий, как система для получения информации о пользователях.

Разрешение имени

Имея иерархию серверов имен, теперь мы рассмотрим вопрос о том, как клиент взаимодействует с этими серверами для разрешения доменного имени. Чтобы проиллюстрировать основную идею, предположим, что клиент хочет разрешить имя `penguins.cs.princeton.edu` относительно набора серверов, упомянутых в предыдущей подглаве. Клиент может сначала отправить запрос, содержащий это имя, одному из корневых серверов (как мы увидим далее, это редко происходит на практике, но этого будет достаточно для иллюстрации базовой операции). Корневой сервер, не имея возможности сопоставить полное имя, возвращает наилучшее совпадение, которое у него есть — NS-запись для `edu`, указывающую на сервер TLD `a3.nstld.com`. Сервер также возвращает все записи, связанные с этой записью, в данном случае A-запись для `a3.nstld.com`. Клиент, не получив нужного ответа, отправляет тот же запрос на сервер имен с IP-адресом `192.5.6.32`. Этот сервер также не может сопоставить полное имя и поэтому возвращает NS и соответствующие A-записи для домена `princeton.edu`. Снова клиент отправляет тот же запрос на сервер с IP-адресом `128.112.129.15` и на этот раз получает NS-запись и соответствующую A-запись для домена `cs.princeton.edu`. На этот раз достигнут сервер, который может полностью разрешить запрос. Финальный запрос к серверу с IP-адресом `128.112.136.10` дает A-запись для `penguins.cs.princeton.edu`, и клиент узнает, что соответствующий IP-адрес — `128.112.155.166`.

Этот пример все еще оставляет несколько вопросов о процессе разрешения имен без ответа. Первый вопрос заключается в том, как клиент изначально находит корневой сервер, или, иначе говоря, как разрешить имя сервера, который знает, как разрешать имена? Это фундаментальная проблема в любой системе именования, и ответ заключается в том, что систему нужно каким-то образом загрузить. В этом случае сопоставление имени с адресом для одного или нескольких корневых серверов хорошо известно; то есть оно публикуется посредством некоторых средств вне самой системы именования.

На практике, однако, не все клиенты знают о корневых серверах. Вместо этого клиентская программа, работающая на каждом интернет-хосте, инициализируется с адресом *локального* сервера имен. Например, все хосты в департаменте компьютерных наук Принстона знают о сервере на `dns1.cs.princeton.edu`. Этот локальный сервер имен, в свою очередь, имеет ресурсные записи для одного или нескольких корневых серверов, например:

```
('root', a.root-servers.net, NS, IN)
(a.root-servers.net, 198.41.0.4, A, IN)
```

Таким образом, разрешение имени фактически включает в себя запрос клиента к локальному серверу, который, в свою очередь, действует как клиент, запрашивающий удаленные серверы от имени оригинального клиента. Это приводит к взаимодействиям «клиент/сервер», проиллюстрированным на рис. 9.18. Одним из преимуществ этой модели является то, что все хосты в Интернете не должны быть в курсе текущего расположения корневых серверов; только серверы должны знать о корне. Вторым преимуществом является то, что локальный сервер видит ответы, возвращающиеся из запросов, которые были отправлены всеми локальными клиентами. Локальный сервер *кеширует* эти ответы и иногда может разрешать будущие запросы, не выходя в сеть. Поле TTL в ресурсных

записях, возвращаемых удаленными серверами, указывает, как долго каждая запись может быть безопасно кэширована. Этот механизм кеширования также может использоваться выше в иерархии, снижая нагрузку на корневые и TLD серверы.

Второй вопрос заключается в том, как система работает, когда пользователь отправляет частичное имя (например, `penguins`) вместо полного доменного имени (например, `penguins.cs.princeton.edu`). Ответ заключается в том, что клиентская программа настроена с локальным доменом, в котором находится хост (например, `cs.princeton.edu`), и она добавляет эту строку к любым простым именам перед отправкой запроса.

Основные выводы

Чтобы убедиться, что вы все поняли: вы только что видели три разных уровня идентификаторов — доменные имена, IP-адреса и физические сетевые адреса, и сопоставление идентификаторов на одном уровне с идентификаторами на другом уровне происходит в разных точках сетевой архитектуры. Во-первых, пользователи указывают доменные имена при взаимодействии с приложением. Во-вторых, приложение использует DNS для перевода этого имени в IP-адрес; именно IP-адрес помещается в каждую дейтаграмму, а не доменное имя. (Кстати, этот процесс перевода включает в себя отправку IP-дейтаграмм по Интернету, но эти дейтаграммы адресованы хосту, который запускает сервер имен, а не конечному пункту назначения.) В-третьих, IP выполняет маршрутизацию на каждом маршрутизаторе, а это часто означает, что он сопоставляет один IP-адрес с другим; то есть он сопоставляет адрес конечного пункта назначения с адресом следующего маршрутизатора. Наконец, IP использует Протокол разрешения адресов (Address Resolution Protocol, ARP) для перевода IP-адреса следующего маршрутизатора в физический адрес этой машины; следующим маршрутизатором может быть конечный пункт назначения или промежуточный маршрутизатор. Фреймы (кадры), отправляемые по физической сети, имеют эти физические адреса в своих заголовках.

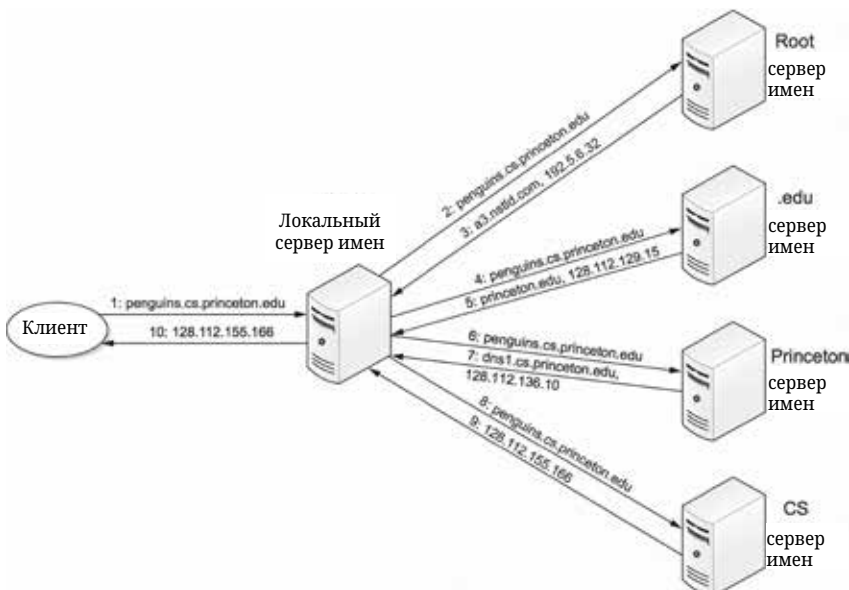


Рисунок 9.18. Разрешение имен на практике, где цифры от 1 до 10 показывают последовательность шагов в процессе.

Глава 9.3.2. Управление сетью (SNMP, OpenConfig)

Сеть представляет собой сложную систему как с точки зрения количества задействованных узлов, так и с точки зрения набора протоколов, которые могут работать на любом из узлов. Даже если вы ограничите себя заботой об узлах в пределах одной административной области, такой как кампус, в ней могут быть десятки маршрутизаторов и сотни — или даже тысячи — хостов, за которыми нужно следить. Если подумать обо всем состоянии, которое поддерживается и управляется на любом из этих узлов — таблицы преобразования адресов, таблицы маршрутизации, состояние TCP-соединений и так далее, — можно прийти в ужас от перспективы необходимости управлять всей этой информацией.

Легко представить себе желание узнать о состоянии различных протоколов на разных узлах. Например, вы можете захотеть мониторить количество абортированных повторных сборок IP-дейтаграмм, чтобы определить, нужно ли корректировать тайм-аут, который собирает мусор из частично собранных дейтаграмм. В качестве другого примера вы можете захотеть отслеживать нагрузку на различных узлах (то есть количество отправленных или полученных пакетов), чтобы определить, нужно ли добавлять новые маршрутизаторы или связи в сеть. Конечно, вы также должны следить за признаками неисправного оборудования и неправильно работающего программного обеспечения.

То, что мы только что описали, является проблемой сетевого управления, которая пронизывает всю сетевую архитектуру. Поскольку узлы, за которыми мы хотим следить, распределены, наш единственный реальный вариант — использовать сеть для управления сетью. Это означает, что нам нужен протокол, который позволяет нам читать и записывать различные части информации о состоянии на разных сетевых узлах. Далее описаны два подхода.

SNMP

Широко используемым протоколом для сетевого управления является SNMP (*Simple Network Management Protocol* — Простой протокол управления сетью). SNMP по сути представляет собой специализированный протокол «запрос/ответ», который поддерживает два типа запросов: GET и SET. Первый используется для получения части информации о состоянии с какого-то узла, а второй — для сохранения новой информации о состоянии на каком-то узле. (SNMP также поддерживает третью операцию, GET-NEXT, которую мы объясним позже.) Далее мы сосредоточимся на операции GET, так как она используется чаще всего.

SNMP используется очевидным образом. Оператор взаимодействует с клиентской программой, которая отображает информацию о сети. Эта клиентская программа обычно имеет графический интерфейс. Можно представить себе этот интерфейс как выполняющий ту же роль, что и веб-браузер. Всякий раз, когда оператор выбирает определенную информацию, которую он или она хочет увидеть, клиентская программа использует SNMP для запроса этой информации с соответствующего узла. (SNMP работает поверх UDP.) Сервер SNMP, работающий на этом узле, получает запрос, находит соответствующую информацию и возвращает ее клиентской программе, которая затем отображает ее пользователю.

Есть только одно усложнение в этом, казалось бы, простом сценарии: как именно клиент указывает, какую информацию он хочет получить, и, соответственно, как сервер знает, какую переменную в памяти прочитать для удовлетворения запроса? Ответ заключается в том, что SNMP зависит от сопутствующей спецификации, называемой *базой управления информацией* (management information base, MIB). MIB определяет конкретные части информации — MIB-переменные, — которые можно получить с сетевого узла.

Текущая версия MIB, называемая MIB-II, организует переменные в различные группы. Вы узнаете, что большинство групп соответствуют одному из протоколов, описанных

в этой книге, и почти все переменные, определенные для каждой группы, должны быть вам знакомы. Например:

- Система — общие параметры системы (узла) в целом, включая расположение узла, время его работы и имя системы.
- Интерфейсы — информация обо всех сетевых интерфейсах (адаптерах), подключенных к этому узлу, например, физический адрес каждого интерфейса и количество пакетов, отправленных и полученных на каждом интерфейсе.
- Преобразование адресов — информация о протоколе разрешения адресов (Address Resolution Protocol), и, в частности, содержимое его таблицы преобразования адресов.
- IP — переменные, связанные с IP, включая таблицу маршрутизации, количество дейтаграмм, успешно пересланных, и статистику о повторной сборке дейтаграмм; включает количество случаев, когда IP отклоняет дейтаграмму по той или иной причине.
- TCP — информация о TCP-соединениях, такая как количество пассивных и активных открытий, количество сбросов, количество тайм-аутов, настройки тайм-аутов по умолчанию и так далее; информация по каждому соединению сохраняется только до тех пор, пока существует соединение.
- UDP — информация о UDP-трафике, включая общее количество отправленных и полученных UDP-дейтаграмм.

Также существуют группы для протокола управления сообщениями в Интернете (ICMP) и самого SNMP.

Возвращаясь к вопросу о том, как клиент указывает, какую именно информацию он хочет получить с узла, наличие списка переменных MIB — это только половина дела. Остаются две проблемы. Во-первых, нам нужен точный синтаксис, чтобы клиент мог указать, какие из переменных MIB он хочет получить. Во-вторых, нам нужна точная форма представления значений, возвращаемых сервером. Обе проблемы решаются с помощью языка абстрактных синтаксических обозначений (ASN.1).

Рассмотрим сначала вторую проблему. Как мы уже видели в предыдущем разделе, ASN.1/Правила базовой кодировки (BER) определяют представление для различных типов данных, таких как целые числа. MIB определяет тип каждой переменной, а затем использует ASN.1/BER для кодировки значения, содержащегося в этой переменной, при передаче по сети.

Что касается первой проблемы, ASN.1 также определяет схему идентификации объектов. MIB использует эту систему идентификации для назначения глобально уникального идентификатора каждой переменной MIB. Эти идентификаторы указываются в нотации «точка», в отличие от доменных имен. Например, 1.3.6.1.2.1.4.3 — это уникальный ASN.1 идентификатор для IP-связанной переменной MIB `ipInReceives`; эта переменная подсчитывает количество IP-дейтаграмм, полученных данным узлом. В этом примере префикс 1.3.6.1.2.1 идентифицирует базу данных MIB (помните, что идентификаторы объектов ASN.1 предназначены для всех возможных объектов в мире), 4 соответствует группе IP, а конечная 3 обозначает третью переменную в этой группе.

Таким образом, сетевое управление работает следующим образом. Клиент SNMP помещает ASN.1 идентификатор переменной MIB, которую он хочет получить, в запрос и отправляет этот запрос на сервер. Сервер затем сопоставляет этот идентификатор с локальной переменной (т.е. с памятью, где хранится значение этой переменной), извлекает текущее значение этой переменной и использует ASN.1/BER для кодировки значения, которое он отправляет обратно клиенту.

Есть еще одна деталь. Многие переменные MIB являются либо таблицами, либо структурами. Такие составные переменные объясняют причину операции SNMP GET-NEXT. Эта операция, применяемая к определенному идентификатору переменной, возвращает зна-

чение этой переменной плюс идентификатор следующей переменной, например, следующего элемента в таблице или следующего поля в структуре. Это помогает клиенту «пройтись» по элементам таблицы или структуры.

OpenConfig

SNMP по-прежнему широко используется и исторически был «тем самым» протоколом управления для коммутаторов и маршрутизаторов, но в последнее время все больше внимания уделяется более гибким и мощным способам управления сетями. Пока нет полного согласия по отраслевому стандарту, но консенсус по общему подходу начинает формироваться. Мы опишем один пример, называемый *OpenConfig*, который получает много внимания и иллюстрирует многие ключевые идеи, которые сейчас рассматриваются.

Общая стратегия заключается в автоматизации сетевого управления настолько, насколько это возможно, с целью исключения человека, подверженного ошибкам, из процесса. Это иногда называется управлением с нулевым вмешательством (*zero-touch management*) и подразумевает две вещи. Во-первых, если исторически операторы использовали такие инструменты, как SNMP, для мониторинга сети, но должны были входить на любое неисправное сетевое устройство и использовать интерфейс командной строки (CLI) для исправления проблемы, то управление с нулевым вмешательством подразумевает, что нам также нужно *программно настраивать* сеть. Другими словами, сетевое управление равносильно чтению информации о состоянии и записи конфигурационной информации. Цель состоит в том, чтобы построить замкнутый контур управления, хотя всегда будут сценарии, в которых оператор должен быть уведомлен о необходимости ручного вмешательства.

Во-вторых, если исторически оператор должен был настраивать каждое сетевое устройство отдельно, то все устройства должны быть настроены согласованным образом, чтобы они могли корректно функционировать как сеть. В результате управление с нулевым вмешательством также подразумевает, что оператор должен иметь возможность объявить свои *намерения* на уровне всей сети, причем инструмент управления должен быть достаточно умным, чтобы выдавать необходимые настройки для каждого устройства в глобально согласованном виде.

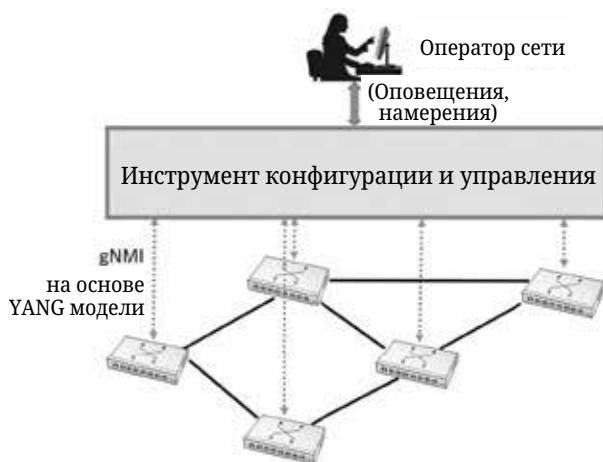


Рисунок 9.19. Оператор управляет сетью с помощью инструмента конфигурирования и управления, который, в свою очередь, программно взаимодействует с базовыми сетевыми устройствами (например, используя gNMI в качестве транспортного протокола и YANG для определения схемы обмена данными).

Рис. 9.19 дает высокоуровневое изображение этого идеализированного подхода к управлению сетью. Мы говорим «идеализированного», потому что достижение истинного управления с нулевым вмешательством пока больше стремление, чем реальность. Но прогресс идет. Например, новые инструменты управления начинают использовать стандартные протоколы, такие как HTTP, для мониторинга и настройки сетевых устройств. Это положительный шаг, потому что он позволяет нам выйти из дела создания очередного протокола «запрос/ответ» и сосредоточиться на создании более умных инструментов управления, возможно, с использованием алгоритмов машинного обучения для определения наличия проблем.

Так же, как HTTP начинает заменять SNMP в качестве протокола для общения с сетевыми устройствами, существует параллельное усилие по замене MIB новым стандартом для информации о состоянии, которую могут сообщать различные типы устройств, а также информации о конфигурации, на которую эти же устройства могут реагировать. Согласование одного стандарта для конфигурации по своей сути является сложной задачей, потому что каждый поставщик утверждает, что его устройство уникально и не похоже на устройства конкурентов. (Это значит, что вызов не совсем технический.)

Общий подход заключается в том, чтобы позволить каждому производителю устройств публиковать *модель данных*, которая определяет параметры конфигурации (и доступные данные для мониторинга) для своего продукта, и ограничить стандартизацию языком моделирования. Основным кандидатом является YANG, который расшифровывается как *Yet Another Next Generation* (еще одно следующее поколение), имя, выбранное для того, чтобы пошутить над тем, как часто требуется переделка. YANG можно рассматривать как ограниченную версию XSD, который, как вы помните, является языком для определения схемы (модели) для XML. То есть YANG определяет структуру данных. Но, в отличие от XSD, YANG не привязан к XML. Он может использоваться в сочетании с различными форматами сообщений, передаваемыми по сети, включая XML, но также ProtoBufs и JSON.

Важно, что переход в этом направлении заключается в том, что модель данных, определяющая семантику переменных, доступных для чтения и записи в программной форме (т.е. это не просто текст в спецификации стандарта). Это не свободный выбор для каждого поставщика, поскольку операторы сети, которые покупают сетевое оборудование, имеют сильный стимул к приведению моделей для аналогичных устройств к единому стандарту. YANG делает процесс создания, использования и модификации моделей более программируемым и, следовательно, адаптируемым к этому процессу.

Здесь и вступает в дело OpenConfig. Он использует YANG в качестве языка моделирования, но также установил процесс для направления отрасли к общим моделям. OpenConfig официально агностичен к механизму RPC, используемому для общения с сетевыми устройствами, но один из подходов, который он активно применяет, называется gNMI (*gRPC Network Management Interface*). Как вы могли догадаться по его названию, gNMI использует gRPC, который, как вы помните, работает поверх HTTP. Это означает, что gNMI также использует ProtoBufs для спецификации данных, которые фактически передаются через HTTP-соединение. Таким образом, как показано на рис. 9.19, gNMI предназначен для использования в качестве стандартного интерфейса управления сетевыми устройствами. Не стандартизированы лишь возможности инструмента управления для автоматизации или точная форма интерфейса, обращенного к оператору. Как и любое приложение, которое стремится удовлетворить потребности и поддерживать больше функций, чем альтернативы, здесь остается много пространства для инноваций в инструментах управления сетью.

Для полноты отметим, что NETCONF — это еще один из пост-SNMP протоколов для передачи информации о конфигурации сетевым устройствам. OpenConfig работает с NETCONF, но наш прогноз указывает на gNMI как на будущее.

В заключение подчеркнем, что происходит кардинальное изменение. В то время как заголовок этой секции указывает на SNMP и OpenConfig как на эквиваленты, точнее будет сказать, что каждый из них представляет разные подходы, но эти подходы существенно отличаются. С одной стороны, SNMP — это просто транспортный протокол, аналогичный gNMI в мире OpenConfig. Исторически он позволял мониторить устройства, но практически не имел ничего общего с их настройкой. (Последнее исторически требовало ручного вмешательства.) С другой стороны, OpenConfig — это в первую очередь усилие по определению общего набора моделей данных для сетевых устройств, аналогично роли MIB в мире SNMP, но OpenConfig (1) основан на моделях с использованием YANG и (2) равно ориентирован как на мониторинг, так и на конфигурацию.

Глава 9.4. Накладные (оверлейные) сети

С момента своего создания Интернет придерживался четкой модели, в которой маршрутизаторы внутри сети отвечают за пересылку пакетов от источника к месту назначения, а прикладные программы работают на хостах, подключенных по краям сети. Парадигма «клиент/сервер», иллюстрированная приложениями, обсуждаемыми в первых двух главах этого раздела, безусловно, соответствует этой модели.

Однако за последние несколько лет различие между *пересылкой пакетов* и *обработкой приложений* стало менее очевидным. Новые приложения распределяются по всему Интернету, и в ряде случаев эти приложения принимают собственные решения о пересылке. Эти новые гибридные приложения иногда могут быть реализованы путем расширения традиционных маршрутизаторов и коммутаторов для поддержки некоторого количества специфичной для приложений обработки. Например, так называемые *коммутаторы уровня 7* стоят перед кластерами серверов и пересылают HTTP-запросы на конкретный сервер в зависимости от запрашиваемого URL. Однако *накладные сети* быстро становятся предпочтительным механизмом для введения новой функциональности в Интернет.

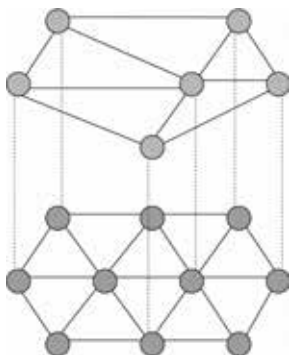


Рисунок 9.20. Оверлейная сеть, расположенная поверх физической сети.

Вы можете рассматривать накладную сеть как логическую сеть, реализованную поверх какой-то основной сети. По этому определению Интернет начинался как накладная сеть поверх линий старой телефонной сети. Рис. 9.20 изображает накладную сеть, реализованную поверх основной сети. Каждый узел накладной сети также существует в основной сети; он обрабатывает и пересылает пакеты специфическим для приложения образом. Связи, соединяющие узлы накладной сети, реализуются как туннели через основную сеть. На одной и той же основной сети могут существовать несколько накладных сетей, каждая из которых реализует свое собственное поведение, специфичное для приложения, и накладные сети могут быть вложенными, одна поверх другой. К слову, все приме-

ры накладных сетей, обсуждаемые в этой главе, рассматривают современный Интернет как основную сеть.

Мы уже видели примеры туннелирования, например, для реализации виртуальных частных сетей (VPN). Краткое напоминание: узлы на обоих концах туннеля рассматривают многопрыжковый путь между ними как единую логическую связь, узлы, через которые проходит туннель, пересылают пакеты на основе внешнего заголовка, не зная, что конечные узлы добавили внутренний заголовок.

Рис. 9.21 показывает три узла накладной сети (А, В и С), соединенных парой туннелей. В этом примере узел накладной сети В может принимать решение о пересылке пакетов от А к С на основе внутреннего заголовка (IHdr), а затем добавить внешний заголовок (OHdr), который идентифицирует С как назначение в основной сети. Узлы А, В и С могут интерпретировать как внутренний, так и внешний заголовок, тогда как промежуточные маршрутизаторы понимают только внешний заголовок. Аналогично А, В и С имеют адреса как в накладной сети, так и в основной сети, но они не обязательно одинаковы; например, их основной адрес может быть 32-битным IP-адресом, в то время как их адрес в накладной сети может быть экспериментальным 128-битным адресом. На самом деле накладная сеть не обязана использовать традиционные адреса и может маршрутизировать пакеты на основе URL, доменных имен, XML-запросов или даже содержимого пакета.

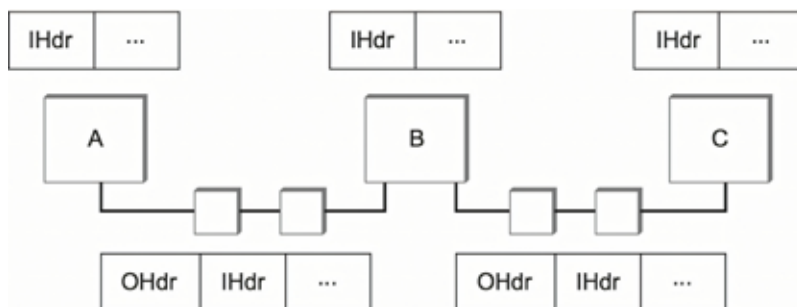


Рисунок 9.21. Оверлейные узлы туннелируют через физические узлы.

Глава 9.4.1. Маршрутизирующие накладные (оверлейные) сети

Самый простой тип накладной сети существует исключительно для поддержки альтернативной стратегии маршрутизации; на узлах накладной сети не выполняется никакая дополнительная обработка на уровне приложений. Виртуальную частную сеть (VPN) можно рассматривать как пример маршрутизирующей накладной сети, но такой, которая не столько определяет альтернативную стратегию или алгоритм, сколько использует альтернативные записи маршрутизации, обрабатываемые стандартным алгоритмом пересылки IP. В данном случае накладная сеть использует «IP-туннели», и возможность использовать эти VPN поддерживается во многих коммерческих маршрутизаторах.

Однако предположим, что вы хотите использовать алгоритм маршрутизации, который коммерческие производители маршрутизаторов не готовы включить в свои продукты. Как бы вы поступили? Вы могли бы просто запустить свой алгоритм на коллекции конечных хостов и туннелировать через интернет-маршрутизаторы. Эти хосты будут вести себя как маршрутизаторы в накладной сети: будучи хостами, они, вероятно, подключены к Интернету только одной физической связью, но как узел в накладной сети они будут подключены к нескольким соседям через туннели.

Поскольку накладные сети, почти по определению, являются способом введения новых технологий, независимых от процесса стандартизации, мы не можем указать на стан-

дартные накладные сети как на примеры. Вместо этого мы иллюстрируем общую идею маршрутизирующих накладных сетей, описывая несколько экспериментальных систем, созданных исследователями сетей.

Экспериментальные версии IP

Накладные сети идеально подходят для развертывания экспериментальных версий IP, которые в конечном итоге захватят мир. Например, IP multicast начался как расширение IP и до сих пор не включен во многие интернет-маршрутизаторы. MBone (multicast backbone) была накладной сетью, реализующей IP multicast поверх уникастовой маршрутизации, предоставляемой Интернетом. Было разработано и развернуто множество мультимедийных конференц-решений для MBone. Например, встречи IETF, которые длятся неделю и привлекают тысячи участников, много лет транслировались через MBone. (Сегодня широкая доступность коммерческих инструментов для проведения конференций заменила подход, основанный на MBone.)

Как и VPN, MBone использовала и IP-туннели, и IP-адреса, но, в отличие от VPN, MBone реализовала другой алгоритм пересылки — пересылку пакетов всем нижестоящим соседям в дереве мультикастовых путей наименьшей длины. Как и в любой накладной сети, маршрутизаторы, поддерживающие мультикаст, туннелируются через устаревшие маршрутизаторы, надеясь, что однажды устаревших маршрутизаторов вообще не останется.

6-BONE была аналогичной накладной сетью, которая использовалась для постепенно развертывания IPv6. Как и MBone, 6-BONE использовала туннели для пересылки пакетов через маршрутизаторы IPv4. Однако, в отличие от MBone, узлы 6-BONE не просто предоставляли новое толкование 32-битных адресов IPv4. Вместо этого они пересылали пакеты на основе 128-битного адресного пространства IPv6. 6-BONE также поддерживала IPv6 multicast. (Сегодня коммерческие маршрутизаторы поддерживают IPv6, но накладные сети все равно являются ценным подходом, пока новая технология оценивается и настраивается.)

Мультикаст на уровне конечных систем

Хотя IP multicast популярен среди исследователей и определенных сегментов сетевого сообщества, его развертывание в глобальном Интернете в лучшем случае было ограниченным. В ответ на это приложения, основанные на мультикасте, такие как видеоконференции, недавно обратились к альтернативной стратегии, называемой *мультикаст на уровне конечных систем*. Идея мультикаста на уровне конечных систем заключается в признании того, что IP multicast никогда не станет повсеместным, и вместо этого следует разрешить конечным хостам, участвующим в конкретном приложении на основе мультикаста, реализовывать свои собственные мультикастовые деревья.

Прежде чем описывать, как работает мультикаст на уровне конечных систем, важно сначала пояснить, что, в отличие от VPN и MBone, мультикаст на уровне конечных систем предполагает, что только интернет-хосты (в отличие от интернет-маршрутизаторов) участвуют в накладной сети. Более того, эти хосты обычно обмениваются сообщениями друг с другом через туннели UDP, а не IP-туннели, что делает их легкими для реализации в виде обычных приложений. Это позволяет рассматривать основную сеть как полностью связанный граф, поскольку каждый хост в Интернете может отправить сообщение любому другому хосту. Говоря абстрактно, мультикаст на уровне конечных систем решает следующую задачу: начиная с полностью связанного графа, представляющего Интернет, цель состоит в том, чтобы найти встроенное мультикастовое дерево, охватывающее всех участников группы.

Обратите внимание, что существует более простая версия этой задачи, обусловленная готовностью использования облачно-хостируемых виртуальных машин (VM) по всему миру. «Конечные системы», поддерживающие мультикаст, могут быть виртуальными машинами, работающими на нескольких сайтах. Поскольку эти сайты хорошо известны и относительно фиксированы, можно построить статическое мультикастовое дерево в облаке, и настоя-

щие конечные хосты просто будут подключаться к ближайшему облачному расположению. Однако для полноты картины ниже описан данный подход в его полном великолепии.

Поскольку мы считаем, что базовый Интернет полностью связан, наивным решением было бы подключить каждый источник напрямую к каждому члену группы. Другими словами, мультикаст на уровне конечных систем можно было бы реализовать, заставив каждый узел отправлять уникастные сообщения каждому члену группы. Чтобы понять проблему при этом подходе, особенно по сравнению с реализацией IP мультикаста в маршрутизаторах, рассмотрим пример топологии на рис. 9.22. На рис. 9.22 показана примерная физическая топология, где R1 и R2 — это маршрутизаторы, соединенные низкоскоростной трансконтинентальной связью; A, B, C и D — конечные хосты; задержки на каналах указаны в виде весов ребер. Предполагая, что A хочет отправить мультикастовое сообщение остальным трем хостам, рис. 9.22 показывает, как бы работала наивная уникастная передача. Это явно нежелательно, так как одно и то же сообщение должно трижды пересекать связь R1-R2, и две копии сообщения пересекают связь R1-R2. На рисунке 9.22 показано мультикастовое дерево IP, построенное протоколом маршрутизации мультикаста на основе векторного расстояния (DVMRP). Ясно, что этот подход устраняет избыточные сообщения. Однако без поддержки маршрутизаторов максимум, на что можно надеяться в случае мультикаста на уровне конечных систем, — это дерево, подобное показанному на рис. 9.22. Мультикаст на уровне конечных систем определяет архитектуру для построения этого дерева.

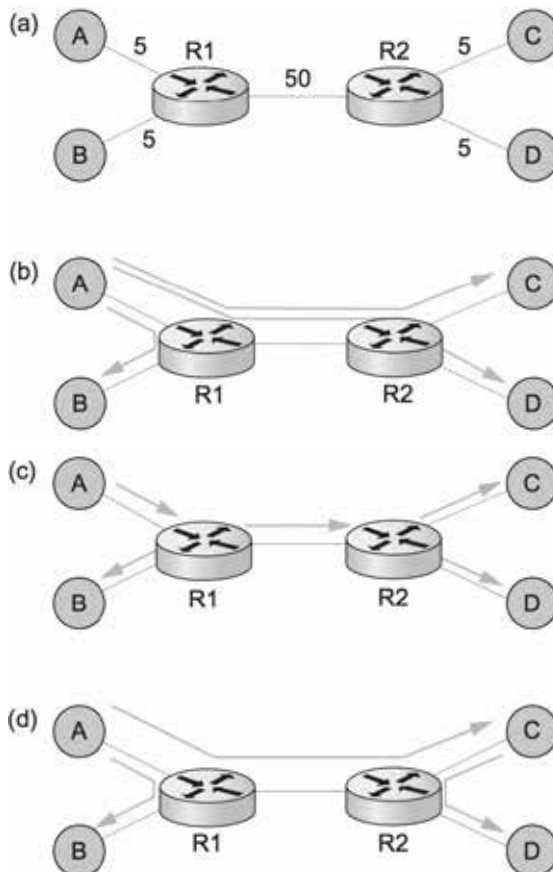


Рисунок 9.22. Альтернативные деревья мультикастовой рассылки, наложенные на физическую топологию.

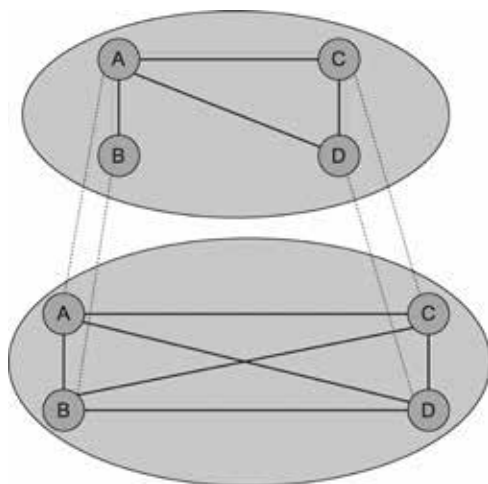


Рисунок 9.23. Мультикастовое дерево, встроенное в оверлейную сеть.

Общий подход заключается в поддержке нескольких уровней накладных сетей, каждая из которых извлекает подграф из накладной сети ниже нее, пока не будет выбран подграф, соответствующий ожиданиям приложения. В частности, для мультикаста на уровне конечных систем это происходит в два этапа: сначала мы строим простую сетевую накладную сеть поверх полностью связанного Интернета, а затем выбираем мультикастовое дерево внутри этой сетки. Идея иллюстрируется на рис. 9.23, снова предполагая четыре конечных хоста A, B, C и D. Первый шаг является критически важным: как только мы выберем подходящую сетевую накладную сеть, мы просто запускаем стандартный алгоритм маршрутизации мультикаста (например, DVMRP) поверх нее для построения мультикастового дерева. Мы также можем игнорировать проблему масштабируемости, с которой сталкивается Интернет-широкий мультикаст, поскольку промежуточную сетку можно выбрать, включив в нее только те узлы, которые хотят участвовать в определенной мультикастовой группе.

Ключом к созданию промежуточной наложенной mesh-сети является выбор топологии, которая примерно соответствует физической топологии основного Интернета, но нам нужно сделать это без информации о том, как на самом деле выглядит основной Интернет, так как мы работаем только на конечных хостах, а не на маршрутизаторах. Общая стратегия заключается в том, чтобы конечные хосты измеряли время задержки до других узлов и добавляли ссылки в mesh только в том случае, если их устраивают результаты измерений. Это работает следующим образом.

Во-первых, при условии, что mesh уже существует, каждый узел обменивается списком всех других узлов, которые он считает частью mesh, с его непосредственно подключенными соседями. Когда узел получает такой список участников от соседа, он включает эту информацию в свой список участников и пересылает полученный список своим соседям. Эта информация в конечном итоге распространяется по всей сети mesh, как в протоколе маршрутизации с вектором расстояния.

Когда хост хочет присоединиться к многоадресной наложенной сети, он должен знать IP-адрес хотя бы одного другого узла, который уже находится в наложенной сети. Затем он отправляет сообщение «join mesh» этому узлу. Это подключает новый узел к mesh через соединение с известным узлом. В общем случае новый узел может отправить сообщение о присоединении нескольким текущим узлам, таким образом подключаясь к mesh через несколько ссылок. Как только узел подключается к mesh через набор ссылок,

он периодически отправляет сообщения «keep alive» своим соседям, уведомляя их о том, что он все еще хочет быть частью группы.

Когда узел покидает группу, он отправляет сообщение «leave mesh» своим непосредственно подключенным соседям, и эта информация распространяется к другим узлам mesh через описанный выше список участников. В качестве альтернативы узел может выйти из строя или просто молча решить покинуть группу, в этом случае его соседи обнаружат, что он больше не отправляет сообщения «keep alive». Некоторые уходы узлов мало влияют на mesh, но если узел обнаружит, что mesh был разделен из-за покинувшего узла, он создает новое соединение с узлом в другой части, отправив ему сообщение «join mesh». Обратите внимание, что несколько соседей могут одновременно решить, что в mesh произошел раздел, и это приведет к добавлению нескольких пересекающихся соединений в mesh.

Как описано выше, у нас получится mesh, который является подграфом оригинального полностью соединенного Интернета, но он может иметь неоптимальную производительность, потому что (1) первоначальный выбор соседей добавляет случайные связи в топологию, (2) восстановление разделов может добавить связи, которые важны в данный момент, но не полезны в долгосрочной перспективе, (3) членство в группе может изменяться из-за динамических присоединений и выходов, и (4) условия в основной сети могут изменяться. Необходимо, чтобы система оценивала ценность каждой связи, что приведет к добавлению новых связей в mesh и удалению существующих связей со временем.

Для добавления новых связей каждый узел i периодически проверяет некоторых случайных членов j , с которыми он в настоящее время не связан в mesh, измеряет время задержки для связи (i, j) , а затем оценивает полезность добавления этой связи. Если полезность выше определенного порога, связь (i, j) добавляется в mesh. Оценка полезности добавления связи (i, j) может выглядеть следующим образом:

```
EvaluateUtility(j)
utility = 0
для каждого члена m не равного i
    CL = текущая задержка до узла m по маршруту через сетку (mesh)
    NL = новая задержка до узла m через сетку (mesh), если добавить
связь (i, j)
    if (NL < CL) then
        utility += (CL - NL)/CL
return utility
```

Решение об удалении связи аналогично, за исключением того, что каждый узел i вычисляет стоимость каждой связи с текущим соседом j следующим образом:

```
EvaluateCost(j)
Cost[i, j] = количество членов, для которых i использует j как следующий скачок
Cost[j, i] = количество членов, для которых j использует i как следующий скачок
return max(Cost[i, j], Cost[j, i])
```

Затем узел выбирает соседа с наименьшей стоимостью и разрывает связь, если стоимость падает ниже определенного порога.

Наконец, так как mesh поддерживается с использованием того, что по сути является протоколом с вектором расстояния, запустить DVMRP для поиска подходящего многоадресного дерева в mesh не составляет труда. Обратите внимание: хотя невозможно доказать, что описанный протокол приводит к оптимальной mesh-сети, что позволяет DVMRP выбрать наилучшее возможное многоадресное дерево, как симуляции, так и обширный практический опыт показывают, что он справляется с этой задачей хорошо.

Устойчивые оверлейные сети

Еще одной функцией, которую можно выполнять с помощью наложенной сети, является поиск альтернативных маршрутов для традиционных одноадресных приложений. Такие наложенные сети используют наблюдение, что неравенство треугольника не выполняется в Интернете. Рис. 9.24 иллюстрирует, что мы имеем в виду. Не редкость найти три узла в Интернете — назовем их А, В и С — такие, что задержка между А и В больше, чем сумма задержек от А до С и от С до В. То есть иногда лучше отправлять пакеты косвенно через промежуточный узел, чем отправлять их напрямую к месту назначения.

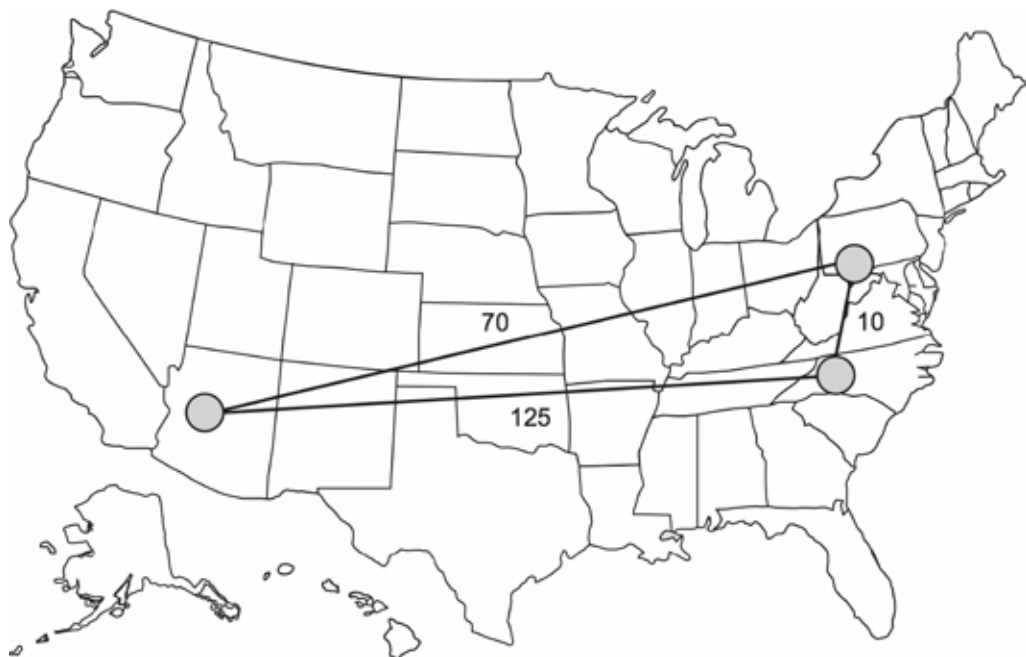


Рисунок 9.24. Неравенство треугольника не обязательно выполняется в сетях.

Как это возможно? Дело в том, что протокол пограничного шлюза (Border Gateway Protocol, BGP) никогда не обещал найти *кратчайший* маршрут между двумя узлами; он только пытается найти *какой-то* маршрут. Это объясняется тем, что маршруты BGP сильно зависят от политических вопросов, например, от того, кто кому платит за перенос трафика. Это часто происходит, например, на точках обмена трафиком между крупными магистральными интернет-провайдерами. В общем, тот факт, что неравенство треугольника не выполняется в Интернете, не должен удивлять.

Как мы можем использовать это наблюдение? Первый шаг — осознать, что существует фундаментальный компромисс между масштабируемостью и оптимальностью алгоритма маршрутизации. С одной стороны, BGP масштабируется до очень больших сетей, но часто не выбирает наилучший возможный маршрут и медленно адаптируется к сбоям в сети. С другой стороны, если бы вас волновал поиск наилучшего маршрута среди небольшого количества узлов, вы могли бы намного лучше контролировать качество каждого маршрута, что позволило бы вам выбирать наилучший возможный маршрут в любой момент времени.

Экспериментальная наложенная сеть, называемая устойчивой наложенной сетью (Resilient Overlay Network, RON), делала именно это. RON масштабировалась только

до нескольких десятков узлов, потому что использовала стратегию $N \times N$ для тщательного мониторинга (посредством активных зондов) трех аспектов качества маршрутов — задержки, доступной пропускной способности и вероятности потерь — между каждой парой узлов. Это позволяло ей выбирать оптимальный маршрут между любой парой узлов и быстро менять маршруты при изменении условий в сети. Опыт показал, что RON смогла обеспечить умеренное улучшение производительности приложений, но, что более важно, она восстанавливалась после сбоев в сети намного быстрее. Например, в течение одного 64-часового периода в 2001 году экземпляр RON, работающий на 12 узлах, обнаружил 32 сбоя длительностью более 30 минут и смог восстановиться после всех из них в среднем менее чем за 20 секунд. Этот эксперимент также показал, что пересылка данных через всего один промежуточный узел обычно достаточна для восстановления после сбоев в Интернете.

Поскольку RON не была разработана как масштабируемый подход, невозможно использовать RON для помощи случайному хосту А в общении со случайным хостом В; А и В должны заранее знать, что они собираются общаться, и затем присоединиться к одному и тому же RON. Однако RON кажется хорошей идеей в определенных условиях, например, при подключении нескольких десятков корпоративных узлов, разбросанных по всему Интернету, или для того, чтобы вы и 50 ваших друзей могли создать свою собственную частную наложенную сеть для выполнения какого-то приложения. (Сегодня эта идея реализуется под маркетинговым названием «программно-определяемая WAN» (*Software-Defined WAN, SD-WAN*).) Однако реальный вопрос заключается в том, что произойдет, если каждый начнет запускать свой собственный RON. Не будет ли избыточная нагрузка от миллионов RON, агрессивно зондирующих маршруты, перегружать сеть, и увидит ли кто-то улучшенное поведение, когда многие RON будут конкурировать за одни и те же маршруты? Эти вопросы остаются без ответа.

Основные выводы

Все эти наложенные сети иллюстрируют концепцию, которая имеет центральное значение для компьютерных сетей в целом: *виртуализацию*. То есть возможно построить виртуальную сеть из абстрактных (логических) ресурсов поверх физической сети, построенной из физических ресурсов. Более того, возможно накладывать эти виртуализированные сети друг на друга, и несколько виртуальных сетей могут сосуществовать на одном уровне. Каждая виртуальная сеть, в свою очередь, предоставляет новые возможности, которые имеют ценность для некоторого набора пользователей, приложений или сетей более высокого уровня.

Глава 9.4.2. Сети peer-to-peer (одноранговые)

Музыкальные приложения для обмена файлами, такие как Napster и KaZaA, ввели термин «peer-to-peer» в популярный лексикон. Но что именно указывает на то, что система является «peer-to-peer»? Конечно, в контексте обмена файлами MP3 это означает отсутствие необходимости скачивать музыку с центрального сайта, а вместо этого возможность доступа к музыкальным файлам непосредственно от того, у кого в Интернете есть копия, хранящаяся на его компьютере. Более широко говоря, мы можем сказать, что сеть peer-to-peer позволяет сообществу пользователей объединять свои ресурсы (контент, хранилище, сетевую пропускную способность, пропускную способность диска, CPU), предоставляя таким образом доступ к большому архиву, более крупным видео/аудиоконференциям, более сложным поискам и вычислениям и т.д., чем любой один пользователь мог бы себе позволить индивидуально.

Довольно часто при обсуждении сетей peer-to-peer упоминаются такие атрибуты, как *децентрализованная* и *самоорганизующаяся*, что означает, что отдельные узлы сами организуются в сеть без какой-либо централизованной координации. Если задуматься, такие термины можно использовать для описания самого Интернета. Однако,

например, Napster не был истинной системой peer-to-peer по этому определению, так как он зависел от центрального реестра известных файлов, и пользователи должны были искать в этом каталоге, чтобы найти, какая машина предлагает определенный файл. Только последний шаг — фактическое скачивание файла — происходил между машинами, принадлежащими двум пользователям, но это лишь немного больше, чем традиционная клиент/серверная транзакция. Единственное различие заключается в том, что сервер принадлежит кому-то из обычных пользователей, а не большой корпорации.

Итак, мы возвращаемся к изначальному вопросу: что интересного в сетях peer-to-peer? Один из ответов заключается в том, что как процесс поиска объекта интереса, так и процесс скачивания этого объекта на вашу локальную машину происходят без необходимости обращаться к централизованному органу, при этом система способна масштабироваться до миллионов узлов. Peer-to-peer система, которая может выполнить эти две задачи децентрализованным образом, оказывается наложенной сетью, где узлы — это те хосты, которые готовы делиться интересующими объектами (например, музыкой и другими файлами), а ссылки (туннели), соединяющие эти узлы, представляют собой последовательность машин, которые вам нужно посетить, чтобы найти нужный объект. Это описание станет яснее после рассмотрения двух примеров.

Gnutella

Gnutella — это ранняя сеть peer-to-peer, которая попыталась различить обмен музыкой (что, вероятно, нарушает чьи-то авторские права) и общий обмен файлами (что должно быть хорошо, так как нас учили делиться с двухлетнего возраста). Что интересно в Gnutella, так это то, что она была одной из первых таких систем, не зависевших от централизованного реестра объектов. Вместо этого участники Gnutella организуются в наложенную сеть, подобную показанной на рис. 9.25. То есть каждый узел, на котором работает программное обеспечение Gnutella (то есть реализует протокол Gnutella), знает о некотором наборе других машин, которые также работают с программным обеспечением Gnutella. Отношение «А и В знают друг друга» соответствует ребрам в этом графе. (Мы поговорим о том, как формируется этот граф, через мгновение.)

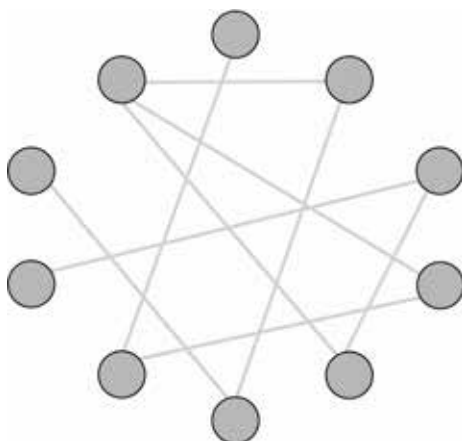


Рисунок 9.25. Пример топологии peer-to-peer сети Gnutella.

Когда пользователь на данном узле хочет найти объект, Gnutella отправляет сообщение QUERY для объекта — например, указывая имя файла — своим соседям в графе. Если у одного из соседей есть этот объект, он отвечает узлу, который отправил запрос,

сообщением QUERY RESPONSE, указывая, где можно скачать объект (например, IP-адрес и номер порта TCP). Этот узел затем может использовать сообщения GET или PUT для доступа к объекту. Если узел не может разрешить запрос, он пересылает сообщение QUERY каждому из своих соседей (кроме того, который отправил ему запрос), и процесс повторяется. Другими словами, Gnutella запрашивает всю наложенную сеть для поиска нужного объекта. Gnutella устанавливает TTL для каждого запроса, чтобы этот процесс не продолжался бесконечно.

В дополнение к TTL и строке запроса каждое сообщение QUERY содержит уникальный идентификатор запроса (QID), но не содержит информации о том, кто является исходным источником сообщения. Вместо этого каждый узел ведет запись недавно виденных сообщений QUERY: как QID, так и сосед (пир, peer), отправивший это сообщение QUERY. Он использует эту информацию двумя способами. Во-первых, если узел когда-либо получает сообщение QUERY с QID, совпадающим с уже виденным недавно, узел не пересылает это сообщение QUERY. Это помогает быстрее разрывать петли пересылки, чем могла бы сделать TTL. Во-вторых, когда узел получает ответ на запрос (QUERY RESPONSE) от соседнего узла, он знает, что нужно переслать ответ обратно соседу, который первоначально отправил сообщение QUERY. Таким образом, ответ возвращается к исходному узлу, и ни один из промежуточных узлов не знает, кто изначально хотел найти этот конкретный объект.

Возвращаясь к вопросу о том, как эволюционирует граф: узел, конечно, должен знать хотя бы об одном другом узле при присоединении к наложенной сети Gnutella. Новый узел присоединяется к наложенной сети хотя бы через одну связь. После этого данный узел узнает о других узлах в результате сообщений QUERY RESPONSE, как для объектов, которые он запрашивал, так и для ответов, которые просто проходят через него. Узел может решать, какие из узлов, обнаруженных таким образом, он хочет оставить в качестве соседей. Протокол Gnutella предоставляет сообщения PING и PONG, с помощью которых узел проверяет, существует ли данный сосед, и получает ответ от этого соседа соответственно.

Очевидно, что Gnutella в описанном виде не является особенно умным протоколом, и последующие системы пытались его улучшить. Одно из направлений для улучшений заключается в том, как распространяются запросы. Затопление (flooding) имеет хорошее свойство: оно гарантирует нахождение нужного объекта за минимальное количество переходов, но оно плохо масштабируется. Возможно пересылать запросы случайным образом или в соответствии с вероятностью успеха, основанной на прошлых результатах. Второе направление — это проактивное реплицирование объектов, так как чем больше копий данного объекта существует, тем легче найти одну из них. Альтернативно можно разработать совершенно другую стратегию, что является темой для следующего обсуждения.

Структурированные наложенные сети

В то время как системы для обмена файлами начали бороться за заполнение пустоты, оставленной Napster, исследовательское сообщество начало изучать альтернативный дизайн для peer-to-peer сетей. Мы называем эти сети *структурированными*, чтобы противопоставить их по сути случайному (неструктурированному) способу, с помощью которого развивается сеть Gnutella. Неструктурированные наложенные сети, такие как Gnutella, используют тривиальные алгоритмы построения и поддержки наложенных сетей, но максимум, что они могут предложить, это ненадежный, случайный поиск. Напротив, структурированные наложенные сети разработаны таким образом, чтобы соответствовать определенной структуре графа, что позволяет надежно и эффективно (с вероятностно ограниченной задержкой) находить объекты в обмен на дополнительную сложность при построении и поддержке наложенной сети.

Если подумать о том, что мы пытаемся сделать на высоком уровне, возникают два вопроса: (1) Как мы сопоставляем объекты с узлами, и (2) Как мы маршрутизируем запрос к узлу, который отвечает за данный объект? Начнем с первого вопроса, который имеет простую формулировку: как мы сопоставляем объект с именем x с адресом некоторого узла n , который может обслужить этот объект? В то время как традиционные peer-to-peer сети не контролируют, какой узел хостит объект x , если бы мы могли контролировать, как объекты распределяются по сети, мы могли бы лучше находить эти объекты в дальнейшем.

Известный метод для сопоставления имен и адресов заключается в использовании хеш-таблицы, так что

$$\text{hash}(x) \rightarrow n$$

означает, что объект x сначала размещается на узле n , и позже клиент, пытающийся найти x , должен будет лишь выполнить хеширование x , чтобы определить, что он находится на узле n . Подход на основе хеширования имеет приятное свойство: он склонен равномерно распределять объекты по набору узлов. Однако простые алгоритмы хеширования страдают от серьезного недостатка: сколько возможных значений n мы должны допускать? (В терминах хеширования сколько должно быть «ведер»?) Наивно можно решить, что есть, скажем, 101 возможное значение хеша, и использовать функцию хеширования по модулю; то есть

```
hash(x)
return x % 101
```

К сожалению, если узлов, готовых разместить объекты, больше чем 101, то мы не сможем воспользоваться ни одним из них. С другой стороны, если выбрать число больше самого большого возможного числа узлов, то некоторые значения x будут хешироваться в адрес для узла, которого не существует. Также есть не менее важная проблема перевода значения, возвращаемого функцией хеширования, в фактический IP-адрес.

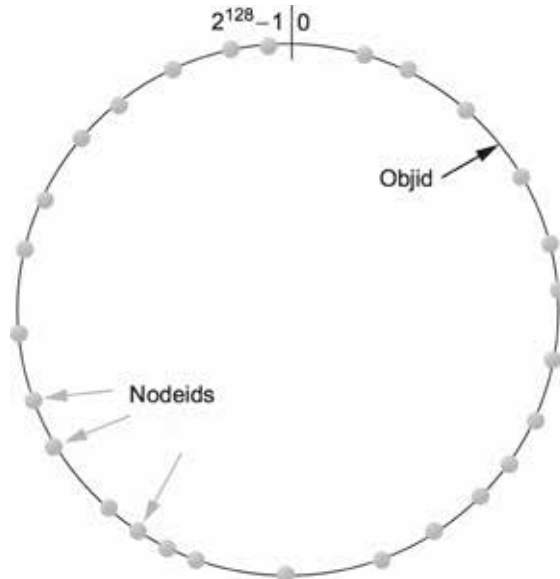


Рисунок 9.26. Как узлы, так и объекты отображаются (хешируются) на пространство идентификаторов, где объекты хранятся в ближайшем узле этого пространства.

Для решения этих проблем структурированные реер-to-реер сети используют алгоритм, известный как *последовательное хеширование*, которое равномерно распределяет множество объектов x по большому пространству идентификаторов. На рис. 9.26 показано 128-битное пространство идентификаторов в виде круга, где мы используем алгоритм для размещения как объектов

$$\text{hash}(\text{ObjectName}) \rightarrow \text{Objid}$$

так и узлов

$$\text{hash}(\text{IP Addr}) \rightarrow \text{Nodeid}$$

на этом круге. Поскольку 128-битное пространство идентификаторов огромно, маловероятно, что объект будет хешироваться в тот же идентификатор, в который хешируется IP-адрес машины. Чтобы учесть эту малую вероятность, каждый объект поддерживается на узле, идентификатор которого является *ближайшим* в этом 128-битном пространстве к идентификатору объекта. Другими словами, идея заключается в использовании высококачественной хеш-функции для отображения как узлов, так и объектов в одно и то же большое, разреженное пространство идентификаторов; затем объекты отображаются на узлы по числовой близости их соответствующих идентификаторов. Как и обычное хеширование, это равномерно распределяет объекты по узлам, но, в отличие от обычного хеширования, при добавлении или удалении узла перемещается только небольшое количество объектов.

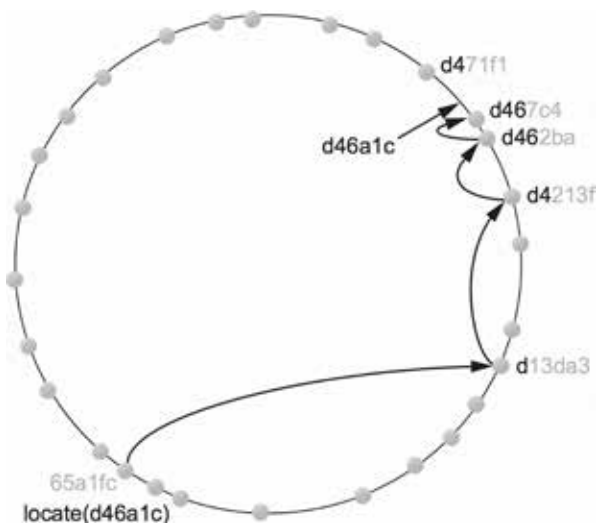


Рисунок 9.27. Объекты находятся путем маршрутизации через одноранговую оверлейную сеть.

Теперь рассмотрим второй вопрос — как пользователь, желающий получить доступ к объекту x , узнает, какой узел ближе всего к ID x в этом пространстве? Один из возможных ответов заключается в том, что каждый узел хранит полную таблицу идентификаторов узлов и их соответствующих IP-адресов, но это было бы непрактично для большой сети. Альтернатива, используемая структурированными реер-to-реер сетями, заключается в том, чтобы маршрутизировать *сообщение к этому узлу*! Другими словами, если мы создадим оверлей разумным образом — то есть выберем записи для таблицы маршрутизации узла разумно, — то мы найдем узел, просто направляя сообщение к нему. В сово-

купности этот подход иногда называют *распределенной хеш-таблицей* (DHT), поскольку концептуально хеш-таблица распределена по всем узлам сети.

Рис. 9.27 иллюстрирует, что происходит для простого 28-битного пространства идентификаторов. Чтобы обсуждение было как можно более конкретным, мы рассмотрим подход, используемый в конкретной peer-to-peer сети под названием *Pastry*. Другие системы работают аналогичным образом.

Предположим, вы находитесь на узле с ID 65a1fc (в шестнадцатеричном формате) и пытаетесь найти объект с ID d46a1c. Вы понимаете, что ваш ID не совпадает с ID объекта, но вы знаете узел, который имеет как минимум общий префикс «d». Этот узел ближе к объекту в 128-битном пространстве идентификаторов, поэтому вы пересылаете сообщение ему. (Мы не приводим формат пересылаемого сообщения, но можно представить его как «найди объект d46a1c».) Предполагая, что узел d13da3 знает другой узел, который имеет еще более длинный общий префикс с объектом, он пересылает сообщение дальше. Этот процесс перемещения ближе в ID-пространстве продолжается, пока вы не достигнете узла, который не знает более близкого узла. Этот узел, по определению, и есть тот, который хранит объект. Помните, что когда мы логически перемещаемся через «пространство идентификаторов», сообщение фактически пересылается от узла к узлу через основную сеть Интернета.

Каждый узел поддерживает как таблицу маршрутизации (подробнее ниже), так и IP-адреса небольшого набора численно больших и малых идентификаторов узлов. Это называется *листовым набором* узла. Значение листового набора заключается в том, что как только сообщение маршрутизируется к любому узлу в том же листовом наборе, что и узел, который хранит объект, этот узел может напрямую переслать сообщение конечному получателю. Другими словами, листовой набор облегчает корректную и эффективную доставку сообщения к численно ближайшему узлу, даже если существует несколько узлов, которые имеют максимальную длину префикса с ID объекта. Кроме того, листовой набор делает маршрутизацию более надежной, поскольку любой узел в листовом наборе может маршрутизировать сообщение так же хорошо, как и любой другой узел в этом наборе. Таким образом, если один узел не может продолжить маршрутизацию сообщения, один из его соседей в листовом наборе может это сделать. В общем, процедура маршрутизации определяется следующим образом:

```
def Route(D):
    if D находится в диапазоне моего leaf set:
        переслать на численно ближайший член в leaf set
    else:
        l = длина общего префикса
        d = значение l-го разряда в адресе D
        if существует RouteTab[l, d]:
            переслать на RouteTab[l, d]
        else:
            переслать известному узлу с общим префиксом не короче
            и численно ближе, чем этот узел
```

Таблица маршрутизации, обозначенная как *RouteTab*, представляет собой двумерный массив. В ней есть строка для каждой шестнадцатеричной цифры в ID (в 128-битном ID их всего 32) и столбец для каждого шестнадцатеричного значения (очевидно, их 16). Каждая запись в строке *i* имеет префикс длиной *i* с этим узлом, и в пределах этой строки запись в столбце *j* имеет шестнадцатеричное значение *j* на позиции *i+1*.

Рис. 9.28 показывает первые три строки примерной таблицы маршрутизации для узла 65a1fcx, где *x* обозначает неуказанный суффикс. Этот рисунок показывает ID-префикс, соответствующий каждой записи в таблице. Он не показывает фактическое значение, содержащееся в этой записи — IP-адрес следующего узла для маршрутизации.

Ряд 0	0	1	2	3	4	5		7	8	9	a	b	c	d	e	f
	x	x	x	x	x	x		x	x	x	x	x	x	x	x	x
Ряд 1	6	6	6	6	6		6	6	6	6	6	6	6	6	6	6
	0	1	2	3	4		6	7	8	9	a	b	c	d	e	f
	x	x	x	x	x		x	x	x	x	x	x	x	x	x	x
Ряд 2	6	6	6	6	6	6	6	6	6	6		6	6	6	6	6
	5	5	5	5	5	5	5	5	5	5		5	5	5	5	5
	0	1	2	3	4	5	6	7	8	9		b	c	d	e	f
	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x
Ряд 3	6		6	6	6	6	6	6	6	6	6	6	6	6	6	6
	5		5	5	5	5	5	5	5	5	5	5	5	5	5	5
	a		a	a	a	a	a	a	a	a	a	a	a	a	a	a
	0		2	3	4	5	6	7	8	9	a	b	c	d	e	f
	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x

Рисунок 9.28. Пример таблицы маршрутизации на узле с идентификатором 65alcx.

Добавление узла в оверлей работает так же, как маршрутизация сообщения «найти объект» к объекту. Новый узел должен знать хотя бы одного текущего члена сети. Он просит этого члена маршрутизировать сообщение «добавить узел» к узлу, численно ближайшему к ID присоединяющегося узла, как показано на рис. 9.29.

Через этот процесс маршрутизации новый узел узнает о других узлах с общим префиксом и начинает заполнять свою таблицу маршрутизации. Со временем, по мере присоединения дополнительных узлов к оверлею, существующие узлы также имеют возможность включить информацию о новом узле в свои таблицы маршрутизации. Они делают это, когда новый узел добавляет более длинный префикс, чем они в настоящее время имеют в своей таблице. Соседи в листовых наборах также обмениваются таблицами маршрутизации друг с другом, и это означает, что со временем информация о маршрутизации распространяется по всему оверлею.

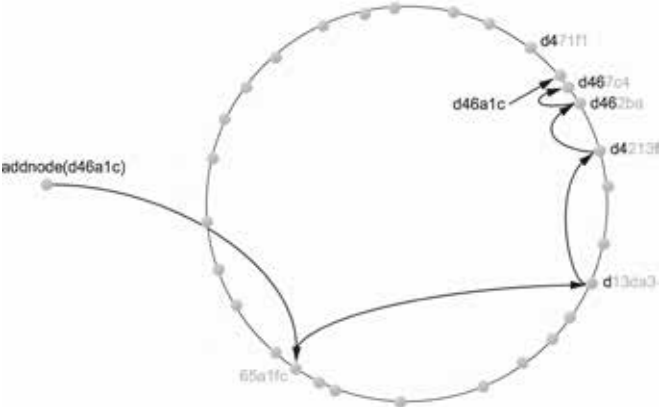


Рисунок 9.29. Добавление узла в сеть.

Читатель, возможно, заметил, что хотя структурированные оверлеи обеспечива-ют вероятностную границу на количество переходов маршрутизации, необходимых для нахождения данного объекта (количество переходов в Pastry ограничено $\log_{16} N$, где N — это количество узлов в оверлее), каждый переход может вносить значительную задержку. Это потому, что каждый промежуточный узел может находиться в случайном месте в Интернете. (В худшем случае каждый узел находится на другом континенте!) Фактически в глобальной сети оверлеев, использующей алгоритм, описанный выше, ожидаемая задержка каждого перехода равна средней задержке среди всех пар узлов в Интернете! К счастью, на практике можно добиться гораздо лучшего результата. Идея заключается в том, чтобы выбирать каждую запись в таблице маршрутизации так, чтобы она указывала на близлежащий узел в основной физической сети среди всех узлов с подходящим для записи префиксом идентификатора. Оказывается, таким образом достигаются задержки от конца до конца, которые не сильно превышают задержку между исходным и целевым узлом.

Наконец, обсуждение до этого момента сосредоточилось на общей проблеме поиска объектов в peer-to-peer (пиринговой) сети. Учитывая такую инфраструктуру маршрутизации, можно строить различные сервисы. Например, сервис обмена файлами использовал бы имена файлов в качестве имен объектов. Чтобы найти файл, они сначала хешируют его имя в соответствующий идентификатор объекта, а затем маршрутизируют сообщение «найти объект» к этому идентификатору. Система также может реплицировать каждый файл на нескольких узлах для повышения доступности. Хранение нескольких копий в листовом наборе узла, к которому обычно маршрутизируется данный файл, было бы одним из способов сделать это. Имейте в виду, что даже если эти узлы являются соседями в пространстве идентификаторов, они, скорее всего, будут физически распределены по всему Интернету. Таким образом, хотя отключение электроэнергии в целом городе может вывести из строя физически близкие копии файла в традиционной файловой системе, одна или несколько копий, вероятно, уцелеют при такой ошибке в сети peer-to-peer.

На основе распределенных хеш-таблиц можно строить не только сервисы обмена файлами. Рассмотрим, например, приложения для многозвездного вещания. Вместо того чтобы строить дерево многозвездного вещания из ячеек, можно было бы построить дерево из ребер структурированного оверлея, тем самым распределяя стоимость создания и обслуживания оверлея между несколькими приложениями и группами многозвездного вещания.

BitTorrent

BitTorrent — это протокол обмена файлами peer-to-peer, разработанный Брамом Коэном. Он основан на репликации файла, точнее, на репликации сегментов файла, которые называются *частями*. Любая конкретная часть обычно может быть загружена с нескольких узлов, даже если только один узел имеет полный файл. Основное преимущество репликации в BitTorrent заключается в избегании узкого места, связанного с наличием только одного источника файла. Это особенно полезно, если учесть, что любой компьютер имеет ограниченную скорость, с которой он может передавать файлы через свой интернет-канал, часто довольно низкую из-за асимметричного характера большинства широкополосных сетей. Преимущество BitTorrent заключается в том, что репликация является естественным побочным эффектом процесса загрузки: как только узел загружает определенную часть, он становится еще одним источником для этой части. Чем больше узлов загружает части файла, тем больше происходит репликации частей, распределяя нагрузку пропорционально, и тем больше доступная общая пропускная способность для совместного использования файла с другими. Части загружаются в случайном порядке, чтобы избежать ситуации, когда узлы оказываются без одних и тех же частей.

Каждый файл делится через свою собственную независимую сеть BitTorrent, называемую роём. (Рой потенциально может делиться набором файлов, но для простоты мы опи-

шем случай с одним файлом.) Жизненный цикл типичного роя следующий. Рой начинается как единственный узел с полной копией файла. Узел, который хочет скачать файл, присоединяется к рю, становясь его вторым членом, и начинает загружать части файла с оригинального узла. Делая это, он становится еще одним источником для загруженных им частей, даже если он еще не скачал весь файл. (Фактически узлы часто покидают рой, как только завершают загрузку, хотя их поощряют оставаться дольше.) Другие узлы присоединяются к рю и начинают загружать части с нескольких узлов, а не только с оригинального узла. См. рис. 9.30.

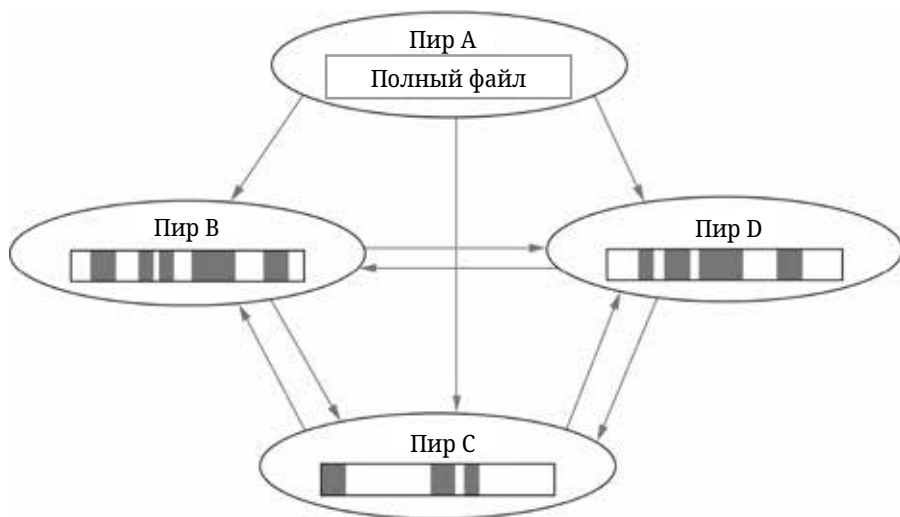


Рисунок 9. 30. Пир в рое BitTorrent скачивает с других пиров, у которых еще может не быть полного файла.

Если файл остается востребованным, и поток новых узлов заменяет те узлы, которые покидают рой, рой может оставаться активным бесконечно; если нет, он может сократиться до включения только оригинального узла до тех пор, пока новые узлы не присоединятся к рю.

Теперь, когда у нас есть обзор BitTorrent, мы можем задаться вопросом, как запросы маршрутизируются к узлам, у которых есть данная часть. Чтобы сделать запросы, потенциальный загрузчик должен сначала присоединиться к рю. Он начинает с загрузки файла, содержащего мета-информацию о файле и рое. Этот файл, который может быть легко реплицирован, обычно загружается с веб-сервера и обнаруживается, следуя ссылкам с веб-страниц. Он содержит:

- Размер целевого файла
- Размер части
- Значения SHA-1 хешей, предварительно рассчитанные для каждой части
- URL трекера роя

Трекер — это сервер, который отслеживает текущее членство роя. Позже мы увидим, что BitTorrent можно расширить, чтобы устранить эту точку централизации, обладающую потенциальным узким местом.

Желающий скачать файл присоединяется к рю, становясь пиром, отправляя сообщение трекеру с указанием своего сетевого адреса и пир-идентификатора, который он сгенерировал случайным образом. Сообщение также содержит SHA-1 хеш основной части файла, который используется как идентификатор роя.

Назовем нового пира Р. Трекер отвечает Р частичным списком пиров, указывая их идентификаторы и сетевые адреса, и Р устанавливает соединения по TCP с некоторыми из этих пиров. Заметьте, что Р напрямую подключен только к подмножеству роя, хотя он может решить связаться с дополнительными пирами или даже запросить больше пиров у трекера. Чтобы установить соединение BitTorrent с конкретным пиром после установления TCP-соединения, Р отправляет свой пир-идентификатор и идентификатор роя, и пир отвечает своим пир-идентификатором и идентификатором роя. Если идентификаторы роя не совпадают или пир-идентификатор ответа не соответствует ожиданиям Р, соединение прерывается.

Получающееся соединение BitTorrent симметрично: каждая сторона может загружать файлы с другой. Каждая сторона начинает с отправки другой битовой карты, показывающей, какие части файла у нее есть, чтобы каждый пир знал начальное состояние другого. Всякий раз, когда загрузчик (D) завершает загрузку очередной части, он отправляет сообщение, идентифицирующее эту часть, каждому из своих напрямую подключенных пиров, чтобы эти пиры могли обновить свое внутреннее представление о состоянии D. Это, наконец, ответ на вопрос о том, как запрос на загрузку части файла направляется к пиру, у которого есть эта часть, поскольку это означает, что каждый пир знает, какие напрямую подключенные пиры имеют эту часть. Если D нужна часть, которой нет ни у одного из его соединений, он может подключиться к большему количеству или другим пирам (он может получить больше пиров от трекера) или заняться другими частями в надежде, что некоторые из его соединений получат эту часть от своих соединений.

Как объекты — в данном случае части файла — сопоставляются с пир-узлами? Конечно, каждый пир в конечном итоге получает все части, так что вопрос действительно заключается в том, какие части у пира есть в данный момент до того, как у него будут все части, или, эквивалентно, в каком порядке пир загружает части. Ответ заключается в том, что они загружают части в случайном порядке, чтобы избежать строгого подмножества или надмножества частей по отношению к своим пирам.

Описанный BitTorrent использует центральный трекер, который представляет собой единую точку отказа для роя и потенциально может быть узким местом производительности. Также предоставление трекера может быть неприятностью для кого-то, кто хотел бы сделать файл доступным через BitTorrent. Новые версии BitTorrent дополнительно поддерживают «рои без трекеров», которые используют реализацию на основе DHT. Программное обеспечение клиента BitTorrent, способное работать без трекера, реализует не только пир BitTorrent, но и то, что мы будем называть *поисковиком пиров* (в терминологии BitTorrent просто «узел»), который пир использует для поиска других пиров.

Поисковики пиров формируют свою собственную накладную сеть, используя собственный протокол поверх UDP для реализации DHT. Более того, сеть поисковиков пиров включает поисковики, связанные с пирами, которые принадлежат к разным роям. Другими словами, в то время как каждый рой формирует отдельную сеть пиров BitTorrent, сеть поисковиков пиров охватывает рои.

Поисковики пиров случайным образом генерируют свои собственные идентификаторы поисковиков, которые имеют тот же размер (160 бит), что и идентификаторы роя. Каждый поисковик поддерживает скромную таблицу, содержащую главным образом поисковиков (и их связанных пиров), чьи идентификаторы близки к его собственному, а также некоторых поисковиков с более далекими идентификаторами. Следующий алгоритм обеспечивает, что поисковики, чьи идентификаторы близки к данному идентификатору роя, вероятно, знают о пирах из этого роя; алгоритм одновременно предоставляет способ их поиска. Когда поисковик F должен найти пиров из определенного роя, он отправляет запрос поисковику в своей таблице, чьи идентификаторы близки к идентификатору этого роя. Если контактированный поисковик знает о каких-либо пирах из этого роя, он отвечает их контактной информацией. В противном случае он отвечает контактной информа-

цией о поисковиках в своей таблице, которые близки к рою, так что F может итеративно опрашивать этих поисковиков.

После того как поиск исчерпан, так как нет более близких к рою поисковиков, F вставляет свою контактную информацию и информацию о своем связанном пире в поисковики, ближайшие к рою. Общий эффект заключается в том, что пиры для определенного роя заносятся в таблицы поисковиков, близких к этому рою.

Описанная выше схема предполагает, что F уже является частью сети поисковиков, что он уже знает, как связаться с некоторыми другими поисковиками. Это предположение верно для установок поисковиков, которые ранее работали, так как они должны сохранять информацию о других поисковиках, даже между запусками. Если рой использует трекер, его пиры могут сообщить своим поисковикам о других поисковиках (в обратном порядке ролей пира и поисковика), так как протокол пира BitTorrent был расширен для обмена контактной информацией поисковиков. Но как новый поисковик может обнаружить других поисковиков? Файлы для роев без трекеров включают контактную информацию для одного или нескольких поисковиков, вместо URL трекера, на случай именно такой ситуации.

Необычным аспектом BitTorrent является то, что он напрямую решает проблему спрашиваемости, или хорошего «сетевого гражданства». Протоколы часто зависят от добросовестного поведения отдельных пиров, не имея возможности его обеспечивать. Например, недобросовестный пир Ethernet мог бы получить лучшую производительность, используя более агрессивный алгоритм отката, чем экспоненциальный откат, или недобросовестный пир TCP мог бы получить лучшую производительность, не сотрудничая в избегании перегрузок TCP.

Хорошее поведение, от которого зависит BitTorrent, заключается в том, что пиры загружают части файлов другим пирам. Поскольку типичный пользователь BitTorrent просто хочет как можно быстрее скачать файл, возникает соблазн реализовать пир, который пытается загрузить все части файла, делая при этом как можно меньше загрузок другим пирам — это плохой пир. Чтобы не допустить плохого поведения, протокол BitTorrent включает механизмы, которые позволяют пирам вознаграждать или наказывать друг друга. Если пир ведет себя плохо, не загружая части другим пирам, второй пир может «задушить» плохого пира: он может решить прекратить загрузку плохому пиру, по крайней мере временно, и отправить ему сообщение об этом. Также существует тип сообщения для уведомления пира о том, что он «задушен». Механизм «душения» также используется пирами для ограничения количества активных соединений BitTorrent, чтобы поддерживать хорошую производительность TCP. Существует множество возможных алгоритмов «душения», и разработка хорошего алгоритма является искусством.

Глава 9.4.3. Сети распространения контента (Content Distribution Networks)

Мы уже видели, как HTTP, работающий поверх TCP, позволяет веб-браузерам получать страницы с веб-серверов. Однако любой, кто когда-либо ждал целую вечность, пока вернется веб-страница, знает, что система далеко не идеальна. Учитывая, что магистраль Интернета сейчас построена из каналов пропускной способностью 40 Гбит/с, не очевидно, почему это должно происходить. В общем, принято считать, что при загрузке веб-страниц в системе есть четыре потенциальных узких места:

- Первая миля. В Интернете могут быть высокоскоростные каналы, но это не поможет вам быстрее скачать веб-страницу, если вы подключены через DSL-линию со скоростью 1,5 Мбит/с или через плохо работающую беспроводную сеть.
- Последняя миля. Канал, который соединяет сервер с Интернетом, может быть перегружен слишком большим количеством запросов, даже если совокупная пропускная способность этого канала довольно высокая.

- Сам сервер. У сервера есть ограниченное количество ресурсов (ЦП, память, пропускная способность диска и т.д.), и он может быть перегружен слишком большим количеством одновременных запросов.
- Точки обмена трафиком. Небольшое количество интернет-провайдеров, которые вместе реализуют магистраль Интернета, могут иметь внутренние каналы с высокой пропускной способностью, но у них мало мотивации обеспечивать высокую пропускную способность для своих коллег. Если вы подключены к провайдеру А, а сервер подключен к провайдеру В, то запрашиваемая вами страница может быть утеряна в точке, где А и В обмениваются трафиком.

С первой проблемой никто, кроме вас, ничего не может сделать, но возможно использовать репликацию для решения оставшихся проблем. Системы, которые это делают, часто называются *сетями распространения контента* (CDN). Akamai управляет, вероятно, самой известной CDN.

Идея CDN заключается в географическом распределении коллекции *серверов-суррогатов*, которые кешируют страницы, обычно поддерживаемые в наборе *серверов бэкенда*. Таким образом, вместо того чтобы миллионы пользователей ждали вечность для связи, когда происходит крупная новость — такая ситуация известна как «вспышечная толпа», — возможно распределить эту нагрузку по многим серверам. Более того, вместо того чтобы пересекать нескольких интернет-провайдеров для доступа, если эти суррогатные серверы случайно распределены по всем магистральным провайдерам, то должно быть возможно достичь одного из них без необходимости пересекать точку обмена трафиком. Очевидно, что поддержание тысяч суррогатных серверов по всему Интернету слишком дорого для любого сайта, который хочет обеспечить лучший доступ к своим веб-страницам. Коммерческие CDN предоставляют эту услугу для многих сайтов, тем самым распределяя стоимость среди многих клиентов.

Хотя мы называем их суррогатными серверами, на самом деле их также можно рассматривать как кеши. Если у них нет страницы, запрошенной клиентом, они запрашивают ее у бэкенд-сервера. На практике же бэкенд-серверы проактивно реплицируют свои данные по суррогатам, а не ждут, пока суррогаты запросят их по требованию. Также только статические страницы, в отличие от динамического контента, распространяются по суррогатам. Клиенты должны обращаться к бэкенд-серверу за любым контентом, который либо часто меняется (например, спортивные результаты и котировки акций), либо создается в результате некоторого вычисления (например, запрос к базе данных).

Наличие большого набора географически распределенных серверов не полностью решает проблему. Для завершения картины CDN также необходимо обеспечить набор *перенаправителей*, которые перенаправляют запросы клиентов на наиболее подходящий сервер, как показано на рис. 9.31. Основная цель перенаправителей — выбрать сервер для каждого запроса, который обеспечит наилучшее *время отклика* для клиента. Второстепенная цель — чтобы система в целом обрабатывала столько запросов в секунду, сколько позволяет поддерживаемое оборудование (сетевые каналы и веб-серверы). Среднее количество запросов, которое может быть удовлетворено за определенный период времени, известное как *пропускная способность* системы, является главным вопросом, когда система находится под высокой нагрузкой, например, когда флеш-мобы обращаются к небольшому набору страниц или атакующий в рамках распределенной атаки отказа в обслуживании DDoS нацелен на определенный сайт, как это произошло с CNN, Yahoo и несколькими другими популярными сайтами в феврале 2000 года.

CDN используют несколько факторов для принятия решения о том, как распределить запросы клиентов. Например, чтобы минимизировать время отклика, перенаправитель может выбрать сервер на основе его *сетевой близости*. В отличие от этого, чтобы улучшить общую пропускную способность системы, желательно равномерно *распределить нагрузку* по набору серверов. И пропускная способность, и время отклика улучшаются,

если механизм распределения учитывает *локальность*; то есть он выбирает сервер, который, вероятно, уже имеет запрашиваемую страницу в своем кеше. Точная комбинация факторов, которую следует использовать в CDN, является предметом дискуссий. В следующей подглаве рассматриваются некоторые из возможных вариантов.

Механизмы

Как было описано выше, перенаправитель — это всего лишь абстрактная функция, хотя «перенаправление» звучит как что-то, что может делать маршрутизатор, так как он логически перенаправляет запросное сообщение так же, как маршрутизатор перенаправляет пакеты. На самом деле существует несколько механизмов, которые можно использовать для реализации перенаправления. Обратите внимание: для целей данного обсуждения мы предполагаем, что каждый перенаправитель знает адрес каждого доступного сервера. (С этого момента мы опустим квалификатор «суррогат» и будем говорить просто в терминах набора серверов.) На практике осуществляется некоторая форма внеполосной связи для поддержания этой информации в актуальном состоянии по мере добавления и удаления серверов.

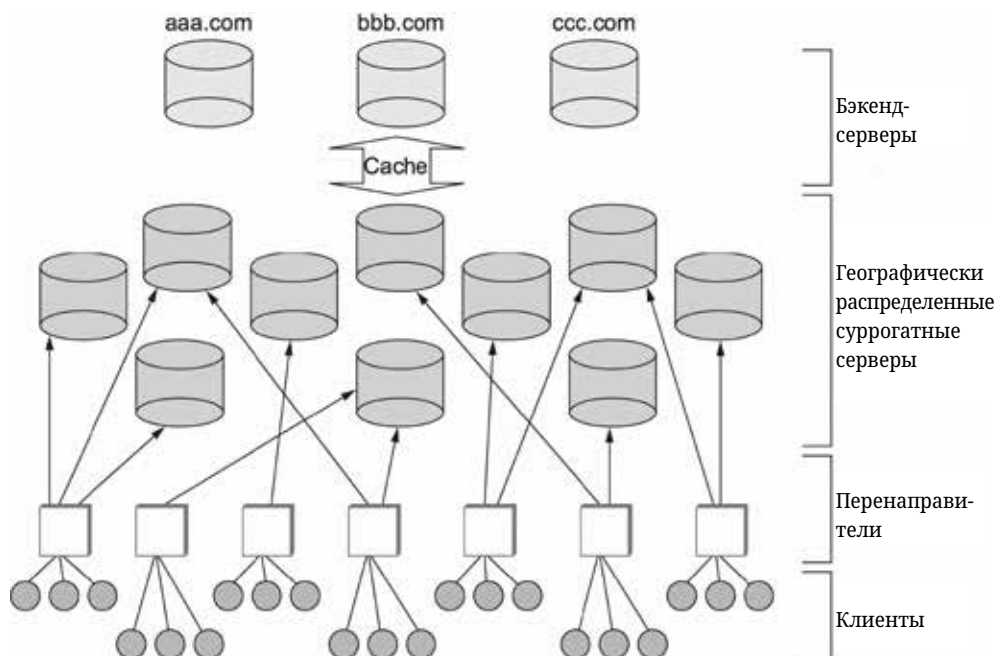


Рисунок 9.31. Компоненты сети распространения контента (CDN).

Во-первых, перенаправление можно реализовать, дополнив DNS, чтобы возвращать клиентам различные адреса серверов. Например, когда клиент запрашивает разрешение имени, DNS-сервер может вернуть IP-адрес сервера, размещающего веб-страницы CNN, который, как известно, имеет наименьшую нагрузку. В качестве альтернативы для данного набора серверов он может просто возвращать адреса по круговой схеме. Обратите внимание, что гранулярность перенаправления на основе DNS обычно находится на уровне сайта вместо конкретного URL. Однако при возврате встроенной ссылки сервер может переписать URL, тем самым фактически указывая клиенту на наиболее подходящий сервер для этого конкретного объекта.

Коммерческие сети доставки контента (CDN) фактически используют комбинацию переписывания URL и перенаправления на основе DNS. По причинам масштабируемо-

сти высокоуровневый DNS-сервер сначала указывает на региональный DNS-сервер, который отвечает фактическим адресом сервера. Чтобы быстро реагировать на изменения, DNS-серверы изменяют время жизни (TTL) возвращаемых ресурсных записей на очень короткий период, например, 20 секунд. Это необходимо, чтобы клиенты не кешировали результаты и, таким образом, не возвращались к DNS-серверу за самым последним сопоставлением URL-сервер.

Другой возможностью является использование функции перенаправления HTTP: клиент отправляет запрос на сервер, который отвечает новым (лучшим) сервером, с которым клиент должен связаться для получения страницы. К сожалению, перенаправление на сервере требует дополнительного времени на пересылку по Интернету, и что еще хуже, серверы могут быть перегружены задачей перенаправления. Вместо этого если есть узел, близкий к клиенту (например, локальный веб-прокси), который знает о доступных серверах, он может перехватить запрос и указать клиенту запрашивать страницу с соответствующего сервера. В этом случае перенаправляющий узел должен находиться в узловой точке, через которую проходят все запросы, или клиент должен сотрудничать, явно обращаясь к прокси (как в случае с классическим, а не прозрачным прокси).

На этом этапе вы можете задаться вопросом, какое отношение CDN имеет к оверлейным сетям, и хотя рассматривать CDN как оверлейную сеть несколько натянуто, у них есть одно важное общее качество. Как и узел оверлейной сети, прокси-перенаправитель принимает решение на уровне приложения. Вместо того чтобы пересылать пакет на основе адреса и своих знаний о топологии сети, он пересылает HTTP-запросы на основе URL и своих знаний о расположении и нагрузке набора серверов. Современная архитектура Интернета не поддерживает перенаправление напрямую — где «напрямую» означает, что клиент отправляет HTTP-запрос на переправитель, который пересылает его к месту назначения, — поэтому перенаправление обычно реализуется косвенно, когда перенаправитель возвращает соответствующий адрес места назначения, а клиент связывается с сервером сам.

Политики

Теперь рассмотрим некоторые примерные политики, которые могут использовать переправители для пересылки запросов. На самом деле мы уже предложили одну простую политику — *циклический перебор* (round-robin). Похожая схема — просто случайный выбор одного из доступных серверов. Оба этих подхода хорошо распределяют нагрузку по CDN, но не особо снижают время отклика, воспринимаемое клиентом.

Очевидно, что ни одна из этих схем не учитывает сетевую близость, но, что также важно, они игнорируют локальность. То есть запросы на один и тот же URL перенаправляются на разные серверы, что делает менее вероятным обслуживание страницы из кеша в памяти выбранного сервера. Это вынуждает сервер получать страницу с диска или даже с бэкэнд-сервера. Как может распределенный набор перенаправителей заставить запросы на одну и ту же страницу направляться на один и тот же сервер (или небольшой набор серверов) без глобальной координации? Ответ удивительно прост: все перенаправители используют некоторую форму хеширования для детерминированного сопоставления URL с небольшим диапазоном значений. Основное преимущество этого подхода в том, что для достижения согласованной работы не требуется связь между перенаправителями; независимо от того, какой перенаправитель получает URL, процесс хеширования выдает один и тот же результат.

Так что же делает хорошую схему хеширования? Классическая схема хеширования по модулю, которая хеширует каждый URL по модулю числа серверов, не подходит для этой среды. Это связано с тем, что при изменении числа серверов расчет по модулю приведет к тому, что уменьшающаяся доля страниц будет сохранять свои серверные назначения. Хотя мы не ожидаем частых изменений в наборе серверов, факт того, что добавление новых серверов в набор вызовет массовое переназначение, нежелателен.

Альтернативой является использование того же алгоритма консистентного хеширования, обсуждаемого в предыдущей главе. В частности, каждый перенаправитель сначала хеширует каждый сервер на единичный круг. Затем для каждого поступающего URL перенаправитель также хеширует URL в значение на единичном круге, и URL назначается серверу, который находится ближе всего на круге к его хеш-значению. Если узел выходит из строя в этой схеме, его нагрузка переходит к его соседям (на единичном круге), так что добавление или удаление сервера вызывает только локальные изменения в назначении запросов. Обратите внимание, что, в отличие от случая одноранговых (peer-to-peer) сетей, где сообщение маршрутизируется от одного узла к другому для нахождения сервера, ID которого ближе всего к объектам, каждый перенаправитель знает, как набор серверов отображается на единичный круг, поэтому они могут независимо выбирать «ближайший» сервер.

Эту стратегию можно легко расширить, чтобы учитывать нагрузку на сервер. Предположим, что перенаправитель знает текущую нагрузку каждого из доступных серверов. Эта информация может быть не совсем актуальной, но можно представить, что перенаправитель просто подсчитывает, сколько раз он перенаправил запрос на каждый сервер за последние несколько секунд, и использует это число как оценку текущей нагрузки на этот сервер. Получив URL, перенаправитель хеширует URL плюс каждый из доступных серверов и сортирует полученные значения. Этот отсортированный список фактически определяет порядок, в котором перенаправитель будет рассматривать доступные серверы. Затем перенаправитель просматривает этот список, пока не найдет сервер, нагрузка на который ниже некоторого порога. Преимущество этого подхода по сравнению с обычным консистентным хешированием заключается в том, что порядок серверов разный для каждого URL, так что если один сервер выходит из строя, его нагрузка равномерно распределяется между другими машинами. Этот подход лежит в основе Протокола маршрутизации массива кешей (CARP) и показан в псевдокоде ниже.

```
SelectServer(URL, S)
    для каждого сервера s из множества серверов S
        weight[s] = hash(URL, address[s])
    sort weight
    отсортировать weight по возрастанию
        if Load(s) < threshold then
            return s
    return сервер с наивысшим weight
```

Когда нагрузка увеличивается, эта схема меняется от использования только первого сервера в отсортированном списке к распределению запросов между несколькими серверами. Некоторые страницы, которые обычно обрабатываются загруженными серверами, также начнут обрабатываться менее загруженными серверами. Поскольку этот процесс основан на совокупной нагрузке сервера, а не на популярности отдельных страниц, серверы, хранящие популярные страницы, могут найти больше серверов, разделяющих их нагрузку, чем серверы, хранящие в совокупности менее популярные страницы. В процессе некоторые непопулярные страницы будут реплицироваться в системе просто потому, что они в основном хранятся на загруженных серверах. В то же время если некоторые страницы станут чрезвычайно популярными, возможно, что все серверы в системе будут нести ответственность за их обслуживание.

Наконец, можно ввести сетевую близость в уравнение как минимум двумя различными способами. Первый способ — это размыть границу между нагрузкой сервера и сетевой близостью, отслеживая, сколько времени сервер тратит на ответ на запросы, и использовать это измерение в качестве параметра «нагрузки сервера» в предыдущем алгоритме. Эта стратегия предпочитает загруженные серверы по сравнению с удаленными или силь-

но загруженными серверами. Второй подход заключается в том, чтобы учитывать близость на более раннем этапе, ограничивая набор серверов, рассматриваемых в вышеуказанных алгоритмах (S), только теми, которые находятся поблизости. Более сложной задачей является определение того, какие из потенциального множества серверов достаточно близки. Один из подходов — выбирать только те серверы, которые доступны в той же ISP, что и клиент. Более сложный подход — взглянуть на карту автономных систем, созданную BGP, и выбирать только те серверы, которые находятся в пределах некоторого количества переходов от клиента, в качестве кандидатов. Нахождение правильного баланса между сетевой близостью и локальностью кеша сервера является предметом текущих исследований.

Перспектива: Облако — новый Интернет

Как мы видели в конце главы 9.1, произошла миграция традиционных интернет-приложений, таких как электронная почта и веб-серверы, с машин, работающих на месте, на виртуальные машины, работающие в облачных инфраструктурах. Это соответствует смене терминологии (от «веб-сервисов» к «облачным сервисам») и многим используемым технологиям (от виртуальных машин к облачным нативным микросервисам). Но влияние облака на то, как сегодня реализуются сетевые приложения, еще больше, чем предполагает эта миграция. В конечном итоге, возможно, самое большое влияние окажет сочетание облачных инфраструктур и оверлейных сетей (подобных тем, что описаны в главе 9.4).

Самое важное, что нужно для эффективного приложения, основанного на оверлейной сети, — это широкое распространение, то есть много точек присутствия по всему миру. IP-маршрутизаторы широко развернуты, поэтому, если у вас есть разрешение использовать их в качестве базовых узлов в вашей оверлейной сети, то у вас все в порядке. Но этого не произойдет, так как нет ни одного оператора сети или администратора предприятия, который позволил бы случайным людям загружать оверлейное программное обеспечение на свои маршрутизаторы.

Ваш следующий выбор может заключаться в краудсорсинге хостинг-сайтов для вашего оверлейного программного обеспечения. Доброта незнакомцев работает, если все вы преследуете общую цель, например, скачивание бесплатной музыки, но новому оверлейному приложению сложно получить вирусную популярность, и даже если это произойдет, обеспечение достаточной емкости в любое время для передачи всего трафика, который генерирует ваше приложение, часто составляет проблему. Это иногда работает для бесплатных сервисов, но не для любого приложения, которое вы надеетесь монетизировать.

Если бы только существовал способ платить кому-то за право загружать и запускать ваше программное обеспечение на серверах, распределенных по всему миру! Конечно, это именно то, что предоставляют облачные инфраструктуры, такие как Amazon AWS, Microsoft Azure и Google Cloud Platform. Для многих облако предлагает, казалось бы, неограниченное количество серверов, но на самом деле не менее важно, если не более важно, где эти серверы расположены. Как мы обсуждали в конце раздела 4, они широко распределены по более чем 150 хорошо связанным сайтам.

Предположим, например, что вы хотите транслировать коллекцию онлайн-видео- или аудиоканалов миллионам пользователей, или вы хотите поддерживать тысячи видеоконференций, каждая из которых соединяет дюжину широко распределенных участников. В обоих случаях вы строите оверлейное мультикастовое дерево (одно на видеоканал в первом примере и одно на сеанс конференции во втором примере), при этом узлы оверлея в дереве расположены в какой-то комбинации из этих 150 облачных сайтов. Затем вы позволяете конечным пользователям, с их общими веб-браузерами или специализированными приложениями для смартфонов, подключаться к выбранным ими мультикастовым деревьям. Если вам нужно хранить некоторый контент видео/аудио для вос-

произведения в более позднее время (например, для поддержки временного сдвига), тогда вы можете также арендовать какое-то хранилище на некоторых или всех этих облачных сайтах, эффективно создавая свою собственную сеть распространения контента.

В долгосрочной перспективе, хотя Интернет изначально был задуман как чисто коммуникационный сервис, с произвольными вычислительными и хранилищными приложениями, которые могли свободно развиваться на его периферии, сегодня программное обеспечение для приложений практически встроено в сеть (распределено по сети), и все труднее определить, где заканчивается Интернет и начинается Облако. Это смешивание будет продолжаться и углубляться по мере того, как облако будет все ближе и ближе к краю сети (например, к тысячам сайтов, где закреплены сети доступа), и экономия от масштаба приведет к тому, что аппаратные устройства, используемые для создания Интернет/облачных сайтов, будут все больше стандартизированы.



*Справочное издание
Анықтамалық басылым*

Серия «Программирование для всех»

Ларри Петерсон, Брюс Дэви

КОМПЬЮТЕРНЫЕ СЕТИ: СИСТЕМНЫЙ ПОДХОД

6-е издание

Ответственный редактор: *А. Ходякова*

Редактор проекта: *А. Дюдина*

Оформление обложки: *А. Забора*

Подписано в печать

Формат 70х100^{1/16}. Бумага офсетная. Печать офсетная.

Гарнитура Noto Serif SemiCondensed.

Усл. печ. л. 39. Тираж 2000 экз. Заказ №

Общероссийский классификатор продукции ОК-034-2014 (КПЕС 2008);

58.11.1 — книги, брошюры печатные.

Изготовлено в 2026 г.

Произведено в Российской Федерации.

Изготовитель: ООО «Издательство АСТ».

129085, Российская Федерация, г. Москва, Звёздный бульвар, дом 21,
строение 1, комната 705, пом. I, 7 этаж.

Адрес места осуществления деятельности по изготовлению продукции:

123112, Российская Федерация, г. Москва, Пресненская набережная, д. 6, стр. 2,

Деловой комплекс «Империя», 14, 15 этаж.

Наш электронный адрес: ask@ast.ru

Наш сайт: www.ast.ru

Интернет-магазин: www.book24.ru

Өндіруші: «Издательство АСТ» ЖШҚ 129085, Ресей Федерациясы,
Мәскеу, Звёздный бульвары, 21-үй, 1-құрылыс, 705-бөлме, I үй-жай, 7-қабат.

Өнім өндіру қызметін жүзеге асыру мекенжайы: 123112, Ресей Федерациясы,

Мәскеу, Пресненская жағ., 6-үй, 2-құр., «Империя» іскерлік кешені, 14, 15-қабат.

Біздің электрондық мекенжайымыз: www.ast.ru E-mail: ask@ast.ru

Интернет-магазин: www.book24.ru Интернет-дүкен: www.book24.kz

Импортер в Республику Казахстан, дистрибьютор и представитель по приёму претензий
на продукцию в Республике Казахстан: ТОО «РДЦ-Алматы».

г. Алматы, ул. Домбровского, 3«а», литер Б, офис 1.

Қазақстан Республикасына импорттаушы дистрибьютор және Қазақстан
Республикасында өнімге шағымдар қабылдау жөніндегі өкіл: «РДЦ-Алматы» ЖШС.

Алматы қ., Домбровский көш., 3«а», Б литері, офис 1.

Тел.: 8(727) 2 51 59 90,91, факс: 8 (727) 251 59 92 ішкі 107;

E-mail: RDC-Almaty@eksmo.kz, www.book24.kz

Өндірілген жылы: 2026. Өнімнің жарамдылық мерзімі шектелмеген.

Сертификаттауға жатпайды.

Ресей Федерациясында өндірілген.

ЛАРРИ ПЕТЕРСОН
БРЮС ДЭВИ

КОМПЬЮТЕРНЫЕ СЕТИ

6-е издание



СИСТЕМНЫЙ
ПОДХОД

Пред вами новое издание классического пособия по конструированию компьютерных сетей, ставшее мировым бестселлером. Авторы подготовили для него полностью обновленный контент с расширенным охватом тем, имеющих в настоящее время первостепенное значение для сетевых специалистов и студентов. Подача материала осуществляется последовательно и системно, благодаря чему читатель получит полное представление об основных «строительных блоках» сетей, причем здесь рассматриваются как аппаратные, так и программные концепции. Кроме того, Интернет представлен в более широком контексте «облака» и формирующих его облачных технологий, а завершается изложение обобщающей заключительной главой о популярных в наши дни сетевых приложениях.

Эта книга, которую признанные эксперты считают наиболее всеобъемлющим пособием по данной теме, будет полезна студентам старших курсов, обучающимся по специальности «Компьютерные сети», отраслевым специалистам, ставящим перед собой задачу повысить уровень своих компетенций, а также всем тем, кто стремится понять принципы работы сетевых протоколов и современных компьютерных сетей и получить полное представление о технологических новациях, кардинально преобразующих наше общество.

книги для любого настроения здесь



ИЗДАТЕЛЬСКАЯ ГРУППА АСТ

www.ast.ru | www.book24.ru

 vk.com/izdatelstvoast

 ok.ru/izdatelstvoast

ISBN 978-5-17-162231-2



9 785171 622312 >