Б. Кришнамурти
Дж. Рексфорд

Web–протоколы

Теория и практика

HTTP/1.1, взаимодействие протоколов, кэширование, измерение трафика

БИНОМ

Web–протоколы
Теория и практика

Web Protocols and Practice

HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement

Balachander Krishnamurthy

Jennifer Rexford

Addison-Wesley

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Б. Кришнамурти, Дж. Рексфорд

Web–протоколы
Теория и практика

HTTP/1.1, взаимодействие протоколов, кэширование, измерение трафика

Перевод
с английского
под редакцией
А.И. Тихонова

Москва
ЗАО "Издательство **БИНОМ**"
2002

УДК 004.057.4
ББК 32.973.26-018.1
К82

Перевод с английского
Задорожного С.С., Левчука Ю.А.

Б. Кришнамурти, Дж. Рексфорд

Web-протоколы. Теория и практика. — М.: ЗАО «Издательство БИНОМ», 2002 г. — 592 с.: ил.

В книге всесторонне рассмотрены компоненты и протоколы, ответственные за передачу Web-содержания. Книга может быть полезной администраторам Web-сайтов, разработчикам, использующим Web-технологии, студентам, изучающим Web и сетевые технологии, а также студентам и специалистам по информационным технологиям. Книга фокусирует внимание на вопросах совершенствования и надежности функционирования Web. В противоположность быстро меняющимся технологиям создания и отображения Web-содержания, коммуникационные протоколы, рассматриваемые в книге, гораздо меньше подвержены изменениям. Функционирование Web и взаимодействие между различными компонентами проиллюстрировано многочисленными примерами. Книга включает развернутые примеры использования протокола HTTP, обзор принципов кэширования и передачи мультимедийных потоков в Web с учетом последних достижений, а также практические примеры использования Web-, прокси-серверов и методов измерения параметров Web-трафика.

ISBN 5-7989-0255-2 (русск.)
ISBN 0-201-71088-9 (англ.)

Authorized translation from the English language edition
© Original Copyright. AT&T Corp., 2001
© Издание на русском языке.
ЗАО «Издательство БИНОМ», 2002

Научно-техническое издание

Балачандер Кришнамурти, Дженифер Рексфорд

Web-протоколы. Теория и практика

Компьютерная верстка *С.В. Лычагина*

Подписано в печать 23.04.2002. Формат 70x100/16. Усл. печ. л. 48,1
Гарнитура Petersburg. Бумага газетная
Печать офсетная. Тираж 3000 экз. Заказ № 1062

ЗАО «Издательство БИНОМ», 2001 г.
103473, Москва, Краснопролетарская, 16
Лицензия на издательскую деятельность № 065249 от 26 июня 1997 г.

Отпечатано с готовых диапозитивов во ФГУП ИПК «Ульяновский Дом печати». 432980, г. Ульяновск, ул. Гончарова, 14

Содержание

Посвящается
ராஜம் பஞ்ஜாபகேசன்
விஜயம் சந்திரமௌலி
— Балачандер Кришнамурти

Моим родителям, Джону и Сьюзан Рексфорд,
за их любовь и поддержку
— Дженифер Рексфорд

Предисловие

Введение

Эта книга описывает технические основы Всемирной паутины (World Wide Web). Мы обсудим технологии передачи, кэширования и оценки параметров сообщений, которые переносят информационное содержимое между Web-сайтами и конечными пользователями. Сообщения передаются между клиентами, прокси-серверами и Web-серверами — тремя основными программными компонентами Web. Формат и способ передачи этих сообщений определяются коммуникационными протоколами, принципы которых описаны в созданных на протяжении ряда лет стандартах. Мероприятия по оценке эффективности и повышению производительности Web основаны на методах сбора и анализа параметров трафика. За счет перемещения Web-содержания ближе к конечным пользователям кэширование сокращает время ожидания на стороне пользователя, а также снижает нагрузку на Web-серверы и сети. Помимо доставки текста и изображений Web-трафик стал также переносить потоки аудио- и видеоинформации. Мультимедийные потоки занимают особое место в коммуникационных протоколах. Все эти вопросы, составляющие техническую основу Web, подробно обсуждаются в книге.

В книге всесторонне рассмотрены системы и протоколы, ответственные за передачу Web-содержания. Эта книга может быть полезной администраторам Web-сайтов, разработчикам, использующим Web-технологии, студентам, изучающим Web и сетевые технологии, а также сообществу исследователей и специалистов по информационным технологиям. Книга фокусирует внимание на вопросах совершенствования и надежности функционирования Web. В противоположность быстро меняющимся технологиям создания и отображения Web-содержания, коммуникационные протоколы, рассматриваемые в книге, гораздо меньше подвержены изменениям. Множество примеров, рассмотрение последних достижений и опыт практического применения иллюстрируют функционирование Web и взаимодействие между различными компонентами. Книга включает развернутые примеры использования протокола HTTP, обзор принципов кэширования и передачи мультимедийных потоков в Web с учетом последних достижений, а также практические примеры ис-

пользования Web-сервера Apache, прокси-сервера Squid и методов измерения параметров трафика. Книга будет весьма полезной для понимания технологий и функционирования Web.

Структура книги

Вводная глава книги содержит обзор эволюции World Wide Web, рассказывает об инфраструктуре именования ресурсов в Web, языке разметки Web-документов и протоколе обмена сообщениями. Далее следуют пять разделов, включающих 14 глав:

- **Программные компоненты.** Эти три главы описывают функционирование клиентов, прокси-серверов и Web-серверов, здесь также рассматриваются сопутствующие темы, такие как сценарии, поисковые серверы, cookies и аутентификация.
- **Web-протоколы.** Это основа книги. В четырех главах данной части представлены основные протоколы (Internet Protocol, Transmission Control Protocol и Domain Name System), здесь вы познакомитесь с HTTP/1.0, получите подробный обзор протокола HTTP/1.1, а также узнаете о взаимодействии между HTTP и TCP.
- **Измерение параметров трафика и рабочей нагрузки.** Две главы третьей части описывают различные методы измерения и анализа Web-трафика, а также содержат обзор ключевых параметров моделей рабочей нагрузки Web, используемых для оценки производительности Web.
- **Web-кэширование и мультимедийные потоки.** Эти две главы предоставляют современный обзор ключевых Web-приложений. Кэширование предусматривает перемещение Web-содержания ближе к пользователю с целью сокращения времени ожидания и снижения загрузки сервера и сети. Мультимедийные потоки предусматривают передачу аудио- и видео данных с воспроизведением их у конечного пользователя.
- **Перспективы.** Эти три главы знакомят с исследовательскими работами по кэшированию, измерению параметров и протоколам с целью предоставить беглый обзор развития технологий в этих трех областях, а также закрепить материал, изученный в предыдущих главах книги.

Для кого написана эта книга

Данная книга является самодостаточной и не предполагает наличие каких-либо предварительных знаний сетевых технологий и Web. Обширная библиография поможет читателям найти дополнительную информацию по тем или иным вопросам. Наибольший интерес книга будет представлять для следующих категорий читателей:

- **Студенты.** Студенты старших курсов и аспиранты могут использовать эту книгу для знакомства с протоколами и измерением параметров трафика в Web. Книга не предполагает предварительного знания студентами сетевых протоколов. Однако предполагается знание основ информатики. В книге обсуждаются текущее состояние проблемы и перспективы развития, что поможет студентам применить полученные ими знания. Ориентация на основные кон-

цепции и эволюцию протоколов гарантирует, что студенты приобретут знания, которые смогут применить при работе с Web-технологиями.

- **Разработчики программных систем.** Для разработчиков в книге имеется всестороннее описание протоколов и программных компонентов Web, включая IP, TCP и DNS, а также их взаимоотношения с Web-клиентами, прокси-серверами и Web-серверами. Кроме того, книга содержит подробное описание принципов измерения параметров Web-трафика и рабочей нагрузки, что поможет разработчикам в оценке и повышении производительности их программных продуктов.
- **Исследователи в области информационных технологий.** Исследователи могут использовать книгу как источник информации о технических аспектах Web и ее связях с Internet. Основной материал книги посвящен устоявшимся технологиям, составляющим основу Web, что дает исследователям необходимую информацию для проведения изысканий в этой области. Дополнительный материал по перспективам исследований позволяет очертить круг задач, решение которых может оказать влияние на развитие Web, а обширная библиография поможет читателям найти публикации и документацию по интересующим их вопросам.
- **Web-администраторы.** Администраторы прокси-серверов и Web-серверов могут углубить свои знания по этим программным компонентам. Книга может послужить в качестве справочного пособия по основным принципам и особенностям протоколов. Акцент на проблемах, связанных с производительностью, поможет администраторам в настройке прокси-серверов и Web-серверов. Кроме того, представлены подробные указания по настройке Web-компонентов. Разделы, посвященные измерению параметров трафика в Web и взаимодействию между HTTP и сетевыми протоколами, помогут администраторам в выявлении проблем, связанных с производительностью.

Книга может быть использована как справочное пособие, как пособие для самостоятельного изучения или как основа для одно- или двухсеместрового курса, посвященного Web и сетевым технологиям. В зависимости от уровня подготовки и интересов читатели могут выбрать тот или иной план чтения книги. Некоторые читатели могут пропустить начальные главы, содержащие элементарные сведения, тогда как другие читатели могут не изучать материал, посвященный перспективам развития технологий.

Благодарности

Эта книга обязана своим появлением на свет не только авторам. Чтобы отметить вклад других людей, мы хотим прежде всего поблагодарить их за потраченное время и усилия, проявленный интерес, усердие и желание помочь нам. Мы осознаем, что эта книга описывает технологии, к которым имеют отношение десятки, если не сотни людей. Некоторые из людей, разработавших концепции, рассматриваемые в книге, явным образом не упоминаются в благодарностях.

Мы благодарны людям, имеющим отношение к созданию, модификации протокола HTTP. Мы в долгу перед людьми, которые проявили живой интерес к нашим усилиям: Роем Филдингом, Джеффом Могулом, Дэвидом Кристолом, Коэном Холтманом, Хенриком Фристик-Нильсеном и Джимом Геттисом.

Замечания Эдама Бредли (главы 6 и 7), Энжа Фельдмана (главы 9 и 10), Салли Флойд (главы 5 и 8), Патрика МакМануса (главы 5–8), Михаила Михалкова (главы 11 и 13), Эриха Наума (глава 8), Марка Ноттингема (глава 11) и Роберта Тау (глава 4) были весьма полезными. Мы выражаем им благодарность за время, которое они потратили на то, чтобы предоставить ссылки на дополнительные материалы, а также за их содействие в обеспечении максимальной ясности изложения.

Большое число рецензентов терпеливо просматривало наши рукописи и вносили конструктивные предложения. Среди них следует отметить Патрика МакМануса, Крейга Уиллза, Анис Шайха, Дэвида Кристола, Коэна Холтмана, Карин Хогстедт, Криса Малли, Йена Купера, Полли Хуанг, Джо Периша, Хенрика Фристик-Нельсена и Дуэйна Уэсселса.

Ряд глав был прочитан нашими коллегами и друзьями, в числе которых Мартин Арлит, Винай Бадами, Стивен Белловин, Эд Блекмонд, С. Байерс, Эрик Чепг, Чак Крэпор, Лори Крэпор, Кристоф Фетцер, Р. Гопалакришнан, Джоэл Готтлиб, Тим Гриффин, Тим Корб, Джефф Корн, Джим Куроуз, Роб Лапфайер, Брайан Ф. Лаву, Карлос Мальцан, Джитендра Пэдхай, Венкат Падманабхан, Реза Реджайе, Сью Рексфорд, Юрий Резник, Луиджи Риццо, Джеис-С. Веклер и Джакобус Ван дер Мерве. Мы также хотели бы поблагодарить рецензентов издательства Addison-Wesley.

Много людей отвечали на специфические вопросы зачастую удивительно подробно и полно. Это Стивен Белловин, Тим Бернерс-Ли, Дэн Коннолли, Салли Флойд, Коэн Холтман, Брюстер Кале, Рохит Хар, Дэвид Кристол, Томас Нартен, Хенрик Фристик-Нильсен, Верн Паксон и Джим Уайтхед.

Уолтер Тичи и Майкл Филиппсен предоставили место для работы и проживания в Карлсруэ во время составления плана книги. Мы благодарим их за теплый прием и возможность воспользоваться информационными ресурсами университета Карлсруэ.

Очень ценной была поддержка и участие Хамида Ахмади, Дэвида Беланджера, Альберта Гринберга и Ким-Фонд Во — команды менеджеров из AT&T Labs-Research. Несколько наших коллег внесли свой вклад в создание условий, необходимых для успешного завершения книги.

Консультации, полученные перед и во время подготовки рукописи, способствовали улучшению изложения. Особо хотелось бы поблагодарить организаторов конференции ACM SIGCOMM за предоставленную возможность проведения консультаций.

Мы хотели бы поблагодарить сотрудников издательства Addison-Wesley за их помощь при подготовке книги к изданию. Мы выражаем признательность Лоринде Черри из AT&T за ее помощь в составлении предметного указателя и Деборе Свейн за помощь в подготовке графиков к главе 10.

Балачандр написал практически всю свою часть книги с помощью компьютера IBM Thinkpad 770ED, на котором была установлена *только* ОС Linux. Книга писалась в самых различных местах земного шара: Рио-де-Жанейро, Фоз де Игуаку, Сантьяго, Лангенстейнбах, Карлсруэ, Мюнхен, Потсдам, Париж и Амстердам. В США это Дин и Делюка (Гринвич Вилледж, Нью-Йорк), Итака, Бостон, Боулдер, различные места в Нью Джерси, Силикон Вэлли и Вашингтон (округ Колумбия). Особо благодарим людей, которые выступили в роли гостеприимных хозяев: Эдуардо Крелл и Вероника Родригес Мунос, Крис и Мэри Маллей, Роб Мэсон, В. Стрикант.

Книга была написана с использованием Emacs под управлением X Window System и напечатана с помощью LaTeX. Подготовленный к печати материал был направлен издателям в виде одного архивированного файла PostScript.

Часть I
Вводная

1

Введение

В первое десятилетие существования World Wide Web происходила корректировка способов создания и обмена информацией, а также решались вопросы глобализации бизнеса. Ванневар Буш в 1945 г. в своей провидческой статье «Как мы мыслим» предложил способ расширения человеческой памяти за счет механических средств [Bus45]. Web можно считать логическим воплощением предвидения Буша. Ключевыми этапами при этом стали разработка гипертекстовых технологий и появление Internet. Книга сосредотачивает внимание на технических аспектах Web, а не на истории возникновения и развития технологии, ее применении и воздействии на общество. В создание технологий, положенных в основу Web, были вовлечены многочисленные исследователи и разработчики из различных организаций, которые все вместе создали систему, используемую сотнями миллионов людей практически во всех странах Земли.

В этой главе рассматриваются ключевые концепции, лежащие в основе Web, глава также является вводной частью книги. Мы начнем с рассмотрения эволюции Web и познакомимся с взаимоотношениями Web с другими системами, разработанными в конце 80-х и в начале 90-х годов, и предназначенными для предоставления непосредственного доступа к информации. Далее мы кратко остановимся на трех основных семантических компонентах Web: именовании ресурсов, языке разметки документов и протоколе обмена сообщениями. Эти компоненты известны по их аббревиатурам: URI, HTML и HTTP, соответственно. Затем будет описана тематика, охватываемая книгой, с учетом ее ориентации на ключевые аспекты, связанные с доставкой Web-содержания. В завершение будет дан краткий обзор книги с разбивкой глав на пять частей: программные компоненты, протоколы, измерение и оценка рабочей нагрузки, приложения и перспективы дальнейших исследований.

1.1. Происхождение и развитие World Wide Web

Впервые предложенная Тимом Бернерс-Ли в 1989 году World Wide Web, или просто Web, представляет собой мир информации, доступный через подключенные к сети компьютеры. Интуитивный графический интерфейс дает возможность пользователям просматривать Web-страницы, щелкая мышью на гиперссылках, не задумываясь при этом о формате или местонахождении содержимого. Помимо просмотра Web-страниц, пользователи могут осуществлять поиск информации, отправлять и получать электронную почту и осуществлять различные деловые операции. Фактически Web является сетевым приложением, которое связывает пользователей с сервисами через компьютеры, разбросанные по всей планете. В этом разделе мы расскажем об эволюции Web до ее современного состояния и

сравним Web с другими конкурирующими системами, разработанными в конце 80-х и начале 90-х годов.

1.1.1. История развития Web

История Web начинается, в некотором смысле, с предложенной Ванневаром Бушем концепции Memex — вспомогательных механических средств «расширения человеческой памяти». Сутью сделанного Бушем в 1945 году призыва к ученым являлось то, что следует сделать имеющиеся знания более доступными для человечества. Он выделил различия между скоростью, с которой эти знания создаются посредством публикаций, и скоростью, с которой человечество способно ориентироваться в лабиринте информации. Говоря словами Буша:

> Memex — это устройство, в котором человек хранит все свои книги и записи, и которое выдает нужную информацию с достаточной скоростью и гибкостью. Оно является расширением и дополнением памяти человека.

Буш предвидел новые способы организации взаимосвязей:

> Появятся совершенно новые формы энциклопедий, содержащих сеть ассоциативных указателей, готовых для загрузки в Memex.

В статье Буша было предсказано всеобъемлющее индексирование текстов и мультимедийных ресурсов с возможностью их быстрого поиска. Статья оказывала влияние на ученых и разработчиков в течение почти 50 лет после своей публикации. Следующим значительным шагом на пути к Web было создание гипертекста. Термин *гипертекст* [Nel67] был введен Тедом Нельсоном в 1965 г. для обозначения информации, *не являющейся записанной последовательно*. Эта информация представляется в виде набора связанных узлов. Читатели могут изучать информацию различными способами, перемещаясь от одного узла к другому. Независимо от этого гипертекст был предложен Дугом Энгельбартом [EE68], изобретателем компьютерной мыши. Гипертекст дает возможность многочисленным авторам читать, писать и корректировать один и тот же документ.

В середине 60-х годов сеть ARPANET была задумана в качестве коммуникационной инфраструктуры для совместного доступа к суперкомпьютерам ученых в США. Министерство обороны США было заинтересовано в создании независимых от производителей оборудования методов передачи данных. В середине и конце 60-х годов стали предприниматься усилия по стандартизации сетевых коммуникационных протоколов. В 70-е годы сеть ARPANET широко использовалась в научных кругах для доступа к удаленным компьютерам, обмена сообщениями электронной почты и передачи файлов между компьютерами. В конце 70-х годов многие университеты и научные учреждения по всему миру могли связываться между собой через ARPANET. Протоколы семейства TCP/IP были стандартизованы в 1980-х годах. Реализация протоколов TCP/IP для различных платформ способствовала быстрому расширению ARPANET в 80-е годы.

Под влиянием технологии гипертекста Тим Бернерс-Ли предложил осуществить связывание информации, хранящейся на различных компьютерах в Европейской лаборатории физики высоких энергий CERN близ Женевы. Первоначально Тим Бернерс-Ли предлагал связывать документы различными способами [BL90]. Это предложение основывалось на системе под названием Enquire Within (Спроси на месте), созданной десятью годами ранее. К этому времени было создано несколько других систем для поиска и доступа к документам в Internet. К таким сис-

темам относятся FTP, Gopher, Archie и WAIS (Wide Area Information Servers). Стандартизованный в 1971 г., протокол передачи файлов FTP (File Transfer Protocol) давал возможность пользователям копировать файлы с и на FTP-серверы [PR85]. FTP-клиент давал возможность пользователю устанавливать с сервером аутентифицированное соединение с помощью указания пользовательского имени и пароля. Особая «анонимная» учетная запись позволяла любым пользователям соединяться с FTP-сервером без пароля. В 70-е и 80-е годы FTP был основным средством для распространения программного обеспечения и больших документов в Internet. В начале 90-х годов на FTP приходилось более половины трафика в Internet.

Получение файлов по FTP требует, чтобы пользователь заранее знал, с каким сервером связаться. Gopher [AAL+92, AML+93] предоставлял пользователям возможность поиска информации в сети компьютеров. Gopher-клиент имеет возможность осуществлять поиск в базах данных по всему миру на основе ключевых слов или тем. Чтобы быть глобально доступным, Gopher-сервер должен быть зарегистрирован как сервер верхнего уровня. WAIS [KM91, Adi94] дает возможность пользователям отправлять запросы к базам данных на удаленных серверах и получать от них ответы. WAIS-поиск возвращал список файлов, упорядоченных по степени их релевантности запросу. Помимо гибкости при поиске, WAIS также предоставлял доступ к файлам изображений. Крупные компании создавали предметные указатели своих информационных ресурсов, доступных через сеть. Проект WAIS начал осуществляться в 1989 г., а к 1992 г. компания WAIS Inc. стала процветающей. На этот момент это была единственная компания, на коммерческой основе занимающаяся технологией доступа к информации в Internet. В 1990 г. компанией Archie было предложено средство для поиска файлов на FTP-серверах в Internet [ED92]. Archie был создан глобальный предметный указатель FTP-серверов, который давал возможность пользователям осуществлять поиск файлов. Также было создано средство под названием Veronica, которое позволяло осуществлять поиск и доступ к ресурсам на серверах Gopher.

Все эти системы конкурировали с Web и какое-то время сосуществовали, пока не были вытеснены Web. Хотя эти системы по-прежнему изредка находят применение, большинство пользователей осуществляет доступ к ним с помощью своих Web-браузеров. У Web имеется несколько преимуществ перед перечисленными выше системами. FTP схож с Web в том, что клиенты выдают запросы на различные ресурсы. Самые первые FTP-клиенты имели относительно сложный интерфейс командной строки, тогда как Web-браузер предлагает простой графический интерфейс. В то время как FTP ограничивается передачей файлов, Web предоставляет доступ к программам и сервисам, таким как сценарии для запросов к базам данных. Хотя Gopher использовался для извлечения документов, Web дает возможность пользователям извлекать, модифицировать и хранить данные, а также активизировать выполнение программ на удаленных компьютерах. Web дает возможность пользователям осуществлять доступ к ресурсам *независимо* от их типа или формата.

Archie, Gopher и WAIS были ориентированы главным образом на текстовые документы. Применение гипертекста в Web позволяет преодолеть очевидные недостатки, присущие меню, которые используются в Gopher. Задача связывания документов распространяется на всех пользователей системы. Хотя Web может быть использована для доступа к серверам Gopher и WAIS, обратное недостижимо из-за примитивного характера протоколов, используемых в этих системах. Первый Web-браузер, разработанный в 1991 г., вышел в свет в январе 1992 г. [BL92c]. Этот

браузер включал поддержку FTP, Gopher и WAIS. Фактически успех Web во многом был обусловлен объединением возможностей конкурирующих систем. Пользователи могли иметь доступ к серверам Archie, Gopher и WAIS через Web-браузеры, а поисковые серверы Web давали возможность пользователям осуществлять поиск Web-страниц по заданным темам. Web стала современной реализацией концепции Memex. С подробной историей Web можно познакомиться непосредственно в Web [Webc].

1.1.2. Web сегодня

С точки зрения миллионов конечных пользователей Web — это просто удобный способ для доступа к информации и осуществления деловых операций с помощью интуитивно понятного графического интерфейса. В этом разделе мы расскажем, как Web в 90-е годы эволюционировала в глобальную систему с разнообразным содержанием и с широкой аудиторией пользователей. Затем мы обсудим, как за этот же период эволюционировал и Web-трафик.

ТЕНДЕНЦИИ РАЗВИТИЯ WEB

После появления первого Web-браузера и сервера в 1991 г. Web продолжала расти невиданными темпами. К началу 1993 г. Web насчитывала приблизительно 50 серверов. Марк Андреесен и Эрик Бина в декабре 1992 г. создали браузер Mosaic и выпустили первую его версию для X-Window весной 1993 г. Появление графического пользовательского интерфейса, который был способен воспроизводить текст и изображения, непосредственно привело к взрывному росту Web. К концу 1993 г. количество Web-серверов увеличилось в 10 раз, а Web-транзакции составляли до 1% трафика Internet. К концу 90-х годов Web-трафик составлял до 75% трафика Internet [CMT98], число пользователей Web возросло до нескольких сотен миллионов, а количество Web-сайтов достигло миллиона. Распределенная сущность Web и использование Web внутри крупных организаций делает точную оценку размеров Web затруднительной. Согласно консервативным оценкам, общедоступными являются свыше миллиарда унифицированных указателей ресурсов (URL), а число URL, доступных внутри частных организаций, не поддается исчислению.

Графический пользовательский интерфейс и относительная легкость публикации Web-содержания стимулировали драматические изменения в способе доступа к информации для различных категорий пользователей. Web создала новые сегменты рынка, дав возможность потребителям просматривать информацию о товарах и осуществлять покупки через сеть. Гипертекст предлагает гибкую возможность для потребителей изучать и сравнивать товары, а электронная коммерция избавила от необходимости в посреднических услугах, непосредственно связав потребителей с продавцами. В результате все больше компаний стали делать информацию о своей продукции доступной в Web. Многие вновь образованные компании использовали Web в качестве главного средства взаимодействия с потребителями. Компании также начали использовать Web для ведения дел друг с другом, минуя традиционные методы взаимодействия. Web превратилась в открытый рынок для взаимодействия между пользователями в форме аукционов, форумов и игр.

Хотя Web изначально разрабатывалась для предоставления всеобщего доступа к информации, она все шире используется внутри организаций для предоставления пользователям личных и конфиденциальных данных. Например, многие компании имеют Web-сайты, которые предоставляют сотрудникам доступ к базам данных, содержащим сведения об их зарплате и социальных выплатах. Число внутренних

Web-сайтов за последние несколько лет значительно возросло. Число Web-страниц внутренних сайтов организаций стало даже превосходить число Web-страниц, доступных для общего пользования. Рост числа внутренних Web-сайтов высветил различие между Web и Internet. Компания может соединить свои компьютеры в частную сеть без предоставления доступа к остальной части Internet. Следует также заметить, что компании, взаимодействующие через Internet, не обязательно действуют как Web-клиенты или Web-серверы. Тем не менее, Internet тесно связана с Web. Internet обеспечивает глобальную коммуникационную инфраструктуру, позволяя Web-клиентам осуществлять доступ к огромному числу Web-серверов по всему миру.

WEB-ТРАФИК

Хотя Web продолжает расширяться как количественно, так и по функциональным возможностям и степени охвата, подавляющее большинство пользовательских запросов делается к относительно небольшому числу Web-сайтов и Web-страниц. Некоторые популярные сайты привлекают чрезвычайно большое число пользователей. К этим сайтам относятся порталы, которые предоставляют пользователям самое разнообразное содержание, поисковые системы, дающие возможность пользователям находить Web-ресурсы, а также популярные сетевые магазины, дающие возможность пользователям покупать товары. Кроме того, особые события, такие как Олимпийские Игры или национальные выборы, приводят к созданию временных Web-сайтов, притягивающих большое количество пользователей на короткое время. Чтобы выдержать столь высокую нагрузку, может потребоваться продублировать Web-содержание на нескольких компьютерах, каждый из которых будет обрабатывать часть запросов. И наоборот, многие Web-сайты получают относительно мало запросов, а доступ ко многим Web-страницам осуществляется редко. Единственный компьютер легко может обрабатывать запросы для нескольких не слишком популярных Web-сайтов.

Первоначально большинство доступных в Web данных представляло собой небольшие текстовые файлы. Затем Web-страницы стали включать изображения в формате GIF и JPEG. На момент подготовки данной книги к публикации текст и изображения представляли примерно одну четвертую и одну вторую Web-запросов, соответственно. Из-за небольшого размера большинства текстовых файлов и файлов изображений, средний объем Web-ресурса составляет около 8 Кб. Несмотря на небольшой размер в среднем, многие ресурсы достаточно объемны. Web не накладывает каких-либо ограничений на размер ресурса. Текстовый файл может состоять из нескольких байтов данных или включать в себя целую книгу. Изображения варьируются от маленьких значков до больших картин. Другие ресурсы, такие как файлы компьютерной анимации или видеоклипы, могут быть очень большими. Размер ресурсов оказывает влияние на объем пространства, выделяемого для их хранения на Web-сервере, на время, необходимое браузеру для загрузки содержимого, а также на нагрузку, которую испытывает сеть при передаче данных от одного компьютера другому.

По мере развития Web все большее число Web-сайтов стало предоставлять динамически генерируемое содержание. Например, поисковая система генерирует список указателей на Web-страницы на основе ключевых слов, введенных пользователем. Содержание в этом случае является результатом выполнения запроса к базе данных. Все большее количество Web-транзакций подразумевает доступ и передачу частных, конфиденциальных данных, таких как номера кредитных карточек. Как следствие, возрастает интерес к проблемам безопасности и аутентификации при Web-транзакциях. Web все шире используется в качестве средства для

создания и передачи данных пользователями, таких как сообщений электронной почты и совместно разрабатываемых документов. Кроме того, Web-браузеры дают возможность пользователям инициировать запросы для передачи потоков аудио- и видеоинформации. Браузер вызывает отдельное приложение для воспроизведения таких данных пользователем; мультимедийный плейер способен также напрямую взаимодействовать с мультимедийным сервером в обход браузера. Инициирование таких запросов из Web-браузера обеспечивает пользователю полноценное восприятие вне зависимости от того, как передаются и отображаются данные.

1.2. Семантические компоненты Web

В Web имеется три семантических компонента: унифицированные идентификаторы ресурсов (Uniform Resource Identifiers — URI), гипертекстовый язык разметки (Hypertext Markup Language — HTML) и протокол передачи гипертекста (Hypertext Transfer Protocol — HTTP). URI представляют собой механизм именования ресурсов в Web с целью их идентификации. HTML — это стандартный язык для создания гипертекстовых документов. HTTP — это язык для взаимодействия между Web-клиентами и Web-серверами.

1.2.1. Унифицированные идентификаторы ресурсов (URI)

Доступ к ресурсам в Web требует их идентификации. Web-ресурс идентифицируется с помощью унифицированного идентификатора ресурса (URI) [BLFM98]. Не являясь какой-либо физической сущностью, URI может быть воспринят как указатель на некий черный ящик, к которому применимы *методы* запросов с целью генерирования потенциально различных ответов в разное время. Метод запроса — это простая операция, такая как выборка, изменение или удаление ресурса. URI идентифицирует ресурс *независимо* от его текущего местонахождения или содержания. На верхнем уровне URI представляет собой простую форматированную строку, такую как **http://www.foo.com/coolpic.gif**. URI обычно состоит из трех частей: протокола для взаимодействия с сервером (например, **http**), имени сервера (например, **www.foo.com**) и имени ресурса на этом сервере (например, **coolpic.gif**). Наиболее популярной формой URI является унифицированный указатель ресурса — Uniform Resource Locator (URL) [BLMM94, Fie95]. Распространенное заблуждение, что URL и URI есть одно и то же, не мешает восприятию Web обывателями. Различия между ними обсуждаются в главе 6 (раздел 6.1.1). В этой книге мы используем популярный термин URL вместо более общего термина URI за исключением тех случаев, когда разница между URI и URL существенна.

1.2.2. Гипертекстовый язык разметки (HTML)

Гипертекстовый язык разметки (Hypertext Markup Language — HTML) обеспечивает стандартное представление гипертекстовых документов в текстовом формате. HTML произошел от более общего языка разметки Standard Generalized Markup Language (SGML). HTML позволяет форматировать текст, встраивать в документ изображения, а также создавать гипертекстовые ссылки на другие документы. Синтаксис HTML достаточно прозрачен и прост в изучении. Простейший HTML-документ — это нечто большее, чем обычный текстовой документ без форматирования или ссылок на другие ресурсы. На протяжении ряда лет были разработаны про-

граммные средства для создания и редактирования HTML-файлов, что избавило большинство людей от необходимости изучать HTML. Эти программные пакеты позволяют переводить документы, созданные в других форматах, в HTML, а также вносить в них дополнения и изменения. Как результат, лишь малое число больших HTML-документов создаются вручную. HTML-файлы предназначены для синтаксического анализа и отображения программами, такими как Web-браузеры, а не для чтения их людьми.

1.2.3. Протокол передачи гипертекста (HTTP)

Функционирование Web зависит от наличия стандартного, устоявшегося способа для взаимодействия Web-компонентов. Протокол передачи гипертекста Hypertext Transfer Protocol (HTTP) представляет собой наиболее распространенный способ передачи ресурсов в Web. HTTP определяет формат и назначение сообщений, которыми обмениваются Web-компоненты, такие как клиенты и серверы. Протокол — это язык, схожий с естественными человеческими языками, за исключением того, что он используется программами. Подобно другим языкам, протокол имеет свой особый синтаксис и семантику, связанные с использованием элементов языка. HTTP определяет синтаксис сообщений и способ интерпретации полей каждой строки сообщения. HTTP представляет собой протокол типа *запрос-ответ* — клиент отправляет сообщение-запрос, а затем сервер откликается сообщением-ответом. Клиентские запросы обычно порождаются действиями пользователя, например, щелчком мышью на гиперссылке или вводом URI в адресной строке браузера. HTTP не сохраняет своего состояния — клиенты и серверы трактуют каждый обмен сообщениями независимо от других, и нет необходимости сохранять какое-либо промежуточное состояние между запросами и ответами.

1.3. Термины и принципы

Этот раздел знакомит с основными понятиями и принципиальными вопросами, раскрываемыми в книге. В таблице 1.1 приведен перечень терминов, получивших широкое употребление как в популярных, так и в технических публикациях. Книга фокусирует внимание на основных принципах и принятых стандартах, а не на приложениях, функционирующих в Web:

- **Web-содержание.** Web предоставляет пользователям доступ к ресурсам посредством обмена HTTP-сообщениями.
- **Программные компоненты.** Web-транзакции состоят из взаимодействий между клиентами, промежуточными звеньями и серверами.
- **Коммуникационная среда.** Internet обеспечивает коммуникационную среду для передачи сообщений между компонентами Web.
- **Стандартизация.** Стандартизация протоколов помогает обеспечить совместимость программных компонентов Web друг с другом.
- **Web-трафик и производительность.** Эффективность работы программных компонентов и сети оказывает существенное влияние на восприятие пользователем информации в Web. Анализ Web-трафика дает ценную информацию, необходимую для повышения эффективности сетевого взаимодействия.
- **Web-приложения.** Кэширование и мультимедийные потоки в основном влияют на производительность Web и на восприятие Web пользователями.

Таблица 1.1. Основные термины, относящиеся к Web

Термин	Определение
WWW/Web	Всемирная паутина (World Wide Web), мир информации, доступ к которому осуществляется через подключенные к сети компьютеры
Гипертекст	Нелинейное связывание близких по смыслу документов для облегчения переходов между ними
Internet	Всемирное объединение взаимодействующих между собой с помощью протокола Internet Protocol (IP) сетей
Web-страница	Документ, доступный в Web с помощью URI
Web-сайт	Набор связанных друг с другом Web-страниц
Браузер	Приложение для запроса и отображения Web-ресурсов

Ключевые термины, относящиеся к этим темам, сведены в таблицы и рассматриваются в отдельных подразделах.

1.3.1. Web-содержание

В таблице 1.2 представлен список терминов, относящихся к Web-содержанию и к протоколам HTTP. Web состоит из набора *ресурсов* или объектов, распределенных по всей Internet. Каждый ресурс представляет собой документ, доступный через сеть или сервис, документ может быть доступен в различных форматах (например, HTML или PostScript). Ресурс может быть статическим файлом или генерироваться динамически во время выполнения запроса. Каждая HTTP-передача связана с одним ресурсом, идентифицируемым URI в сообщении-запросе. Web-страница состоит из контейнерного ресурса, такого как HTML-файл, который может включать ссылки на один или несколько *встроенных* ресурсов, таких как изображения или анимация. Загрузка Web-страницы предусматривает отдельные HTTP-передачи для контейнера и каждого из встроенных ресурсов.

Таблица 1.2. Терминология, относящаяся к Web-ресурсам и HTTP-сообщениям

Термин	Описание
Ресурс	Сетевой объект данных или сервис, идентифицируемый с помощью URI
Сообщение	Основная единица сетевого взаимодействия в HTTP
Отправитель/ получатель	Компоненты, ответственные за отправку/получение сообщения
Заголовок	Управляющая часть сообщения
Содержимое	Информация, передаваемая в теле сообщения
Вышерасположенный/ нижерасположенный	Направление потока сообщений (от вышерасположенного отправителя к нижерасположенному получателю)

Каждая HTTP-передача состоит из двух *сообщений*: сообщения-запроса, отправляемого клиентом, и соответствующего сообщения-ответа сервера. Клиент является *отправителем* сообщения-запроса и *получателем* сообщения-ответа. Аналогично,

сервер является *получателем* сообщения-запроса и *отправителем* сообщения-ответа. Отправитель сообщения является вышерасположенным для получателя, а получатель является нижерасположенным для отправителя. Сообщение представляет собой последовательность байтов, начинающуюся с необязательного *заголовка*, который содержит управляющую информацию. HTTP-заголовки имеют формат ASCII; например, заголовок Date, присутствующий в сообщении-запросе и в сообщении-ответе, имеет следующий формат:

```
Date: Sat Oct 28 2000 11:29:32 GMT
```

Кроме того, сообщение-запрос или сообщение-ответ может иметь *тело* — непосредственное представление ресурса. Например, ответ на запрос на **http://www.foo.com/coolpic.gif** может содержать GIF-изображение в качестве тела сообщения.

1.3.2. Программные компоненты

В таблице 1.3 приведены основные программные компоненты Web и некоторые широко используемые термины, имеющие к ним отношение. *Агент пользователя* инициирует HTTP-запрос и обрабатывает ответ. Наиболее типичным примером агента пользователя является Web-браузер, который отправляет запросы от имени пользователя и выполняет множество других задач, таких как отображение Web-страниц и хранение пользовательских предпочтений. *Клиент* представляет собой программу, которая отправляет HTTP-запрос и получает ответ. *Исходный сервер* — это программа, которая предоставляет или генерирует Web-ресурс. *Сервер* — это программа, которая получает HTTP-запрос и отправляет ответ. На практике клиент может отправлять HTTP-запрос на *промежуточное звено* — другой компонент Web на пути к исходному серверу. Например, сотрудники компании могут настроить свои браузеры на использование прокси-сервер, которому будут направляться запросы. *Прокси-сервер* играет роль и клиента, и сервера. Прокси-серверы могут выполнять множество функций, таких как фильтрация запросов к нежелательным Web-сайтам, предоставление клиентам определенной степени анонимности и кэширование популярных ресурсов. HTTP не сохраняет состояние, поэтому у сервера отсутствует информация о запросе после отправки ответа. Сервер может проинструктировать агента пользователя хранить определенную информацию между серией запросов и ответов, поместив ее в *cookies*.

Таблица 1.3. Терминология, относящаяся к программным компонентам Web

Термин	Описание
Агент пользователя	Клиентская программа, которая инициирует запрос (например, браузер)
Web-клиент	Программа, которая отправляет HTTP-запрос Web-серверу
Web-сервер	Программа, которая получает HTTP-запрос от Web-клиента и передает ему ответ
Исходный сервер	Сервер, где располагается или динамически создается запрашиваемый ресурс
Промежуточное звено	Web-компонент на пути между агентом пользователя и исходным сервером (например, прокси-сервер или шлюз)

Термин	Описание
Прокси-сервер	Промежуточная программа, функционирующая как сервер для клиента и как клиент для сервера
Cookie	Служебная информация, передаваемая между агентом пользователя и исходным сервером

1.3.3. Сеть

В таблице 1.4 приведены основные термины, относящиеся к сетевым протоколам, составляющим основу Web. Клиент и сервер обычно выполняются как программы на *хостах*. Как правило, клиент и сервер размещаются в различных местах в Internet, хотя в настоящее время отмечается быстрый рост применения Web внутри организаций. Отправка и получение HTTP-сообщений требует, чтобы два хоста могли идентифицировать друг друга и обмениваться информацией. Развитие Internet привело к появлению стандартных протоколов, поддерживающих множество сетевых сервисов, таких как FTP, электронная почта и Web. На нижнем уровне Internet предоставляет основной коммуникационный сервис — доставку *пакетов* данных от одного хоста другому. Протокол *Internet Protocol (IP)* осуществляет доставку отдельных пакетов; при этом отправляющий и принимающий хосты идентифицируются 32-битными *IP-адресами*. Эти адреса обычно представляются в нотации, когда каждая из четырех восьмибитных составляющих адреса записывается в виде десятичного числа. Эти составляющие отделяются друг от друга точками (например, 10.243.74.5 или 10.4.170.124).

Таблица 1.4. Терминология, относящаяся к Internet и протоколам

Термин	Описание
Хост	Компьютер, подключенный к сети
Пакет	Основная единица передачи информации в Internet
IP	Internet Protocol — протокол, который осуществляет доставку отдельных пакетов между хостами
IP-адрес	32-битный числовой адрес, идентифицирующий хост[1] в Internet
Имя хоста	Чувствительная к регистру строка, идентифицирующая хост в Internet
DNS	Система именования доменов (Domain Name System), распределенная инфраструктура для преобразования доменных имен хостов в IP-адреса
TCP	Transmission Control Protocol — протокол, который реализует концепцию надежного двунаправленного соединения
Соединение	Логический коммуникационный канал между двумя хостами

Простая служба доставки пакетов не удовлетворяет потребностям большинства сетевых приложений, включая Web. Web-клиент идентифицирует Web-сервер по *доменному имени хоста* (например, **www.att.com** или **users.berkeley.edu**), а не по IP-адресу, а приложения обмениваются HTTP-сообщениями, а не IP-пакетами. Чтобы соединить эти части друг с другом, используется система именования доме-

[1] Точнее IP-адрес идентифицирует сетевой интерфейс. — *Прим. ред.*

нов *Domain Name System* (DNS) и протокол управления передачей *Transmission Control Protocol* (TCP). Перед обращением к Web-серверу Web-клиент сначала преобразует с помощью DNS доменное имя хоста **www.att.com** в IP-адрес. Web-клиент осуществляет системный вызов для обращения к DNS-серверу, который возвращает IP-адрес **www.att.com**. Используя этот IP-адрес, Web-клиент инициирует взаимодействие с Web-сервером. Клиент и сервер устанавливают *TCP-соединение* — логический коммуникационный канал, который обеспечивает двунаправленное взаимодействие между двумя приложениями. Реализованный в операционных системах двух хостов, TCP скрывает детали отправки и получения данных через Internet. После того, как соединение установлено, клиент может использовать TCP-соединение для отправки HTTP-запроса серверу, а сервер может отреагировать, передав HTTP-ответ.

1.3.4. Стандартизация

В таблице 1.5 приведены основные термины, относящиеся к стандартизации протоколов. Браузеры, прокси- и Web-серверы взаимодействуют между собой, чтобы предоставить пользователю доступ к ресурсам в Web. Хотя каждый Web-сайт может иметь свой набор ресурсов с различным сочетанием типов содержания, доступ к этим ресурсам должен быть независимым от их атрибутов. Стандартизация протоколов жизненно необходима, чтобы компоненты взаимодействовали между собой должным образом. Клиент, отправляющий запрос, должен иметь возможность ясно представлять тот спектр ответов, которые он может получить, а сервер должен иметь возможность однозначно интерпретировать запрос. Хотя программное обеспечение Web может быть создано различными компаниями или исследовательскими организациями, взаимодействие между компонентами должно быть предсказуемым. Написание компонентов программного обеспечения Web и построение Web-сайтов должно осуществляться с помощью четко определенных интерфейсов. Стандарт протокола дает возможность гарантировать, что компоненты могут взаимодействовать друг с другом, а процесс стандартизации позволяет приводить вновь разрабатываемые компоненты в соответствие со стандартом.

Таблица 1.5. Терминология, относящаяся к стандартам протоколов Internet

Термин	Описание
IETF	Проблемная группа проектирования Internet (Internet Engineering Task Force) — открытое сообщество, занимающееся развитием и совершенствованием Internet
Рабочая группа	Группа IETF, занимающаяся разработкой определенных стандартов
Рабочий проект (Internet Draft)	Неофициальная версия документов по стандарту, отражающая, что работа над стандартом продолжается
RFC	Запрос на комментарий (Request For Comments), официальный документ, связанный со стандартами Internet
MUST, SHOULD и MAY	ОБЯЗАТЕЛЬНО, ЖЕЛАТЕЛЬНО и ВОЗМОЖНО — уровни требований к совместимости со спецификацией протокола

Internet не имеет единого «центра управления»; различные хосты и сети придерживаются стандартов протоколов добровольно. Организация *Internet Engineering Task Force* (IETF) [Bra96b] представляет собой открытое сообщество разработчи-

ков, поставщиков программных продуктов и исследователей, которые способствуют развитию и функционированию Internet. Описания стандартов в публикуемых документах составляют основу для взаимопонимания. IETF формирует стандарты с помощью официальных публикаций, называемых *Request for Comments* (RFC — запрос на комментарии). Первый документ RFC был опубликован в 1969 г.

Документы IETF на начальном этапе представляют собой *рабочие проекты* (Internet Drafts), которые являются неофициальными документами, подвергаемыми корректировкам на основе замечаний, получаемых от членов сообщества. Не все рабочие проекты превращаются в RFC. RFC делятся по темам и направлениям: стандарты, история, информация и исследования. Документы стандартов обычно создаются *рабочей группой (Working Group)* IETF. Требования совместимости со стандартом имеют различные уровни: *MUST (ОБЯЗАТЕЛЬНО)*, *SHOULD (ЖЕЛАТЕЛЬНО)* и *MAY (ВОЗМОЖНО)*. Любая совместимая реализация должна удовлетворять всем требованиям уровня MUST. Реализация может считаться условно совместимой, если она удовлетворяет всем требованиям уровня SHOULD. Требования уровня MAY соблюдать необязательно. Эти уровни требований помогают обеспечить взаимную совместимость различных реализаций. Документы стандартов проходят через три стадии: Proposed Standard (Предложение по стандарту), Draft Standard (Проект стандарта) и Internet Standard (Стандарт Internet), постепенно проходя путь от хорошо разработанной спецификации до высококачественного, зрелого в техническом отношении документа, основанного на практическом опыте применения. Некоторые RFC отражают текущий практический опыт (best current practices — BCP) и известны как BCP-документы. Стандарты действуют не вечно — они могут быть отменены и заменены следующей спецификацией.

Чтобы содействовать росту Web, в 1994 г. был основан World Wide Web Consortium (W3C). W3C было разработано множество стандартов, получивших название Рекомендаций, которые имели отношение к Web. W3C сосредотачивает свое внимание на представлении Web-содержания и связанных с этим технологиях (например, языке HTML), а не на аспектах, относящихся собственно к сети. W3C работает над вопросами архитектуры, проблемами, связанными с пользовательским интерфейсом, форматами, языками, юридическими вопросами и вопросами издательской деятельности, а также проблемами доступности Web для людей с ограниченными возможностями (инвалидов).

1.3.5. Web-трафик и производительность

Таблица 1.6 содержит термины, имеющие отношение к Web-трафику и производительности. Широкая популярность Web привела к быстрому росту числа пользователей и Web-сайтов. Однако возможности сети и серверов не безграничны. Желание пользователей быстро получать ответы стало причиной более пристального внимания к проблемам производительности. *Время ожидания* — это время между началом действия, например, отправкой сообщения-запроса и первым признаком получения ответа. *Время ожидания на стороне пользователя* — это задержка между моментом, когда пользователь активизирует гиперссылку, и началом отображения запрашиваемого содержимого в окне браузера. Каждый программный компонент и протокол характеризуются определенным временем ожидания на стороне пользователя. Например, браузер должен построить HTTP-запрос, определить IP-адрес сервера, установить TCP-соединение с сервером, передать запрос, дождаться ответа и, наконец, отобразить ответ пользователю. Большое время ожидания на стороне пользователя может обусловливаться различными факторами, та-

кими как задержки при обращении к DNS, «заторы» в сети или перегруженность сервера.

Таблица 1.6. Терминология, относящаяся к Web-трафику и производительности

Термин	Описание
Время ожидания	Время между началом действия и первым признаком ответа
Время ожидания на стороне пользователя	Время между действием пользователя и началом отображения содержимого
Пропускная способность	Объем трафика, который может быть передан за единицу времени
Рабочая нагрузка	Набор входных данных, получаемых компонентом Web за определенное время
Регистрация в журнале	Запись транзакций, выполненных компонентом Web

Кроме того, браузер может работать через Internet-соединение с относительно низкой *пропускной способностью*, например, через модем со скоростью передачи 28,8 Кбит/с. Передача ресурса большого объема через модем вносит заметную задержку, даже если Web-сервер и остальная часть Internet загружены мало. Анализ производительности сложной и разнородной системы, каковой является Web, на практике представляет собой достаточно непростую задачу. Все же общее представление о системе и осведомленность в принципах ее функционирования имеют важное значение для выявления проблем с производительностью и совершенствования технологий. Измерение параметров и анализ Web-трафика необходимы для того, чтобы охарактеризовать работу пользователей и свойства ресурсов, доступных в Web. Большинство действий по анализу Web-трафика заносятся в *журналы регистрации*, которые обеспечивают запись информации HTTP-передачах, выполняемых программными компонентами Web. Анализ журналов регистрации полезен для понимания характеристик *рабочей нагрузки* программных компонентов Web. Рабочая нагрузка определяется набором всех входных данных (например, HTTP-запросов), получаемых компонентом за определенное время. Характеристики рабочей нагрузки, такие как время между запросами, а также размеры и степень популярности различных ресурсов, имеют важное влияние на производительность Web-протоколов, программных компонентов и сети.

1.3.6. Web-приложения

В таблице 1.7 приведены термины, относящиеся к Web-приложениям. С ростом популярности Web кэширование и мультимедийные потоки стали важными приложениями. Рост Web привел к большой загруженности Web-серверов и Internet и, как следствие, к увеличению времени ожидания на стороне пользователя при доступе к Web-содержимому. Кэширование перемещает содержимое ближе к пользователю. Это относительно простая идея, существовавшая уже на первых порах развития Web. *Кэш* представляет собой локальное хранилище сообщений. Кэш может размещаться в браузере пользователя, на Web-сервере или на компьютере, расположенном на пути между пользователем и Web-сервером. Хотя кэширование повышает производительность, кэшированные ответы могут отличаться от тех, которые имеются на исходном сервере. Механизм *согласования кэша* используется для обеспечения совпадения кэшированного сообщения с сообщением на сервере. Аль-

терпативой кэшированию является *репликация*, предусматривающая явное дублирование Web-содержания на нескольких серверах. Клиентский запрос направляется одной из реплик. Вместо репликации всего содержания Web-сайта может осуществляться избирательная доставка содержания от имени исходного сервера. Подобный прием называется *распределением Web-содержания*.

Таблица 1.7. Терминология, связанная с Web-кэшированием и организацией мультимедийных потоков

Термин	Описание
Кэш	Хранилище сообщений, используемое для сокращения времени ожидания на стороне пользователя, а также снижения нагрузки на сеть и на сервер
Согласование кэша	Механизм снижения вероятности возврата устаревших сообщений из кэша
Репликация	Дублирование ресурсов на нескольких исходных серверах
Распределение Web-содержания	Доставка ресурсов от имени исходного сервера
Поток аудио/видеоданных	Последовательность звуковых сэмплов или видеокадров
Потоковая передача	Одновременная передача информации сервером и воспроизведение клиентом аудио/видео данных
Медиаплейер	Приложение для воспроизведения мультимедийных потоков

Web предоставляет доступ к разнообразным ресурсам безотносительно к их формату и местоположению. В последние несколько лет наблюдается рост популярности аудио и видео в Web. В отличие от традиционного Web-содержания, *потоки аудио* или *видеоданных* состоят из последовательностей звуковых сэмплов или видеокадров, занимающих определенный период времени. В приложениях для воспроизведения мультимедийных потоков, таких как музыкальные проигрыватели или видео по запросу, клиент воспроизводит сэмплы или кадры по мере их поступления от сервера вместо того, чтобы загрузить все содержимое целиком, а затем начать его воспроизведение. Такие приложения обычно поглощают значительную часть пропускной способности сети и чувствительны к задержкам при получении аудиосэмплов и видеокадров. Хотя HTTP и может быть использован для доставки мультимедийного содержания, по большей части передача мультимедийных данных лишь инициируется с помощью HTTP. Затем отдельное вспомогательное приложение, *медиаплейер*, взаимодействует с мультимедийным сервером, используя свой набор протоколов, которые лучше подходят для работы с потоками аудио и видео.

1.4. Темы, оставшиеся нераскрытыми

Представление технической основы Web — достаточно непростая задача. Быстрый рост Web привел к беспрецедентному технологическому развитию, включая идеи, заимствованные и адаптированные из более ранних протоколов, языков и приложений. Эта книга сосредотачивает свое внимание на протоколах и программ-

ных компонентах, имеющих отношение к передаче Web-содержания; в связи с этим, в книге не рассматриваются достаточно глубоко другие темы, такие как:

- Языки, включая Hypertext Markup Language (HTML) и XML (eXtensible Markup Language).
- Perl и другие языки сценариев для Web-приложений.
- Проблемы безопасности в Web.
- Социальные проблемы, а также проблемы выбора и реализации стратегий, относящихся к конфиденциальности, безопасности и доступности.
- Вопросы администрирования и настройки прокси- и Web-серверов.
- Программные продукты конкретных производителей, включая прокси- или Web-серверы.

Подробное рассмотрение этих вопросов не входит в задачу данной книги. Подробную информацию по ним вы можете найти в других книгах. Например, есть множество книг, где описывается HTML или настройка Web-серверов. Наконец, мы решили не уделять особого внимания ряду вопросов, таких как описание программных продуктов различных производителей, поскольку такая информация быстро устаревает. Вместо этого мы сосредоточили внимание на концепциях и технологиях.

1.5. Краткий экскурс по книге

В этом разделе мы совершим краткий экскурс по книге. Читатели могут принять решение пропустить определенные главы книги, либо изучать ее полностью в порядке следования глав. В данной главе был дан общий обзор Web и тем, затрагиваемых в книге. Последующие главы книги поделены на пять основных частей:

- **Программные компоненты Web.** Во второй части книги рассматриваются ключевые программные компоненты Web: клиенты, прокси- и Web-серверы. Широкое распространение браузера Mosaic способствовало успешному развитию Web. Последующее создание прокси-серверов способствовало масштабируемости. Современные Web-серверы далеко ушли от своих первых реализаций и в настоящее время способны обслуживать десятки миллионов запросов ежедневно. В этой части книги содержится как исторический обзор эволюции компонентов, так и обзор ключевых принципов их функционирования. Хотя каждый из компонентов рассматривается по отдельности в соответствующих главах, показана их взаимосвязь на классическом примере стандартной Web-транзакции. Три главы предоставляют необходимую базу для перехода к более детальному рассмотрению инфраструктуры Web далее в этой книге. Сначала вы познакомитесь с клиентами, затем с прокси- и, наконец, с Web-серверами, т.е. пройдете тем путем, который обычно проходят сообщения. Помимо наиболее популярного Web-клиента — браузера, мы также познакомим вас со специализированными клиентами, такими как спайдерами, сканирующими сеть, которые дали возможность реализовать в Web популярные поисковые приложения. Рассматриваются также вопросы, связанные с cookies, которые используются для сохранения информации при обмене Web-сообщениями. Эти вопросы рассматриваются как применительно к клиенту, так и применительно к серверу. Поскольку эта часть книги предшествует главам, посвященным протоколам, мы отложим обсуждение проблем, относящихся к Web-протоколам. На примере популярного Web-сервера Apache рассматривается архитектура и функциональные возможности Web-сервера.

- **Web-протоколы.** Доставка Web-содержания зависит от ряда стандартных протоколов: IP, TCP, DNS, которые обеспечивают основные коммуникационные сервисы, и HTTP, представляющего собой протокол прикладного уровня. Эти протоколы эволюционировали на протяжении многих лет в ходе процесса стандартизации, осуществляемого IETF, и все вместе образуют устойчивую инфраструктуру для запроса и передачи Web-ресурсов. Глубокое обсуждение этих протоколов формирует основу книги и служит важным базовым материалом для остальных глав. Рассмотрение ведется по принципу снизу — вверх, начиная с главы, посвященной сетевым протоколам: IP, DNS, TCP, а также различным протоколам прикладного уровня, предшествовавшим HTTP. Следующие две главы посвящены подробному рассмотрению протокола HTTP — как HTTP/1.0, так и более новому стандарту HTTP/1.1. Материал поделен на две главы, чтобы показать эволюцию протокола с течением времени на фоне быстрого роста Web. В ходе обсуждения раскрывается процесс эволюции протокола, анализируются проблемы, свойственные ранним версиям протокола. Две главы, посвященные HTTP, дополняют описание протокола в соответствии с документами RFC 1945 и RFC 2616. Эта часть книги заканчивается главой, где рассмотрено взаимодействие между HTTP и TCP, которое оказывает важное влияние на эффективность Web-трафика в Internet.
- **Измерение параметров и характеристики Web-трафика.** В этой части книги мы представляем подробный обзор имеющихся технологий для сбора и анализа параметров Web-трафика. Прокси- и Web-серверы создают записи в журналах регистрации при выполнении HTTP-транзакций. Измерения также могут осуществляться путем пассивного мониторинга ссылок в сети или активного генерирования запросов к целевым серверам. С первых дней существования Web исследователи и разработчики протоколов осуществляли анализ измеренных данных для определения характеристик Web-трафика и поиска методов повышения производительности Web. Производительность Web зависит от того, как поведение пользователей связано с базовыми протоколами и программными компонентами. Измерение параметров и анализ Web-трафика играет также важную роль в создании эталонных тестов для сравнения различных реализаций прокси- и Web-серверов. В первой главе обсуждаются три основных этапа в измерении параметров Web-трафика: мониторинг передачи сообщений, генерирование записей о проведенных измерениях и предварительная обработка данных перед их анализом. Мы рассмотрим регистрацию параметров для клиентов, прокси- и Web-серверов, а также мониторинг пакетов и активные измерения. Далее мы обсудим, как преодолевать ограничения, свойственные каждому из методов измерений, и рассмотрим практические примеры, демонстрирующие их реальное применение. Вторая глава посвящена построению моделей рабочей нагрузки, которые позволяют выявить наиболее существенные параметры Web-трафика. В подробном обзоре характеристик Web-трафика особое внимание уделено применению HTTP, ключевых атрибутов Web-ресурсов и влиянию поведения пользователей на Web-трафик. Мы также выясним, как характеристики рабочей нагрузки могут изменяться при дальнейшем развитии Web.
- **Web-приложения.** Пятая часть книги фокусирует внимание на двух ключевых приложениях — кэшировании и мультимедийных потоках, — которые оказывают значительное влияние на доставку Web-содержания. Быстрый рост Web привел к резкому увеличению Internet-трафика. Ни одна другая технология не развивалась столь быстро. В настоящее время три из четырех

пакетов в трафике Internet приходятся на Web. Такой прорыв сделал кэширование насущной необходимостью. В главе, посвященной кэшированию, дается обзор основных принципов и методов кэширования. Рассматриваются имеющиеся на сегодняшний день проблемы, связанные с кэшированием, в свете протоколов кэширования, аппаратных и программных средств кэширования и с учетом последних веяний, касающихся распределения Web-содержания. В следующей главе представлены вопросы, связанные с доставкой аудио- и видеосодержания, которое является быстро растущим сегментом Web-трафика. По мере того, как все больше пользователей получают доступ к соединениям с Internet с высокой пропускной способностью, мультимедийные потоки становятся все более распространенным явлением. В главе обсуждаются уникальные требования, предъявляемые к аудио- и видеоданным, в сравнении с традиционным содержанием в виде текста и изображений. Далее мы рассмотрим протоколы, предназначенные для передачи мультимедийных потоков, в том числе достаточно глубоко познакомимся с протоколом Real Time Streaming Protocol (RTSP), который заимствовал несколько ключевых принципов от HTTP/1.1.

- **Перспективы исследований.** Последняя часть книги посвящена перспективам исследований по трем важным и актуальным темам: кэширование, измерение параметров и протоколы. Вместо представления сформировавшихся и общепринятых идей, эти главы предоставляют краткий обзор технологий в процессе их развитии. В этих главах объясняются мотивы и разъясняются отдельные составляющие исследований, а также приводятся оценки тех или иных идей. Идеи, рассмотренные здесь, получили всеобщее признание, но не были широко реализованы на момент издания книги. Для каждой темы рассматриваются факторы, влияющие на возможную реализацию идей или на основные выводы по результатам исследований. Даже если идея никогда не применялась на практике, процесс нахождения и оценки решения имеет непреходящее значение для стимулирования новых идей. В каждом из разделов в этих трех главах рассматривается отдельная подтема или исследование. Главы не предоставляют полный обзор исследований, проводящихся по каждой из тем. Вместо этого в них главным образом нашли выражение наши пристрастия и предположения относительно того, какие темы исследований станут наиболее важными. Кроме того, при выборе тем исследований предпочтения отдавались тем из них, которые связаны с ключевыми принципами и стандартными протоколами, рассмотренными в предыдущих частях книги.

Все пять частей книги, объединенные воедино, представляют описание технологий, составляющих основу Web.

Часть II
Компоненты программного обеспечения Web

2

Web-клиенты

В предыдущей главе вы узнали о трех основных видах программных компонентах Web: клиентах, прокси- и Web-серверах. Большинство пользователей Web непосредственно имеют дело с Web-клиентами и почти никогда с прокси- или Web-серверами. Web-серверы являются необходимым компонентом, прокси-серверы играют важную роль, а сеть необходима для передачи данных. И все же *браузер* — наиболее известный тип Web-клиента — это то, что конечные пользователи реально *используют* для взаимодействия с Web.

Можно сказать, что появление Web-браузера Mosaic [Mos] явилось главной причиной, благодаря которой Всемирная паутина стала частью повседневной жизни миллионов людей. Прошло всего три года после создания браузера Mosaic, а число пользователей Web достигло десятков миллионов. Ни один другой программный компонент не смог привлечь так много новых пользователей к новой или уже существующей технологии за такой короткий промежуток времени. Как следствие, возросло число документов, доступных в Internet. Несмотря на то, что Web существовала уже за несколько лет до начала широкого использования Mosaic, браузер существенно способствовал привлечению новых пользователей.

Хотя с точки зрения пользователя браузер является центральной (и даже единственной) частью его восприятия Web, другие клиенты играют столь же важную, хотя и не такую очевидную роль. В этой главе рассматриваются три вида Web-клиентов: браузер, как наиболее популярный вид Web-клиента; *программы-пауки (спайдеры)* и их роль в качестве ключевой составляющей поисковых систем в Web, а также менее известное программное обеспечение — *агенты пользователя*.

Глава начинается с рассмотрения происхождения Web-клиента — программы, которая отправляет Web-запросы и принимает ответы. Популярность графических растровых дисплеев и простота навигации сыграла ключевую роль в эволюции браузеров. Для подробного обзора роли браузеров при взаимодействии с Web используется классический пример. Далее рассматривается настройка браузера, которая заключается в задании атрибутов или параметров, которые являются специфическими для протокола HTTP. Браузеры выполняют несколько функций, которые не связаны с протоколом HTTP, например, вызов программ для интерпретации и отображения ответов. Рассматривается также возможность настройки этих действий.

Браузер является наиболее близким к пользователю из трех основных видов программных компонентов. Проблемы безопасности, связанные с доступом к ресурсам и загрузкой сценариев или других исполняемых программ, являются неотъемлемой составной частью взаимодействия браузера с Web. После рассмотрения вопросов безопасности мы обсудим роль cookies в управлении состоянием между Web-транзакциями. Исследуется важная, но по большей части скрытая роль браузера при работе с cookies.

Спайдер — это разновидность клиентской программы, которая не взаимодействует с пользователями напрямую. Спайдеры необходимы для поиска и индексирования ресурсов. Рассмотрев, как осуществляется поиск, мы расскажем о той нише, которую спайдеры занимают в Web.

Помимо браузеров и спайдеров, большая роль отводится агентам — программам, функционирующим в интересах пользователей в специфических приложениях, включая средства для участия в аукционах и средства фильтрации данных, которые менее известны. Мы рассмотрим некоторые популярные виды таких программных компонентов. В заключение мы остановимся на общих принципах построения клиентов, а также на их практическом применении.

2.1. Клиент как программа

Web-клиент представляет собой разновидность программного обеспечения. Главная задача Web-клиента — отправлять Web-запросы от имени пользователя и принимать ответы. Клиент также используется для многих других целей. Web была разработана после появления традиционной архитектуры программного обеспечения клиент-сервер, в которой клиент связывается с сервером и отправляет ему запрос, после чего получает ответ сервера. На момент создания Web другие системы клиент-сервер (о чем уже говорилось в главе 1) предлагали услуги, аналогичные тем, которые предоставляла Web. Имелась возможность посылать запросы на поиск ресурсов путем обращения к индексам, в которых и осуществлялся поиск. Любая из обращавшихся с запросом программ являлась клиентом. Web развивалась вместе с этими системами (Archie, Gopher, группы новостей и т.д.), поэтому сначала Web-клиент был во многом схож с другими аналогичными компонентами в традиционной схеме клиент-сервер.

В типовой системе клиент-сервер клиенты являются относительно простыми компонентами. Клиент формирует запрос, посылает его серверу, а затем читает, анализирует и отображает ответ. В типовых системах клиент-сервер интеллектуальная часть расположена на сервере. Для Web же это не совсем так. Серверы по-прежнему берут на себя значительную часть функций. Однако на практике Web-клиенты являются достаточно сложными компонентами. Эта сложность проистекает не из тех базовых задач, которые Web-клиент должен выполнять: построение должным образом оформленного Web-запроса, установка соединения и взаимодействие с Web-сервером через надежное соединение транспортного уровня. По большей части сложность клиентской программы определяется факторами, сопровождающими базовый обмен запрос-ответ, — возможностями настройки при создании запроса, интерпретации ответа и его отображении.

2.2. Эволюция браузеров

Первоначально браузер был простой программой, которая давала возможность пользователям получать ресурсы из любой точки Web; то есть он был всего лишь воплощением Web-клиента. Через очень короткое время он стал основой практически для всех взаимодействий с компьютером для многих пользователей. Пользователи применяли браузер для отправки и получения электронной почты, работы с группами новостей, участия в интерактивных форумах и доступа к Web. Браузер часто считается *пользовательским агентом*, а не просто клиентом, поскольку он *инициирует* Web-запрос.

Большинству пользователей браузеров не известно о разнице между Internet и Web. Они также не знают, что большинство приложений, которые они используют, включая электронную почту, группы новостей и т.д., существовало задолго до Web. В этой главе мы особое внимание уделим той роли, которую браузеры играют во взаимодействии с Web. Помимо этого мы обсудим некоторые вопросы, связанные с настройкой браузера пользователем.

Растровые графические дисплеи появились в массовом порядке в середине 80-х годов. Графический интерфейс стал обязательным для любого приложения, претендующего на популярность среди пользователей. Преимущества графического интерфейса над текстовым уже тогда были широко известны. Однако многие пользователи компьютеров еще не имели возможности работать с растровыми дисплеями и часто использовали текстовый интерфейс для доступа к своим файлам. Это нашло отражение в конце 80-х годов в виде интерфейсов к различным средствам доступа к информации, таким как Archie, Gopher и Wide Area Information Server (WAIS) (см. главу 1). Пользователи перешли от доступа к локальным файлам к доступу ко всем имеющимся в Internet ресурсам. Способ доступа также изменился от доступа к ресурсу с заранее известным местоположением на доступ к ресурсам путем поиска в Internet.

Пользователи выигрывают оттого, что им известно множество ресурсов, доступных в данный момент времени, а также имеется информация о последних обращениях к этим ресурсам при работе с группой ресурсов. При доступе к большей коллекции ресурсов в Internet возможность быстро перемещаться между ресурсами приобретает существенное значение. Возможность перемещения между группами ресурсов и между отдельными ресурсами требует наличия ясного и интуитивно понятного пользовательского интерфейса. Коллекция ресурсов в данном месте может быть организована иным способом, чем в других местах. Хотя пользователи получили доступ к большому количеству ресурсов по всему миру, их способность обрабатывать информацию увеличилась не столь заметно. Средство, которое дает возможность пользователям перемещаться между коллекциями ресурсов, должно удовлетворять следующим требованиям:

- Иметь представление о текущем контексте (где находится пользователь и что доступно из этого места).
- Знать, куда пользователь может перейти дальше, и где пользователь был до этого момента.
- Иметь возможности настройки средств навигации и отображения ресурсов, к которым осуществляется доступ.
- Быть способным осуществлять поиск в коллекции ресурсов.

Первый браузер был создан в лаборатории CERN с использованием графического интерфейса пользователя NeXTStep [BLCGP92] и продемонстрировал мощные возможности использования гипертекстовых ссылок для просмотра коллекций документов. Он получил название **WorldWideWeb** и был написан создателем Web Тимом Бернерс-Ли. Были также созданы клиентские программы, которые могли работать и в текстовом режиме. С точки зрения Web-сервера клиент был лишь программой, независимо от интерфейса, предоставляемого конечному пользователю. Работа над созданием первого текстового браузера под названием *linemode* [Pel91] началась в конце 1990-го года, а доступен он стал в марте 1991 г. Браузер LineMode был браузером, который давал возможность пользователям перемещаться по набору связанных ресурсов, и служил в качестве средства извлечения однородной информации. LineMode гарантировал доступ к Web для всех пользовате-

лей, независимо от терминала, который они использовали. Пользователи могли перемещаться по коллекции Web-документов с помощью только клавиатуры и всего двух управляющих последовательностей: возврата каретки и перевода строки. Ссылка, например, отображалась в виде числа в квадратных скобках. Пользователь мог ввести число с клавиатуры, чтобы перейти к документу, на который указывала ссылка. В клиентской программе имелось ровно семь команд [Pel91], таких как **List** (для вывода списка всех ссылок в документе), **K** для выполнения поиска по ключевому слову и **Recall** для вывода списка документов, просмотренных в сеансе на данный момент (прототип журнала регистрации событий).

Первый браузер представлял собой нечто большее, чем демонстрацию правильности концепции. Он заложил основные принципы, используемые при создании современных браузеров. Если отвлечься от внешнего оформления, главным достоинством современных браузеров является их способность легко перемещаться по ссылкам в Internet, — эта возможность была реализована на самых первых порах существования Web. Другими словами, браузер стал реализацией концепции, в соответствии с которой пользователям предоставлялась возможность быстро перемещаться между коллекциями документов и интуитивно понятно осуществлять эти действия.

После LineMode появилось несколько других браузеров, от браузера Curses с навигацией с помощью клавиш управления курсором до семейства браузеров, работающих под управлением различных популярных систем построения оконного интерфейса: X11 Window System (Viola, tkWWW, MidasWWW), браузера, написанного для компьютеров Macintosh, и даже браузера, написанного на Perl. Появление браузера Mosaic стало значительным шагом вперед, а Mosaic стал основным широко используемым браузером. Текстовой браузер Lynx появился в 1992 г. и до сих пор находит ограниченное применение. Некоторые из разработчиков Mosaic затем создали браузер Netscape, а несколькими годами позже был создан браузер Internet Explorer. На момент подготовки этой книги к публикации (на протяжении последнего года или около этого) в браузеры не было добавлено каких-либо значительных новых функций. Другими словами, браузеры достигли зрелого уровня. В то же время Web-серверы постоянно модифицируются с целью увеличения производительности, надежности и предоставления различного уровня сервиса различным клиентам.

2.3. Функции браузера, относящиеся к Web

Браузер прежде всего реализует функции Web-клиента: формирует и отправляет HTTP-запрос, затем получает, синтаксически анализирует и отображает ответ. Сеанс работы браузера представляет собой набор запросов, посылаемых пользователем, возможно, на основе ответов, полученных на каждом из этапов. Сеанс работы браузера может длиться как несколько минут, так и несколько часов. На рис. 2.1 показаны различные шаги процесса работы с Web-запросом. Несколько упрощая, можно сказать, что сначала осуществляется синтаксический анализ указанного пользователем URL (Uniform Resource Locator — унифицированный указатель ресурса), определяется IP-адрес Web-сервера, с которым следует связаться, затем устанавливается соединение с сервером, после чего посылается HTTP-запрос. Заметим, что некоторые из этапов могут не выполняться для каждого запроса из-за использования *кэширования* — браузер может иметь копию ответа, полученного ранее, что избавляет от необходимости отправлять запрос на этот ресурс снова. Подробнее кэширование будет рассмотрено в разделе 2.3.3.

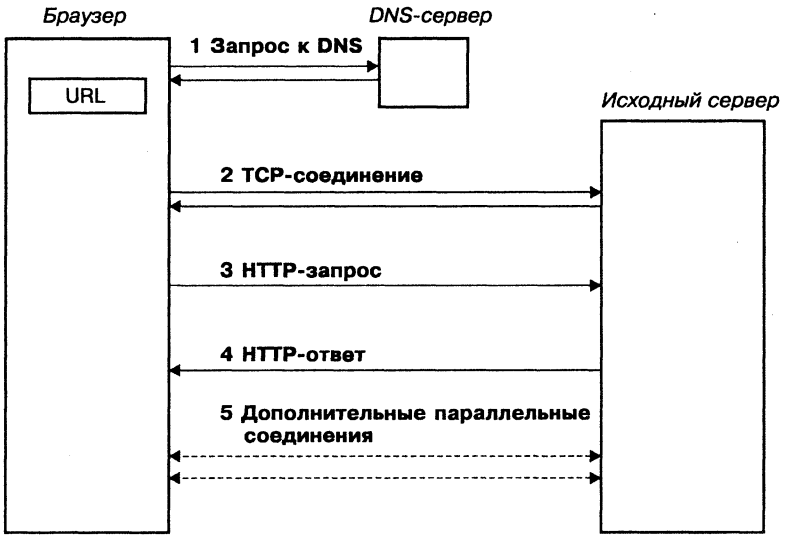


Рис. 2.1. Основные шаги работы браузера

URL может быть предоставлен различными способами; чаще всего это щелчок мышью на имеющейся на странице гиперссылке. URL может быть также введен в адресной строке браузера или выбран из файла *закладок* (файла, содержащего URL и краткие описания часто посещаемых ресурсов) или из журнала браузера. Браузеры воспринимают **http** как протокол по умолчанию, если не указано иное, а также могут завершать ввод пользователя, предлагая на выбор недавно использованные URL. В таблице 2.1 представлены некоторые действия пользователя, которые могут породить Web-запрос.

Таблица 2.1. Действия пользователя, которые могут породить запрос

Вариант ввода	Способ создания запроса
Кнопка Forward/Backward (Вперед/Назад)	Механизм запоминания последних обращений пользователя
Выбор гиперссылки	Из соответствующей строки URL в HTML-тексте
Закладки (избранное)	Из пользовательского набора часто используемых гиперссылок
Щелчок на кнопке Submit (Отправить), нажатие клавиши Return или Enter	URL извлекается из соответствующей формы
Адресная строка	Пользовательский ввод
Различные меню	Из выбранного пункта меню
Изображения	Щелчок мышью автоматически генерирует запрос
Обновление	Текущий URL

В начале раздела мы рассмотрим классический пример, иллюстрирующий функции Web-браузера. Далее мы познакомимся с типичным способом построения запросов — заполнением формы. Затем мы рассмотрим роль, которую играет кэширование при работе с браузером. После этого мы оценим роль прокси-сервера, если таковой имеется на пути между клиентом и сервером. Затем будет рассмотрены функции браузера в построении заголовков HTTP-запроса. Наконец, мы обсудим, как браузер обрабатывает HTTP-ответ.

2.3.1. Классический пример, иллюстрирующий функции Web-браузера

Мы будем использовать нижеследующий пример на протяжении всей книги. Документ, в который встроено несколько изображений, должен быть загружен клиентом с исходного сервера. В этом примере мы предполагаем, что клиент напрямую взаимодействует с исходным сервером без промежуточных звеньев, таких как прокси-серверы. Кроме того, мы предполагаем, что кэширование ресурсов не используется.

Обычно в браузере отображается видимая часть гиперссылки, в то время как URL скрыт от пользователя. Видимая часть гиперссылки обычно является пояснением, хотя иногда используются фразы типа «щелкните здесь». Видимая часть гиперссылки обычно определенным образом выделяется, например, цветом и/или подчеркиванием. Пользователь активизирует гиперссылку, щелкая на ней мышью или вводя URL в предусмотренном для этого адресном поле в браузере. Предположим, что пользователь выбрал ссылку **http://www.bar.com/foo.html**. Браузер выполняет синтаксический анализ URL **http://www.bar.com/foo.html**. Первая часть, предшествующая символу ':', соответствует протоколу, который браузер будет использовать для выборки ресурса **www.bar.com/foo.html**. В данном случае это **http**, хотя, как мы увидим далее, могут быть использованы и другие протоколы, такие как File Transfer Protocol (FTP) или Telnet. Сам ресурс **www.bar.com/foo.html** имеет две части: начальную часть, состоящую из доменного имени компьютера (**www.bar.com**), на котором размещен Web-сервер, и имени ресурса (**/foo.html**), доступного на сервере **www.bar.com**. Браузер делает запрос к серверу доменных имен (DNS) для преобразования доменного имени **www.bar.com** в IP-адрес сервера (этап 1 на рис. 2.1). После получения IP-адреса компьютера **www.bar.com** браузер устанавливает соединение на транспортном уровне с Web-сервером, используя протокол Transmission Control Protocol (TCP) (этап 2). После успешного установления соединения браузер отправляет сформатированный HTTP-запрос на ресурс **/foo.html** (этап 3). Web-сервер **www.bar.com** возвращает текущее содержания ресурса **/foo.html** (этап 4).

Таблица 2.2. Ресурсы, используемые в примере, и их тип содержания

Ресурс	Описание	Тип содержания
/foo.html	Контейнерный HTML-документ	HTML
/foo1.gif	Встроенное изображение	GIF
/foo2.gif	Встроенное изображение	GIF
/foo3.jpg	Встроенное изображение	JPEG
/book.cgi	Исполняемый сценарий	CGI
/mp.tv	Мультимедийный документ	Специальный формат

Браузер осуществляет синтаксический анализ ответа и может устанавливать дополнительные соединения для загрузки встроенных ресурсов. В этом примере мы полагаем, что в **/foo.html** имеются три встроенных ресурса. На странице также имеется ссылка на ресурс, представляющий собой CGI-сценарий (об этом пойдет речь в главе 4, раздел 4.2.3), и ссылка на мультимедийный объект. В таблице 2.2 перечислены ресурсы, используемые в примере. Браузер может сразу начать отображение частично загруженного содержимого **/foo.html**, даже если он устанавливает дополнительные соединения для запроса встроенных изображений[1]. Дополнительные соединения, необходимые для загрузки изображений, могут быть установлены параллельно. Соединения не обязательно могут устанавливаться с тем же исходным сервером, поскольку некоторые из встроенных ресурсов документа **/foo.html** могут размещаться на других серверах. Браузер также может начать отображение каждого из изображений по мере их получении. Задача браузера завершается после того, как он получит все встроенные изображения и отобразит их. На рис. 2.2 показано, как контейнерный документ **foo.html** должен выглядеть вместе с тремя встроенными изображениями, формой и ссылкой на мультимедийный документ.

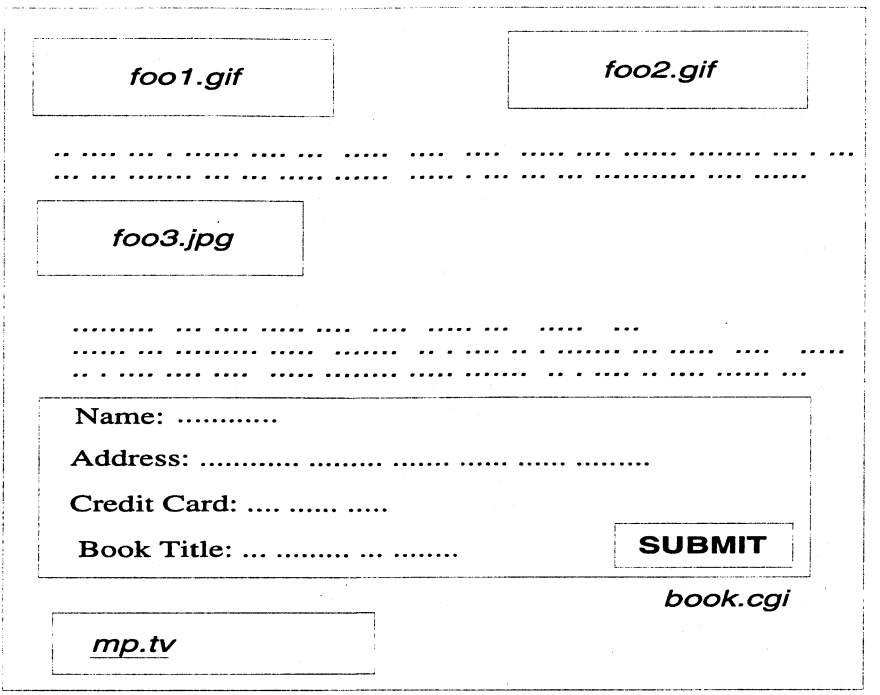


Рис. 2.2. Контейнерный документ **foo.html**

2.3.2. Выдача запроса браузером

Большинство пользователей делают запрос, щелкая на видимой части гиперссылки на Web-странице или вводя URL в адресной строке браузера. Однако в некоторых случаях построение запроса представляет собой несколько более сложный

[1] Для этого в теге <img> необходимо задать ширину и высоту изображения. — *Прим. ред.*

процесс. Рассмотрим пример, связанный с заказом книг в Web, для чего требуется заполнить форму, состоящую из набора полей, значения которых должны быть выбраны из меню или введены. Типичными примерами создания запросов в результате заполнения пользователем формы являются приложения электронной коммерции и поисковые системы. Одна из причин построения запросов с помощью форм состоит в том, что значительную часть всех запросов в Web получают поисковые сайты. Пользователь обычно делает запрос на поиск, заполняя поисковую форму и щелкая на кнопке передачи данных, либо нажимая клавишу Enter. Отправка данных формы или щелчок на гиперссылке приводят к одному и тому же действию — построению соответствующим образом сформатированного HTTP-запроса.

Вернемся к предыдущему примеру, в котором имеется ссылка на исполняемый на сервере сценарий **/book.cgi**. Страница заказа книг включает в себя форму, которую пользователь заполняет информацией, необходимой для заказа книги. Набор полей, заполняемых пользователем, обычно содержит имя, адрес, номер кредитной карты и название книги. Заполненная форма может быть отправлена исходному серверу по меньшей мере двумя способами. В первом способе значение каждого поля и соответствующее имя поля трактуются как пары (имя, значение). Коллекция пар (имя, значение) включается в качестве параметров в заголовок HTTP-запроса **GET**. Например, если одно из полей в форме есть поле имени, а пользователь вводит в него Noam Chomsky, пара может быть включена в URL в виде **http://www.foo.com/book.cgi?name=Noam+Chomsky**. Другой способ — включить данные формы в тело HTTP-сообщения с помощью другого метода запроса (**POST**).

2.3.3. Кэширование в браузере

В этом разделе мы вкратце остановимся на кэшировании в Web применительно к браузерам. Кэш — это локальное хранилище данных, которое может использоваться для уменьшения времени ожидания пользователем ответа от сервера. В браузере обычно применяются два вида кэширования: с использованием памяти выполняющегося процесса и с использованием дискового пространства, выделяемого для кэширования. Если сеть сильно загружена или если Web-сервер активно используется, получение ресурсов с сайта может занять длительное время. Поскольку большинство пользователей часто посещают одни и те же Web-сайты, браузеру выгодно кэшировать последние загруженные страницы. Если пользователь попытается вернуться к Web-странице, которую он недавно посетил, страница может быть отображена из кэша браузера. Привычка пользователей возвращаться к недавно просмотренной странице столь распространена, что в большинстве браузеров предусмотрена кнопка Back (Назад), дающая возможность пользователю возвращаться к последним запрошенным страницам.

Предположим, что одно из встроенных изображений ресурса **/foo.html**, скажем, **/foo1.gif**, было сохранено в кэше браузера. Чтобы повторно вывести изображение, браузеру не нужно выполнять отдельный запрос на него. Поскольку отдельный HTTP-запрос не нужен, нет необходимости и в запросе к DNS на преобразование доменного имени исходного сервера в IP-адрес, в установлении TCP-соединения для отправки HTTP-запроса и в передаче потока байтов, соответствующих изображению **/foo1.gif**, через сеть. Все это приводит к значительному сокращению времени ожидания на стороне пользователя — интервала времени между щелчком пользователя на гиперссылке и началом отображения страницы в окне браузера.

Однако существует возможность, что ресурс был изменен с момента его кэширования. Браузеру может понадобиться проверить, что кэшированный ответ по-прежнему *актуальный*, сопоставив кэшированную копию с текущей копией на исходном сервере. Подобная проверка получила название *проверки актуальности кэша*. Если версия на исходном сервере более новая, то кэшированная копия *устарела*. Например, ресурс **/foo1.gif** мог измениться с момента его последней загрузки.

Считается, что кэш поддерживает *актуальность* кэшированных ресурсов, если при этом делается так, что кэшированные ресурсы являются актуальными. Кэш, который осуществляет проверку кэшированной версии на соответствие версии на исходном сервере каждый раз, когда выполняется запрос на кэшированный ресурс, поддерживает *высокую* степень актуальности кэша. Если кэш использует эвристический подход для определения, по-прежнему ли считать актуальным кэшированный ответ, то поддерживается *низкая* степень актуальности кэша. В последнем случае ответы могут возвращаться без сопоставления с ресурсом на исходном сервере. Для поддержания актуальности кэшированных ресурсов используется несколько эвристических алгоритмов. Проверка кэшированных ресурсов может выполняться периодически через фиксированные интервалы времени, либо через меняющиеся интервалы времени в зависимости от атрибутов ресурса (размер, время последней модификации, тип содержимого и т.д.). Решение, следует ли возвращать кэшированный ресурс, принимается в зависимости от стратегии проверки актуальности, используемой браузером в данный момент. Браузер может решить, что изменение изображений происходит не так часто, как изменение текстовых ресурсов, и использовать кэшированное значение без проверки его актуальности с помощью обращения к исходному серверу. Хотя браузер может и не предполагать, что ресурс **/foo.html** по-прежнему актуален, он вполне может допустить, что изображение **/foo1.gif** в последнее время не менялось. При этом возникает возможность возврата устаревшего ресурса в качестве ответа, если браузер поддерживает низкую степень актуальности кэша.

Пользователь может обойти эвристическую проверку достоверности кэша браузером и принудить браузер отправлять запрос непосредственно на исходный сервер. Например, в браузере Netscape это делается с помощью щелчка на кнопке **Reload** при предварительно нажатой клавише **Shift**.

При записи в кэш ответа исходного сервера в него включается значение «времени последней модификации», указывающее, когда ресурс был в последний раз модифицирован на исходном сервере. Кэш будет хранить это значение и включать его в заголовок запроса как часть процесса проверки актуальности. Если первоначальный ответ сервера не содержит времени последней модификации, кэш присваивает значение эвристически. Например, кэш может присвоить значение времени, соответствующее времени получения ответа. Исходный сервер сравнивает текущее время последней модификации ресурса со значением для кэшированного ресурса, чтобы дать знать, если ресурс был изменен.

Если запрашиваемый ресурс не содержится в кэше браузера, запрос передается серверу, который может быть или не быть исходным Web-сервером.

Подробнее о кэшировании мы поговорим в главе 11.

2.3.4. Заголовки сообщения-запроса

Итак, мы узнали, как может быть выбран URL, как форма может быть преобразована в запрос, и как браузер принимает решение об отправке запроса после проверки его кэшированной копии. Теперь мы рассмотрим, как действие пользователя

при работе с браузером преобразуются в реальный HTTP-запрос. Сообщение HTTP-запроса представляет собой единицу коммуникационного взаимодействия между клиентом и сервером. Сообщение-запрос состоит из заголовка запроса и необязательного тела запроса. Типичные заголовки запроса включают информацию, идентифицирующую агента пользователя, допустимые форматы кодирования ответа и идентифицирующую информацию, указывающую, что пользователь имеет права доступа к запрашиваемому ресурсу. Заголовки запроса могут составляться, исходя из параметров окружения. Например, в заголовок включается информация об агенте пользователя, такая как версия браузера или операционной системы компьютера, на котором выполняется браузер. Другой пример — адрес электронной почты пользователя, полученный из параметров настройки браузера. Полный запрос составляется с учетом особенностей версии HTTP, реализованной в браузере. В главах 6 и 7 мы подробно обсудим различные заголовки сообщения-запроса.

2.3.5. Обработка ответов

На заключительном этапе браузер должен обработать ответ, полученный от сервера. Браузер получает ответ и осуществляет его синтаксический анализ, чтобы определить, что следует отображать (и есть ли, что отображать). Ответ может быть отображен в том же окне браузера, в котором был сформирован запрос, либо в другом окне. Выбор другого окна может быть сделан либо пользователем, либо задан в гиперссылке. Например, некоторые браузеры по умолчанию пытаются отобразить ответ в том же окне, если была нажата левая кнопка мыши, и открывают новое окно, если была использована средняя кнопка мыши. Некоторые гиперссылки автоматически приводят к отображению ответа в отдельном окне. Ответ может содержать сценарий (например, JavaScript), который приводит к отображению результата его выполнения в другом окне.

Отображение ответа определяются пользовательскими предпочтениями, такими как размер шрифта или цвет (подробнее об этом говорится в разделе 2.4). Отображение ответа сервера может зависеть от параметров, входящих в заголовок запроса, — например, таким образом задается язык ответа или формат кодирования. Управление содержанием и его отображением определяется следующим:

- Действиями пользователя (например, щелчком левой или средней кнопкой мыши).
- Предпочитаемым пользователем форматом содержания, указанным в заголовке запроса, сформированного и отправленного браузером.
- Ответом на выбор гиперссылки.
- Возможной адаптацией исходного сервера к представлению содержания в формате, желательном для пользователя.
- Желанием автора содержания отображать содержание в отдельном окне.

Конечный результат, таким образом, зависит от различных программных компонентов, а также возможностей их настройки и степени управляемости на каждом из этапов.

Подробное описание различных действий, которые браузер должен выполнять для воспроизведения страницы, выходят за рамки данной книги. Заинтересованным читателям можно порекомендовать обратиться к книгам, посвященным популярным браузерам, таким как Netscape Navigator или Internet Explorer.

2.4. Настройка браузера

Браузер, подобно любому другому интерактивному программному обеспечению, допускает настройку пользователем. Возможность настройки была характерной чертой интерактивных систем задолго до появления Web. Пользователи настраивали систему под свои нужды, чтобы обеспечить интуитивно понятный интерфейс. С появлением настраиваемых менеджеров окон стало возможным иметь различные конфигурации. Пользователи стараются сделать свое взаимодействие с новыми интерактивными системами похожим на взаимодействие с теми системами, работа с которыми им привычна. Возможности настройки часто определяют, насколько быстро пользователи смогут адаптироваться к новой системе. В широком смысле есть два вида настроек. К первому относятся внешние, видимые свойства, такие как размер и цвет. Ко второму относятся внутренние семантические свойства, такие как выбор языка или управление кэшированием ответов. Диапазон настраиваемых свойств достаточно велик. Это объясняется тем, что браузеры превратились в инструмент для выполнения множества приложений большим числом пользователей. Браузеры используются в качестве основного инструмента для взаимодействия пользователя с Internet.

Браузер принимает во внимание пользовательские предпочтения на всех этапах взаимодействия. Например, пользователь может указать, следует ли направлять запрос через прокси-сервер или отправлять его непосредственно исходному серверу. Аналогично, запрос может нести информацию о желании пользователя получить ответ, отличный от того, который определяется установленным по умолчанию языком или форматом кодирования.

Пользователь должен знать о настройках браузера, чтобы интерпретировать ответ должным образом. Например, если браузер использует кэшированный ответ, поскольку удаленный сервер в данный момент не доступен, пользователь может захотеть узнать, не устарел ли этот ответ. Если ответ использует определенный набор шрифтов или определенный набор символов, то он должен воспроизводиться с использованием соответствующих атрибутов, доступных локально. Восприятие пользователем содержания определяется группой параметров, используемых для настройки. Следует заметить, что на практике пользователи обычно *не* настраивают какие-либо параметры браузера. Такие решения чаще всего принимаются системными администраторами на стороне клиента, и лишь малое число пользователей изменяют параметры, принятые по умолчанию. Системные администраторы, в свою очередь, также часто пользуются настройками по умолчанию программного обеспечения.

В этом разделе мы рассмотрим, как браузеры настраиваются в соответствии с предпочтениями пользователя, связанными как с внешним видом, так и с семантическими параметрами взаимодействия с Web. После этого будет рассмотрена настройка функций браузера, не связанных с протоколом HTTP.

2.4.1. Внешний вид

В браузерах имеется несколько возможностей настройки их внешнего вида, например, набора отображаемых кнопок и т.д. Ответы, отображаемые на экране, имеют несколько свойств, учитывающих пользовательские настройки. Например, размер окна браузера изменяется в зависимости от характеристик компьютера пользователя, а вид отображаемой информации определяется размерами окна браузера и используемыми шрифтами. Разные пользователи выбирают различные настройки

для своих браузеров. С ростом популярности браузеров последние превратились в инструмент взаимодействия с большим числом приложений. Помимо просмотра Web-страниц, многие пользователи используют браузер для отправки и приема сообщений электронной почты, в качестве планировщика, адресной книги, для распечатки файлов и т.д. Каждое из этих приложений имеет свой собственный набор свойств, и многие из них можно настраивать.

Ниже приведен список настраиваемых пользователем свойств, которые оказывают влияние на отображение Web-страниц.

- **Внешний вид.** К этой группе свойств относятся размер окна браузера, отображение панелей инструментов и полос прокрутки. Это общие свойства, имеющиеся в графических интерфейсах большинства платформ. Многие элементы, связанные с внешним видом, являются функциональными возможностями, предоставляемыми оконным менеджером, который управляет отображением информации. Единственные атрибуты, которые влияют на отображение Web-страниц, связаны с областями окна браузера, предназначенными для ввода URL и отображения списка закладок.
- **Отображение встроенных изображений на странице.** На Web-страницах имеется как текст, так и изображения; следовательно, браузеры по умолчанию будут загружать и отображать все ресурсы. Однако изображения требуют больше времени для загрузки, поскольку их размер в среднем больше, чем размер текстовых ресурсов, а Web-страницы могут иметь несколько встроенных изображений. Многие пользователи, использующие низкоскоростные соединения с Internet, часто не хотят долго ждать загрузки полного содержимого страницы — текста и изображений. Другие пользователи могут быть не заинтересованы в просмотре изображений, информационное содержимое которых представляется им не слишком важным. Браузер дает возможность пользователям указать, хотят ли они, чтобы изображения автоматически загружались, или же нет. В большинстве браузеров предусмотрена кнопка, с помощью которой пользователи могут запросить загрузку изображений для определенной страницы, изначально загруженной без изображений. Помимо задержки при загрузке изображений, браузер должен *отображать* изображения, что вносит дополнительную задержку. Однако задержка, связанная с низкой скоростью соединения, обычно является превалирующей.
- **Шрифты.** Текстовый материал часто отображается с помощью нескольких шрифтов. Пользователи, использующие различные шрифты при вводе, естественно ожидают того же при отображении материала. Применительно к Web, браузер должен выбирать между имеющимися на локальном компьютере шрифтами и использовать их для отображения информации на загруженной странице. Он должен выбрать гарнитуру, размер, набор символов, использовать шрифты с символами фиксированной или переменной ширины. Загруженная Web-страница может содержать указания, какие шрифты применять, но пользователь может заместить эти указания своими. Например, некоторые пользователи предпочитают из эстетических соображений использовать моноширинные шрифты.
- **Цвет.** Простейшие опции предусматривают выбор цвета символов и цвета фона окна браузера при отображении страницы либо гиперссылок, имеющихся на странице. Браузеры часто предоставляют пользователям возможность изменения цветов для загружаемых страниц. Один из способов — дать пользователю возможность выбора палитры (заданного набора цветов) элементов

страницы. Другой способ — разрешить пользователю непосредственно задавать цвета элементов Web-страницы в соответствии с собственным вкусом. Пользовательские дисплеи различаются по цветовому разрешению и контрастности. Браузеры могут попытаться подобрать наилучшую цветовую настройку, хотя подобный автоматический выбор применяется редко, поскольку требует изменения настроек по умолчанию, которые могут быть более удобными для пользователя.

Другой способ задания свойств заключается в применении каскадных таблиц стилей Cascading Style Sheets (CSS) [CSSa]. Каскадные таблицы стилей были введены World Wide Web Consortium (W3C) в 1994 г. как единый способ описания способа отображения Web-документов. CSS избавляет автора содержания от необходимости использовать теги HTML для настройки отображения документа. Можно задавать свойства для отображения документа на дисплее пользователя или в печатной форме. Краткое руководство по CSS можно найти в [CSSb].

2.4.2. Семантические настройки

Помимо различных свойств, определяющих внешний вид загруженного документа, в браузере имеется несколько семантических параметров, например, адрес используемого прокси-сервера или набор символов представления содержания, которые также допускают настройку пользователем. Некоторые семантические параметры имеют отношение к настройкам протокола, что требует понимания принципов функционирования HTTP (об этом пойдет речь в главах 6 и 7). Простым примером настраиваемого свойства является язык, используемый при получении запрошенного ресурса, если доступно более одного языка. Выбор языка, сделанный в браузере, переводится в соответствующий синтаксис HTTP и включается в заголовки запроса. Например, если пользователь указал швейцарский диалект немецкого языка (**de-CH**) в качестве предпочтительного языка, следующий заголовок HTTP-запроса

```
Accept-Language: de-CH
```

будет добавлен в список заголовков, передаваемых с каждым запросом. Сервер, получающий запрос, может использовать заголовок **Accept-Language** при создании ответа. Подобные заголовки выражают лишь предпочтения пользователя, и серверы не обязаны корректировать свои ответы с учетом этих заголовков.

В браузере могут быть настроены пять категорий семантических свойств:

- Параметры соединения (прокси-сервер).
- Содержание или выбор ресурсов (допустимые языки и наборы символов).
- Кэширование (см. раздел 2.3.3).
- Обработка запросов (об этом пойдет речь далее в разделе 2.4.3).
- Cookies (они будут рассматриваться в разделе 2.6).

Первые две категории будут рассмотрены ниже, о других речь пойдет далее в других разделах. Браузер разрешает пользователю указывать, следует ли использовать прокси-сервер в качестве промежуточного звена между пользователем и Internet. При использовании прокси-сервера в настройках браузера необходимо указать его IP-адрес или доменное имя. Информация о прокси-сервере может быть указана для взаимодействий по протоколу HTTP, а также для взаимодействия по таким протоколам, как FTP или Gopher. Средства настройки обладают достаточ-

ной гибкостью, чтобы дать возможность пользователям указать специфические домены (разделы в иерархии доменных имен в Internet), для которых прокси-сервер должен или не должен применяться. Например, браузер может быть настроен таким образом, чтобы прокси-серверу передавались только запросы на страницы из домена **cnn.com** (это все URL с суффиксом **cnn.com**). Запросы ко всем другим доменам будут направляться в обход прокси-сервера, что дает возможность пользователям просматривать сайты, избегая проблем, связанных с функционированием прокси-сервера.

Пользователи могут отключать ряд автоматических действий, выполняющихся при загрузке страницы. Например, загрузка страницы может вызвать загрузку в браузер и исполнение Java-апплета. Пользователь может отключить этот режим, сделав соответствующую настройку в браузере. Проблема при таком подходе состоит в том, что пользователи могут лишиться ряда функциональных возможностей, предоставляемых приложением, поскольку ряд популярных приложений в Web используют данную технологию. В браузере можно разрешить выполнение сценариев только для определенных приложений, таких как чтение групп новостей или электронной почты. В разделе 2.5 рассматриваются проблемы безопасности при загрузке ресурсов, связанные с тем, что некоторые действия могут инициироваться автоматически.

В некоторые новые версии популярных браузеров встроен механизм автоматической настройки, который сам устанавливает умалчиваемые значения различных семантических атрибутов. Например, возможно автоматическое указание прокси-сервера на основе IP-адреса клиента путем загрузки сценария с Web-сервера и установки значений по умолчанию.

2.4.3. Настройка в браузере функций, не связанных с протоколами

Браузер может использовать несколько вспомогательных программ для обработки ответов, выбор и режим функционирования таких программ обычно настраиваются. Если ответом на запрос является текстовый документ или изображение, браузеру известно, как отображать ответ. Однако некоторые ответы требуют использования *вспомогательных* приложений для интерпретации ответа. Например, представим себе, что пользователь хочет загрузить документ в формате Portable Document Format (PDF) либо в формате PostScript (PS). Одной из возможностей для браузера является загрузка содержимого, сохранение документа в локальном файле и предоставление пользователю возможности вызвать отдельную программу для отображения содержимого. Однако браузер может инициировать вспомогательную программу, которая объединяет действия по сохранению загруженного ресурса и отображению содержания. Например, браузер может вызвать *acroread* — популярную программу, способную отображать PDF-файлы. Точно так же браузер может вызвать программу *ghostview* для отображения файлов PostScript. Решение вызвать определенное вспомогательное приложение принимается либо в результате анализа расширения файла ресурса (**.pdf** для PDF-файлов и **.ps** для файлов PostScript), либо в результате анализа информации о типе содержания ресурса, имеющейся в заголовке ответа. Взаимосвязь между типом ресурса и вспомогательным приложением настраивается отдельно.

Таблица 2.3. Вспомогательные приложения, запускаемые в зависимости от типа файла/содержания

Тип содержания	MIME-тип	Вспомогательное приложение
Данные, сжатые с помощью Zip	application/x-zip-compressed	*gunzip*/WINZIP32
Документ PostScript	application/postscript	*ghostview*/GSVIEW32
Документ Word	application/msword	*catdoc*/WINWORD
Документ PDF	application/pdf	*acroread*/ACRORD32
Аудио/видео	video/x-mpeg-2	*raplayer*/MPLAYER2

В таблице 2.3 приведены несколько стандартных типов содержания файлов, распознаваемых браузером, и запускаемые в ответ вспомогательные приложения. Второй столбец описывает MIME-тип документа, т.е. стандартный способ представления типа документа. Типы документов кодируются с помощью так называемых многоцелевых расширений электронной почты (MIME — Multipurpose Internet Mail Extensions). В третьем столбце содержатся примеры широко известных программ UNIX и Windows, которые служат в качестве вспомогательных приложений.

Как показано в таблице 2.3, если MIME-типом ресурса является **application/postscript**, для просмотра загруженного документа PostScript вызывается программа *ghostview*. Программа *ghostview* открывает отдельное окно и имеет собственный пользовательский интерфейс, позволяющий изменить увеличение документа или распечатать его содержимое. Вспомогательное приложение, таким образом, способно расширить базовые функциональные возможности Web-браузера. По мере появления новых MIME-типов будут создаваться новые вспомогательные приложения, которые дадут возможность пользователю взаимодействовать с содержимым. Браузеры будут продолжать оставаться основным интерфейсом для загрузки содержания и запуска соответствующих вспомогательных приложений.

Возьмем другое популярное приложение в Web — загрузку и воспроизведение мультимедийных данных (например, аудио и видео). Вместо того чтобы встраивать средства для интерпретации мультимедиа в браузер, последний просто активизирует соответствующее приложение-проигрыватель. На рис. 2.3 показано, как это осуществляется.

Пользователь выбирает ресурс **http://www.bar.com/foo.ra**, а браузер передает HTTP-запрос исходному серверу **www.bar.com** на ресурс **/foo.ra** (этап 1). Исходный сервер отправляет обратно HTTP-ответ (этап 2), но содержимым ответа является лишь указатель на информацию. Ответ предназначен для вспомогательного приложения, а не для Web-браузера. Обычно ответом является URL, такой как **pnm://ra-ms.com/foo.ra**, где **pnm** означает протокол «Progressive Networks Media», а **ra-ms.com** представляет собой имя сервера, на котором размещен ресурс **/foo.ra**.

Поскольку браузер был настроен для вызова вспомогательной программы, то исходя из типа файла, он вызывает клиентскую программу **real-audio**, которая связывается с мультимедийным сервером **ra-ms.com** (этап 3) и начинает загрузку аудио. Клиентская программа для воспроизведения аудио на пользовательском компьютере может открыть дополнительные окна для регулировки громкости или других параметров звука. После загрузки HTTP-ответа, содержащего только URL **pnm://ra-ms.com/foo**, заканчивается часть оригинального запроса, которая относится к **http://www.bar.com/foo.ra**. Браузер активизирует аудиоклиент, который

самостоятельно осуществляет связь с мультимедийным сервером. Аудиоклиент имеет свой собственный пользовательский интерфейс (регулировка громкости, пауза, перемотка и т.д.) и может интерпретировать данные, полученные с сервера. Аудиоклиент может использовать другой протокол (отличный от HTTP) для получения данных. Таким образом, браузер служит для взаимодействия с Web-ресурсом **http://www.bar.com/foo.ra**, а аудиоклиент принимает управление от браузера для выполнения задачи загрузки и воспроизведения аудио.

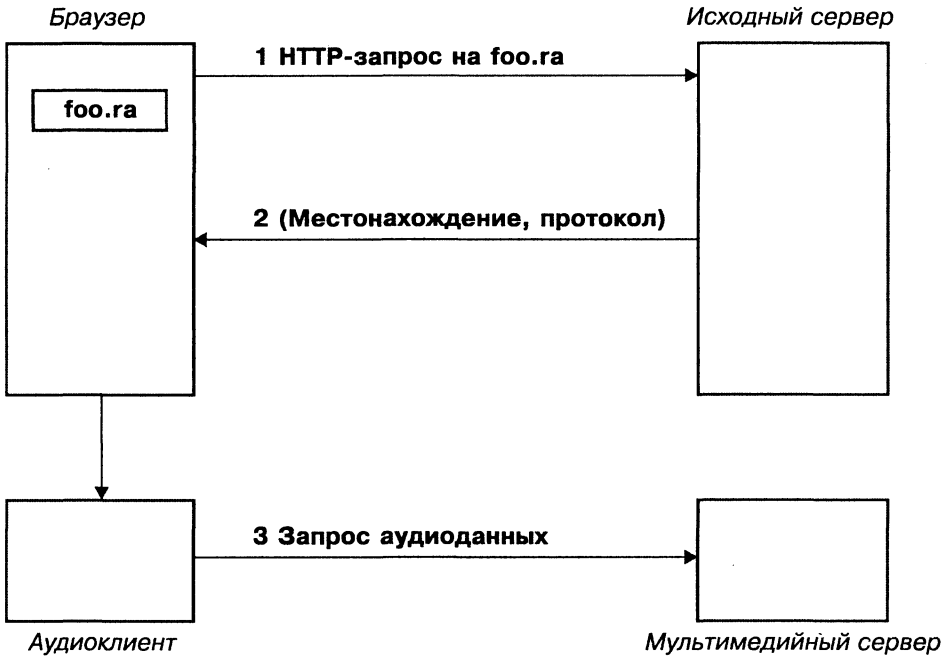


Рис. 2.3. Воспроизведения аудио

Подключаемые модули (plug-ins) представляют собой фрагменты кода, предназначенные для интерпретации определенных видов данных браузером. В отличие от сценариев, интерпретируемых браузером, подключаемые модули обычно безопасны. Некоторые подключаемые модули могут нести в себе вирусы, хотя на практике через подключаемые модули распространяется лишь малое число вирусов, поскольку они должны устанавливаться с ведома пользователя. Примером популярного подключаемого модуля является *Shockwave*, который необходим браузеру для отображения некоторых видов анимации.

Не все ответы могут быть обработаны обработчиками в браузере. С некоторыми приложениями приходится взаимодействовать по протоколам, отличным от HTTP. Способность браузера использовать различные протоколы позволила ему стать универсальным пользовательским интерфейсом для широкого набора приложений.

2.5. Вопросы безопасности при работе с браузером

Браузер представляет собой интерфейс между пользователем и Web и по мере необходимости отправляет информацию о пользователе, позволяя при этом настраивать параметры безопасности. В то же время браузер следит за тем, чтобы удаленным программам не предоставлялся несанкционированный доступ к компьютеру пользователя.

Запуск программы без четкого представления ее функциональных возможностей несет в себе риск нарушить систему безопасности. Первоначально Web-браузер представлял собой относительно простую программу. Сейчас же исходный код браузера содержит десятки тысяч строк, что увеличивает риск нарушения защиты. Годы, потраченные на изучение проблем безопасности, показали, что труднее всего обеспечить безопасность, если применяется сложное программное обеспечение. Браузер используется *главным образом* для доступа к различным сайтам Web. При этом каждый сайт использует другой сложный программный компонент (Web-сервер), что также снижает безопасность.

Проблемы безопасности в браузере главным образом связаны с доступом к дисковой памяти и к вычислительным ресурсам компьютера. Хотя многие документы, загружаемые в компьютер пользователя, являются статическими документами, состоящими из текста и изображений, растет число документов, содержащих исполняемый код Java, Javascript, Visual Basic Script, Perl и т.д., а также компоненты ActiveX. Ряд проблем, связанных с безопасностью, достаточно широко освещается в технической литературе и в популярных изданиях. Некоторые из этих проблем очень серьезные, а за последние несколько лет были выявлены новые проблемы. Угрозы безопасности могут варьироваться от несанкционированного доступа к пользовательским файлам до более серьезных проблем, таких как незаконное использование вычислительных ресурсов компьютера пользователя. Обновление версий Netscape Navigator и Internet Explore во многом связано с проблемами безопасности. Хотя пользователям не обязательно разрешать автоматическое выполнение кода на их компьютерах, в ряде браузеров допустимы некоторые или все возможности выполнения загружаемого программного кода. Пользователь может изменить параметры поведения по умолчанию. Для более глубокого рассмотрения проблем безопасности в Web читателю можно посоветовать обратиться к другим книгам по данной проблеме [Ste98a, RGR97].

Если компьютер пользователя не подключен к локальной сети и тем самым не связан с другими компьютерами, при загрузке программ из Internet опасности подвергаются лишь вычислительные и файловые ресурсы компьютера пользователя. Если же пользовательский компьютер подключен к локальной сети, а также имеются совместно используемые файловые или другие ресурсы, риску подвержены и другие компьютеры. Даже нормальная работа компьютера может быть нарушена в результате воздействия злоумышленников, осуществляющих атаку *отказа от обслуживания*. При этом атакуемый компьютер использует все ресурсы процессора, дисковой подсистемы для взаимодействия с атакующей программой, ресурсов на обслуживание каких-либо других пользователей просто не остается. Кроме того, могут заниматься ресурсы процессора на компьютере пользователя, либо организовываться передача файлов по электронной почте с пользовательского компьютера.

Ниже мы кратко рассмотрим риски для системы безопасности, связанные с применением некоторых популярных в Web языков программирования.

Java. Java является одной из первых систем программирования, дающих возможность выполнения программ на компьютере пользователя в результате загрузки ресурса извне. Java относительно безопасна из-за принятого в Java принципа выполнения загруженных из Web программ в специальной среде, называемой «песочницей», используемой для ограничения доступа к ресурсам пользовательского компьютера. *Апплет* Java представляет собой небольшое приложение, созданное для выполнения ограниченной задачи с ограниченным доступом к пользовательским ресурсам. Апплетам Java, загружаемым пользовательским компьютером, обычно не разрешается осуществлять операции чтения или записи в локальной файловой системы. Подход с использованием интерпретируемого байт-кода в Java уменьшает возможность доступа к ресурсам компьютера. Однако несмотря на ограниченные возможности доступа апплетов Java, исследования показали, что ненадлежащее использование протоколов Internet может привести к появлению брешей в системе безопасности. Хотя апплет не способен удалить файл на компьютере пользователя, он может инициировать атаку отказа от обслуживания, путем использования практически всех вычислительных ресурсов пользовательского компьютера.

JavaScript. JavaScript гораздо проще Java и используется в HTML-документах. Большинство браузеров могут выполнять сценарии JavaScript. Функции JavaScript синтаксически анализируются браузером, после чего инициируются определенные действия. Проблемы безопасности связаны со способностью JavaScript выполнять команды без ведома пользователя. Например, загруженный сценарий JavaScript может отправлять сообщения электронной почты. Пользовательские файлы являются доступными для сценариев, поэтому произвольное выполнение кода JavaScript, если только пользователь не уверен, что выполнение сценариев не несет опасности, является рискованным. Браузеры допускают настройку, в соответствии с которой выполнение сценариев JavaScript разрешается только для определенных приложений.

ActiveX. Элементы управления ActiveX схожи с апплетами Java. Элементы ActiveX не используют «песочницу»; вместо этого они применяют механизм *сертификатов*. Сертификаты представляют собой заверенные свидетельства, удостоверяющие, что данное программное обеспечение создано определенной организацией или разработчиком. Если сертифицированный элемент управления несет в себе риск нарушения системы безопасности, сертификат может быть отозван, а компания, создавшая данный элемент управления, может не получить сертификата в дальнейшем. Модель сертификации требует участия специальных *доверенных* организаций, осуществляющих выдачу сертификатов. Хорошо себя зарекомендовавший производитель программного обеспечения может, тем не менее, непредумышленно выпустить программное обеспечение, способствующее появлению брешей в системе безопасности. Подход с применением сертификатов может быть также использован для программ на Java и JavaScript.

2.6. Cookies

HTTP не сохраняет свое состояние, поэтому Web-серверу не нужно хранить какую-либо информацию о прошлых или будущих запросах. Однако на стороне сервера может иметься существенная причина для сохранения информации о состоянии между запросами в ходе сеанса или даже между сеансами. Например, может оказаться необходимым предоставлять доступ к ряду страниц на сервере только

определенной группе пользователей. Если ввод идентификационной информации необходим при каждом обращении пользователя к любой из этих страниц, возникает дополнительная нагрузка как на пользователя (ему необходимо вводить эту информацию), так и на сервер (для обработки этой информации). Дополнительные транзакции также снижают пропускную способность сети. Сохранение определенной информации между запросами сокращает непроизводительные затраты для пользователя, сети и сервера. Браузер играет важную роль в предоставлении необходимой информации о состоянии вместе с пользовательским запросом.

Cookies являются одним из средств сохранения состояния HTTP [MF00]. Cookie [Netd, KM00] — это информация о состоянии, которая передается Web-сервером браузеру и хранится на компьютере пользователя в интересах сервера. Механизм работы с информацией о состоянии HTTP дает возможность клиентам и серверам сохранять информацию за пределами запроса и ответа на него. Cookies впервые были применены Netscape в 1994 г. [Netd]. Затем организацией Internet Engineering Task Force (IETF) был начат процесс стандартизации. Механизм работы с состоянием HTTP, формализующий использование cookies, подробно описан в документе RFC 2965 [KM00], выпущенном в октябре 2000 г. и представляющем собой предложение по стандарту (Proposed Standard) IETF. RFC 2965 отражает опыт различных реализаций.

В этом разделе мы сначала остановимся на причинах использования cookies. Далее будет исследован механизм использования cookies в браузере. Cookies имеют весьма важное значение в контексте обеспечения конфиденциальности пользователя. В этой связи в научном сообществе имеются различные мнения. Раздел заканчивается рассмотрением различных проблем нарушения конфиденциальности, связанных с cookies. Роль приложений на стороне сервера в создании, передаче и использовании cookies обсуждается в главе 4 (раздел 4.2.4).

2.6.1. Причины использования cookies

Сервер передает cookie браузеру вместе со своим ответом, требуя от браузера включать cookie в последующие запросы к серверу. Когда пользователь в следующий раз посещает Web-сайт, браузер включает информацию из cookie в заголовок запроса. Таким образом, Web-сервер способен различать пользователей в ходе сеанса и между различными сеансами. Информация, отправленная в cookie, может быть уникальной для каждого посетителя Web-сайта, что создает условие для индивидуального обслуживания пользователей.

Многие Web-сайты (например, The New York Times, **http://www.nytimes.com**) требуют, чтобы cookies использовались браузером при загрузке страниц. Сайты могут требовать, чтобы пользователи идентифицировали себя именами и паролями. Если это необходимо делать при каждой загрузке страниц с этого сайта, пользователь вряд ли захочет работать с таким сайтом. Вместо этого необходимая идентификационная информация может передаваться автоматически через cookie.

Типичный пример использования cookie — «магазинная тележка», в которую пользователь помещает купленные им товары. Имеются Web-сайты, на которых пользователи могут выбирать товары, которые они планируют купить, например, книги или компакт-диски. По мере выбора пользователем товаров виртуальная магазинная тележка содержит множество выбранных на данный момент товаров.

Без cookies Web-серверу пришлось бы сохранять состояние всех своих пользователей (их может быть тысячи) на своем компьютере в течение длительного периода времени. Содержимое магазинной тележки или перечень приобретенных

в последнее время товаров являются примерами информации о состоянии. В других случаях может сохраняться более подробная информация, например, все купленные пользователем товары или страницы, которые пользователь посетил в ходе последнего сеанса. Реальное состояние не обязательно сохраняется в cookies полностью; cookies часто используются в качестве индекса для базы данных, в которой Web-сервером сохраняется информация о состоянии пользователя. Сохранение информации о пользователе дает возможность приложению совместно использовать ее с другими приложениями, в том числе с другими серверами. Подобное совместное использование информации без ведома пользователя ведет к угрозе конфиденциальности пользователя.

2.6.2. Использование cookies в браузере

На рис. 2.4 представлен клиент, отправляющий запрос исходному серверу A (этап 1). Исходный сервер в ответ включает заголовок (**Set-Cookie**) со значением cookie (**XYZ**) (этап 2). Во все последующие запросы к исходному серверу A клиент включает cookie (этап 3, передача **Cookie** с запросом в заголовке).

Заметим, что клиент не интерпретирует строку cookie (XYZ) при ее сохранении и включении в последующие запросы. Сервер свободен в построении строки, предназначенной клиенту, сформировавшему запрос, и в изменении механизма построения строки cookie. На рисунке также показано, что обмен cookies фактически осуществляется без ведома пользователя, если только пользователь не потребовал уведомлять его каждый раз при передаче cookie. Такое уведомление мешает работе пользователя, поэтому лишь малая часть пользователей, осведомленных об этой возможности, применяет ее на практике.

Cookies первоначально хранятся в оперативной памяти компьютера пользователя и записываются на долговременное устройство хранения (файлы на диске) при выходе из браузера. В соответствии с указаниями по реализации в [KM00] агенты пользователя могут хранить часто используемые cookies столько, сколько нужно, однако существует ряд ограничений. Cookie может иметь размер до 4 Кбайтов. Браузеры дают возможность сохранять максимум 20 cookies на сервер (или домен) и не более 300 cookies, чтобы избежать перегрузки.

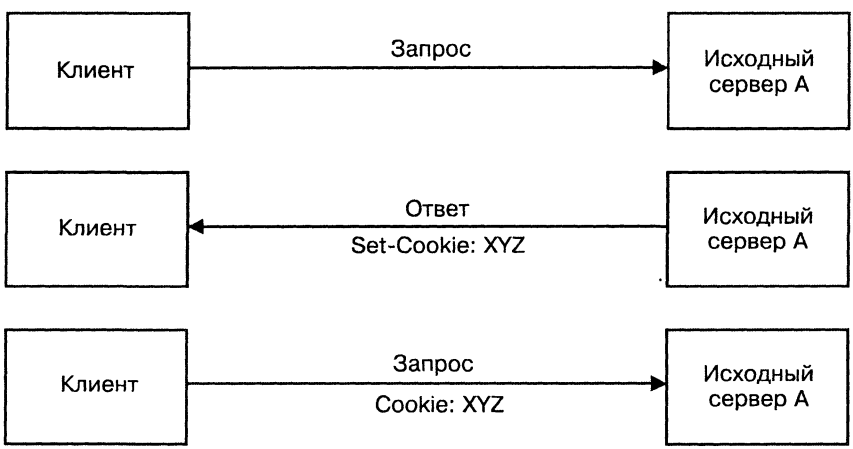


Рис. 2.4. Обмен cookies между клиентом и сервером

2.6.3. Контроль пользователя над cookies

Как и другие семантические свойства, управление которыми может осуществляться в браузере (см. раздел 2.4.2), cookies допускают значительную степень контроля со стороны пользователя. Пользователи могут:

- **Решать, принимать ли какие-либо cookies вообще.** Подобное решение может сделать для пользователя невозможным загрузку страниц с некоторых сайтов.
- **Устанавливать ограничения на размер и количество принимаемых cookies.** Тем самым пользователь управляет пространством, которое будет выделено для cookies на его компьютере, и уменьшает вероятность размещения произвольно больших cookies.
- **Решать, принимать ли cookies от всех сайтов, или только от определенных сайтов/доменов:** Это дает возможность пользователю принимать cookies с нужных сайтов и отклонять cookies с других сайтов.
- **Ограничить время жизни cookies только данным сеансом.** Более детальное управление на уровне сеанса дает возможность пользователю разрешать прием cookies только для выполнения определенной задачи. В конце сеанса осуществляется возврат к режиму по умолчанию, в соответствии с которым cookies для последующих сеансов не принимаются.
- **Потребовать, чтобы cookies исходили от того же сервера, с которого была получена текущая страница.** Тем самым гарантируется, что пользователь будет знать, откуда поступили cookies, а другим сайтам, с которыми браузер мог автоматически связаться, будет запрещено посылать cookies. Например, когда браузер загружает контейнерный документ, встроенные изображения могут извлекаться автоматически. Встроенные изображения могут размещаться на сервере, отличном от того, на котором находится контейнерный документ.

2.6.4. Проблемы нарушения конфиденциальности, связанные с cookies

Пожалуй, ни одна другая технология не вызывала столько разногласий, сколько cookies. Cookies широко используются, но в то же время считается, что они могут привести к нарушению конфиденциальности пользователя. Прежде всего, если применение cookies разрешено браузером по умолчанию, многие пользователи могут даже не подозревать, что они используют cookies. Cookies передаются *открытым текстом* — другими словами, в незашифрованном виде. «Подслушивающая» программа, способная перехватывать сетевой трафик, может узнать содержимое cookies. Создается возможность модификации информации cookies по мере прохождения пакетов по сети. Таким образом, конфиденциальную информацию не следует передавать открытым текстом в составе cookies, если только между клиентом и сервером не установлено защищенное коммуникационное взаимодействие. Помимо раскрытия личной информации пользователя, модификация значений cookies может также привести к изменениям на стороне сервера. Исходный сервер может использовать фрагменты cookies в качестве индексов для обращения к внутренней базе данных, и изменение cookies может привести к непредумышленной модификации базы данных без вмешательства со стороны пользователя. Многие пользователи не знают, кто имеет доступ к информации, извлеченной из cookies, и для чего используется эта информация. Например, информация из cookies может совместно использоваться компаниями, а относящиеся к пользователю данные (профили) могут быть незаконно использованы.

Проблема еще больше усугубляется, если информация из cookies передается серверу, отличному от исходного сервера, которому изначально был послан запрос. Сбор информации третьей стороной может привести к сложно обнаруживаемым и серьезным проблемам с пользовательской безопасностью. Например, агент пользователя U загружает страницу со встроенными изображениями с сервера S. Если встроенные изображения находятся на другом сервере, скажем E, браузер посылает запросы на сервер E, который может передать cookies. Теперь браузер будет иметь cookies и от S, и от E, хотя пользователь явно не обращался к серверу E. Механизм управления промежуточным состоянием HTTP требует, чтобы агент пользователя прекратил прием и отправку cookies в случае *не поддающейся проверке* транзакции, т.е. пользователю не предоставляется возможность просмотра URL, запрошенного от его имени. Другими словами, запрос к URL делается без ведома пользователя, и cookies принимаются с сайта, который напрямую пользователем не посещался.

Рассмотрим пример. Пусть компания DoubleClick занимается сбором информации. Предположим, пользователь отправляет запрос на поиск информации в популярную поисковую систему и получает десять гиперссылок на страницы с искомой информацией. URL на эти страницы будут не в виде **http://www.foo1.com/foo.html**, **http://www.foo2.com/bar.html** и т.д. Вместо этого они могут иметь следующий вид:

```
http://ad.doubleclick.net/x26362d2/www.foo1.com/foo.html
```

Теперь, когда пользователь щелкает мышью на одной из гиперссылок, запрос посылается на сервер **ad.doubleclick.net**. Сервер **ad.doubleclick.net** записывает информацию из cookie пользователя. Информация из cookie связывает два компонента: пользователя, отправившего запрос, и посещенный им сайт **www.foo1.com**. Далее запрос переадресовывается на сервер **www.foo1.com**. Переадресация представляет собой вид HTTP-ответа, который инструктирует браузер отправить другой запрос с иным URL. Переадресация запросов осуществляется *прозрачно* для большинства пользователей. Если различные серверы, которые пользуются услугами DoubleClick, сотрудничают между собой и совместно используют информацию из cookies одного и того же пользователя, они могут составить себе более подробный «портрет» пользователя.

Пользователь не может управлять использованием собранной из cookies информации. Было бы полезным при взаимодействии агента пользователя и сервера уведомлять пользователя и запрашивать у него разрешение перед использованием информации. Подобный подход называется моделью *подтвержденного участия (opt-in)*; т.е. пользователи явным образом соглашаются предоставить информацию о себе. Альтернативой является схема *уклонения от участия (opt-out)*, согласно которой поставщики содержания предоставляют пользователям возможность исключения их личных данных из собранной информации. Несмотря на то, что информация о пользователях не собирается автоматически, большинство пользователей не уклоняются от участия в сборе информации о себе. Причинами здесь являются инертность и отсутствие технической грамотности при слепом следовании инструкциям.

Основная проблема с cookies состоит в том, что многие пользователи просто не имеют понятия, какие действия выполняются от их имени. Даже если им известно о наличии cookies, они могут не знать, что можно запретить работу с cookies или использовать их избирательно. Ряд сайтов предоставляет подробную информацию о проблемах, связанных с cookies, и возможном некорректном использовании cookies. Читатели могут получить дополнительную информацию из других источников [Jun, Muf, Coo].

2.7. Спайдеры

Спайдер (spider) — это программа, используемая для получения некоторых или всех ресурсов с большого числа Web-сайтов. На первых порах спайдеры [Fic94] использовались в качестве вспомогательного средства при обслуживании Web-сайтов. В настоящее время спайдеры в основном используются для сбора информации в интересах поисковых систем. До сих пор мы обсуждали роль Web-браузеров — наиболее распространенного типа Web-клиентов. Поиск информации в Web также остается одним из популярных приложений, что и явилось причиной для создания таких инструментов, как спайдеры. Далее мы подробно рассмотрим клиенты-спайдеры и поговорим об их использовании в поисковых системах.

В главе 1 мы говорили о роли, которую играли предшествующие Web-системы в оказании помощи пользователям при поиске интересующих их документов в Internet. Эти системы имели каталоги, служившие в качестве предметных указателей для наборов документов. Gopher и WAIS имели дополнительные возможности для индексирования документов, хотя при этом каждый сайт должен был регистрировать свое содержимое на центральных серверах.

2.7.1. Поиск в Web

Системы, связанные с Web, такие как Gopher, были также ориентированы на поиск информации. Централизованный характер Gopher с его требованиями глобальной регистрации создали условия для всеобъемлющего поиска, однако необходимость обслуживания центрального реестра стало серьезной помехой в достижении успеха. С самого начала поиск активно использовался приложениями в Web. При быстром увеличении числа Web-сайтов, пользователей и Web-страниц, отсутствовала расширяемая технология, позволяющая отдельным пользователям или владельцам сайтов ориентироваться в быстро растущем наборе доступных документов. Децентрализованная суть Web сделала поиск сайтов и страниц необходимостью. Если пользователь ищет вхождения строки в небольшом файле, можно выполнить поиск по запросу. Выполнение же поиска в наборе из нескольких тысяч файлов потребует времени. Если же следует просмотреть десятки тысяч файлов, размещенных на тысячах компьютеров, задержка для пользователя будет еще больше. Наконец, если сотни тысяч пользователей ищут текстовые строки в десятках миллионов документов, размещенных на миллионах компьютеров, проблема становится значительно более сложной. Именно эту проблему призваны решать поисковые системы.

Один из способов существенно ускорить поиск — это иметь упорядоченный набор указателей вхождений искомых строк на их позиции в документах. Такой набор указателей называется *инвертированным индексом*. Например, предметный указатель в конце этой книги является подмножеством инвертированного индекса; он отсылает обратно к страницам, где встречаются указанные термины. Авторы могут принять решение не использовать все слова, которые встречаются в книге, в предметном указателе. Предметный указатель при этом получился бы слишком большим и не слишком полезным. Предметный указатель в книге указывает только на основные вхождения проиндексированных слов, в то время как инвертированный индекс обычно содержит все вхождения каждого из индексированных терминов. Слова, которые исключены из процесса индексирования, называются *сорными словами (stop words)*. В качестве примеров можно привести часто встречающиеся слова, такие как «the» и «end».

Первоначально, когда коллекция документов на удаленных сайтах была небольшой, имело смысл создавать локальные инвертированные индексы на каждом из удаленных сайтов. Однако с резким расширением Web создание локальных инвертированных индексов стало нецелесообразным. Лучшим выходом стало иметь несколько централизованных предметных указателей, которые могли бы использоваться миллионами пользователей. Для осуществления масштабного поиска необходимы два компонента: *спайдеры* и *поисковые системы*. В этом разделе мы познакомимся со спайдерами и увидим, как они используется совместно с поисковыми системами.

2.7.2. Клиент-спайдер

Спайдер — это ключевой инструмент для поиска в Web. Как упоминалось ранее, спайдер представляет собой программу, которая получает некоторые или все ресурсы с большого числа сайтов, главным образом с целью создания инвертированных индексов, которые позднее будут использоваться поисковыми приложениями. Подобно другим Web-клиентам, спайдер формирует HTTP-запросы для доступа к ресурсам Web-сайта и осуществляет синтаксический анализ ответов. Главными различиями между спайдером и браузером являются гораздо большее число сайтов, к которым осуществляется обращение и посылаются запросы, отсутствие какого-либо отображения ответов и достаточно необычное использование ответов.

На практике, однако, с сайтов может запрашиваться только часть ресурсов. Многие спайдеры, например, не запрашивают изображения или мультимедийные ресурсы. Это делается, если спайдер используется для построения индекса только текстовых ресурсов.

Допустим, сайт **www.kandrse.com** желает предоставить поисковый сервис, сходный с тем, который предоставляют популярные системы **Excite** и **AltaVista**. Сайт **www.kandrse.com** использует спайдер для загрузки страниц, подлежащих индексированию. Спайдер обычно начинается с базового списка популярных сайтов — известного как *начальный список (start-list)*, за которым следуют все URL, которые будут найдены в составе сайтов начального списка. В качестве примера начального списка можно привести перечень категорий, присутствующий на популярных сайтах, таких как Yahoo. Построение начального списка популярных сайтов и распределение задач представляют собой наиболее интересную техническую задачу при создании спайдеров.

Спайдер загружает начальную страницу сайта (например, **http://www.cnn.com/**) и просматривает все встроенные в нее гиперссылки. Для каждой гиперссылки он может загрузить соответствующую страницу, — это называется *обходом сначала в ширину (breadth-wise)*. Осуществляется переход по каждой гипертекстовой ссылке в *составе* сайта (т.е. гиперссылкам, имеющим суффикс **cnn.com/**, например, **http://www.cnn.com/nebraska.html** или **http://www.cnn.com/weatherpage.html**), а затем переходы по гиперссылкам на полученной странице и т.д. Есть и другой способ: спайдер может отобрать первую встроенную гиперссылку и, предполагая, что гиперссылка обращается также к HTML-документу, осуществить синтаксический анализ документа и перейти по первой встроенной гиперссылке (если она имеется). Подобный подход называется *обходом сначала в глубину (depth-wise)*. Здесь следует позаботиться о недопущении замкнутых циклов в результате перехода по гиперссылке, которая ранее уже была обработана. Множество страниц может иметь встроенные ссылки на одну и ту же страницу, либо страница в глубине иерархии сайта может иметь ссылку на страницу верхнего уровня. Например, страница

http://www.cnn.com/foreign/latvia/riga/opera/2000/schedule.html может иметь ссылку на **http://www.cnn.com**. Спайдер может также использовать другие алгоритмы для пропуска нежелательных страниц или сайтов; например, содержимое сайта может быть признано неподобающим. Многие спайдеры также принимают решение осуществлять индексирование лишь до определенного уровня глубины сайта. В результате сочетания обходов в ширину и в глубину могут быть собраны все ресурсы сайта. Время последнего посещения сайта записывается и используется при принятии решения о повторном его индексировании.

Спайдер может попытаться загрузить все ресурсы сайта за один раз, либо, что является более правильным, разбить эту задачу на отдельные части и выполнять ее в течение некоторого периода времени. В противном случае исходный сервер сайта может оказаться слишком загруженным при получении запросов от спайдера и не сможет обслуживать запросы от обычных пользователей. Большинство используемых в Web спайдеров не возвращаются к одному и тому же сайту чаще одного или двух раз в минуту. Спайдеры могут не загружать определенные в ресурсы в поисковую систему **www.kandrse.com**. Например, если сайт поисковой системы намеревается построить инвертированный индекс только текстовых ресурсов, загружать изображения нет надобности.

После того как сайт был проиндексирован, спайдер должен периодически повторно посещать сайт, поскольку содержимое последнего может изменяться. Однако содержимое одних сайтов может меняться не столь часто, как содержимое других сайтов. Спайдер должен обладать достаточным интеллектом, чтобы переиндексировать сайты в соответствии со скоростью изменения информации на них. Это позволяет сократить объем работы и избежать излишних обращений к сайту. Кроме того, некоторые составные части сайта могут меняться чаще, чем другие. Спайдер должен, таким образом, учитывать структуру информации на сайте и просматривать регулярно меняющиеся части более часто.

В то время как статические ресурсы могут быть достаточно легко проанализированы спайдером, динамические ресурсы обычно не индексируются. Ряд ресурсов сайта может динамически создаваться с помощью CGI-сценариев (см. главу 4, раздел 4.2.3). Сценарии часто используют параметры, а спайдер «не знает», какие значения параметров следует использовать при вызове. Основным назначением индексирования сайтов является возможность возвращать список гиперссылок на страницы в ответ на запрос. Таким образом, даже если бы это было возможно, в индексировании динамических ресурсов нет особого смысла. Если спайдер «замечает», что сайт состоит главным образом из динамических ресурсов, он может не индексировать статические страниц сайта, а также динамически генерируемые ресурсы, которые не изменяются в зависимости от параметров запросов. Иногда на сайте имеются ресурсы, которые недостижимы через ссылки извне. Спайдеры не знают о существовании таких ресурсов и не индексируют их.

Сайт может оказаться не в состоянии противодействовать индексированию себя спайдером. Хотя многим сайтам индексирование выгодно (многие создатели Web-страниц хотели бы, чтобы их страницы увидело как можно больше людей), некоторые сайты предпочитают, чтобы их «не беспокоили». Для разрешения этой проблемы существует ряд соглашений. Нет простого способа сделать так, чтобы ресурс был доступен любым клиентам кроме спайдеров, поскольку вряд ли можно различать клиентов, посещающих сайт. Каждый запрос, поступающий от клиента, рассматривается как независимый запрос, и серверам пришлось бы хранить значительный объем информации о состоянии, чтобы отслеживать частоту поступления запросов от определенных клиентов в надежде обнаружить спайдер. Кроме того,

подобный механизм может оказаться бесполезным, если спайдеры не посылают все свои запросы с одного IP-адреса, либо меняют частоту передачи запросов конкретному сайту. Даже если исходному серверу известно, что клиентом является спайдер, обеспечить селективный доступ может оказаться затруднительным, поскольку серверу придется проверять каждый входящий запрос, чтобы выяснить, не поступил ли он от спайдера. Подобное действие может увеличить время ожидания на стороне клиентов.

Существует два соглашения, которым обычно следуют сайты, чтобы каким-то образом контролировать работу индексирующих их спайдеров. Спайдеры должны иметь стимул вести себя надлежащим образом; в противном случае поисковая система может заслужить плохую репутацию. На уровне сайта это может быть реализовано следующим образом: администратор Web-сайта ведет файл с именем **robots.txt**. Робот — это одно из названий автоматизированного клиента (такого, как спайдер), а файл **robots.txt** содержит правила доступа, которым должны следовать роботы. Web-сайты используют этот файл в соответствии со стандартом Robot Exclusion Standard [RES]. Файл содержит список каталогов, которые спайдеры не должны посещать, а также спецификацию агентов пользователя, к которым эти ограничения применяются.

Например, рассмотрим следующий файл **robots.txt**:

```
User-agent: *
Disallow: /stats
Disallow: /cgi-bin/
Disallow: /Excite/
```

В нем указывается, что всем агентам пользователя разрешается загружать ресурсы с сайта для индексирования. Если в поле **User-agent** присутствуют одна или несколько строк (вместо "*"), например:

```
User-agent: ArachnoPhobia, BlackWidow
```

то два указанных агента пользователя распознаются как клиенты, которые не могут обращаться к сайту для индексирования. Web-сервер не осуществляет какой-либо явной проверки или действий по запрету доступа.

Строки **Disallow** используются для указания каталогов (**/states**, **/cgi-bin**, **/Excite**), которые не должны просматриваться программами-роботами, в том числе спайдерами. Опять-таки, сервер не может реализовать такие ограничения. Стандарт Robot Exclusion Standard [RES] определяет соглашения, которым должен следовать «добропорядочный» робот. Побудительным мотивом для роботов следовать стандарту является то, что сайтам часто известен набор ресурсов, которые нет смысла индексировать. В приведенном выше примере каталог **cgi-bin** скорее всего содержит группу ресурсов, которые при вызове с различными значениями параметров с большой вероятностью будут возвращать различные результаты.

Таблица 2.4. Некоторые поисковые системы

Поисковая система	Имя агента	Доменное имя спайдера
AltaVista (обычный спайдер)	Scooter/2.0 G.R.A.B. X2.0 Scooter/1.0 scooter@pa.dec.com	scooter.pa-x.dec.com scooter*.av.pa-x.dec.com
Euroseek	Arachnoidea (arachnoidea@euroseek.com)	*.euroseek.net (infra.euroseek.net)

Поисковая система	Имя агента	Доменное имя спайдера
Excite	ArchitextSpider	crawl*.atext.com
Google	BackRub/2.1	google.com
Inktomi	Slurp/2.0	*.inktomi.com
Infoseek	Infoseek Sidewinder/0.9	*.infoseek.com
Lycos	Lycos_Spider (T-Rex)	lycosidae.lycos.com
Northern Light	Gulliver/1.2	taz.northernlight.com

Второй способ сообщить роботам о том, какие ресурсы не следует индексировать — воспользоваться HTML-тегом META. Например, тег

```
<META NAME="ROBOTS" CONTENT="NOINDEX, NOFOLLOW">
```

информирует робота, что текущий ресурс не должен индексироваться *и* не следует осуществлять переходы по гиперссылкам в ресурсе. При синтаксическом анализе HTML-документа спайдеры просматривают значение атрибута CONTENT тега META с целью выяснения, могут ли они индексировать этот документ или осуществлять переходы по имеющимся в нем гиперссылкам.

В таблице 2.4 представлен список известных спайдеров [Spi], используемых некоторыми популярными поисковыми системами. В первом столбце указано название поисковой системы, во втором столбце — имя, присвоенное спайдеру, а в третьем столбце — доменное имя компьютера, на котором размещен спайдер. Информация предоставляется в виде идентификационных данных, которые Web-сайт может использовать, для обеспечения не слишком частого доступа спайдера к сайту. Большинство известных спайдеров следуют стандарту Robot Exclusion Standard. К главным отличиям между спайдерами относятся общее число посещаемых сайтов, общее количество индексируемых ресурсов и частота переиндексирования. Ряд современных спайдеров способны осуществлять индексирование миллиардов Web-страниц и делать их доступными для поиска. Сравнение эффективности различных поисковых систем можно найти в [LG99]. Большинство компаний не раскрывают информацию о своих спайдерах, поскольку среди поисковых систем имеется сильная конкуренция. Подробную информацию о расширяемом средстве сканирования Web с описанием его возможностей можно найти в [HN99].

2.7.3. Использование спайдеров в поисковых системах

Спайдеры помогают поисковым системам индексировать страницы Web-сайтов. В зависимости от сложности спайдера, размера начального списка и доступных спайдеру ресурсов, инвертированный индекс может быть создан для всех или для некоторых страниц. За последние 20 лет были созданы сложные алгоритмы построения предметных указателей для больших наборов документов [WMB99].

Сайты поисковых систем являются, пожалуй, наиболее популярными в Web. Сайты *порталов*, такие как **yahoo.com**, **excite.com, google.com** и **altavista.com**, являются отправными точками для множества приложений. О порталах подробнее рассказывается в главе 4, раздел 4.1. Практически все порталы имеют пользовательский интерфейс для поиска. Поисковая система предоставляет пользователю возможность для ввода одного или нескольких ключевых слов. Ключевые слова ищутся в инвертированном индексе, после чего возвращаются указатели на содержащие эти слова документы (если таковые имеются). Поиск может быть ограничен

одной базой данных (например, содержащей ссылки на группы новостей или сводки показателей стоимости акций), либо поиск осуществляется по всему индексированному содержимому Web. Многие пользователи тратят значительное время на поиск. Одной из причин этого является отсутствие хорошего единого предметного указателя для Web. К наиболее популярным сайтам относится **Yahoo!**, который индексирует сайты, но не содержимое всех страниц. Другими словами, для целей индексирования отбираются только основные страницы сайтов. Поисковые сайты, такие как **AltaVista** и **Google**, индексируют отдельные страницы в составе Web-сайтов. Таким образом, сайты, подобные **Yahoo!**, могут использоваться для поиска общей информации по теме, не вдаваясь в подробности, тогда как **AltaVista** и **Google** более результативны для поиска отдельных документов.

Список возвращенных документов (или указателей на документы) называется результирующим множеством. Поисковые системы различаются по уровню сложности. Большинство поисковых систем предоставляет простые функции поиска, с помощью которых в предметном указателе ищется одно или несколько ключевых слов и возвращаются указатели на документы, в которых найдено *любое* вхождение ключевого слова. Наличие нескольких ключевых слов интерпретируется как требование выполнения логического оператора *or (или)*. Поисковая система **Google** [Goo] предоставляет пользователям простой интерфейс. Искомые термины объединяются с помощью логического оператора *and (и)*, при этом возвращаются ссылки только на документы, содержащие *все* ключевые слова. Усовершенствованная версия поисковой системы **AltaVista** [Alt] и ее вариант **Raging [Rag]** имеют более сложный интерфейс: пользователи могут использовать любую комбинацию операторов *and (и)*, *or (или)*, *not (не)* и *near (около)*. Оператор *not (не)* представляет собой унарное отрицание, и документы, содержащие искомый термин, исключаются из результирующего множества. Оператор *near (около)* используется для задания расстояния между ключевыми словами в документе Путем сочетаний различных операторов можно получить результирующее множество, соответствующее требованиям пользователя и содержащее умеренное число документов.

Двумя основными показателями поисковых систем, выработанными в результате многолетних исследований, являются *полнота (recall)* и *точность (precision)*. Полнота оценивает широту охвата искомого множества, т.е. объем результирующего множества как функция от размера списка документов, в которых встречаются искомые ключевые слова. Если объем результирующего множества большой, пользователи могут быть уверены, что они получили полный ответ. К сожалению, принимая во внимание неоднозначность естественного языка, большинство «ответов» в результирующем множестве могут не устраивать пользователя.

Предположим, например, что пользователь хочет узнать имя создателя храма Парфенон в Греции. Строка поиска «создатель Парфенона» может вернуть множество страниц, относящихся к издательской компании «Парфенон», к греческим ресторанам под названием «Парфенон», а также к различным архитекторам. Показателем качества ответа является *точность* — уместность документов в данном результирующем множестве. Часто полнота и точность являются взаимоисключающими: возврат меньшего числа документов может увеличить точность, но полнота при этом уменьшится. Аналогично, более широкое результирующее множество в целом снижает точность. Противоречие между точностью и полнотой во многом остается неразрешимым, и многие поисковые системы тратят массу усилий, пытаясь повысить релевантность, возвращая, тем не менее, результирующее множество достаточно большого объема. Сложность запросов на естественном языке и отсутствие универсальных методов их обработки приводит к низкому качеству резуль-

татов. Подавляющее большинство запросов — на сегодняшний день, около 90% — к популярным поисковым системам, таким как **AltaVista** или **Google**, состоят из одного термина.

После выполнения поиска и формирования списка всех документов, в которых присутствуют ключевые слова, а также фильтрации по показателям полноты и точности с целью получения желаемого размера результирующего множества, поисковая система должна *ранжировать* результаты. Ранжирование — это определение порядка, в котором будут возвращаться результаты поиска. Если мощность множества результата для искомой строки «создатель Парфенон» составила 823, многие поисковые системы могут возвратить только первые 200 соответствий. Однако определение первых 200 соответствий требует ранжирования записей в результирующем множестве. Ранжирование по частоте вхождений искомых ключевых слов в документе не обязательно является хорошим показателем. Ранжирование на основе частоты вхождений ключевых слов является рискованной, поскольку некоторые авторы документов намеренно заполняют свои документы ключевыми словами, известными как «ловушки запросов». Например, поскольку значительное число запросов направлено на поиск материалов порнографического содержания (эта тема не слишком широко обсуждается в научной литературе), создатели таких Web-страниц добавляют на свои страницы множество наиболее распространенных ключевых слов. Достаточно легко можно сделать эти ключевые слова невидимыми при воспроизведении HTML-документа, установив для них нулевой размер шрифта или сделав так, чтобы текст сливался с фоном. Другими словами, эти ключевые слова воспринимаются только программными компонентами поисковых систем, осуществляющими ранжирование. Более интеллектуальные поисковые системы уменьшают значение частоты вхождений терминов при использовании его в качестве параметра для ранжировании. Хотя ранжирование — достаточно известная и очевидная концепция, используемая при поиске информации, исследования показали, что большинство пользователей просматривают лишь несколько элементов в результирующем множестве. На деле большинство популярных поисковых систем возвращает по 10 результатов за раз, хотя результирующее множество может содержать тысячи документов. Большинство пользователей обычно не идут дальше второй страницы результирующего множества; т.е. просматривается максимум 20 записей. Популярная поисковая система Google дает возможность пользователям непосредственно переходить к первому документу в ранжированном списке результатов в обход страницы с результатами поиска.

В последнее время в поисковых системах было реализовано несколько новых идей. Например, поисковые системы, такие как Google и Clever [Cle], использовали понятие *авторитетных (authoritative)* Web-страниц при ранжировании страниц для повышения релевантности ответов. Многие пользователи считают, что Web-сайт газеты New York Times является полезным сайтом и добавляют на свои собственные страницы ссылки на этот сайт. Количество ссылок на страницу является показателем как популярности, так и в некоторой степени доверия пользователей к информации на этой странице. Если набор страниц ранжирован на основе числа ссылок, указывающих на эти страницы, то страницы, количество ссылок на которые наибольшее, возвращаются пользователю в первую очередь. Подобная классификация использует неявно выраженную степень доверия к странице пользователей. Такая информация собирается спайдерами и используется для ранжирования результатов поиска. При этом существенно возрастает потенциальная релевантность результатов поиска, поскольку обычно люди не создают ссылки на мало полезные сайты. Программы разбиения сайтов на категории используют специфиче-

ские характеристики сайтов, такие как большое число маленьких изображений и наличие определенных ключевых слов. Поисковые системы могут учитывать категории сайтов и возвращать более качественные результаты поиска. Набор ключевых слов, введенных пользователем для поиска, может использоваться для определения, что же действительно ищет пользователь. Например, при поиске домашней страницы пользователя в качестве ключевого слова наиболее часто указывается имя пользователя. Таким образом, если выражение для поиска представляется схожим с именем пользователя, оптимальным при ранжировании будет помещение на первое место домашней страницы пользователя (если она существует).

Интервал времени между обработкой документа спайдером, составлением поисковой системой инвертированного индекса и использованием его при поиске может быть различным для различных поисковых систем. Обычно он находится в пределах от одной до двух недель. Если за это время документ изменяется, искомые ключевые слова могут больше не присутствовать в документе, когда пользователь осуществляет поиск. Что еще хуже, документ может быть удален с сайта. Web-сайт может быть недоступен во время проведения поиска. В действительности, принимая во внимание эти возможности, некоторые поисковые системы кэшируют копии искомых ресурсов во время индексирования. Пользователь может, по крайней мере, увидеть версию документа, имевшуюся на момент индексирования. Часто кэшируется только текст HTML, а встроенные изображения игнорируются.

Спайдер — довольно активно работающий клиент в смысле частоты и числа запросов. Он играет важную роль в одном из наиболее популярных приложений в Web — поиске. Совершенствование алгоритмов функционирования и эффективности спайдеров может оказать значительное влияние на Web.

2.8. Интеллектуальные агенты и браузеры специального назначения

После широкого распространения поисковых систем естественным продолжением развития стало появление программ, инициирующих поиск в интересах пользователей. Пользователи могут указать предпочитаемые ими поисковые системы и порядок ранжирования при настройке программы, настройки сохраняются в профиле пользователя. Подобные программы, называемые *агентами*, выполняют поиск, сопоставляют результаты и отображают их в соответствии с профилем пользователя. Взаимодействие между пользователем и поисковыми агентами отличается от взаимодействия между пользователем и браузером.

В этом разделе мы рассмотрим следующие вопросы:

- **Интеллектуальные агенты.** Эти агенты предназначены для решения специфических задач, таких как поиск, участие в аукционах и т.д.
- **Браузеры специального назначения.** Эти браузеры призваны осуществлять просмотр Web-страниц в автономном режиме, а также организовывать совместные действия пользователей по просмотру Web-содержания.

2.8.1. Интеллектуальные агенты

Интеллектуальные агенты работают от имени пользователей и пытаются предоставить эффективный сервис различным группам пользователей. Мы рассмотрим два вида интеллектуальных агентов:

- Системы метапоиска, которые инициируют поиск с помощью нескольких поисковых систем.
- Аукционные агенты, которые выполняют функции назначения цены в интересах пользователя.

СИСТЕМЫ МЕТАПОИСКА

Системы метапоиска передают указанные пользователем ключевые слова нескольким популярным поисковым системам. Результаты либо объединяются, либо группируются по поисковым системам. Система метапоиска дает возможность пользователю принимать решение, какой из алгоритмов ранжирования, принятых отдельными поисковыми системами, использовать, а также выбирать поисковые системы. Агент может отфильтровывать недоступные сайты, возвращенные в результатах поиска, либо недоступные страницы на доступных сайтах. Пользователь может сразу увидеть общие результаты для нескольких поисковых систем. Пользователь может выбирать результаты поиска, если они были высоко оценены более чем одной поисковой системой. По природе своей работы, система метапоиска должна ожидать ответов от нескольких поисковых систем, что вносит дополнительную задержку. Кроме того, при отправке запросов множеству поисковых систем сеть испытывает дополнительную нагрузку. Системы метапоиска не столь популярны, как обычные поисковые системы. Пользователи предпочитают применять привычные поисковые системы, вместо того, чтобы сортировать результаты, полученные от различных поисковых систем, имеющих свои собственные критерии ранжирования.

Агент может самостоятельно загружать страницы и представлять результаты не в виде указателей на страницы, а в виде реального содержания. Агенты могут продолжать поиск в автономном режиме и находить новые страницы, которые были созданы после того, как были заданы ключевые слова для поиска. Собранная информация может позднее стать доступной пользователю по электронной почте или получена при последующем обращении к агенту.

Системы метапоиска и другие поисковые агенты посылают обычные HTTP-запросы подобно любому другому клиенту за исключением того, что они представляют собой конкретные информационные запросы от имени одного или нескольких пользователей, а адресуются определенному набору исходных серверов. Такие агенты могут специальным образом организовывать свое соединение с этими исходными серверами и уметь осуществлять синтаксический анализ ответов. Агенты имеет более узкое применение, чем браузеры, но, тем не менее, реализуют базовые возможности Web-клиента.

АУКЦИОННЫЕ АГЕНТЫ

Поисковый агент выполняет действия в интересах пользователя весьма ограниченным и строго направленным способом. Поисковый агент передает запросы, получает результаты и, возможно, фильтрует их. Однако некоторые приложения требуют большей степени интеллектуальности и альтернативных действий — агенты могут реально *действовать* в интересах пользователей, например, назначая цену на

сайтах электронных аукционов. Аукционные сайты дают возможность пользователям назначать цену и ждать, чтобы посмотреть, было ли предложение цены принято на момент закрытия аукциона. В этом промежутке другие участники могут предлагать свою цену, что потребует от пользователя назначить более высокую цену, если он заинтересован в приобретении выставленного на аукцион товара. Поскольку у пользователя может не быть возможности постоянно реагировать на изменение ситуации и помнить время окончания торгов, будет полезно поручить выполнение этой задачи *аукционному агенту*, действующему в интересах пользователя. Аукционный агент не слепо исполняет отданные команды, а *реагирует* на ответы и предпринимает дополнительные действия. Пользователь может задать верхний предел цены. Следует сказать, что подобного рода инструментальные средства не слишком распространены.

Подобно поисковому агенту аукционный агент посылает соответствующим образом составленные запросы HTTP-клиента на аукционные сайты и выполняет синтаксический анализ ответов. Агент может быстро среагировать на основе полученного ответа и установленных пользователем ограничений, отправив дополнительные предложения цены. Часть программного обеспечения аукционного агента, относящаяся к Web-клиенту, является довольно простой, в то время как составляющая, выполняющая внутреннюю работу по синтаксическому анализу ответов и принимающая решения о способах реагирования, является более сложной.

2.8.2. Браузеры специального назначения

Браузер специального назначения представляет собой браузер, модифицированный для специального применения. Мы рассмотрим три вида браузеров специального назначения:

- **Кобраузер (co-browser).** Это браузер, помогающий пользователю осуществлять опережающую загрузку необходимых ресурсов.
- **Браузер для совместной работы нескольких пользователей (collaborative browser).** Расширение кобраузера, позволяющее группе пользователей совместно осуществлять просмотр Web-содержания.
- **Оффлайновый браузер.** Браузер, который дает возможность пользователю в автономном режиме (без подключения к сети) просматривать набор ранее загруженных страниц.

КОБРАУЗЕР

Кобраузер — это вспомогательная программа, которая при обращении к определенному ресурсу избирательно запрашивает другие взаимосвязанные ресурсы. Это делается в предположении, что пользователь, обратившийся к определенному ресурсу, может также быть заинтересованным и в других ресурсах. Программа ответственна за определение, какие ресурсы могут быть связаны с данным, и за создание ссылок между связанными ресурсами. В статическом варианте это может быть связанный с определенным ресурсом набор ссылок. В динамическом варианте кобраузер активно загружает связанные ресурсы и предоставляет их пользователю. Примером кобраузера может послужить система Letizia [Lie95], которая загружает ресурсы, идентифицируемые по гиперссылкам на странице в предположении, что пользователь вероятнее всего выберет одну из них в дальнейшем. В главе 13 (раздел 13.3) мы более подробно обсудим упреждающую загрузку ресурсов.

БРАУЗЕР ДЛЯ СОВМЕСТНОЙ РАБОТЫ НЕСКОЛЬКИХ ПОЛЬЗОВАТЕЛЕЙ

Агент может работать в интересах многих пользователей. Агент может помогать совместной работе нескольких пользователей над одной задачей. Браузер для совместной работы может вести наблюдение за действиями нескольких пользователей и пытаться выявить пользователей со схожими интересами. Например, если пользователь часто обращается к определенной электронной доске объявлений и проявляет интерес к небольшой группе музыкантов, агент может свести пользователя с другими пользователями, имеющими схожие вкусы. Подобная фильтрация содержания стала достаточно типичной в Web. Некоторые популярные Web-сайты информируют покупателей, купивших продукт, о связанных продуктах на основе действий других пользователей. Агент наблюдает за покупками пользователей, производит анализ и, когда ему удается выявить пользователей со схожими интересами, пользователь уведомляется о других товарах, приобретенных пользователями с похожими интересами.

Агент-браузер для совместной работы позволяет нескольким пользователям осуществлять совместный просмотр. Система Let's Browse [LDV99] представляет собой расширение кобраузера Letizia, упомянутого выше. Группа людей может осуществлять совместный просмотр, получая сведения об обращениях друг друга к ресурсам. Система может рекомендовать информацию о действиях одного пользователя другим пользователям в группе. После упреждающей загрузки группы ссылок в документе могут быть избирательно отфильтрованы на базе того, по каким ссылкам был осуществлен переход другими пользователями группы.

ОФФЛАЙНОВЫЙ БРАУЗЕР

Оффлайновый браузер — это программа, которая способна загружать полное содержимое одного или нескольких сайтов либо по требованию, либо на периодической основе. Оффлайновые браузеры способны связывать несколько целевых запросов в одну последовательность и загружать ответы на компьютер пользователя для просмотра сайта в автономном режиме, когда пользователь отключен от сети. Оффлайновые браузеры обычно начинают работу с заданного URL сайта и действуют как спайдер, за исключением того, что переходы осуществляются по всем гиперссылкам. Например, если содержимое сайта **www.site.com** должно быть загружено для последующего просмотра, оффлайновый браузер будет делать обход по всем встроенным гиперссылкам документов на **www.site.com**, но при условии, что все URL имеют префикс **www.site.com**. Если гиперссылка указывает на **www.offsite.com**, она будет проигнорирована. Осуществляя обход в ширину или в глубину, браузер может захватить все связанное содержимое сайта **www.site.com**. Сайт может быть воссоздан на стороне клиента с переименованием всех URL, чтобы обеспечить локальный просмотр сайта клиентом. Поскольку все содержимое сайта было скопировано на диск клиента, все Web-ссылки могут быть переименованы в статические файловые ссылки. Когда пользователь просматривает сайт в автономном режиме, файлы могут служить в качестве кэшированной копии ресурсов, не требуя какой-либо повторной проверки актуальности ресурсов путем обращения к исходному серверу.

2.9. Резюме

Клиент — это программное обеспечение общего назначения, а браузер — наиболее популярный вид клиента. Различные виды клиентов отличаются по принципам их функционирования и по способу практического использования. Взаимодействуя непосредственно с пользователем, клиент должен отвечать требованиям безопасности пользователя и снижать риск возможного раскрытия конфиденциальной информации.

Браузер загружает и отображает ресурсы в интересах пользователей. Браузер является графическим интерфейсом пользователя для работы в Web, и одно это уже делает его чрезвычайно популярным программным компонентом. В настоящее время широко используются только два Web-браузера: Netscape Navigator и Microsoft Internet Explorer.

Web является децентрализованной и постоянно меняющейся структурой, не имеющей центральной базы данных URL или соответствующего содержания. Спайдеры осуществляют просмотр Web-страниц и содействуют в поиске. Спайдеры осуществляют периодический обход Web-ресурсов с селективной выборкой, в результате чего формируется инвертированный индекс. Вследствие наличия временного лага между моментом индексирования ресурсов спайдером и поиском пользователями этих ресурсов, результаты поиска могут быть неактуальными. Семантическая точность текущей версии ресурса не может быть гарантирована — искомые пользователем ключевые слова могут уже не присутствовать в текущей версии ресурса. Все популярные поисковые системы, такие как **www.altavista.com** и **www.google.com**, используют спайдеры для поддержания актуальности своих предметных указателей.

Агенты действуют на основе ранее заданных ограничений и посылают меньше запросов, чем спайдеры. Агент отличается от спайдера большей направленностью и конкретностью действий. Спайдер менее «разборчив» в выборе ресурсов. В Web повседневно используется несколько популярных агентов, например, аукционные агенты.

3

Прокси-серверы

Традиционная модель коммуникационного взаимодействия клиент/сервер предусматривает передачу запросов от клиента серверу и ответов обратно клиенту. Между клиентом и сервером нет никаких промежуточных звеньев. Применительно к Web пользователи с помощью своих агентов посылают запросы Web-серверам, а ответы возвращаются непосредственно браузерам. Наличие промежуточного звена может уменьшить число нежелательных коммуникационных взаимодействий на обеих сторонах. Например, промежуточное звено (прокси-сервер, называемый также сервером-посредником) может иметь собственный кэш и доставлять ответы клиенту без обращения к исходному Web-серверу. Время ожидания, затрачиваемое клиентом, сокращается, если прокси-сервер находится ближе к клиенту. Сокращается также нагрузка на сеть, поскольку сообщению требуется пройти более короткий путь. В то же время прокси-сервер способен снизить загрузку Web-сервера, которому придется иметь дело с меньшим числом обращений. В качестве прокси-сервера может выступать программа, находящаяся на том же компьютере, либо на компьютере, отличном от того, на котором был сделан запрос.

Прокси-серверы используются и при работе и с другими протоколами, которые были популярны во времена создания Web. Например, протоколы File Transfer Protocol (FTP) и Network News Transfer Protocol (NNTP) предусматривают использование *прокси-серверов* — программ, которые действуют от имени группы клиентов при взаимодействии их с серверами. Эти прокси-серверы берут на себя часть работы, выполняемой серверами.

В этой главе мы обсудим роль, которую играют прокси-серверы, начиная с первого опыта их применения и заканчивая сегодняшним днем, когда они стали играть важную роль в результате взрывного увеличения Web-трафика. Прокси-сервер действует как сервер для клиента и как клиент для других прокси- или Web-серверов. Основная роль, которую играет прокси-сервер, состоит в том, что он является промежуточным интерфейсом для многих клиентов в их взаимодействии с другими серверами. В качестве внешнего интерфейса прокси-серверы могут обеспечивать совместный доступ и анонимность клиентов. Прокси-серверы могут присутствовать в различных точках пути между клиентами и Web-серверами. Они могут быть расположены близко к клиенту, к Web-серверу или находиться в любой промежуточной точке между ними. Число прокси-серверов определяет число Web-компонентов, через которые должны пройти сообщения. Находясь на пути запросов и ответов, прокси-серверы могут функционировать в качестве фильтров, определяя, какие запросы следует передать дальше и какие ответы возвращать клиентам. Прокси-серверы могут модифицировать запросы и ответы, проходящие через них. Например, прокси-серверы могут преобразовывать Web-запрос клиента и направлять его на FTP-сервер, а затем возвращать ответ клиенту как обычный HTTP-ответ.

Мы начнем с истории и эволюции промежуточных компонентов Web, включая прокси-серверы, шлюзы и туннели. Основное внимание в этой книге уделяется собственно прокси-серверам, поэтому в оставшейся части главы речь пойдет исключительно о них. Далее будет представлена укрупненная классификация прокси-серверов. Область применения прокси-серверов следующая: совместный доступ к ресурсам, кэширование ответов, преобразование и фильтрация запросов и ответов. После этого достаточно широкого обзора применения прокси-серверов мы сузим область рассмотрения до роли прокси-серверов применительно к протоколу HTTP. Эта роль заключается в обработке запросов и ответов HTTP, а также в решении специфических задач, когда он действует и как сервер, и как клиент. На пути между клиентом и Web-сервером может присутствовать несколько прокси-серверов, поэтому далее мы рассмотрим, как прокси-серверы взаимодействуют между собой. Связь может быть линейной, если прокси-серверы в группе последовательно соединены в цепочку. Группа прокси-серверов также может быть связана с одним прокси-сервером более высокого уровня, который может быть, в свою очередь, связан с другими прокси-серверами. Далее мы обсудим, как нужно настраивать клиенты для доступа к прокси-серверам. Будет рассмотрена роль прокси-серверов в обеспечении безопасности и конфиденциальности. В последнее время в Web приобретают популярность нетрадиционные прокси-серверы, такие как перехватывающие и обратные прокси-серверы. Мы познакомимся с такими прокси-серверами и узнаем, чем они отличаются от традиционных прокси-серверов. В заключение мы подытожим основные принципы функционирования прокси-серверов, а также их практического использования.

3.1. История и эволюция прокси-серверов

Хотя первоначально Web-сообщения передавались между пользовательским агентами и Web-серверами непосредственно, промежуточные компоненты быстро нашли применение. К трем наиболее распространенным промежуточным компонентам относятся собственно прокси-серверы, шлюзы и туннели. В этом разделе мы рассмотрим процесс эволюции этих компонентов-посредников.

Прокси-серверы стали играть важную роль уже в 1994 г. К этому привел рост популярности Web. В первом проекте документа, описывающего синтаксис унифицированных идентификаторов ресурсов (URI) [BL94], упоминалась роль прокси-сервера как *шлюза*, т.е. HTTP-сервера, через который проходят сообщения и где может быть осуществлено преобразование протоколов. В целях безопасности многим организациям не разрешалось иметь непосредственный выход в Internet. Вместо этого все коммуникационные потоки проходили через *межсетевой экран* или *брандмауэр* [BC95] — компьютер, который выполнял роль «стража». Трафик к месту назначения и от него, представляющийся небезопасным, отфильтровывался межсетевым экраном. Поскольку весь трафик в Internet и из Internet проходил через межсетевой экран, последний естественным образом исполнял функции прокси-сервера. Изначальным назначением прокси-сервера было предоставить доступ Web-клиентам, находящимся за межсетевым экраном организации [LA94]. Первоначальное назначение прокси-серверов состояло в том, чтобы клиенты не теряли каких-либо функциональных возможностей при передаче своих запросов и получении ответов через межсетевой экран. Самый первый прокси-сервер в действительности представлял собой шлюз в CERN (Европейский центр физики высоких энергий) [LA94], где впервые в мире был реализован Web-сервер. Роль прокси-серверов в сокрытии информации и инкапсуляции подробно описана в [Sha86].

В первом проекте документа [BLFN95], описывающем основные положения протокола HTTP/1.0, говорилось о роли прокси-серверов как промежуточных коммуникационных компонентов для различных других протоколов, таких как Simple Mail Transfer Protocol (SMTP) [Pos82], NNTP [KL86], FTP [PR85], Gopher [AML+93] и WAIS [PFG+94]. Эти протоколы являются основой таких популярных приложений, как электронная почта, группы новостей, передача файлов и т.д. Проект спецификации HTTP/1.0 [BLFN95] также содержал формальное определение прокси-сервера:

> Промежуточная программа, которая действует и как сервер, и как клиент для целей передачи запросов. Прокси-серверы часто используются для передачи сообщений через межсетевой экран. Прокси-сервер принимает запросы от клиентов и либо обрабатывает их самостоятельно, либо передает запросы (возможно, с преобразованием) другим серверам.

Это определение по большей части сохранено и в последующих спецификациях HTTP. В данной книге мы будем использовать именно это определение.

Итак, прокси-серверы, поначалу используемые в качестве шлюзов, в настоящее время стали составной частью пользовательского «восприятия» Web. Запрос от клиента может завершиться на прокси-сервере, если требуемый ответ содержится в кэше прокси-сервера. В противном случае запрос будет перенаправлен на другой сервер, который может быть конечным сервером в цепочке, так и не быть им. Конечный сервер фактически отвечает на клиентский запрос. На рис. 3.1 демонстрируется прокси-сервер, играющий роль клиента и сервера. Как показано на рисунке, клиент 1 запрашивает ресурс B. Запрос направляется прокси-сервером серверу 1, а ответ возвращается клиенту. Этот запрос не кэшируется. Запрос от клиента 2 на ресурс A направляется серверу 2, но ответ кэшируется на прокси-сервере и возвращается клиенту. Теперь, когда клиент 3 запрашивает ресурс A, запрос прокси-сервером не перенаправляется серверу 2, вместо этого клиенту 3 возвращается копия ресурса A из кэша.

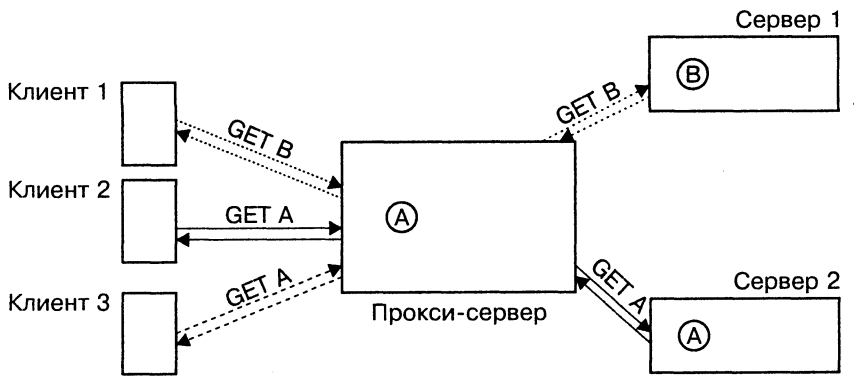


Рис. 3.1. Прокси-сервер в качестве промежуточного звена между клиентом и сервером

Прокси-сервер представляет собой не только промежуточное звено между отправителем и получателем HTTP-сообщения. Шлюз — это сервер, который обычно действует как передаточное звено для серверов, работающих с другими протоколами, например, серверов электронной почты или FTP-серверов. При направлении

сообщения такому серверу шлюз осуществляет преобразование HTTP-запроса к требуемому протоколу. Ответ возвращается шлюзу, который после выполнения соответствующего преобразования возвращает ответ отправителю запроса. Однако между прокси-сервером и шлюзом имеются различия. При использовании прокси-сервера отправителю следует знать, что прокси-сервер может либо сам ответить на запрос, либо переадресовать запрос. В случае использования шлюза имеет место несколько другая картина. Для отправителя шлюз действует как исходный сервер. Клиенту, обращающемуся с запросом, необязательно знать, что где-то на пути сообщения имеется шлюз. Шлюз к другому сервису, например, к электронной почте или группам новостей, должен принимать входящий запрос и преобразовывать его в формат, используемый сервисом, которому направляется запрос.

Туннель — это промежуточное звено, которое передает битовые данные между двумя точками соединения. Туннель, в отличие от прокси-сервера или шлюза, не выполняет синтаксический анализ и не интерпретирует HTTP-сообщение, передаваемое через него вслед за строкой запроса (первой строки HTTP-сообщения) для указания адреса компьютера, с которым необходимо установить соединение. Туннель переходит в активное состояние в начале сеанса взаимодействия. В активном состоянии туннель не воспринимается как часть HTTP-взаимодействия. Туннель активизируется в начале сеанса и в активном состоянии не считается частью коммуникационного взаимодействия, поскольку он не просматривает, не создает и не изменяет передаваемое информационное содержимое. Туннели не кэшируют ответы, тогда как шлюзы и прокси-серверы могут это делать. Еще более важно, что время существования туннеля, в отличие от прокси-сервера или шлюза, равно времени существования канала связи между конечными точками соединений. Если оба конца соединений закрыты, туннель прекращает свое существование.

В главе 7 (раздел 7.5) обсуждается концепция долговременных HTTP-соединений, сохраняющихся и после выполнения транзакции запрос–ответ. Прокси-серверы, шлюзы и туннели играют важную роль в поддержании долговременного соединения между клиентом и сервером, если они присутствуют на пути передачи сообщений. Все промежуточные компоненты должны удовлетворять правилам относительно передачи заголовков HTTP-сообщений. Прокси-серверы и шлюзы также нужны для того, чтобы скрывать определенную идентификационную информацию (например, IP-адреса отправителей) при использовании их для передачи сообщения через межсетевой экран. Аналогично последняя версия протокола HTTP, HTTP/1.1, требует, чтобы шлюзы и прокси-серверы идентифицировали себя в том случае, если они являются частью HTTP-транзакции. Промежуточные компоненты должны проявлять осторожность при корректировке тела сообщения. Например, сообщение может включать контрольную сумму для проверки целостности сообщения, которая может стать некорректной в случае изменения тела сообщения.

Хотя имеются только три крупные категории промежуточных компонентов, в этой главе, да и во всей остальной книге мы ограничимся рассмотрением только прокси-серверов. Прокси-сервер — это промежуточное звено в Web, которое получило достаточно широкое распространение. Шлюзы и туннели используются гораздо реже.

3.2. Высокоуровневая классификация прокси-серверов

По принципу работы прокси-серверы можно разделить по двум ключевым признакам. Во-первых, некоторые прокси-серверы имеют ассоциированные с ними кэши, а другие — нет. Во-вторых, вне зависимости от кэширования, некоторые прокси-серверы модифицируют проходящие через них сообщения, тогда как другие этого не делают.

3.2.1. Кэширующие прокси-серверы

Разница между обычными и кэширующими прокси-серверами весьма важна. Обычный прокси-сервер просто пересылает запросы и ответы. Кэширующий прокси-сервер способен поддерживать собственное хранилище ответов, полученных ранее. Когда прокси-сервер получает запрос, который может быть удовлетворен кэшированным ответом, запрос не пересылается, а ответ возвращается прокси-сервером. Как мы увидим далее в этой главе, для того, чтобы кэшированный ответ мог быть возвращен, должны быть выполнены определенные условия. Мы используем термин *кэширующий прокси-сервер* для прокси-сервера, который имеет ассоциированный с ним кэш.

3.2.2. Прозрачный прокси-сервер

По принципу передачи сообщений прокси-серверы можно разделить на две группы: прозрачные и непрозрачные. Различие между ними связано с модификацией проходящих через прокси-сервер сообщений. *Прозрачный прокси-сервер* модифицирует запрос или ответ лишь в меру необходимости. Примером такого изменения сообщения прозрачным прокси-сервером может служить добавление идентификационной информации о себе или сервере, от которого сообщение было получено. Подобная информация может даже являться обязательной для протокола HTTP. В разделе 3.8 мы поговорим о неправильном использовании термина «прозрачный прокси-сервер» в различных сферах Web-индустрии для обозначения прокси-серверов, которые правильнее было бы назвать перехватывающими прокси-серверами.

Непрозрачный прокси-сервер способен модифицировать запрос и/или ответы. Примером такого изменения запроса является анонимизация, в соответствие с которой информация о клиенте прокси-сервера скрывается. Примером изменения ответа может послужить преобразование формата — изображение конвертируется из одного формата в другой для уменьшения размера ответа. Другой пример непрозрачного прокси-сервера — прокси-сервер, осуществляющий перевод документа с одного языка на другой. Имеются правила, являющиеся общими для прокси-серверов обоих видов. В то же время с каждым видом прокси-серверов связаны свои собственные правила. Прозрачный прокси-сервер должен обеспечить, чтобы длина содержимого сообщения не изменялась при передаче сообщения через прокси-сервер. Заметим, что прозрачные и непрозрачные прокси-серверы отличаются от шлюзов и туннелей. Оба вида прокси-серверов могут, в отличие от туннелей, иметь ассоциированный с ними кэш. Оба вида прокси-серверов действуют как промежуточное звено между Web-клиентом и Web-сервером; т.е. обмен сообщениями осуществляется в формате HTTP.

3.3. Применение прокси-серверов

Ниже мы рассмотрим различные варианты использования прокси-серверов, помимо предоставления клиентам доступа к ресурсам в Web. Это совместное использование ресурсов, кэширование, анонимизация, трансформация запросов и/или ответов и, наконец, фильтрация запросов/ответов.

3.3.1. Совместный доступ к Web

Прокси-сервер действует как внешний интерфейс для группы клиентов и дает им возможность осуществлять совместный доступ к Web. Клиенты совместно осуществляют доступ к Internet через прокси-серверы и могут совместно использовать ресурсы. Если несколько клиентов запрашивают один и тот же ресурс с исходного сервера, между прокси-сервером и исходным сервером может быть установлено одно соединение. Альтернативой может служить установка отдельных соединений между различными клиентами и исходным сервером, что увеличивает нагрузку на исходный сервер. Однако в том случае, когда выполняются запросы на *различные* ресурсы, прокси-сервер может последовательно упорядочить их. Если имеется задержка в получении ответа для первого запроса, второй ответ также будет задержан. Каждое соединение между клиентом и прокси-сервером является в некотором смысле *локальным* соединением (т.е. расстояние в сети между ними небольшое), а более длинный маршрут между прокси-сервером и исходным сервером совместно используется всеми клиентами. Таким образом запросы и ответы, которые различаются для клиентов, проходят короткое расстояние, в то время как запросы/ответы, которые являются общими для всех клиентов, совместно используют более длинный путь между прокси-сервером и исходным сервером.

Многие провайдеры Internet и корпорации принудительно осуществляют доступ пользователей к Web через прокси-серверы. Хотя некоторым пользователям могут быть известны способы обхода прокси-серверов и у них может иметься разрешение делать это, большинство пользователей провайдеров Internet и поставщиков информационного содержимого (таких как America Online) осуществляют выход в Internet через прокси-серверы. Ряд таких прокси-серверов являются кэширующими прокси-серверами.

3.3.2. Кэширование ответов

Одной из главных задач прокси-серверов является кэширование. Кэширование представляет собой сохранение ранее полученного ответа для последующего использования, когда клиенты запрашивают один и тот же ресурс. Кэш возвращает ответ, если ответ все еще считается *актуальным* (т.е. исходный сервер санкционирует ответ, который и будет возвращен). Кэширование подробно рассматривается в главе 11. Функция кэширования для прокси-сервера является необязательной; т.е. прокси-сервер выполняет роль кэша в дополнение к его роли как сервера для клиентов, находящихся за ним, и клиента для исходных серверов. Многие прокси-серверы в Internet играют роль некэширующих прокси-серверов. Не все клиенты получают выигрыш от наличия кэширующего прокси-сервера. Например, если клиентом является программа, осуществляющая сканирование Web-ресурсов, не имеет смысла осуществлять запросы через кэш прокси-сервера, поскольку способы ее работы с Web-ресурсами существенно отличаются от поведения обычных клиентов. Для большинства ответов вероятность, что они снова будут востребованы,

в этом случае весьма мала. Принимая во внимание, что большинство прокси-серверов имеют ограниченный объем дискового пространства для кэширования ответов, следует избегать кэширования ответов на запросы, сделанные программами, осуществляющими сканирование Web-ресурсов.

3.3.3. Анонимизация клиентов

Прокси-серверы играют важную роль в анонимизации клиентов, находящихся за ними. Когда Web-запрос поступает на исходный сервер через прокси-сервер, исходный сервер воспринимает прокси-сервер как клиента, обращающегося с запросом. Когда исходный сервер пытается идентифицировать имя компьютера, с которого получен запрос, он обнаруживает прокси-сервер вместо клиентов, которые находятся за прокси-сервером. Учитывая, что за прокси-сервером могут иметься сотни клиентских компьютеров, у исходного сервера может отсутствовать возможность различать конкретных клиентов. Анонимизация доступа может иметь очень важное значение для определенных клиентов. Некоторые пользователи могут осуществлять доступ к сайтам, которые содержат конфиденциальную информацию (например, связанную с состоянием здоровья), и могут не захотеть раскрывать себя. Конечно, прокси-серверу известно, какой клиент осуществляет доступ к ресурсу, но никому другому это не известно, если только прокси-сервер не сообщит данную информацию. Обычно организации, поддерживающие прокси-серверы, располагают средствами управления ими. Поставщики услуг Internet могут обеспечить сохранение конфиденциальности и неразглашение информации о работе в Web отдельных клиентов.

Некоторые прокси-серверы могут быть явным образом настроены так, чтобы не обеспечивать анонимность и добавлять заголовок, указывающий на клиента, от имени которого пересылается сообщение. В общем случае, хотя прокси-серверы в целом обеспечивают анонимность, в запросе имеются, по крайней мере, два фрагмента информации, в которых имеются данные о пользователе:

- Первый фрагмент информации — это заголовок **User-Agent**, присутствующий в запросах, в которых идентифицируется браузер пользователя и, возможно, операционная система компьютера пользователя.
- Второй фрагмент информации — это данные, используемые для идентификации состояния сеанса клиента. Эти данные часто хранятся в cookies (о чем шла речь в разделе 2.6 главы 2) или в других полях идентификатора сеанса, явно устанавливаемых исходным сервером. Прокси-сервер передает запросы от многих пользователей и тем самым предоставляет для них некоторую степень анонимности. Исходным серверам не известны IP-адреса компьютеров клиентов, поскольку запросы приходят через прокси-сервера. Хотя cookies не обязательно раскрывают информацию о пользователе, они снижают степень анонимизации, обеспечиваемую прокси-сервером.

Обладая такой информацией, исходный сервер может легко отслеживать отдельных пользователей, несмотря на то, что они остаются за прокси-сервером. Если для исходного сервера доступна и другая информация о пользователе, исходный сервер может легко связать ее с информацией из cookies, чтобы уникально идентифицировать пользователя.

3.3.4. Преобразование ответов и запросов

Прокси-сервер может модифицировать запрос или ответ, а в некоторых случаях и ответ, и запрос. Клиенты могут информировать прокси-сервер о своих предпочтениях и возможностях своих собственных компьютеров или соединений. Прокси-сервер может попытаться адаптировать ответ к возможностям конкретного клиента. Например, пользователь, подключенный через соединение с низкой пропускной способностью, может захотеть получать ответы в сжатом виде с целью ускорения передачи. Прокси-сервер осуществляет модификацию избирательно, в зависимости от запросов и от ответов.

Прокси-сервер может также преобразовывать запрос вне зависимости от возможностей клиента. Прокси-сервер способен включать в запрос информацию, указывающую на его возможности. Например, прокси-сервер и исходный сервер могут иметь возможность применять определенный алгоритм сжатия, который значительно уменьшает размер тела запроса или ответа. Клиент не участвует в этой части обмена сообщениями, а отсутствие у клиента поддержки алгоритма сжатия не мешает прокси-серверу и исходному серверу более эффективно использовать пропускную способность сети. Аналогично, прокси-сервер и исходный сервер могут работать с более новой версией HTTP, имеющей дополнительные возможности. Прокси-сервер может обладать способностью кодировать запрос или обрабатывать ответ в более современном формате. Преобразования не оказывает влияния на семантику запроса или ответа.

3.3.5. Шлюзы к системам, не являющимся HTTP-серверами

Прокси-сервер может также играть роль промежуточного звена при работе с другими системами, не поддерживающими HTTP. Web была создана с целью обеспечения беспрепятственного доступа к ресурсам в Internet. Важно, что прокси-сервер, как компонент Web, облегчает такой доступ. Например, прокси-сервер может действовать как промежуточное звено между Web-клиентом и FTP-сервером. Прокси-сервер может действовать так, как он действует в случае применения промежуточного звена Web; клиентский запрос пересылается на сервер, а ответ посылается обратно клиенту. Однако вследствие того, что Web-клиент и FTP-сервер используют разные протоколы, прокси-сервер должен преобразовать HTTP-запрос клиента в FTP-запрос и переформатировать ответ FTP-сервера в HTTP-ответ. Таким образом прокси-сервер играет роль шлюза. Отсутствие информации о типе содержания в запросах, производящихся с использованием других протоколов (например, FTP), может сделать эту задачу достаточно сложной.

Предположим, что Web-клиент посылает запрос FTP-серверу **ftp://ftp.research.att.com/F** через прокси-сервер. Запрос, направляемый прокси-серверу, представляет собой обычный HTTP-запрос. Прокси-сервер сначала конвертирует запрос Web-клиента в FTP-запрос, который может быть преобразован в последовательность команд FTP. Для некоторых протоколов преобразование запроса может состоять просто в обрамлении HTTP-запроса другими заголовками. Однако в случае протокола FTP прокси-сервер должен сначала установить FTP-соединение, действуя как FTP-клиент. В отличие от HTTP, в FTP для взаимодействия между клиентом и сервером используются отдельные каналы для данных и управления, о чем более подробно рассказывается в главе 5 (раздел 5.4.2). Таким образом, прокси-серверу следует открыть соединение управления для авторизации, а затем ус-

тановить соединение и для передачи содержимого файла F, запрошенного Web-клиентом. Аналогичным образом прокси-сервер может действовать как шлюз к серверу электронной почты с целью получения почтового ресурса M. На рис. 3.2 показано, как прокси-сервер выступает в качестве шлюза к FTP-серверу и серверу электронной почты, в то же время играя роль прокси-сервера для HTTP.

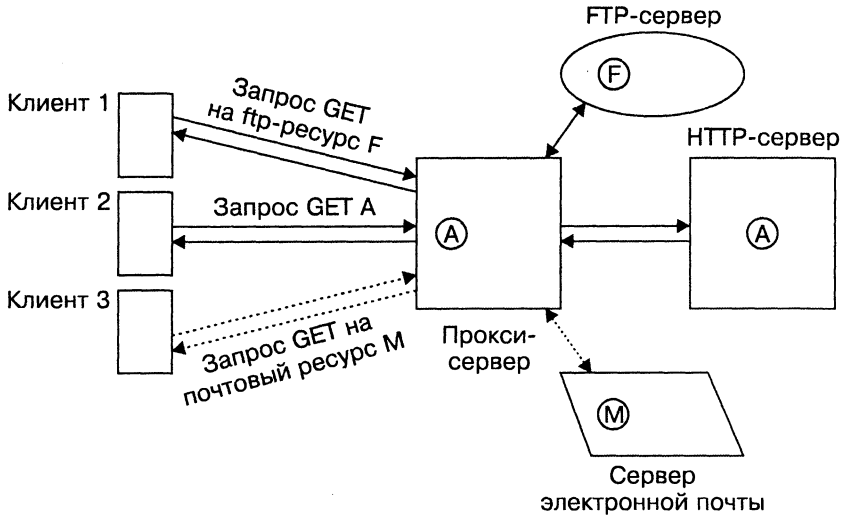


Рис. 3.2. Прокси-сервер, действующий в качестве шлюза к FTP, HTTP-серверам и серверам электронной почты

С точки зрения Web-клиента был сделан Web-запрос ресурса **ftp://ftp.research.att.com/F**, после чего ресурс **F** был получен. Этапы обработки FTP-запроса — установление соединения, авторизация, отправка последовательности команд и закрытие соединения — выполняются прокси-сервером. Прокси-сервер должен возвратить файл Web-клиенту в виде HTTP-ответа с соответствующим кодом ответа и HTTP-заголовками ответа.

Порой прокси-сервер может играть роль пассивного передаточного звена. Например, при обслуживании защищенного соединения поверх протокола Secure Socket Layer (SSL) прокси-сервер действует как туннель. SSL выполняет шифрование содержимого сообщения. Прокси-сервер передает биты сообщения в обоих направлениях. Программа прикладного уровня, каковой является прокси-сервер, может нарушать требования к конфиденциальности, которые предъявляются к SSL. Даже если прокси-серверу разрешено было «увидеть» лишь унифицированный указатель ресурса (URL), это уже может нести некоторую информацию, снижающую степень конфиденциальности. На самом деле практически все, что может «увидеть» прокси-сервер при работе с SSL, — это адрес целевого сервера и номер порта. Способность действовать в качестве туннеля расширяет область применения прокси-серверов. Подробнее о SSL рассказывается в главе 6 (раздел 6.4).

3.3.6. Фильтрация запросов и ответов

Прокси-сервер исполняет роль «стража», отфильтровывая ненадлежащие запросы и ответы. Прокси-сервер может отфильтровывать запросы на основе адресов сайтов и

ответы на основе определенных их параметров, например, размера ответа. Организация может устанавливать правила применительно к пользователям, запрещающие им посещение определенных сайтов, например, развлекательных (т.е. игровых сайтов, сайтов с анекдотами, порнографических сайтов), электронных аукционов и сайтов по трудоустройству. Прокси-сервер может сопоставлять каждый запрос со списком сайтов, которые организация считает нежелательными для посещения, и отказывать в обработке таких запросов, отправляя сообщение об ошибке. Сравнение может выполняться по полному URL (например, **http://www.unitedmedia.com/dilbert**), но чаще всего осуществляется сравнение с частью URL, которая представляет собой имя сервера (**www.unitedmedia.com**). При фильтрации вполне очевидна возможность излишне «запретительного» характера таких мер. В то же для некоторых сотрудников организации могут иметься вполне разумные причины для посещения «запретных» сайтов. Чтобы решить данную проблему, надо либо предоставить таким клиентам возможность обойти прокси-сервер, либо прокси-сервер должен устанавливать различные правила доступа для определенных клиентов на основе их IP-адресов. Прокси-сервер — это специализированное программное средство, поэтому он допускает значительную гибкость при его настройке. Многие хорошо известные продукты (например, Netnanny, SurfWatch и др.), которые фильтруют запросы к нежелательным сайтам, сопоставляют запрашиваемый URL с набором шаблонов. Другой причиной, по которой прокси-сервер фильтрует запросы к сайту, может стать то, что обслуживание ответов требует значительных ресурсов. Ограничивая набор сайтов, разрешенных клиентам для доступа, прокси-сервер может реально повысить производительность для других клиентов.

Фильтрация запросов на основе URL — это только один из примеров использования фильтрации. Другой пример — фильтрация запросов к поисковым системам при указании определенных ключевых слов в строке поиска. Это заставляет прокси-сервер сначала проверить, является ли сайт, к которому осуществляется обращение, поисковой системой, а затем сопоставить искомый аргумент с набором ключевых слов для поиска, являющихся нежелательными. Еще один пример фильтрации запросов — удаление определенных заголовков в Web-запросе, которые могут содержать сведения о клиенте, такие как адрес электронной почты пользователя. Как мы увидим далее в главе 7, стандарт протокола HTTP определяет правила для прокси-серверов, в соответствии с которыми осуществляется удаление или изменение заголовков в сообщении-запросе.

При фильтрации ответов прокси-сервер обычно удаляет определенные ответы, содержащие данные в форматах, вызывающих подозрения. Некоторые вирусы передаются с помощью ресурсов определенных форматов, и прокси-сервер может уделить дополнительное внимание ответам в подобных форматах. Прокси-сервер может проверять такие ответы на вирусы, прежде чем возвратить ответы клиенту. Помимо текста и изображений, довольно часто в Web в качестве ответов возвращаются программы. В качестве примеров можно привести апплеты Java или сценарии JavaScript. Интеллектуальный прокси-сервер способен выполнять проверки таких программ, чтобы удостовериться, что они не могут нанести вред. Другим критерием может быть размер ответа. Предположим, что прокси-серверу известно о низкой пропускной способности находящегося за ним клиента, и запросы, длина которых превышает определенное значение, выявляются. Прокси-сервер может принять решение изменить формат (например, сжать ответ) перед его передачей клиенту.

3.4. Роль прокси-серверов в обработке запросов и ответов HTTP

В этом разделе мы рассмотрим роль прокси-серверов в обработке запросов и ответов HTTP. Начнем мы с рассмотрения различных этапов передачи запросов и ответов, которые имеют место в случае наличия прокси-сервера между браузером и Web-сервером. Далее мы посмотрим, как прокси-сервер обрабатывает запросы и ответы. После этого мы исследуем специфические задачи функционирования прокси-сервера как Web-сервера. Затем мы рассмотрим роль прокси-сервера в качестве Web-клиента и имеющиеся при этом отличия от передачи запросов исходному серверу от имени клиентов. Наконец, мы рассмотрим пример использования прокси-сервера.

3.4.1. Этапы обмена запросами и ответами при наличии прокси-сервера

На рис. 3.3 представлены этапы взаимодействий, имеющие место при наличии прокси-сервера между клиентом и исходным сервером. Это модифицированная версия примера, представленного на рис. 2.1 (глава 2, раздел 2.3).

После того, как пользователь выбрал URL, браузер обращается к серверу доменных имен (DNS) для определения IP-адреса прокси-сервера, для работы с которым он был настроен, а затем устанавливает TCP-соединение с прокси-сервером (этапы 1 и 2). HTTP-запрос, поступивший от браузера (этап 3), заставляет прокси-сервер обратиться к DNS за IP-адресом исходного сервера (этап 4). Браузер и прокси-сервер могут использовать различные DNS-серверы. Прокси-сервер устанавливает TCP-соединение с исходным сервером, а затем посылает HTTP-запрос исходному серверу (этапы 5 и 6). Ответ от исходного сервера направляется обратно прокси-серверу, который пересылает его браузеру (этапы 7 и 8). На этой стадии браузер может установить необязательные параллельные соединения с прокси-сервером, а прокси-сервер может установить свои собственные необязательные параллельные соединения с исходным сервером (этапы 9 и 10).

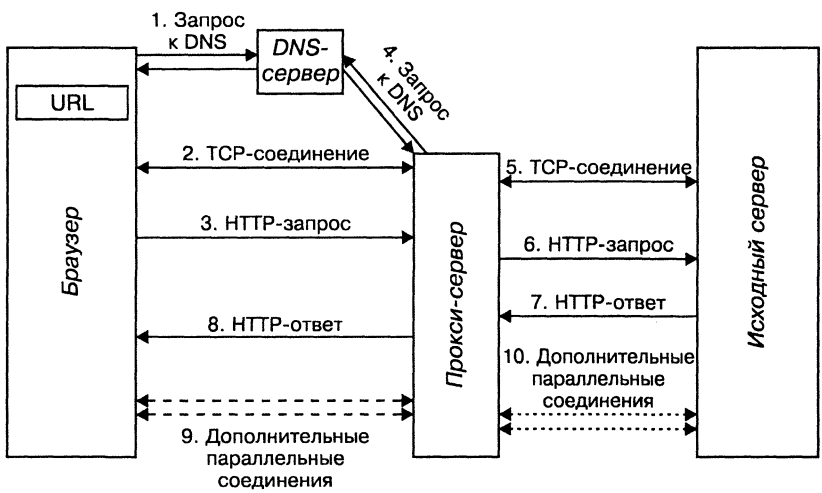


Рис. 3.3. Этапы взаимодействия браузера с исходным сервером при наличии прокси-сервера

3.4.2. Обработка запросов и ответов HTTP

Поскольку прокси-сервер является ключевым компонентом при выполнении Web-транзакций, прокси-серверы действуют в соответствии с правилами протокола HTTP. В главе 7 (раздел 7.11) мы обсудим особые правила, определяющие поведение прокси-сервера. Если обобщить, прокси-сервер должен выполнять несколько синтаксических и семантических требований. Синтаксические требования обычно связаны с заголовками, которые прокси-сервер добавляет к сообщениям, и с модификацией имеющихся заголовков. Семантические требования включают в себя корректную обработку запросов и ответов, а также поддержку ограничений, действующих при возврате кэшированных ответов.

Прокси-сервер будет передавать любые заголовки HTTP-сообщений, которые он не понимает, если не задействованы какие-либо механизмы протокола, препятствующие этому. Прокси-сервер может также добавлять или модифицировать заголовки в ответе или запросе. Требования можно разделить на следующие категории:

Идентификация. Прокси-сервер может идентифицировать себя в поле заголовка, чтобы другие компоненты в цепочке запрос-ответ знали, что сообщение прошло через прокси-сервер. В последней версии протокола (HTTP/1.1) все прокси-серверы обязаны идентифицировать себя. Одна из причин такого требования — необходимость выявления возможных петель на пути передачи сообщений.

Изменение номера версии. Прокси-сервер может изменять строку запроса (первую строку HTTP-запроса), которая включает метод запроса, URI и номер версии. Например,

```
GET /foo/bar.html HTTP/1.0
```

заголовок, отправленный клиентом, может быть заменен прокси-сервером HTTP/1.1 на

```
GET /foo/bar.html HTTP/1.1
```

перед тем, как сообщение будет отправлено дальше. Прокси-сервер может применить новые возможности HTTP/1.1 (см. главу 7) к сообщению, передаваемому между прокси-сервером и исходным сервером, несмотря на то, что клиент работает с более старой версией протокола. Если сообщение получено прокси-сервером от отправителя, использующего более новую версию протокола, прокси-сервер может использовать более старую версию протокола для передачи сообщения далее. Прокси-серверу не разрешается сообщать неверные данные относительно своего номера версии протокола, поскольку это может вызвать проблемы при интерпретации HTTP-сообщения. Если прокси-сервер изменяет номер версии, некоторые поля заголовка могут оказаться изменены, а некоторые — остаться неизменными. Для пересылки запросов должны быть соблюдены дополнительные требования HTTP/1.1. В главе 6 (раздел 6.5) подробно обсуждается, как должным образом интерпретировать номера версий HTTP.

Добавление обязательной информации о ресурсе. При возврате ответа из кэша прокси-серверу может быть предписано добавлять некоторую дополнительную информацию о ресурсе. Например, это может быть время, прошедшее с того момента, как ресурс был проверен на актуальность на исходном сервере.

Семантическая нейтральность. Одной из сложных задач, которые стоят перед прокси-сервером, является реакция на проблемы, имеющие место в запросах и ответах. Прокси-серверу нельзя принимать семантические решения, например, объявление запроса некорректным. Существует хрупкий баланс между тем, что ожидается от прокси-сервера, и тем, что он в действительности может сделать, столкнувшись с некорректным запросом или ответом. Обязанности прокси-сервера можно

выразить следующим образом: выяснить, может ли запрос быть обработан локально (если прокси-сервер является кэширующим и ответ может быть возвращен из кэша), и, если нет, переслать запрос дальше. Прокси-сервер может вести себя как туннель и «вслепую» переслать запрос, получив сообщение, которое он не может понять. Подобное поведение прокси-сервера как туннеля наиболее соответствует ожиданиям и клиента, и исходного сервера.

Обслуживание задержек и буферизация. В качестве промежуточного звена прокси-сервер отвечает за обслуживание соединения с исходным сервером в процессе формирования ответа. Ответ может генерироваться динамически. В этом случае прокси-сервер должен обслуживать соединения, буферизовать фрагменты ответа по мере их поступления от отправителя, пересылать буферизованные фрагменты получателю и ожидать отклика на завершение получения ответа клиентом, прежде чем закрыть соединения. Требования к буферизации также применяются и при передаче информации между клиентом и исходным сервером в прямом направлении. Однако запросы обычно имеют меньший размер, чем ответы, и не потребляют столько ресурсов прокси-сервера. Риск занесения в буфер прокси-сервера очень больших объемов данных привел к тому, что часть спецификации протокола HTTP, относящаяся к передаче сообщений, была прописана особенно тщательно. В главе 7 (раздел 7.6) эта проблема обсуждается более подробно.

Управление состоянием и стратегии поведения. Действуя как интерфейс для множества клиентов, прокси-сервер должен обслуживать состояния многочисленных приходящих запросов и ответов. Прокси-сервер не может закрыть соединение, пока обмен по нему не будет закончен. Если пользователь щелкает мышью на кнопке Stop в браузере и прерывает клиентский запрос в ходе его выполнения, прокси-сервер должен закрыть соединение с сервером. Тот же самый запрос может быть сделан тем же клиентом (или каким-либо другим) позднее. Если ответ от исходного сервера был полностью загружен в кэш прокси-сервера, то последующий запрос может быть обработан прокси-сервером локально, без пересылки данных по сети. Принципы принятия решений, кэшировать ли загруженные запросы даже в том случае, если клиент разрывает соединение, не специфицированы в протоколе. Эти действия должны выполняться кэширующим прокси-сервером, исходя из оптимального времени ожидания ответа пользователем и пропускной способности сети.

Чтобы прокси-сервер эффективно функционировал в качестве промежуточного звена, он должен соблюдать баланс между своими ролями сервера и клиента. Прокси-серверу предъявляется иной набор критериев эффективности, нежели исходному серверу и клиенту. Тогда как исходные серверы принимают и отвечают на многие тысячи запросов, прокси-сервер должен обслуживать двунаправленные соединения и состояния соединений, ассоциированные с многими клиентами и серверами. Как мы увидим позднее, для передачи множества запросов и ответов может быть использовано одно соединение на транспортном уровне. Исходный сервер, имея дело с тысячами запросов в секунду, может принять решение закрыть неактивное соединение раньше других. С другой стороны, прокси-сервер может использовать другой критерий при принятии решения о продолжительности времени бездействия, прежде чем закрыть соединение с клиентом, поскольку клиент с большей вероятностью повторно воспользуется своим соединением с прокси-сервером.

Проблемы практической реализации. Прокси-сервер должен иметь возможность поддерживать соединения с тысячами клиентов в течение различных периодов времени. В то же время он должен обслуживать соединения с сотнями исходных серверов (или прокси-серверов). Поскольку сервер обслуживает соединения с тысячами клиентов, прокси-сервер сталкивается со всеми проблемами, присущими Web-серве-

ру. Прокси-сервер должен принять соединение, создать программный поток или процесс для обслуживания запроса и сохранять соединение до тех пор, пока запрос не будет обработан. Позже он может занести информацию о запросе в системный журнал и выполнить все необходимые действия по освобождению ресурсов, связанных с сохранением состояния процесса или выделенными буферами. Такие операции должны быть выполнены в процессе буферизации ответов от других серверов для других запросов. Количество соединений, доступных прокси-серверу на транспортном уровне, ограничено, поэтому новые соединения не могут быть приняты, пока не будут закрыты некоторые из старых соединений. Аналогично, поскольку прокси-сервер буферизует ответы и по возможности кэширует их, здесь также могут возникать проблемы с ресурсами для их размещения. Подробнее о проблемах, с которыми сталкивается сервер при обслуживании множества одновременных запросов, рассказано в главе 4 (раздел 4.4).

Работа с cookies. В главе 2 (раздел 2.6) мы узнали, как клиент использует cookies. В главе 4 (раздел 4.4) мы увидим, как серверы работают с cookies. Поскольку прокси-сервер действует и как клиент, и как сервер, он видит cookies в сообщениях-ответах и возвращает cookies в сообщениях, пересылаемых исходному серверу, если они присутствуют в запросах. Например, клиент отправляет запрос на ресурс исходному серверу S через прокси-сервер P. Исходный сервер S отправляет cookie в заголовке **Set-Cookie** обычным образом. С точки зрения исходного сервера прокси-сервер P является клиентом, на котором будет размещен cookie. Прокси-сервер передает заголовок **Set-Cookie** клиенту для сохранения [KM00]. Когда клиент в следующий раз связывается с исходным сервером, клиент использует URL для принятия решения, какой cookie включить в свой заголовок **Cookie**. Прокси-сервер передает этот заголовок без изменений. На деле прокси-серверу запрещается добавлять свои собственные связанные с cookies заголовки в любом направлении. Таким образом простое присутствие прокси-сервера на маршруте не влияет на семантику cookies и для клиента, и для сервера. Однако Web-сервер может проинструктировать прокси-сервер не кэшировать определенные заголовки. Например, можно указать, чтобы заголовок **Set-Cookie** не кэшировался (см. главу 7, раздел 7.3.3, где описаны различные механизмы для управления кэшированием).

3.4.3. Прокси-сервер в роли Web-сервера

Когда прокси-сервер играет роль Web-сервера, его главное назначение — принимать и обрабатывать запросы клиентов. Кэширующий прокси-сервер, действующий как Web-сервер для клиентского запроса, имеет возможность проверить, может ли запрос быть выполнен без обращения к исходному серверу из кэша. Клиент может захотеть получить ответ без потенциальных задержек при пересылке запроса. Существуют особые механизмы протокола HTTP, которые позволяют клиенту потребовать, чтобы сообщение *не* пересылалось прокси-сервером дальше. В таких случаях прокси-сервер должен сам ответить на запрос. Прокси-сервер проверяет, имеется ли ответ в его кэше. Если прокси-сервер не может найти ответ в кэше, он должен ответить сообщением об ошибке.

Если требования отвечать локально не предъявляется, прокси-сервер проверяет, имеется ли ответ в его кэше и является ли ответ актуальным. В противном случае прокси-сервер пересылает запрос исходному серверу и ожидает ответа. Ответ может быть возвращен исходным сервером или другим прокси-сервером, расположенным на пути к исходному серверу. Когда ответ поступает прокси-серверу, он

возвращается клиенту. В зависимости от политики кэширования, проводимой прокси-сервером, ответ может быть помещен в кэш.

Рассмотрим рис. 3.3 в предположении, что запросы клиента проходят через прокси-сервер, который способен напрямую отправлять запросы исходному серверу. Прокси-сервер может иметь кэшированную копию одного из встроенных в страницу **foo.html** ресурсов, допустим, что **foo2.gif**. Получив запрос на ресурс **foo2.gif,** прокси-сервер ищет ресурс в своем кэше. Если ресурс найден в кэше, прокси-серверу необходимо удостовериться, что кэшированный ответ совпадает с тем, который мог бы быть получен при обращении к исходному серверу. Кэширующий прокси-сервер может воспользоваться простой эвристикой, считая ресурсы, кэшированные в течение последних нескольких минут, как полученные обращением к исходному серверу. Затем прокси-сервер может вернуть ответ из своего кэша, не пересылая запрос исходному серверу. Однако выбор интервала времени, в течение которого ресурс считается актуальным, существенно зависит от природы ресурса — так курс акций нельзя считать актуальной информацией уже через несколько секунд.

С точки зрения клиента прокси-сервер ведет себя как исходный сервер. Главным различием здесь является уменьшение времени ожидания при возврате кэшированного ответа, поскольку прокси-сервер обычно находится гораздо ближе к клиенту, чем исходный сервер. Между кэширующим прокси-сервером и исходным сервером в сети передается меньшее число байтов, что помогает уменьшить вероятность возникновения узких мест в сети.

3.4.4. Прокси-сервер в роли Web-клиента

Прокси-сервер играет роль клиента, когда пересылает запрос исходному серверу. Существуют две причины, по которым прокси-сервер пересылает запрос:

- Прокси-сервер является кэширующим прокси-сервером, но он не может удовлетворить запрос из своего кэша.
- Прокси-сервер не является кэширующим прокси-сервером или явным образом проинструктирован клиентом пересылать запрос.

В роли клиента прокси-сервер должен выполнять все задачи, присущие клиенту. Ему нужно определить IP-адрес исходного сервера, обратившись к DNS-серверу, открыть соединение с исходным сервером, передать запрос и получить возвращенный исходным сервером ответ. Ответ возвращается клиенту, который отправил запрос прокси-серверу. Если ответ ссылается на встроенные ресурсы, клиент может также отправить прокси-серверу запросы на эти ресурсы. Прокси-сервер обрабатывает каждый из этих запросов аналогично тому, как он обрабатывал первый запрос, получает ответы от сервера и пересылает ответы клиенту.

Прокси-сервер может иметь несколько другие возможности по сравнению с клиентом. Так, прокси-сервер может иметь лучшее подключение к Internet, чем клиент. Для прокси-сервера является типичным использование более старой версии протокола, чем клиентом. Это главным образом обусловлено тем, что новые версии браузеров распространяются быстро и бесплатно. Обновление программного обеспечения промежуточных компонентов занимает больше времени. На деле в настоящее время в Web имеется гораздо больше пользовательских агентов и Web-серверов, чем прокси-серверов, работающих под HTTP/1.1. Если версия протокола прокси-сервера является более старой, чем версия у клиента, прокси-сервер не будет способен использовать новые функциональные возможности протокола. Любой запрос, использующий более позднюю версию, должен быть сведен про-

кси-сервером к более старой версии, прежде чем прокси-сервер осуществит его пересылку.

Прокси-сервер не имеет права изменять семантику запроса. В противном случае клиент, сделавший запрос, может получить не тот ответ, который ожидал, либо получить его в формате, который не сможет обработать. Например, Web-сервер и прокси-сервер могут использовать определенный способ кодирования ответа (допустим, новый алгоритм сжатия). Однако прокси-сервер может изменить ответ, чтобы клиент смог его понять. Номер версии клиента дает прокси-серверу информацию об его функциональных возможностях. Таким образом, прокси-серверу возможно придется осуществить декомпрессию сжатого сообщения перед передачей сообщения клиенту.

По сравнению с клиентом, прокси-сервер взаимодействует с гораздо большим числом серверов и по природе своей деятельности с большей вероятностью обращается к одному и тому же Web-сайту много раз. Можно избежать многочисленных обращений к DNS для преобразования доменного имени одного и того же сайта в IP-адрес, если это уже недавно было сделано от имени другого клиента. Заметим, что обращения к DNS выполняются прокси-сервером только в том случае, если он является последним в цепочке прокси-серверов между пользовательским агентом и Web-сервером; в противном случае он просто пересылает запрос следующему прокси-серверу в цепочке.

В качестве Web-клиента прокси-сервер принимает решения об открытии и о поддержании соединений со многими серверами. Стандартный клиент, действующий от имени одного пользователя, должен поддерживать одновременно небольшое число соединений. Действуя от имени нескольких клиентов, прокси-сервер должен иметь возможность открывать и поддерживать большое число соединений одновременно. Прокси-серверу приходится выделять значительные ресурсы для обслуживания состояний от имени этих соединений, загружать ресурсы с множества серверов и буферизировать данные. Если ответ генерируется динамически, прокси-сервер должен буферизировать ответ на длительное время, прежде чем он сможет закончить пересылку ответа.

3.4.5. Пример использования прокси-сервера

Мы рассмотрим использование прокси-сервера, взяв за основу пример из второй главы (раздел 2.3). Клиент загружает документ с несколькими встроенными изображениями. Теперь предположим, что браузер был настроен для работы через прокси-сервер.

Допустим, что изображение **foo1.gif** находится в кэше браузера, а изображение **foo2.gif** — в кэше прокси-сервера. Прокси-сервер может воспользоваться своей стратегией актуализации содержимого кэша, чтобы решить, может ли кэшированная копия **foo2.gif** быть возвращена клиенту без повторной ее проверки на исходном сервере. Если прокси-сервер считает, что ответ не является больше актуальным, он может обновить кэшированный ответ, отправив модифицированный запрос исходному серверу. Модифицированный запрос к исходному серверу требует, чтобы содержимое ресурса было возвращено только в том случае, если ресурс был изменен с того момента, как он был кэширован прокси-сервером. Если прокси-сервер отправляет запрос и получает новую копию ресурса, ему нужно кэшировать ответ и переслать ответ клиенту. Что касается клиента, то он получает ресурс **foo2.gif** от прокси-сервера. Клиент не имеет понятия о действиях, предпринятых прокси-сервером для получения ресурса. Таким образом, при запросе ресурса **foo2.gif** прокси-сервер действует как Web-клиент.

Теперь предположим, что ресурс **foo3.gif** не содержится в кэше прокси-сервера. Прокси-сервер в этом случае действует как Web-клиент и отправляет запрос исходному серверу, где размещен данный ресурс. Если пользователь обращается к ресурсу **amex.cgi**, заполняя форму, или выбирает ресурс **mp.tv**, эти запросы также пересылаются через прокси-сервера. Прокси-сервер открывает отдельное соединение с соответствующим исходным сервером в каждом из этих случаев. Ресурс **amex.cgi** может потребовать значительного времени на обработку. При этом прокси-сервер должен будет поддерживать соединения с исходным сервером и с клиентом открытыми.

3.5. Цепочки прокси-серверов и иерархии

На первых порах применения прокси-серверов при взаимодействии между пользовательскими агентами и серверами на пути между ними присутствовал единственный прокси-сервер. В скором времени на пути между пользовательским агентом и исходным сервером стали появляться несколько прокси-серверов. Причиной этому послужила структура организаций, использующих Web. Например, в университете может быть прокси-сервер, через который проходят все Web-транзакции, но на некоторых факультетах университета могут быть свои собственные прокси-серверы. Выход университета в Internet может осуществляться через провайдера, также имеющего собственный прокси-сервер. Наконец, выход в Internet исходного сервера может также осуществляться через прокси-сервер. В результате этого пользовательский запрос и ответ исходного сервера проходят через несколько прокси-серверов, образующих цепочку. В настоящее время в Web вполне обычна ситуация, когда запрос проходит через группу связанных между собой прокси-серверов. Агент пользователя может быть непосредственно настроен для взаимодействия с прокси-сервером для всех своих операций с Web. На рис. 3.4 представлена конфигурация такой системы.

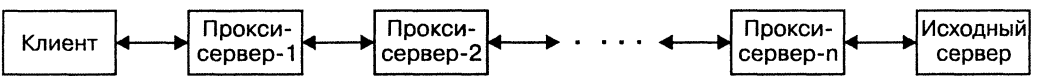


Рис. 3.4. Цепочка прокси-серверов в Web

Как видно из рис. 3.4, сообщение от агента пользователя может проходить через несколько прокси-серверов, прежде чем достигнет исходного сервера. Часто реальная структура может отличаться от простой линейной структуры, представленной на рис. 3.4. Так, промежуточные серверы могут иметь другие серверы, подключенные к ним с обеих сторон. Довольно распространенной является структура, когда группа серверов внутри организации образует иерархию. Например, региональный прокси-сервер может быть связан с другими региональными прокси-серверами, которые, в свою очередь, могут быть связаны с национальными прокси-серверами. Отметим, что в этом случае, в отличие от структуры на рис. 3.4, представляющей собой линейную цепочку, позади регионального или национального прокси-сервера могут располагаться другие прокси-серверы. Подобная иерархическая структура прокси-серверов широко распространена и очень полезна в странах, где стоимость доступа к Internet высока. Ограничив в разумных пределах число подключений на региональном или национальном уровне, организации могут существенно сократить время ожидания пользователей и стоимость соединений. Группа прокси-сер-

веров может обслуживать больший объем кэшей для хранения ответов. Однако при наличии нескольких прокси-серверов при исполнении запроса возникает необходимость поиска нужного кэша.

3.6. Настройка прокси-сервера

В главе 2 (раздел 2.4) мы узнали, что пользователь может настроить браузер и задать различные физические и семантические параметры. В случае использования прокси-серверов пользователи обычно не могут влиять на его настройки. Например, пользователь обычно не может изменять специфические для прокси-сервера параметры, такие как размер кэша или частоту обновления содержимого. В браузере можно задать, следует ли использовать прокси-сервер в качестве промежуточного компонента. Клиент может обойти прокси-сервер и выбрать режим, при котором все запросы направляются исходному серверу непосредственно, хотя этому может помешать присутствие межсетевых экранов. Многие браузеры могут быть настроены на использование прокси-сервера. В некоторых организациях пользователям разрешается изменять настройки по умолчанию с помощью выбора, использовать ли прокси-сервер вообще, использовать определенный прокси-сервер для всех запросов или для всех запросов за исключением тех, которые адресуются определенным Web-серверам. Последняя из этих опций предоставляет пользователю наибольшие возможности. Пользователь может разрешить использование прокси-сервера для сайтов, которые не нарушают его конфиденциальности, или для популярных сайтов, ресурсы которых могут содержаться в кэше прокси-сервера. Для других доменов пользователи могут обходить прокси-сервер. Большинство пользователей не изменяют настройки по умолчанию, а многие пользователи могут даже не знать, что у них есть возможность изменять настройки по умолчанию. Для некоторых организаций попытки пользователей изменить настройки браузера, связанные с прокси-сервером, могут оказаться тщетными. Администраторы организации могут использовать стратегию, благодаря которой все пользовательские запросы должны обязательно проходить через прокси-сервер.

Серьезные проблемы возникают при настройке мобильных клиентов, когда при настройке необходимо указывать один из нескольких прокси-серверов. В связи с этим последнее время ведутся работы по созданию средств для автоматического поиска нужного прокси-сервера. Вместо того чтобы просто найти прокси-сервер, задействуется более совершенный механизм, который позволяет найти информацию о настройках, которая, в свою очередь, позволяет использовать наиболее подходящий в данный момент прокси-сервер. Эта работа пока находится на стадии экспериментов [CGC+00], использует протокол Dynamic Host Configuration Protocol [Dro97] и связана применением DNS [Moc87a].

3.7. Проблемы, связанные с конфиденциальностью

В главе 2 мы обсуждали проблемы, связанные с конфиденциальностью, применительно к браузерам. Прокси-серверы играют более важную роль с точки зрения безопасности. Являясь промежуточным звеном для множества пользователей, прокси-сервер осведомлен о многих деталях их поведения:

- ресурсах, к которым осуществляется доступ;
- частоте доступа;
- заголовках и теле запроса;
- заголовках и теле ответа.

Эта осведомленность потенциально несет угрозу для безопасности пользователя. Однако прокси-сервер часто воспринимается как *доверенный* промежуточный компонент, если пользователи настроили свои браузеры для доступа через прокси-сервер. Пользователям известно, кто ответственен за собранную информацию. Как мы увидим в разделе 3.8, существуют ситуации, в которых у пользователей отсутствует информация о том, что их запросы и ответы проходят через прокси-серверы. В таких случаях неосведомленность о наличии промежуточного звена вызывает необходимость задуматься, кто имеет доступ к информации о работе пользователя.

Прокси-серверам известен IP-адрес клиента. Если клиентом является однопользовательский компьютер, или если в пользовательских запросах присутствует одно и то же значение поля **User-Agent**, прокси-сервер может определить, что группа запросов исходит от одного и того же пользователя. Прокси-сервер не всегда может точно знать, кто осуществляет доступ к Web-ресурсам через него. Если прокси-сервер расположен перед многопользовательским компьютером, он не всегда в состоянии точно установить, какой пользователь отправил запрос. В многопользовательской среде для идентификации пользователя может быть использована другая информация, например, системная информация о времени входа пользователя в систему и выхода из нее. Подобные приемы часто используются в случаях расследования преступлений. Полиция в ряде штатов США использует журналы регистрации для отслеживания пользователей. Другой уязвимой с точки зрения безопасности информацией является перечень ресурсов, к которым осуществлялся доступ. Выборочный просмотр ресурсов, к которым обращался пользователь, может создать «портрет» пользователя. Люди, осуществляющие поиск информации по определенным темам, связанным с определенными заболеваниями, либо посещающие Web-сайты по трудоустройству, должны в определенной степени считаться с риском, что об их проблемах и намерениях станет известно другим лицам.

Содержимое запроса может также содержать дополнительную информацию о пользователе, такую как день рождения или номер его кредитной карты. Эта информация извлекается из форм, заполняемых повсеместно при регистрации для участия в конференциях и списках рассылки, покупке товаров, заказе авиабилетов. Очевидным решением, все чаще используемым многими Web-сайтами, является использование шифрования на основе протокола SSL, в этом случае прокси-сервер используется в качестве туннеля. Аналогичным образом, содержимое ответа также несет информацию о пользователе.

3.8. Другие виды прокси-серверов

Мы рассмотрели различные роли прокси-серверов, в которых значение данного термина было достаточно прояснено. Однако имеются и другие виды прокси-серверов. Мы рассмотрим два из них: *обратные* прокси-серверы и *перехватывающие* прокси-серверы. Мы обсудим причины, по которым такие прокси-серверы находят применение, а также выясним, чем они отличаются от традиционных прокси-серверов, рассмотренных ранее в этой главы. Важно отметить, что хотя термины обратный и перехватывающий прокси-сервер используются достаточно часто, эти про-

кси-серверы не всегда соответствуют спецификации протокола HTTP. Что еще более важно, их поведение пока не стандартизовано.

3.8.1. Обратные прокси-серверы или серверы-заместители

Изначально прокси-серверы размещались как можно ближе к пользователю. Администраторы конфигурировали прокси-серверы для локальных сетей или пользователей небольших организаций. По мере роста популярности Web ряд Web-сайтов начал притягивать к себе большое число пользователей. Чтобы уменьшить нагрузку на серверы, прокси-серверы пришлось размещать ближе к исходным серверам. Такие прокси-серверы получили название *обратных прокси-серверов*, поскольку они располагались на другом конце цепочки запрос-ответ по сравнению с традиционными прокси-серверами, располагающимися ближе к пользователю. Другой причиной, по которой прокси-серверы стали размещать перед исходным сервером (или группой серверов, обслуживающих сайт), стало желание сделать исходные серверы «невидимыми» для входящих запросов. Обратные прокси-серверы защищают исходный сервер от прямых атак извне. Промежуточный компонент перед исходными серверами может также оказать помощь в распределении нагрузки между группой серверов, обслуживающих активно посещаемый сайт. Прокси-сервер в этом случае переадресует запросы на другие компьютеры. Обратный прокси-сервер представляется клиенту как исходный сервер. Обратный прокси-сервер действует как внешний интерфейс для одного или нескольких исходных серверов, которые могут находиться за сетевым экраном. Клиентский запрос передается обратному прокси-серверу, который пересылает его исходному серверу. Обратный прокси-сервер может также иметь кэш. При пересылке запроса обратный прокси-сервер действует как туннель; т.е. биты, передаваемые в обоих направлениях, остаются без изменений. Извне можно видеть лишь один IP-адрес обратного прокси-сервера.

Заметим, что хотя термин «обратный прокси-сервер» стал достаточно популярным, в действительности необходимости в применении отдельного термина нет. С точки зрения клиента он взаимодействует с исходным сервером. Обратный прокси-сервер является лишь шлюзом, а взаимодействие между обратным прокси-сервером и внутренним исходным сервером (серверами) скрыто от клиента. Фактически обратный прокси-сервер может не использовать HTTP для взаимодействия с исходным сервером (серверами), находящимися позади него. Для обратных прокси-серверов в настоящее время используется и новый термин — *сервер-заместитель*.

3.8.2. Перехватывающие прокси-серверы

Если прокси-сервер явно предназначен для размещения перед Web-клиентом (т.е. агентом пользователя, трактующим прокси-сервер как сервер), то все запросы и ответы к клиенту и от клиента будут проходить через прокси-сервер. Вне зависимости от конфигурации локальной сети, прокси-сервер пропускает через себя все передаваемые между Internet и клиентом пакеты. У клиента отсутствует возможность миновать прокси-сервер и напрямую связаться с исходными серверами. Во многих организациях данная конфигурация используется по умолчанию. Однако в том случае, если у пользователей имеется несколько способов выхода в Internet или обхода прокси-сервера, прокси-сервер не сможет пропускать через себя *все* запросы и ответы.

Часть трафика, относящаяся к Web, может быть извлечена путем просмотра сетевого трафика. Прокси-сервер, который либо напрямую просматривает сетевой трафик и перехватывает Web-трафик, либо получает перенаправленный трафик от элементов сети, осуществляющих перехват трафика, называется *перехватывающим* прокси-сервером. Перехватывающий прокси-сервер затем предпринимает попытку сформировать ответ на запрос локально, либо переадресовывает запрос. Чтобы перехватывающий прокси-сервер пропускал все запросы и ответы, он должен быть установлен таким образом, чтобы через него проходил единственный путь выхода в Internet от пользователей. Для использования перехватывающего прокси-сервера необходимо, чтобы на маршрутизаторе было предусмотрено отслеживание сетевого трафика. Поскольку перехват выполняется на более низком уровне стека протоколов, весьма затруднительно реализовать обход перехватывающего прокси-сервера для определенных запросов. Клиенты не могут принимать решение о переадресации каких-либо своих запросов и ответов. Как следствие, никакой настройки клиентов не требуется.

В разделе 3.2 мы познакомились с термином «прозрачный прокси-сервер». Прозрачность прокси-сервера определяется тем, что происходит с сообщениями, а не тем, является ли наличие прокси-сервера невидимым для клиента или для пользователя. Перехватывающие прокси-серверы изначально назывались прозрачными прокси-серверами. Позднее была внесена ясность, и перехватывающие прокси-серверы получили наименование, которое более к ним подходит.

3.9. Резюме

Прокси-сервер представляет собой популярный и полезный промежуточный компонент Web, который дает возможность большому числу клиентов позади него осуществлять совместный доступ к Internet. Прокси-серверы осуществляют фильтрацию запросов и ответов, а также в некоторой степени обеспечивают конфиденциальность пользователей. Кэширование помогает значительно уменьшить время ожидания для клиентов. Прокси-серверы гарантируют неизменность семантики пересылаемых запросов/ответов, выступая в роли клиента или сервера. Поскольку практически все популярные провайдеры используют прокси-серверы, большая часть запросов к Web-серверам исходит от прокси-серверов. В Web достаточно часто используются иерархические структуры прокси-серверов на местном, региональном и национальном уровнях.

4

Web-серверы

Web-сервер — это программа, которая создает и возвращает ответы на запросы Web-ресурсов клиентами. Эта глава описывает работу Web-серверов, завершая тем самым обзор программных компонентов Web: клиентов, инициирующих запросы, прокси-серверов, которые действуют как промежуточные звенья, и Web-серверов, создающих ответы. Обработка клиентского запроса включает в себя несколько ключевых действий: синтаксический анализ сообщения-запроса, проверку полномочий, связывание URL в запросе с ресурсом в файловой системе сервера, построение сообщения-ответа и возврат сообщения-ответа клиенту, обратившемуся с запросом. Сервер может генерировать сообщение-ответ различными способами. В простейшем случае сервер лишь извлекает файл, ассоциированный с URL, и возвращает содержимое клиенту. В других случаях сервер может вызвать сценарий, который связывается с другими серверами или базами данных для построения сообщения-ответа.

Эта глава начинается с рассмотрения отличий между *Web-сайтом* и *Web-сервером*. Затем поясняется, как сервер обрабатывает клиентский запрос. При этом затрагиваются проблемы аутентификации и способы динамического формирования запросов. HTTP представляет собой протокол, не сохраняющий состояния, поэтому серверу нет необходимости сохранять информацию между последовательными запросами. Будет описано, как сервер может сохранять некоторую информацию с целью снижения нагрузки при обработке будущих запросов. Чтобы обслуживать несколько клиентов одновременно, серверу нужно иметь эффективный способ переключаться между обработкой различных запросов. Мы обсудим и сравним два основных подхода: серверы с *управлением по событиям* и серверы с *управлением по процессам*, а также различные гибридные подходы. Системные ресурсы, которые требуются серверу, зависят от степени популярности Web-сайта. Мы кратко остановимся на проблемах установки нескольких серверов на одном компьютере, управления копиями одного сайта на нескольких компьютерах. С целью проиллюстрировать работу реального Web-сервера в конце главы приводится практический пример на базе Web-сервера Apache.

4.1. Web-сайты и Web-серверы

Web-сайт состоит из набора Web-страниц, связанных с именем хоста, тогда как Web-сервер представляет собой программу, которая обрабатывает клиентские запросы на Web-ресурсы. В этом разделе мы рассмотрим функциональные возможности различных видов Web-сайтов и обсудим основные принципы работы Web-серверов. Мы также поясним различие между Web-сайтом и Web-сервером и между Web-сервером и платформой, на которой последний функционирует.

4.1.1. Web-сайт

Действия пользователя при работе в Web заключаются в загрузке Web-страниц с различных Web-сайтов. Web-сайты существенно отличаются друг от друга по предоставляемой информации. Ниже приводится классификация Web-сайтов.

- **Университетский Web-сайт.** Университет может иметь свой Web-сайт, который предоставляет информацию об учебных и научно-исследовательских программах, содержит фотографии и контактную информацию. Основной университетский сайт может быть связан с Web-сайтами факультетов. Эти Web-сайты могут содержать списки рабочих и домашних телефонов, информацию об определенных учебных курсах, а также личные домашние страницы преподавателей и студентов.
- **Web-сайт фирмы-производителя.** Компания может иметь Web-сайт, описывающий ее продукцию и оказываемые услуги. На сайте могут также быть помещены подробные описания продукции, информация о ценах, а также средства для размещения заказов. Подобная информация может представлять собой всего лишь преобразованные печатные материалы в виде набора статических файлов и изображений. В качестве дополнения сайт может давать пользователям возможность размещать заказы на товары и услуги путем взаимодействия с базой данных, которая содержит информацию о наличии и количестве товаров, а также о заказах потребителей. Кроме того, сайт может давать возможность пользователям осуществлять поиск в базе данных.
- **Корпоративная интрасеть.** Многие компании имеют внутренние Web-сайты, которые недоступны для посторонних лиц. Эти сайты обычно предоставляют доступ к каталогам, внутренним базам данных. Для доступа к информации, как правило, требуется указать имя пользователя и пароль. До появления Web доступ к таким базам данных для сотрудников был затруднен. Доступ к базам данных через внутренние Web-сайты способствует повышению эффективности ведения бизнеса и дает возможность быстрого и удобного получения информации.
- **Web-сайты бизнес-бизнес (B2B).** Компании, которые зависят друг от друга по роду своей деятельности, также могут использовать Web-сайты для координации совместных действий, поскольку при этом достигается высокая эффективность при меньших затратах, чем в случае взаимодействия посредством традиционного документооборота, телефона или факса. Например, одна компания (допустим, производитель автомобилей) может размещать на Web-сайте другой компании заказы (например, на комплектующие). Выполнение подобных транзакций между компаниями через Web упрощает процесс размещения заказов и отчетность.
- **Различные мероприятия.** Некоторые Web-сайты посвящены разовым мероприятиям, таким как спортивные состязания или национальные выборы. Эти сайты обычно существуют относительно недолго, а содержимое их быстро изменяется. Рассмотрим Web-сайт, посвященный Олимпийским играм. За несколько недель до начала соревнований на сайт обращается небольшое число пользователей. Затем на сайте размещается информация о месте и времени проведения соревнований, а также об их участниках. После начала Олимпийских игр на сайте размещается актуальная динамически генерируемая информация о спортивных событиях. В это время сайт пользуется чрезвычайной популярностью. Возможны всплески числа запросов непосредственно перед или

сразу после популярного соревнования. Запросы могут приходить от пользователей со всего мира и в любое время суток.

- **Портал.** Портал представляет собой централизованное хранилище обновляемой информации, к которой более или менее регулярно обращается большое число пользователей. Например, портал может предоставлять доступ к новостям, информации о погоде, программах телевидения. Возможна настройка способа представления информации пользователями. Например, пользователя может интересовать погода в Токио, результаты последних бейсбольных матчей, текущие цены на определенные акции. Кроме того, портальный сайт предоставляет гиперссылки, организованные по категориям, обеспечивая удобный доступ пользователей. Разработчики портала разбивают эти ссылки по категориям и проверяют их.
- **Поиск.** Пользователю может понадобиться найти информацию, которая не подпадает под имеющиеся на сайте категории. Поисковая система дает возможность пользователю ввести запрос и возвращает перечень гипертекстовых ссылок на Web-страницы, которые содержат введенные пользователем слова. В противоположность Web-сайтам, которые предоставляют доступ к внутренним базам данных, поисковая машина осуществляет запросы к индексированному множеству Web-ресурсов, которое собирают и обновляют специальные программы-спайдеры, о чем говорилось во второй главе (раздел 2.7). Главная задача поисковой системы — создавать содержательные ответы на пользовательские запросы.
- **Шлюз к другим сервисам.** Многие Web-сайты предоставляют доступ к другим сервисам, например, к группам новостей, электронной почте, доскам объявлений, дискуссионным группам, службам мгновенных сообщений. Например, Web-сайт новостной сети дает возможность пользователям подписываться на группы новостей, читать и публиковать в них статьи. Это требует хранения на сайте информации о группах новостей и статьях, прочитанных каждым пользователем. Аналогично, Web-сайт может обеспечивать внешний интерфейс для электронной почты, который предоставляет возможность пользователям читать, хранить, создавать и отправлять сообщения. Дискуссионные группы предоставляют пользователям возможность общаться друг с другом. Сетевые газеты могут предоставлять читателям форумы для обсуждения и обмена мнениями о последних событиях.

Степень загрузки Web-сайта может определяться различными факторами, в том числе популярностью сайта, разнообразием состава пользователей, количеством и объемом Web-ресурсов, динамически генерируемым содержанием, а также необходимостью ограничить доступ для определенных пользователей.

4.1.2. Web-сервер

Web-сервер — это программа, которая обрабатывает HTTP-запросы на определенные ресурсы. Создание и передача Web-страницы может потребовать обращения к нескольким серверам, выполнения сценариев и запросов к базам данных. Рассмотрим классический пример, когда пользователь обращается к Web-сайту **www.bar.com**. Ресурс **http://www.bar.com/foo.html** может содержать несколько встроенных изображений, которые автоматически загружаются браузером. Эти встроенные изображения не обязательно размещены на **www.bar.com**. Например, HTML-файл **http://www.bar.com/foo.html** может иметь встроенное изображение

http://images.bar.com/foo3.gif, которое размещено на **images.bar.com**. Изображения могут размещаться и поддерживаться другой компанией, **www.images.com**, для чего используется встроенная ссылка **http://www.images.com/ fooimages/foo3.gif**. Размещение изображений на отдельном сервере дает возможность основному Web-серверу обрабатывать большой поток запросов на небольшие и часто меняющиеся HTML-документы.

Web-сервер функционирует на компьютере, имеющем доступ к сети. Компьютер имеет один или несколько процессоров, память и жесткие диски для хранения статических документов и сценариев. Платформы различаются по своей вычислительной мощности, способам подключения к сети и объему накопителей, выбор платформы определяется типом Web-ресурсов и ожидаемой интенсивностью клиентских запросов. Web-сервер взаимодействует с процессором, дисковой памятью и сетевым соединением через посредство операционной системы. Для Web-сайтов, обслуживающих большое число пользователей, Web-сервер устанавливается на отдельном компьютере. В других случаях на компьютере может также выполняться множество других приложений, например, сервер электронной почты или FTP-сервер, либо пользовательские приложения. Для выполнения пользовательских запросов сервер может исполнять сценарии, которые взаимодействует с другими серверами или системами управления базами данных. При взаимодействии с другими серверами может использоваться как HTTP, так и другие протоколы.

4.2. Выполнение клиентского запроса

Web-серверы предоставляют доступ к разнообразным ресурсам: от статических файлов до сценариев, которые динамически генерируют ответы. В этом разделе мы рассмотрим этапы выполнения клиентского запроса. Далее мы обсудим, как Web-серверы осуществляют аутентификацию пользователей с целью ограничения доступа к определенным ресурсам. Затем будет описано, как Web-серверы динамически генерируют содержимое путем синтаксического анализа HTML-файлов и вызова сценариев. Наконец, мы поговорим о роли cookies в сохранении состояния между последовательно выполняемыми одним и тем же пользователем запросами.

4.2.1. Этапы выполнения клиентского запроса

На верхнем уровне Web-сервер выполняет следующие действия по обработке HTTP-запроса:

1. **Чтение и синтаксический анализ сообщения HTTP-запроса.** Сервер читает сообщение запроса, отправленное клиентом. Заголовок сообщения содержит управляющую информацию, например, запрашиваемую операцию (например, **GET**) и URL запрашиваемого ресурса (например, **/foo.html**). Сервер может извлекать другие поля заголовков, которые влияют на построение сообщения-ответа.
2. **Преобразование URL в имя файла.** Сервер преобразует URL в имя файла соответствующего ресурса. URL может иметь прямое отношение к структуре базовой файловой системы. Например, Web-ресурсы могут располагаться в каталоге, например, **/www**, тогда URL **http://www.bar.com/foo/index.html** будет соответствовать файлу **/www/foo/index.html**.
3. **Разрешение на выполнение запроса.** Прежде чем сформировать сообщение-ответ, сервер проверяет, имеет ли клиент разрешение на доступ к ресур-

су. Хотя многие Web-ресурсы доступны для любых пользователей, сервер может ограничить доступ к некоторым ресурсам на основе информации о правах доступа в заголовке HTTP-запроса.

4. **Формирование и передача ответа.** Сервер генерирует сообщение-ответ, которое содержит заголовок с информацией о состоянии (например, наличие ошибки, связанной с несанкционированным доступом или обращением к несуществующему ресурсу, переадресация клиента к ресурсу с другим URL, успешный ответ, содержащий запрашиваемый ресурс). Кроме того, заголовок может содержать метаданные о ресурсе, такие как его длина и формат.

В определенный момент в ходе обработки запроса сервер может записать информацию о запросе и ответе в свой журнал. Например, сервер может сформировать запись в журнале перед или после передачи сообщения-ответа.

В качестве примера обработки запроса обратимся к классическому примеру, представленному ранее во второй главе (раздел 2.3.1). Клиент отправляет HTTP-запрос на ресурс **http://www.bar.com/foo.html** серверу **www.bar.com**. Сервер читает сообщение для определения имени **/foo.html** запрашиваемого ресурса. Затем сервер идентифицирует соответствующий файл **/www/foo.html** и определяет, что для доступа к этому ресурсу авторизация не требуется. Далее сервер инициирует системные вызовы для получения атрибутов ресурса, таких как размер файла и дата последней модификации. Эти атрибуты фигурируют в заголовке HTTP-ответа (поля заголовка **Content-Length** и **Last-Modified**) вместе с другой информацией, такой как информация о состоянии ответа, идентификационные данные сервера и текущее время. После создания заголовка ответа сервер передает заголовок и содержимое файла обратившемуся с запросом клиенту.

После получения сообщения-ответа клиент осуществляет синтаксический анализ HTML-файла и выдает HTTP-запросы для каждого из встроенных изображений **foo1.gif**, **foo2gif** и **foo3.ipg**. Сервер обрабатывает эти запросы точно таким же образом, как и запрос на **foo.html**. При ответе на каждый запрос сервер извлекает соответствующий файл, формирует заголовок ответа, после чего передает заголовок и данные. Содержимое любого из этих файлов может изменяться, не оказывая влияния на работу сервера. Кроме того, сервер не устанавливает каких-либо взаимоотношений между ресурсами на Web-странице. Такие взаимоотношения устанавливаются автором документа **foo.html** путем указания ссылок на каждое из изображений в HTML-файле. Браузер инициирует извлечение этих изображений путем отправки HTTP-запросов серверу, а сервер независимо обрабатывает каждый из этих запросов.

Сообщение-ответ сервера не всегда содержит ресурс. В качестве примера предположим, что в браузере имеется кэшированная копия документа **foo.html**. Не зная, был ли ресурс изменен сервером, браузер может отправить запрос на подтверждение актуальности кэшированной копии. Браузер включает время последней модификации кэшированного ресурса в заголовок запроса. После получения такого запроса сервер определяет время последней модификации файла **/www/foo.html**. Если оба времени совпадают, значит, что ресурс не был изменен с момента последнего его запроса браузером. Сервер отправляет сообщение-ответ с заголовком, который указывает, что запрашиваемый ресурс не был модифицирован, инструктируя браузер использовать кэшированную копию. В этом случае сообщение-ответ не включает в себя содержимое ресурса. С другой стороны, если файл **/www/foo.html** был изменен, сообщение-ответ сервера будет включать содержимое файла.

4.2.2. Управление доступом

Web-сервер может ограничивать доступ пользователей к определенным ресурсам. Управление доступом требует сочетания *аутентификации (проверка подлинности)* и *авторизации (проверки полномочий)*. При аутентификации определяется пользователь, инициировавший запрос, а при проверке полномочий выясняется, может ли пользователь иметь доступ к определенному ресурсу. Сначала мы рассмотрим, как Web-сервер осуществляет аутентификацию пользователя с помощью агента пользователя. Затем мы обсудим, как Web-сервер определяет, может ли пользователь иметь доступ к запрашиваемому ресурсу.

АУТЕНТИФИКАЦИЯ

Большинство систем клиент–сервер аутентифицируют пользователя, запрашивая у него имя и пароль. В этой модели на сервере имеется файл, содержащий имена и пароли всех зарегистрированных пользователей; с целью защиты информации пароли могут храниться в зашифрованном виде. Пользователь вводит имя и пароль в начале сеанса работы с сервером. Сервер проверяет это имя и пароль и сохраняет информацию о пользователе на данный сеанс. В определенный момент пользователь или сервер завершает сеанс. Для дальнейшего доступа пользователю необходимо инициировать новый сеанс работы с сервером, что подразумевает повторный ввод имени и пароля.

В Web отсутствует понятие сеанса работы пользователя с сервером. У пользователя создается впечатление сеанса работы с Web-сайтом, состоящего из последовательности запросов к Web-страницам. Сервер же, со своей стороны, видит лишь поток независимых HTTP-запросов. В известном смысле каждый запрос принадлежит своему собственному сеансу. Следовательно, Web-сервер должен выполнять аутентификацию для *каждого* запроса на ресурс, для которого предусмотрены ограничения доступа. Однако пользователь может посчитать обременительным вводить имя и пароль при каждом HTTP-запросе, включая запросы на встроенные изображения. Вместо этого сервер предписывает агенту пользователя включать информацию об имени и пароле в заголовок HTTP-запроса. Пользователь вводит имя и пароль, которые сохраняются до тех пор, пока пользователь не закроет браузер. Запоминание агентом пользователя имени и пароля дает возможность серверу выполнять аутентификацию, по-прежнему трактуя каждый запрос независимо. Любые промежуточные компоненты между агентом пользователя и исходным сервером просто пересылают информацию о пользователе.

Предположим, что пользователь щелкает мышью на гиперссылке, которая соответствует защищенному ресурсу на сервере **www.bar.com**. Браузер отправляет на Web-сервер HTTP-запрос. Сервер распознает, что запрашиваемый ресурс имеет ограничение на доступ. Сервер возвращает HTTP-ответ, который указывает, что запрос требует аутентификации. Ответ также указывает, какой *вид* аутентификации требуется. Сервер может выбрать один из многих типов аутентификации, включая обычную (basic) или с помощью дайджеста (digest) [FHBH+99]. Ответ также указывает на *область* (realm) — строку, которая объединяет множество ресурсов сервера. Каждое множество может иметь свою схему аутентификации с различными именами и паролями. Далее браузер запрашивает у пользователя имя, пароль и включает эту информацию в последующие запросы. Подробнее о HTTP-аутентификации будет рассказано в главе 7 (раздел 7.10).

ПРОВЕРКА ПОЛНОМОЧИЙ

Процесс аутентификации дает возможность серверу идентифицировать пользователя, обратившегося с HTTP-запросом. Для управления доступом к Web-ресурсам сервер должен реализовать стратегию проверки полномочий (авторизации). Серверу нужен эффективный способ определения, *какие* аутентифицированные пользователи могут иметь доступ к определенному ресурсу. Подобные стратегии поведения являются составной частью настройки сервера. Такая стратегия поведения обычно находит выражение в виде *списков управления доступом*, которые содержат пользователей, которым разрешен или запрещен доступ к ресурсу. Детали того, как создавать списки управления доступом и как хранить имена пользователей и пароли, зависят от программного обеспечения сервера. Вместо того чтобы иметь отдельный список управления доступом для каждого ресурса, на сервере может быть сделана настройка по умолчанию для всех ресурсов в каталоге или для всех ресурсов сервера. Ресурсы, требующие аутентификации, могут быть размещены в отдельном каталоге (например, **/www/private**).

Проверкой полномочий называется процесс применения этих стратегий управления доступом. При обработке запроса сервер должен определить, является ли запрашиваемый ресурс защищенным списком управления доступом. Например, ресурс **http://www.bar.com/private/index.html** может быть связан с HTML-файлом, хранящимся по адресу **/www/private/index.html**. Доступ к каждому подкаталогу может разрешен только для определенных пользователей. В связи с этим сервер должен определять стратегию управления доступом для каждого каталога на пути к запрашиваемому файлу. Если доступ к каталогу ограничен, сервер должен выполнить анализ списка пользователей с целью определения, может ли запрос быть выполнен. В дополнение к проверке имени пользователя сервер может разрешать или запрещать доступ к ресурсу на основе другой информации, ассоциированной с HTTP-запросом, такой как имя хоста или IP-адрес запросившего клиента. Однако реализация стратегии поведения при проверке полномочий, основанная на имени клиента или адресе, является довольно рискованной, поскольку некоторые запросы могут быть отправлены от имени прокси-сервера, а не агента пользователя. На самом деле пользователь, которому отказано в доступе к ресурсу, мог настроить браузер для отправки запросов через прокси-сервер, которому разрешен доступ к ресурсу.

Аутентификация HTTP-запросов может существенно повысить нагрузку на Web-сервер. Сервер должен проверить имя пользователя и пароль для идентификации пользователя и должен осуществить анализ аутентификационной информации в каждом каталоге на пути к запрашиваемому ресурсу. При сложных настройках сервера процесс проверки полномочий обходится достаточно дорого. Нагрузка может быть снижена за счет выполнения этих функций только по мере необходимости. Например, стратегия поведения для каталога верхнего уровня, например, **/www/public**, может предусматривать, что ни один из подкаталогов не требует проверки прав доступа. Это дает возможность серверу избежать дополнительных проверок в подкаталогах. Администратор сервера может ограничить проверку полномочий только для небольшого числа особо важных ресурсов, которые могут быть помещены в особый подкаталог. Это позволит минимизировать накладные расходы на аутентификацию и проверку полномочий, ограничивая в то же время доступ к данным там, где это необходимо.

4.2.3. Динамическое создание ответов

Помимо доставки статического содержимого, в Web имеются возможности для динамического создания ресурсов и доступа к ним. Эти возможности отличают Web от первых сервисов в Internet по передаче файлов. Динамически генерируемые ответы создаются различными способами. *Включения на стороне сервера* предписывают Web-серверу настраивать статические ресурсы с помощью директив, содержащихся в файле. В противоположность этому, *серверный сценарий* представляет собой отдельную программу, которая создает запрашиваемый ресурс. Программа может выполняться как в адресном пространстве сервера или как отдельный процесс, который взаимодействует с сервером. Динамическое создание сообщения-ответа обеспечивает высокую степень гибкости, но приводит к увеличению нагрузки на сервер, а также снижает безопасность.

ВКЛЮЧЕНИЯ НА СТОРОНЕ СЕРВЕРА

Создатели Web-содержимого часто осуществляют настройку Web-страницы на конкретного пользователя, обращающегося с запросом. Например, Web-страница может отображать текущее время, IP-адрес или имя клиента. Создатель Web-содержимого не обладает этой информацией на момент создания HTML-файла. Вместо этого файл может включать в себя директивы или макросы, инструктирующие сервер вставить информацию в процессе обработки запроса. Отвечая на клиентский запрос, сервер осуществляет синтаксический анализ файла и замещает текст каждого макроса. Макросы представляют собой специальные теги в файле, которые интерпретируются сервером. Например, файл может содержать директиву

```
<!-- #echo var="LAST_MODIFIED"-->
```

которая включает время последней модификации файла в ответ, отправляемый клиенту. Информация появляется в *теле* ответа в виде HTML-текста и будет отображена пользователю. Это никак не связано с включением сервером метаданных в заголовок HTTP-запроса.

Макросы могут ссылаться на множество разнообразных переменных, содержащих информацию о HTTP-запросе. Помимо простого замещения, макрос может требовать, чтобы сервер вставил в документ другой файл. Например, документ **/www/foo.html** может содержать макрос, который вставляет содержимое другого файла, **/www/infoo.html**, в HTML-код сообщения-ответа. Если обобщить, макрос может инструктировать сервер вызывать программу, указанную в макросе, отличную от программы, указанной в URL сообщения-запроса. Включения на стороне сервера предлагают создателям Web-содержания относительно простой способ настройки их документов путем внесения незначительных дополнений в HTML-код. Создателю содержимого при этом не нужно знать о других способах динамического создания HTML-содержания. Однако серверные включения требуют, чтобы Web-сервер осуществлял синтаксический анализ HTML-файлов и выполнял действия, запрашиваемые макросом. Это приводит к большему времени ожидания на стороне пользователя по сравнению с традиционной загрузкой статического HTML-файла.

Выполнение синтаксического анализа для каждого запроса может привести к неоправданному увеличению нагрузки сервера, особенно если большинство файлов не содержит макросов. Вместо этого сервер определяет, требует ли файл синтаксического анализа, основываясь на URL. В соответствии с принятым соглашением, такие ресурсы имеют расширение **.shtml** вместо **.html**. Другим получившим распространение файловым расширением является **.php**. Оно принято для страниц, обрабатывае-

мых PHP (Hypertext Preprocessor или Personal Home Page). PHP представляет собой язык сценариев, который не зависит от платформы, сценарии встраиваются в HTML-код и обладают более широкими возможностями по сравнению с простым замещением. Интерпретатор PHP выполняется под управлением Web-сервера, интерпретатор осуществляет синтаксический анализ и обработку файлов. PHP-файл может содержать данные, отправленные пользователем в HTML-формах. Так, если пользователь ввел имя и номер телефона, эти переменные могут быть включены в HTML-файл, возвращаемый сервером. В PHP также имеются гибкие средства поддержки работы с базами данных. Еще одним популярным расширением файлов является **.asp**. Оно принято для Microsoft Active Server Pages (ASP). ASP, подобно PHP, встраивает переменные и операторы сценариев непосредственно в HTML-файл. Однако ASP может использоваться только Web-серверами на платформе Microsoft.

СЕРВЕРНЫЕ СЦЕНАРИИ

Вместо того чтобы встраивать информацию в HTML-файл, некоторые программы могут генерировать весь ресурс. В этом случае URL в HTTP-запросе соответствует программе, а не документу. Программа может решать различные задачи, такие как доступ к информации в базах данных или создание ответа, содержание которого зависит от клиента, обращающегося с запросом. Поддержка выполнения сценариев по своему принципу аналогична разрешению пользователям «заходить» на сервер и выполнять программы. Однако при этом имеется несколько важных различий. Вместо прямого доступа к компьютеру сервера, пользователь указывает нужный сценарий и его параметры в HTTP-запросе. Сценарии формируют выходные данные в формате (например, HTML), который может быть интерпретирован Web-браузером. С точки зрения клиента и пользователя подобные динамически генерируемые ответы не отличаются от статических ресурсов за исключением того, что при этом может увеличиться время ожидания, связанное с выполнением сценария.

Исполнение сценария на сервере также отличается от загрузки программы (например, написанной на Java) для ее выполнения в Web-браузере. Выполнение программы в браузере больше подходит для приложений, которые взаимодействуют с браузером и с пользователем. В противоположность этому серверные сценарии имеют доступ к данным, которые могут быть недоступны где-либо еще. Например, Web-сервер может активизировать сценарий, выполняющий запрос пользователя на книги по определенной тематике. Ответ на запрос может вызвать необходимость обращения к данным, содержащим конфиденциальную информацию. Выполнение запроса на клиенте потребовало бы копирования всех этих данных. Кроме того, потребовалась бы программа, способная выполнить поиск. Выполнение программы как сценария на сервере гарантирует, что программа и данные не будут незаконно использованы. Программа со временем может быть модифицирована в случае обнаружения ошибок. В нее могут быть внесены усовершенствования, при этом пользователям не нужно устанавливать на своих компьютерах новые версии. Данные также могут быть изменены «незаметно» для пользователя.

Теоретически код, который генерирует динамический ответ, может быть интегрирован с программным обеспечением Web-сервера. Однако это нежелательно, поскольку потребует внесения изменений в программное обеспечение сервера каждый раз, когда добавляется новая функциональная возможность. Это также требует, чтобы каждый разработчик приложений был знаком с программным обеспечением сервера. Желательно было бы иметь одну группу программистов, разрабатывающих программное обеспечение Web-сервера, и другую группу программистов, потенциально более крупную, создающих приложения для динамического создания содер-

жания. Четкое разделение программного обеспечения сервера и сценариев имеет большое значение. Главная задача Web-сервера — связать запрашиваемый URL с соответствующим сценарием и передать данные в сценарий. Главная задача сценария — обработать входные данные и создать содержание для клиента. Сервер может взаимодействовать со сценарием различными способами:

- **Отдельный процесс, инициированный сервером.** Сценарий может выполняться как отдельный процесс, инициированный сервером для создания запрашиваемого ресурса. Наличие отдельного процесса изолирует сервер от сценария, но при этом для каждого запроса приходится затрачивать дополнительные усилия на создание и уничтожение процесса. Это традиционный подход, принятый для интерфейса Common Gateway Interface (CGI).
- **Программный модуль, выполняющийся в том же процессе.** Сценарий может быть оформлен как отдельный программный модуль, выполняющийся как часть Web-сервера. Вызов модуля в процессе сервера позволяет избежать дополнительных затрат, имеющих место при создании отдельного процесса, и уменьшить требования к системным ресурсам сервера. Подобный подход был применен в Netscape Server Application Programming Interface (NSAPI), Microsoft Internet Server Application Programming Interface (ISAPI), в модуле Apache *mod-perl,* а также в *сервлетах* Java Sun Microsystems.
- **Непрерывный процесс, взаимодействующий с сервером.** Сценарий может выполняться в отдельном процессе, который обрабатывает множество запросов в течение длительного периода времени. Сервер взаимодействует с процессом, передавая ему параметры и получая выходные данные. Наличие отдельного выполняющегося процесса избавляет от необходимости создавать и уничтожать процесс для каждого запроса; кроме того, процесс может обслуживать взаимодействие с другими сервисами, такими как системы управления базами данных. Подобный подход реализован в FastCGI [Fcg].

Разработка методов взаимодействия между Web-серверами и сценариями стала за последние несколько лет сферой значительной коммерческой активности. Это привело к появлению большого количества разнообразных интерфейсов прикладного программирования для написания сценариев.

ПЕРЕДАЧА ДАННЫХ В СЦЕНАРИЙ И ИЗ СЦЕНАРИЯ

Отделение сценариев от Web-сервера требует наличия строго определенного интерфейса для передачи данных между двумя компонентами программного обеспечения. Первым делом сервер должен определить, что запрашиваемый ресурс представляет собой сценарий, а не документ. URL, указывающие на сценарии, обычно содержат символ "?" или подстроку "cgi", "cgi-bin" или "cgibin". Вообще говоря, связывание URL со сценарием определяется настройками сервера. Например, сервер может быть настроен таким образом, что любой URL с определенным расширением (например, **.cgi**) или размещенный в определенном подкаталоге (например, **/www/scripts** или **/www/cgibin**), будет восприниматься как сценарий. После проверки того, что URL соответствует сценарию, сервер проверяет разрешение на выполнение сценария. Затем сервер запускает сценарий и ожидает его завершения, после чего отправляет ответ клиенту.

Наличие четко определенного интерфейса для обмена данными имеет важное значение для организации совместной работы сервера и сценария. Реализации интерфейса различаются в зависимости от используемой технологии. Тем не менее, все методы едины в том, что сервер передает входные данные сценарию и получает

от него выходные данные. В качестве иллюстрации рассмотрим подход, используемый в популярном интерфейсе Common Gateway Interface (CGI) [Gun96]. CGI определяет интерфейсы для большого числа платформ. Операционные системы отличаются способами, используемыми для обмена данными между приложениями. Это затрудняет реализацию единого интерфейса, применимого для всех платформ. Web-сервер на базе UNIX направляет данные сценарию через стандартный ввод и переменные окружения, а получает данные от сценария через стандартный вывод. Языки программирования, такие как Perl или C, имеют функции, которые возвращают значения указанных переменных окружения. В последующем обсуждении мы сосредоточим внимание на взаимодействии между UNIX-сервером и CGI-сценарием.

Таблица 4.1. Примеры переменных окружения CGI

Тип	Переменная	Пример
Информация о сервере	SERVER_NAME	www.bar.com
	SERVER_SOFTWARE	Apache/1.2.6
	SERVER_PROTOCOL	HTTP/1.0
	SERVER_PORT	80
	DOCUMENT_ROOT	/www
	GATEWAY_INTERFACE	CGI/2.0
Информация о клиенте	REMOTE_ADDR	10.9.57.188
	REMOTE_HOST	users.berkelly.edu
Информация о запросе	CONTENT_TYPE	text/html
	CONTENT_LENGTH	158
	REQUEST_METHOD	GET
	QUERY_STRING	name=Noam+Chomsky
	ACCEPT_LANGUAGE	de-CH
	HTTP_USER_AGENT	Mozilla/2.0

Web-сервер предоставляет сценарию разнообразную информацию, как показано в таблице 4.1. Информация о сервере содержит имя (доменное имя хоста или IP-адрес), название и номер версии программного обеспечения сервера, название и версию протокола, номер порта сервера и корневой каталог для ресурсов, размещенных на Web-сайте. Информация о клиенте включает в себя IP-адрес и имя компьютера клиента. Информация о запросе включает тип содержимого запроса, длину запроса и желательный формат для запрашиваемого ресурса. Другие поля HTTP-запроса доступны через переменные, имена которых начинаются с HTTP. Например, HTTP-запрос может содержать заголовок **User-Agent**, который идентифицирует тип браузера, сделавшего запрос; эта информация будет доступной через переменную окружения HTTP_USER_AGENT. Аналогично, переменная HTTP_COOKIE указывает на cookie, если они включены в HTTP-запрос. Переменные окружения дают возможность сценарию менять свои действия в зависимости от сервера, обратившегося с запросом клиента, и заголовков запроса. Например, сценарий может прочесть файл **neatdata.txt** из корневого каталога **/www** и преобразовать его содержимое для соз-

дания HTML-файла с IP-адресом клиента и приветствием на швейцарском диалекте немецкого языка.

CGI также предоставляет сценарию способ получать данные от пользователей, которые вводят их в HTML-формах. Вернувшись к примеру, предположим, что на Web-странице **http://www.bar.com/foo.html** отображается поле ввода, где пользователь может ввести текст и отправить его серверу нажатием кнопки Submit. Предположим, что пользователь ввел "Noam Chomsky" и щелкнул мышью на кнопке Submit, что заставляет браузер отправить HTTP-запрос **GET** на ресурс **http://www.bar.com/book.cgi?name=Noam+Chomsky**, либо запрос **POST** с URL **http://www.bar.com/book.cgi,** содержимое запроса передается в теле сообщения. После получения запроса сервер вызывает сценарий **/www/book.cgi**. В дополнение к установке переменных окружения сервер передает сценарию значения параметров через дополнительную переменную окружения (QUERY_STRING для метода GET) или через стандартный ввод (для метода POST).

Сценарий просматривает клиентский ввод (если он имеется) и переменные окружения. При обработке запроса сценарий может взаимодействовать с другими серверами или базами данных. Например, сценарий **/www/book.cgi** может выдать запрос к базе данных для получения информации о книгах, написанных Ноамом Хомски. При записи сообщения-ответа в стандартный поток вывода сценарий может создать HTTP-заголовки от имени сервера. Например, сценарий может создать заголовок, в котором указываются на размер и тип тела сообщения (например, GIF-изображение размером 3576 байтов), или предоставить URL, который переадресует клиента к другой Web-странице. Web-сервер может быть настроен для получения заголовка ответа, предоставленного сценарием, без модификации; это называется *сценарием без синтаксического анализа заголовка.* В других случаях Web-сервер анализирует выходные данные сценария для включения в них отсутствующей информации (например, заголовка **Date** с текущими датой/временем) до пересылки ответа клиенту.

ВЛИЯНИЕ СЦЕНАРИЕВ НА ПРОИЗВОДИТЕЛЬНОСТЬ И БЕЗОПАСНОСТЬ

Выполнение сценариев на сервере оказывает существенное влияние на производительность и безопасность. При запуске сценария и передаче данных в обоих направлениях потребляются ресурсы сервера. Кроме того, сам сценарий может выполнять вычисления, осуществлять доступ к базам данных, либо связываться с другими серверами. Эти операции создают задержки ответов на клиентские запросы. Рассмотрим пользовательский запрос, который требует взаимодействия с базой данных, размещенной на другом компьютере, а не на компьютере сервера. Сценарий, выполняющийся на компьютере сервера, должен связаться с удаленным компьютером для передачи запроса к базе данных. Ответ задерживается на время, необходимое для передачи запроса по сети и операций с базой данных. Кроме того, сценарии обычно пишутся на языках, таких как Perl, которые компилируются во время выполнения. Затраты на компиляцию сценария увеличивают время ожидания ответа. Предварительно компилируемые языки, такие как C, более эффективны в этом смысле, но обычно программирование на них более трудоемко, особенно при работе со строками, что чаще всего встречается в Web-сценариях.

Запуск сценариев на выполнение с помощью URL значительно расширяет функциональные возможности Web. Однако при написании сценария следует учитывать, какие функции он выполняет и какие системные ресурсы потребляет. В самом худшем случае плохо написанный сценарий может никогда не закончить свою работу и расходовать значительную долю ресурсов процессора и памяти. Как ре-

зультат, Web-серверы обычно накладывают ограничения на ресурсы процессора и память, предоставляемые сценарию. Следует также иметь в виду, что администратор Web-сайта может запретить использование сценариев или наложить ограничения на круг лиц, которым разрешается устанавливать новые сценарии. Использование сценариев также сопровождается проблемами, связанными с безопасностью. Сценарий может выполнять многочисленные функции, неожиданные для разработчиков Web-сервера. Сценарий может осуществлять доступ к файлам на компьютере сервера. Плохо написанный сценарий может изменить данные или вернуть запросившему клиенту информацию, не подлежащую разглашению.

Сценарии, которые получают данные от пользователя, требуют дополнительных предосторожностей, чтобы гарантировать, что пользователь не сможет через сценарий выполнить несанкционированные действия. Рассмотрим пример сценария, который дает возможность пользователю вводить произвольные команды для их выполнения на компьютере сервера. Пользователь может с помощью такого сценария удалить файлы на компьютере сервера или передать пользователю конфиденциальную, не предназначенную для него информацию (например, с помощью электронной почты). Подобных ситуаций можно избежать несколькими способами. Во-первых, сценарий может быть написан таким образом, чтобы ограничить вводимую пользователем информацию. Например, Web-страница может предоставлять пользователю возможность выбора из ряда предопределенных действий или ввода списка ключевых слов вместо команд для выполнения их на компьютере сервера. Во-вторых, сценарий может осуществлять предварительную обработку пользовательских данных, чтобы избежать выполнения операций, которые дают несанкционированный доступ к программам или данным. В-третьих, администратор сайта может ограничить доступ к самим сценариям. Например, сценариям может быть запрещен доступ к файлам, содержащим важные данные (например, файлу паролей), и к определенным командам (например, отправка электронной почты или удаление файлов). Администратор сайта может также принудить сценарий выполняться с теми же правами доступа, которые имеет человек, написавший сценарий, предотвращая тем самым доступ сценария к файлам, к которым не имеет доступа автор сценария.

4.2.4. Создание и использование cookies

Cookies дают возможность сохранять информацию о пользователе в течение определенного времени, о чем шла речь в главе 2 (раздел 2.6). Web-сайты используют cookies для отслеживания пользователей и хранения информации о транзакциях, которые выполняются с помощью многочисленных HTTP-взаимодействий. Cookies обычно создаются, используются и модифицируются сценариями, вызываемыми для динамического создания ответов, а не Web-сервером.

Web-сайт может адаптировать содержание передаваемое пользователю, обратившемуся с запросом. Например, сайт электронной коммерции может возвратить Web-страницу с персонифицированным приветствием, которое содержит имя пользователя и рекомендации по возможным покупкам. Отслеживание пользователей дает возможность Web-сайту создавать профили отдельных пользователей и накапливать статистику о поведении пользователей при обращениях к сайту, например, среднее время, проведенное пользователем на сайте. Однако HTTP-запрос не предоставляет достаточной информации для идентификации пользователей. Множество пользователей могут просматривать Web-содержимое с одного компьютера или направлять свои запросы через один и тот же прокси-сервер. Кроме

того, IP-адрес компьютера может время от времени изменяться, если провайдер Internet динамически предоставляет IP-адрес при соединении пользователя с Internet. Теоретически сайт может потребовать от пользователя указания уникального имени и, возможно, пароля для каждого запроса, но это будет обременительно для пользователя и вызвать у него раздражение.

Вместо этого можно указать браузеру включать уникальный cookie в каждый HTTP-запрос. Вернувшись к нашему примеру, предположим, что Web-сервер получает HTTP-запрос на ресурс **http://www.bar.com/book.cgi?name=Noam+Chomsky**. Запрос содержит cookie. Сервер вызывает сценарий **/www/book/cgi** и передает cookie через переменную окружения HTTP_COOKIE. Сценарий может использовать cookie для определения, какие рекламные объявления и предложения следует поместить на Web-страницу. Например, предложения могут быть связаны с книгами по темам, соответствующим предыдущим покупкам данного пользователя. С точки зрения браузера cookie представляет собой произвольную строку символов. Сценарий же, со своей стороны, связывает с этой строкой определенный смысл. В простейшем случае строка представляет собой просто уникальный идентификатор пользователя, такой как имя пользователя или числовой код. Если запрос не содержит cookie, сценарий может создать новый cookie и включить его в заголовок сообщения-ответа

```
Set-Cookie: Customer="user17"; Version="1"; Path="/book"
```

Последующие запросы от этого пользователя будут содержать cookie

```
Cookie: Customer="user17"; Version="1"; Path="/book"
```

Сценарий может использовать cookie как идентификатор пользователя при взаимодействии с внутренней базой данных. Например, сайт электронной коммерции может использовать базу данных, хранящую информацию о последних заказах и текущем содержимом магазинной тележки каждого пользователя. База данных сохраняет свое состояние между последовательными запросами в отличие от Web-сервера, который обрабатывает каждый запрос независимо, вызывая сценарий, взаимодействующий с базой данных.

В других ситуациях cookie может содержать дополнительную информацию, такую как имя пользователя и цвета, которым он отдает предпочтение. Это полезно для динамического создания Web-содержимого, настраиваемого на основе указанных атрибутов. Например, Web-страница может содержать персональное приветствие, включающее имя пользователя. Cookie может также содержать информацию о предыдущих действиях пользователя по просмотру Web-сайта. Хранение накопленной информации в cookie может избавить от необходимости сохранять информацию о пользователе во внутренней базе данных. Например, в cookie сайта электронной коммерции может храниться текущий список заказанных пользователем товаров. Допустим, на сайте электронной коммерции пользователь заполняет различные формы для заказа книг. Пользователь добавляет книгу в список заказанных товаров, что заставляет браузер послать HTTP-запрос, содержащий cookie, ассоциированный с этим Web-сайтом. Сценарий создает сообщение-ответ, включающий в себя заголовок

```
Set-Cookie: Order="Chomsky_Bio"; Version="1"; Path="/book"
```

Затем пользователь добавляет в список еще одну книгу, что инициирует HTTP-запрос, включающий в себя

```
Cookie: Customer="user17"; Version="1"; Path="/book"
        Order="Chomsky_Bio"; Version="1"; Path="/book"
```

Процесс продолжается, если пользователь заказывает другие книги. В этом примере все важные данные включены в cookie. Однако при таком подходе cookie может иметь слишком большие размеры. Браузер может не разрешать хранения cookie, превышающего некоторую максимальную длину. Чтобы избежать создания больших cookies, сценарий может использовать cookie лишь для хранения идентификатора пользователя, а содержимое списка заказов пользователя хранить в базе данных на сервере.

Браузер трактует cookie как строку, передаваемую от имени пользователя. Однако достаточно искушенные пользователи могут совместно использовать, создавать или модифицировать cookies. Предположим, что Web-сайт не позволяет пользователю загрузить определенный ресурс, если HTTP-запрос не содержит cookie. Пользователи могут сформировать свои собственные cookies, чтобы не допустить отслеживания сайтом их запросов. Кроме того, один пользователь может «позаимствовать» допустимый cookie у другого пользователя, модифицировав строку в серверном ответе **Set-Cookie**. Обратившись к предыдущему примеру, можно предположить, что искушенный пользователь отправит HTTP-запрос, содержащий cookie, заменив строку **user17** на **user8**. Здесь **user8** соответствует корректному пользователю. Чтобы устранить подобные проблемы, cookie может включать некоторую зашифрованную информацию, не дающую пользователям возможность создавать корректные cookie от своего имени. Как часть обработки запроса сценарий может осуществлять проверку cookie на корректность. Хотя это предотвращает использование фиктивных cookies, все еще остается риск, что пользователь воспользуется cookie, принадлежащем кому-либо другому. Чтобы предотвратить подобные действия, для cookie может быть назначено время жизни. По истечении времени жизни cookie пользователю необходимо принять новый cookie для получения доступа к сайту.

4.3. Совместное использование информации несколькими запросами

Хотя Web-сервер обрабатывает каждый запрос независимо, он может сохранять некоторую информацию с целью уменьшения издержек при обслуживании последующих запросов. Обрабатывая запрос, сервер может сохранить тело HTTP-ответа или часть HTTP-ответа в оперативной памяти для дальнейшего использования. Аналогично, сервер может сохранять информацию, сформированную в процессе обработки запроса. В обоих случаях сервер должен принять меры, чтобы не отправить клиенту устаревшую информацию.

4.3.1. Совместное использование HTTP-ответов несколькими запросами

На практике лишь небольшая часть ресурсов Web-сайта используется при обработке большинства запросов. Сервер может уменьшить издержки на обработку этих запросов, сохранив часто запрашиваемые ресурсы в оперативной памяти. Предположим, что сервер получает запрос на ресурс **http://www.bar.com/foo.html**, представляющий собой статический файл. Чтобы обработать запрос, сервер должен открыть и прочитать файл, соответствующий этому URL. При чтении файла данные копируются с диска в оперативную память. Обращение к диску увеличивает нагрузку системы и приводит к задержке при обработке запроса. В идеале следующий запрос на

/foo.html не должен вызывать повторное открытие и чтение файла. Вместо этого сервер может передать данные непосредственно из оперативной памяти. Это называется *кэшированием на стороне сервера*, поскольку данные с диска кэшируются в оперативной памяти сервера.

Кэширование ресурса на Web-сервере несколько отличается от кэширования в браузере или в прокси-сервере. С целью увеличения производительности сервер хранит данные в оперативной памяти. Серверу необходимо убедиться, что копия в оперативной памяти совпадает с данными, хранящимися на диске. Если содержимое файла изменяется, сервер не должен передавать копию, хранящуюся в оперативной памяти. Чтобы избежать передачи устаревшего ответа, сервер может проверить время последней модификации файла на диске до возврата кэшированной копии; если файл был изменен, сервер может удалить кэшированную копию из оперативной памяти. Альтернативой является уведомление сервера системой о том, что файл был модифицирован. При этом сервер удаляет кэшированное содержимое из памяти. В данном случае сервер может посылать ответ запросившему клиенту без явной проверки времени последней модификации файла.

Кроме кэширования статических файлов, Web-сервер может также сохранять в оперативной памяти динамически генерируемые ответы. Рассмотрим Web-сайт, который предоставляет пользователю возможность вводить строку и получать гиперссылки на Web-страницы, содержащие ее. Многие запросы пользователей содержат одну и ту же строку для поиска. Вместо того чтобы обрабатывать каждый запрос независимо, сервер может сохранить результаты выполнения таких запросов в оперативной памяти. Сервер может обработать результаты, соответствующие кэшированному запросу, вместо того, чтобы вызывать сценарий, взаимодействующий с внутренней базой данных. Результат выполнения запроса может состоять из нескольких Web-страниц, и пользователь может выбрать вторую страницу после того, как просмотрит первую. Сервер может сформировать вторую страницу на основе кэшированных результатов выполнения первого запроса вместо того, чтобы генерировать содержимое второй Web-страницы «с нуля». Кэширование динамически генерируемых ответов потенциально уменьшает нагрузку на сервер особенно для сайтов, задача которых в основном состоит в динамическом формировании ответов на клиентские запросы. Как и в случае статического содержания, серверу нужно обеспечить, чтобы кэшированный результат был актуальным. Добавление новых данных в базу данных приводит к необходимости удалить результаты выполнения предыдущих запросов из оперативной памяти.

4.3.2. Хранение метаданных между запросами

В дополнение к кэшированию Web-ресурсов, сервер может сохранять информацию, сформированную в процессе обработки HTTP-запроса, одним из следующих способов.

- **Преобразование URL в путь к файлу в файловой системе.** Сервер должен установить соответствие между URL в сообщении-запросе и путем к файлу в файловой системе сервера. Кэширование результатов преобразования избавляет от необходимости повторно выполнять его для последующих запросов того же URL. Однако сервер должен гарантировать, чтобы кэшированная информация была актуальной; внесение изменений в настройки сервера может привести к изменению связи между URL и путем в файловой системе сервера.

- **Управляющая информация о ресурсе.** Сервер может также кэшировать дескриптор файла, идентифицирующий местонахождение файла в файловой системе. Кроме того, сервер может кэшировать основные атрибуты файла, такие как размер и время последней модификации. Это сокращает затраты на построение заголовка HTTP-ответа для будущих запросов на этот же ресурс. Однако сервер должен гарантировать, что кэшированные атрибуты не будут использоваться, если файл будет изменен.
- **Заголовки HTTP-ответа.** Сервер может кэшировать часть заголовка HTTP-ответа. Многие заголовки в сообщении-ответе несут в себе атрибуты запрашиваемого ресурса (например, размер, тип содержимого и время последней модификации). Кэширование этих заголовков избавляет сервер от необходимости формировать их при обработке будущих запросов. Однако сервер не должен кэшировать весь заголовок ответа, поскольку некоторые поля (например, дата/время ответа) могут измениться с момента предыдущего запроса на тот же ресурс.

Кэширование информации между последовательными ответами для одного и того же ресурса снижает нагрузку на сервер, давая ему возможность обработать больше запросов.

Помимо кэширования информации между несколькими запросами на *один и тот же* ресурс, сервер может кэшировать определенную информацию для запросов на различные ресурсы:

- **Текущие дата/время.** Текущие дата/время присутствуют в заголовке HTTP-ответа и в журнале сервера. Определение текущего времени требует активизации системного вызова (например, *gettimeofday()* в UNIX), что сказывается на увеличении загрузки сервера. Значение времени в заголовке указывается с точностью до секунды. В то же время сервер за секунду может обработать сотни запросов. Вместо того чтобы повторно делать системный вызов для каждого запроса, сервер может воспользоваться одним и тем же значением времени для серии запросов. Например, сервер может делать системный вызов после каждой группы из нескольких запросов, а также при относительном бездействии. Однако кэширование результатов системного вызова в течение достаточно большого времени может ограничить точность информации о значении времени, записываемом в журнал сервера.
- **Доменное имя клиента.** В некоторых случаях обработка запроса может зависеть от доменного имени клиента, сделавшего запрос. Например, сервер может ограничить доступ к ресурсу или произвести настройку ответа для клиентов из определенных доменов (например, **.edu** или **.competitor.com**). Преобразование IP-адреса в доменное имя компьютера клиента требует от сервера активизации системного вызова (например, *gethostbyaddr()*). За счет кэширования этой информации сервер может избежать повторения процесса преобразования IP-адреса в имя для будущих запросов с того же IP-адреса. Однако через некоторое время IP-адрес и доменное имя могут перестать соответствовать друг другу.

Кэширование информации между запросами на разные ресурсы позволяет повысить эффективность работы сервера.

4.4. Архитектура сервера

Web-сервер обычно обрабатывает множество клиентских запросов одновременно. Эти запросы должны осуществлять совместный доступ к процессору, к дисковой и оперативной памяти, сетевому интерфейсу сервера. В этом разделе мы обсудим способы распределения системных ресурсов между конкурирующими клиентскими запросами. Сервер с *управлением по событиям* осуществляет обработку запросов в одном процессе, который попеременно обслуживает различные запросы, в то время как сервер с *управлением по процессам* создает для каждого запроса отдельный процесс. В гибридных схемах делаются попытки использовать преимущества каждого из этих подходов. Хотя указанные подходы были достаточно подробно исследованы и до появления Web, выбор архитектуры сервера оказывает существенное влияние на производительность.

4.4.1. Архитектура серверов с управлением по событиям

Простейшим подходом является сервер с единственным процессом, который обслуживает один запрос за раз. Процесс принимает клиентский запрос, генерирует и передает ответ клиенту, после чего переходит к следующему запросу. Естественным расширением для Web-сервера, обрабатывающего запросы в одном процессе, является переключение с одного обрабатываемого запроса на другой. Вместо того чтобы обработать запрос с начала до конца, процесс периодически выполняет небольшую часть работы для каждого обрабатываемого запроса. Подобный подход с управлением по событиям наиболее приемлем в тех случаях, когда каждый запрос требует небольшого объема работы. Обработка Web-запроса легко подразделяется на небольшое количество действий: соединение с клиентом, чтение HTTP-запроса, нахождение соответствующего ресурса и передача ответа. Во многих случаях сервер может завершить каждое из этих действий за короткий период времени.

Однако некоторые из этих действий могут породить задержки, неподконтрольные Web-серверу. Например, сервер не может начать генерировать ответ, пока не будет получено сообщение-запрос от клиента. Точно так же, извлечение данных с диска может задержать передачу сообщения-ответа. Ожидая завершения операции, сервер должен иметь возможность переключиться на другой запрос. Это требует тщательного анализа операций, которые могут заблокировать процесс на сервере. Web-сервер активизирует системные вызовы операционной системы для выполнения низкоуровневых задач, относящихся к сетевым соединениям и операциям с дисковой памятью. Чтобы избежать ожидания завершения выполнения этих задач, Web-сервер с управлением по событиям обычно осуществляет неблокирующие системные вызовы, которые позволяют вызвавшему процессу продолжать выполнение, ожидая ответа от операционной системы. При завершении выполнения системной операции возбуждается событие, которое должно быть обработано серверным процессом.

В любой заданный момент времени сервер может иметь одно или более событий, связанных с различными Web-запросами, обработка которых не закончена. Обработка событий может инициировать дополнительные системные вызовы, которые, в свою очередь, приводят к возбуждению новых событий. Например, предположим, что клиент запрашивает статический ресурс, хранящийся на диске сервера. После получения клиентского запроса и преобразования URL в путь к файлу в файловой системе сервер инициирует событие для извлечения соответствующего файла с диска. В процессе ожидания поступления данных с диска сервер может обработать другой запрос. Обработка всех запросов в одном процессе дает возмож-

ность серверу последовательно упорядочить действия, которые модифицируют одни и те же данные. Это особенно важно применительно к Web, поскольку HTTP-запросы могут инициировать создание или модификацию Web-ресурсов. Предположим, что два пользователя пытаются одновременно изменить один и тот же ресурс. Сервер с управлением по событиям может легко сделать так, чтобы одна операция записи завершилась до того, как будет начата следующая. Кроме того, обработка запросов в рамках одного процесса облегчает совместное использование данных разными запросами. Предположим, например, что два пользователя выдают один и тот же запрос на поиск. Сервер может кэшировать результат выполнения первого запроса и вернуть тот же самый ответ для второго запроса.

Выполнение запросов в одном процессе с управлением по событиям имеет смысл, если каждое событие вносит небольшую задержку, величина которой не может превышать некоторого значения. Например, синтаксический анализ HTTP-запроса представляет собой относительно простую задачу. Однако другие операции, такие как ожидание получения HTTP-запроса, могут обусловить длительную задержку. К счастью, операционная система берет на себя эти функции, давая возможность серверному процессу обрабатывать в это время другие запросы. Тем не менее, некоторые HTTP-запросы вносят более существенную задержку, чем другие. Написание серверного программного обеспечения, которое позволяет явно переключаться между запросами, существенно осложняет разработку программных средств для сохранения промежуточных результатов каждого запроса. Например, динамически генерируемые Web-ответы обычно требуют от сервера выполнения сценария. Сервер может не знать заранее, как долго будет выполняться сценарий. Если допустить неопределенно долгое выполнение сценария, то это может привести к большим задержкам для других запросов. Вместо этого следует выполнять сценарий в отдельном процессе, чтобы дать возможность серверу обработать другие запросы.

Сервер с управлениям по событиям существенно зависит от эффективной поддержки операционной системой неблокирующих системных вызовов. Поддержка неблокирующих системных вызовов отличается для различных операционных систем. На практике неблокирующие операции, которые взаимодействуют с сетевым интерфейсом, обладают хорошим быстродействием. Взаимодействие с дисковой подсистемой вызывает больше проблем. Например, неблокирующий системный вызов для чтения файла с диска может, тем не менее, заблокировать вызвавший процесс в ходе выполнения операции с диском. Это может снизить производительность Web-сервера и увеличить время ожидания ответа. Подобные ограничения операционной системы вместе с проблемами при разработке серверного программного обеспечения являются доводами против архитектуры с управлением по событиям. Как результат, большинство высокопроизводительных Web-серверов не используют подобную архитектуру. Несмотря на это, некоторые экспериментальные Web-серверы реализуют модель с управлением по событиям, а элементы, присущие подходу с управлением по событиям, находят применение в других архитектурах.

4.4.2. Архитектура серверов с управлением по процессам

Альтернативой архитектуре с управлением по событиям является архитектура, в которой сервер создает отдельный процесс для каждого запроса. При этом подходе отдельный процесс выполняет все действия по обработке данного запроса. Выполнение множества процессов дает серверу возможность обрабатывать множество запросов одновременно. В противоположность подходу с управлением по событиям, модель с управлением по процессам зависит от возможностей операционной системы

осуществлять переключения между различными процессами. Операционная система выполняет процесс некоторый период времени (допустим, 10 мс или 100 мс), прежде чем переключиться на другой процесс. При этом создается впечатление, что каждый процесс выполняется на отдельном компьютере. Кроме того, если процесс блокируется ожиданием завершения системного вызова, операционная система начинает выполнять другой процесс. Например, в то время как один процесс ожидает получения данных с диска, может выполняться другой процесс.

Web-сервер с управлением по процессам обычно имеет один главный процесс, который прослушивает новые клиентские соединения. Для каждого нового соединения главный процесс создает отдельный процесс для обслуживания соединения. После синтаксического анализа клиентского запроса и передачи ответа процесс завершается. Завершение процесса после обработки запроса защищает систему от определенных разновидностей ошибок программирования. Процесс, который не освобождает ресурсы памяти, может постепенно захватывать все больше памяти. Это называется *утечкой памяти*. При завершении процесса операционная система автоматически освобождает ресурсы, выделенные процессу. Простота такого подхода дает возможность разработчикам сосредоточиться на базовой составляющей серверного программного обеспечения (синтаксический анализ запросов и создание ответов). Большинство первых Web-серверов следовали этой модели, в том числе серверы CERN и NCSA [KMR95].

Затраты на создание и завершение процесса составляют значительную часть работы при обработке HTTP-запроса для небольшого статического ресурса. Для снижения этих затрат при запуске сервера могут быть созданы несколько процессов. После создания процесс обслуживает запросы один за другим. При установлении нового соединения главный процесс предоставляет существующий, неактивный процесс для обработки запроса, а не создает новый процесс. Помимо снижения нагрузки на сервер, использование существующего процесса сокращает время ожидания пользователем. Однако подобный подход чувствителен к ошибкам программирования, ведущим к утечкам памяти. Даже если программное обеспечение Web-сервера свободно от таких ошибок, многие Web-сайты используют библиотеки или сценарии, написанные сторонними программистами. Такое программное обеспечение может вызвать утечки памяти. Для решения подобных проблем сервер может завершать процесс после обработки определенного числа запросов. В качестве альтернативы Web-сервер может воспользоваться одним из усовершенствованных методов управления памятью, которые ограничивают объем оперативной памяти, выделяемый каждому процессу. Подробнее об этом будет говориться при рассмотрении Web-сервера Apache в разделе 4.6.1.

Несмотря на повышение производительности, которое достигается при использовании пула заранее созданных процессов, подход с управлением по процессам имеет несколько ограничений. Во-первых, переключение с одного процесса на другой сопровождается дополнительными накладными расходами. Операционная система должна сохранить информацию о выполняющемся процессе, обновить различные таблицы и списки перед загрузкой информации о следующем процессе; в подходе с управлением по событиям структуры данных для активных запросов хранятся в одном процессе. В модели с управлением по процессам переключение с одного процесса на другой может потребовать загрузки данных с диска в оперативную память, или из оперативной памяти в кэш процессора. Большинство Web-запросов не требуют значительных вычислений, поэтому переключение между процессами составляет достаточно большую часть работы, выполняемой сервером. Во-вторых, дополнительные затраты возникают, если процесс предполагает

совместное использование данных. Каждый процесс представляет собой отдельную программу со своим адресным пространством. В Web большинство запросов выполняется для относительно небольшого числа ресурсов. Хранение этих часто запрашиваемых ресурсов в оперативной памяти дает возможность избежать затрат на извлечение или создания ответов «с нуля» для каждого запроса. Для сервера с управлением по процессам совместное использование ресурсов в оперативной памяти требует координации между процессами.

4.4.3. Серверы с гибридной архитектурой

Попытка объединить сильные стороны моделей с управлением по событиям и с управлением по процессам привела к созданию серверов с гибридной архитектурой. Например, модель с управлением по процессам может быть переработана таким образом, чтобы каждый процесс мог обслуживать более одного запроса одновременно. Фактически каждый процесс при этом становится сервером с управлением по событиям, который попеременно переключается между небольшим количеством запросов в наборе. Тем самым сокращается число процессов и снижаются затраты на совместное использование информации несколькими запросами, направляемыми одному процессу. Однако такой подход сохраняет ряд недостатков, присущих подходам с управлением по событиям и с управлением по процессам. Подобно серверам с управлением по событиям, такой подход может привести к тому, что задержка в при обработке одного запроса может привести к задержкам в обслуживании других запросов. Ограничение числа запросов, ассоциированных с каждым процессом, уменьшает вероятность возникновения подобных проблем. Как и серверы с управлением по процессам, гибридный сервер несет дополнительные затраты при переключении между процессами и совместном использовании данных несколькими процессами. Тем не менее, обслуживание нескольких запросов в одном процессе сокращает количество процессов и дает возможность совместного использования ресурсов внутри процесса.

Второй подход позволяет избежать затрат на переключение между процессами. Некоторые операционные системы позволяют процессу иметь множество независимых *программных потоков*[1]. Каждый программный поток выполняется в адресном пространстве породившего его процесса. Многопоточный Web-сервер может создать программный поток для каждого запроса. Сервер будет иметь один процесс, подобно серверу с управлением по событиям, с множеством различных программных потоков, подобно серверу с управлением по процессам. В отличие от сервера с управлением по событиям, многопоточному процессу не нужно координировать переключения между программными потоками явным образом. Когда один программный поток выполняет блокирующий системный вызов, в том же процессе могут выполняться другие программные потоки. Затраты на переключение между программными потоками существенно меньше, чем на переключение между процессами, поскольку программные потоки в процессе совместно используют адресное пространство процесса. Это дает возможность программным потокам совместно использовать и информацию о запросах. Например, программные потоки могут иметь совместный доступ к часто используемым ресурсам и заголовкам HTTP-ответов. Тем не менее, программные потоки должны использовать средства синхронизации для совместного доступа к данным. Разработка программного обеспечения для многопоточного сервера обычно является слож-

[1] Иногда программный поток (thread) называют облегченным или легковесным процессом. — *Прим. ред.*

ной задачей. Программные потоки, выполняющиеся в одном процессе, не защищены друг от друга; без тщательной координации один программный поток может испортить данные, необходимые другому программному потоку. Кроме того, отсутствие поддержки программной многопоточности в некоторых операционных системах затрудняет разработку многопоточного сервера, который мог бы выполняться на различных платформах.

Третий подход объединяет явные преимущества моделей с управлением по событиям и с управлением по процессам в гибридной схеме [PDZ99]. Подход с управлением по событиям хорошо использовать для обработки запросов, которые не потребляют значительных ресурсов процессора и дисковой подсистемы. Многие Web-запросы порождают короткие сообщения-ответы, не требующих доступа к дисковой памяти. Допустим, клиент запросил несуществующий ресурс. Распознав, что URL не соответствует какому-либо файлу или сценарию на сайте, сервер отвечает коротким сообщением об ошибке. Кроме того, на сервере в оперативной памяти может быть кэшировано множество часто используемых Web-ответов, что дает возможность избежать обращения к дисковой памяти. Это предполагает гибридную архитектуру сервера, которая по умолчанию применяет подход с управлением по событиям. Сервер имеет главный процесс с управлением по событиям, который обслуживает начальные этапы обработки каждого запроса. Если запрос требует значительных вычислений (например, если сервер должен активизировать сценарий) или доступа к дисковой памяти (например, при обработке запросов, требующих извлечения файлов с диска), то главный процесс передает выполнение операции, требующий много времени, вспомогательному процессу. Процесс с управлением по событиям может передать ответ клиенту, используя неблокирующие системные вызовы.

4.5. Размещение Web-серверов

До сих пор в этой главе подразумевалось, что каждый Web-сервер выполняется на отдельном компьютере. Однако это не всегда так. На практике на одном компьютере может быть размещено содержимое многих Web-сайтов. Кроме того, популярный Web-сайт может быть реплицирован на нескольких компьютерах.

4.5.1. Несколько Web-сайтов на одном компьютере

На первых порах существования Web отдельные лица и учреждения, желающие иметь Web-сайт, устанавливали программное обеспечение Web-сервера на отдельном компьютере. Администрирование Web-сайтов осуществлялось локально. По мере коммерциализации Internet все чаще используется подход, при котором Web-сайты устанавливаются на компьютерах и обслуживаются хостинговой компанией. Компания, занимающаяся Web-хостингом, может иметь группу компьютеров в одном или в нескольких местах. Разделение ролей создателя содержания и хостинг-провайдера имеет несколько преимуществ.

- Хостинговая компания берет на себя всю заботу по эксплуатации Web-серверов, избавляя организации от необходимости иметь опытных сотрудников, занимающихся обслуживанием Web-серверов.

- Хостинговая компания обеспечивает вычислительную и сетевую инфраструктуру для поддержки всех потребностей Web-сайта, избавляя создателей информационного содержания от необходимости приобретения оборудования и программного обеспечения.
- Хостинговая компания может разложить затраты на вычислительные и сетевые ресурсы на большое число Web-сайтов.
- Владелец содержания защищен от риска раскрыть свои конфиденциальные данные в Internet, разрешив доступ к своим компьютерам. Кроме того, трафик к и от Web-сайта не создает нагрузку на IP-сеть организации.

Однако владелец содержания становится существенно зависимым от хостинговой компании в плане предоставления эффективного и надежного доступа к Web-сайту. Помимо того, внесение изменений требует определенной координации действий с хостинговой компанией.

Хостинговая компания обычно поддерживает множество Web-сайтов и дает возможность создателям содержания обновлять его. Многие Web-сайты не слишком популярны, чтобы потреблять все ресурсы процессора, дисковой подсистемы и оперативной памяти компьютера. Некоторые сайты очень часто посещаемы в определенное время суток и относительно мало загружены в остальное время. Например, сайты электронной коммерции могут быть активными в течение дня, а развлекательные сайты — в вечернее время. Размещение нескольких Web-серверов на одном компьютере позволяет хостинговой компании эффективно снижать затраты. Кроме того, объединение на компьютере Web-сайтов, которые используют различные ресурсы, позволяет воспользоваться преимуществами единой, целостной системы. Например, сайт с динамическим содержанием может потреблять значительные процессорные ресурсы, в то время как сайт с большим объемом статического содержания может существенно загружать дисковую подсистему. Выполнение этих сайтов на одном компьютере позволит эффективно использовать и процессор, и дисковую подсистему.

При размещении и обслуживании нескольких сайтов на одном компьютере требуется иметь способ направления клиентских запросов соответствующему сайту. Пусть компания **hostmany.com** размещает содержание, предоставляемое компаниями **foo.com** и **bar.com**, на одном компьютере. Самый простой подход — иметь компьютер, выполняющий один сервер **www.hostmany.com**, который принимает запросы, адресованные и тому, и другому сайту. Например, компании могут иметь отдельные наборы файлов на диске, а все URL должны начинаться с **http://www.hostmany.com/foo/** или с **http://www.hostmany.com/bar/** для указания, к какому набору файлов на сервере должен быть осуществлен доступ. Однако этот подход для многих создателей содержания может оказаться неприемлемым. Большинство создателей содержания предпочитает иметь собственные доменные имена, зарегистрированные в Internet. Например, две компании могут захотеть, чтобы их Web-сайты носили доменные имена **www.foo.com** и **www.bar.com**.

Чтобы позволить использование отдельных доменных имен, хостинговая компания может установить и исполнять несколько Web-серверов на одном компьютере. Такие серверы называются *виртуальными серверами*. Каждый виртуальный сервер имеет свое собственное дерево документов, параметры настройки и работает, как если бы он был единственным Web-сервером на компьютере. Предположим, что серверы **www.foo.com** и **www.bar.com** выполняются на одном и том же компьютере. Запрос **http://www.foo.com/index.html** будет означать обращение к ресурсу **/index.html** на виртуальном сервере **www.foo.com**, в то время как запрос

http://www.bar.com/index.html означает обращение к ресурсу **/index.html** на виртуальном сервере **www.bar.com**. При отправке HTTP-запроса Web-клиент должен определить IP-адрес Web-сервера. На первых порах существования Web каждый виртуальный сервер должен был иметь собственный IP-адрес, чтобы гарантировать, что клиентский HTTP-запрос дойдет до нужного сервера. Например, серверам **www.foo.com** и **www.bar.com** могли быть присвоены адреса 10.63.127.8 и 10.63.127.9, соответственно, несмотря на то, что два виртуальных сервера выполняются на одном компьютере.

Многие операционные системы предусматривают поддержку назначения множества IP-адресов одному сетевому интерфейсу. Трафик, отправленный на один из таких адресов, достигает нужного компьютера; затем операционная система направляет данные соответствующему виртуальному серверу. Такой подход эффективен, но не слишком производителен. Во-первых, операционная система может накладывать ограничения на число серверных процессов или на число IP-адресов для одного компьютера. Во-вторых, назначение отдельного IP-адреса каждому Web-серверу приводит к нерациональному использованию дефицитных в настоящее время IP-адресов. Быстрый рост числа Web-сайтов в 90-х годах прошлого века обострил обе эти проблемы. Это вызвало необходимость внесения изменений в протокол HTTP, разрешающих нескольким виртуальным серверам иметь один и тот же IP-адрес, о чем пойдет речь в главе 7 (раздел 7.8).

4.5.2. Несколько компьютеров для одного Web-сайта

Наиболее популярные Web-сайты получают чрезвычайно большое число клиентских запросов, которые не могут быть обслужены одним компьютером. Обслуживание большого потока запросов обычно требует репликации содержания на нескольких компьютерах. Простейший способ репликации содержания — это выделить отдельное доменное имя для каждой реплики. Предположим к примеру, что имеется Web-сайт, на котором размещено для загрузки пользователями новое программное обеспечение, например, последняя версия Web-браузера. Пользователь может выбирать из списка один из сайтов-*зеркал*. Это помогает распределить нагрузку между сайтами. В списке также может быть указано географическое местоположение различных реплик, давая возможность пользователю получать содержимое с ближайшего сервера. Это снижает нагрузку на сеть и уменьшает задержку при загрузке содержания. При загрузке больших программных пакетов эти преимущества перевешивают неудобства, связанные с необходимостью для пользователя выбирать реплику вручную.

Однако наличие отдельного сервера для каждой реплики имеет несколько недостатков. Ручной выбор сервера утомителен для пользователя, особенно при просмотре Web-содержания. В идеале репликация содержания на различных компьютерах должна быть незаметной для пользователя. Ручной выбор не обязательно равномерно распределяет нагрузку. Отдельный пользователь не осведомлен о решениях, принятых другими пользователями, или о текущей загрузке серверов. Наличие различных доменных имен для каждой реплики порождает ситуацию, когда несколько URL ссылаются на одно и то же содержание. Рассмотрим Web-страницу **bar.html**, которая реплицирована на два сервера: **www1.big.com** и **www2.big.com**. Предположим, что ресурс **http://www1.big.com/bar.html** был сохранен в кэше браузера. В дальнейшем пользователь может выдать запрос на ресурс **http://www2.big.com/bar.html**. Однако браузер не знает, что этот URL ссылается на то же самое содержимое и, следовательно, не сможет вернуть кэшированное содержимое.

Использование одного доменного имени для всех реплик устраняет эти проблемы. Однако при этом требуется эффективный способ доставки клиентского запроса к отдельной реплике. Простейший подход состоит в том, чтобы назначить различные IP-адреса каждой из реплик. Например, **www.big.com** может быть размещен на двух компьютерах с IP-адресами 10.198.3.47 и 10.198.3.48, соответственно. Преобразование доменного имени сервера в определенный IP-адрес выполняется сервером доменных имен (DNS), о чем будет рассказано в главе 5 (раздел 5.3.5). При альтернативном подходе имя **www.foo.com** может быть ассоциировано с одним IP-адресом, который соответствует серверу-заместителю, расположенному перед комплексом Web-хостинга, как показано на рис. 4.1. Этот сервер-заместитель может принимать решения, какая реплика будет обрабатывать клиентский запрос. Выбор реплики может основываться на различных критериях, таких как нагрузка на серверы или запрашиваемые URL. В настоящее время ведутся исследования по разработке оптимальной стратегии выбора реплики, о чем подробнее будет говориться в главе 11 (раздел 11.13).

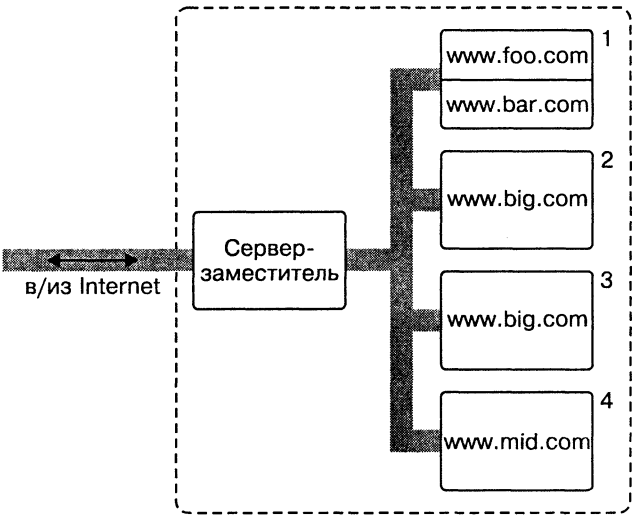


Рис. 4.1. Хостинговая система с сервером-заместителем перед четырьмя Web-серверами

Репликация одного и того же содержания на нескольких серверах сопровождается рядом проблем. Последовательность запросов от одного и того же клиента может не быть обработана одной и той же репликой. Чрезвычайно важно, чтобы каждая реплика генерировала один и тот же ответ на один и тот же запрос. Каждый сервер должен иметь однозначное соответствие между URL и содержанием, а также выдавать идентичный HTTP-ответ. Если содержание изменяется, новые данные должны быть доступны от каждой реплики. Если сервер извлекает ресурсы из собственной файловой системы, любое изменение в файле требует обновления каждой реплики. Кроме того, реплики должны иметь одинаковые стратегии управления доступом. Пользователю не должен быть разрешен доступ к ресурсу для одной реплике и запрещен для другой. Это требует, чтобы на каждым сервере имелась одинаковая информация об имени пользователя и пароле, а также одинаковая стратегия управления доступом. Аналогично, если Web-сайт использует cookies, реплики должны иметь соглашение о том, как их генерировать и интерпретировать.

4.6. Практический пример. Web-сервер Apache

Чтобы проиллюстрировать работу Web-сервера, мы рассмотрим практический пример использования популярного, свободно распространяемого сервера Apache [Apa, LL99]. Выпущенная в 1995 г. первая версия (0.6.2) сервера Apache была основана на созданном ранее Web-сервере NCSA [KBM94]. Название Apache было связано с тем, что разработчики широко используют технологию «программных заплаток» (patches), и является сокращением от "a patchy server". Программное обеспечение было разработано несколькими добровольцами, образовавшими группу Apache Group [MFH00]. Сервер Apache в настоящее время является наиболее популярным Web-сервером [Netc]. Также широко используется коммерческое программное обеспечение Web-серверов, главным образом разработанных Microsoft и Netscape, которое нашло наибольшее применение при создании больших коммерческих Web-сайтов.

В этом разделе мы познакомимся с основными возможностями версии 1.3.3 сервера Apache. Установка сервера Apache состоит в простом копировании соответствующих исполняемых файлов на компьютер. Однако в большинстве случаев Web-администраторы приспосабливают программное обеспечение к требованиям своих сайтов. Эта настройка обычно заключается в задании параметров, которые влияют на выделение ресурсов, интерпретацию HTTP-запросов, генерацию заголовков HTTP-ответов, управление доступом и регистрацию действий сервера. В этом разделе мы постараемся в общих чертах рассказать, как работает Web-сервер, а также проиллюстрировать, какие возможности настройки доступны администратору сайта. Хотя обсуждение будет посвящено серверу Apache, многое из сказанного можно отнести и к другим Web-серверам.

4.6.1. Управление ресурсами

Сервер Apache 1.3.3 назначает отдельный процесс каждому соединению, в противоположность недавно разработанному серверу Apache 2.0, который реализован как многопоточный. Ресурсы размещаются в динамических областях (пулах), которые автоматически освобождаются после завершения процесса.

МОДЕЛЬ ПРОЦЕССА

Сервер Apache использует модель с управлением по процессам с главным процессом, который назначает процесс каждому новому соединению. Вместо того чтобы создавать новый процесс для каждого нового соединения, процесс-родитель создает несколько дочерних процессов при запуске сервера. Количество начальных дочерних процессов (**StartServers**) представляет собой один из параметров настройки, относящихся к дочерним процессам. Эти параметры приведены в таблице 4.2. Сервер поддерживает баланс между слишком большим и слишком маленьким числом процессов. Сервер накладывает ограничение на число одновременно выполняющихся процессов (**MaxClients**), которое задается при компиляции программного обеспечения Web-сервера. Теоретически сервер может всегда работать с данным числом процессов, даже если большинство из них неактивны и ожидают новых соединений. Наличие неактивных процессов позволяет избежать затрат на создание новых процессов при поступлении дополнительных запросов. Однако наличие большого числа неактивных процессов приводит к нерациональному использованию ресурсов операционной системы.

Сервер Apache накладывает ограничения на минимальное и максимальное количество неактивных процессов (**MinSpareServers** и **MaxSpareServers**). Каждые несколько секунд родительский процесс определяет, сколько дочерних процессов являются неактивными. Затем родитель создает или завершает дочерние процессы, в зависимости от того, является ли это число слишком маленьким или слишком большим. Уничтожение неактивных процессов возвращает операционной системе ресурсы, которые могут быть использованы активными процессами, и сокращает нагрузку на систему. Если количество неактивных процессов мало, создание новых процессов подготавливает сервер к обслуживанию будущих клиентских запросов. Порождая дополнительные процессы до поступления запросов, сервер гарантирует, что создание процесса не задержит обработку новых клиентских запросов. Такой подход к созданию и завершению процессов является эффективным способом добиться компромисса между относительно небольшим числом неактивных процессов и созданием новых процессов. Помимо периодического удаления неактивных процессов, сервер Apache накладывает настраиваемое ограничение на количество HTTP-запросов, обрабатываемых каждым дочерним процессом (**MaxRequestsPerChild**). Дочерний процесс завершается при достижении этого ограничения.

Таблица 4.2. Основные параметры настройки для дочерних процессов и сетевых соединений

Параметр	**Описание (значение по умолчанию в Apache 1.3.3)**
StartServers	Начальное число дочерних процессов (5)
MaxClients	Максимальное количество дочерних процессов (256)
MinSpareServers	Минимальное число неактивных дочерних процессов (5)
MaxSpareServers	Максимальное число неактивных дочерних процессов (10)
MaxRequestsPerChild	Максимальное число запросов на дочерний процесс (30)
ListenBacklog	Максимальное число ожидающих соединений (511)
SendBufferSize	Размер буфера передачи TCP (размер по умолчанию для ОС)
MaxKeepAliveRequests	Максимальное число запросов на соединение (100)
KeepAliveTimeout	Максимальное время, в течение которого соединение может оставаться неактивным (15 с)

Сервер Apache имеет определенные значения по умолчанию для каждого из настраиваемых параметров. Однако оптимальные значения параметров зависят от совокупности клиентских запросов. Возьмем сервер, который получает запросы от большого числа клиентов, подключенных в Internet с помощью соединений с низкой пропускной способностью. Это является аргументом в пользу того, чтобы разрешить серверу иметь относительно большое количество активных процессов. Причины здесь две: во-первых, ограниченная пропускная способность клиентов требует от сервера передачи данных с низкой скоростью, т.е. на обслуживание каждого запроса уходит больше времени. Во-вторых, серверу нужно выполнить достаточно большое число последовательных передач, чтобы эффективно использовать свое подключение к Internet. Другой пример — сервер, который исполняет сценарий практически для каждого HTTP-запроса. Выполнение сценария требует значительных ресурсов процессора сервера. Для обеспечения разумного времени отклика на клиентский запрос может потребоваться наложить относительно строгое ог-

раничение на число одновременно выполняющихся процессов. Оптимизация производительности сервера и времени ожидания пользователей требует подбора настраиваемых параметров.

Сервер Apache имеет возможности настройки параметров, значения которых определяются сетевым соединением, обслуживаемым каждым из дочерних процессов. Протокол Transmission Control Protocol (TCP) координирует доставку сообщений с запросами и ответами, о чем подробнее пойдет речь в главе 5 (раздел 5.2). Родительский процесс прослушивает клиентские запросы для установления TCP-соединений. Когда все дочерние процессы обрабатывают соединения, операционная система поддерживает очередь ожидающих соединений. Сервер Apache накладывает ограничение на число соединений в очереди (**ListenBacklog**). Операционная система имеет буфер передачи для хранения исходящих данных для каждого установленного соединения. Сервер Apache может изменять размер этого буфера (**SendBufferSize**). Увеличение размера буфера передачи позволяет повысить производительность, особенно в случае большой задержки при обмене данными между сервером и клиентом. Наконец, сервер ограничивает число HTTP-запросов для одного TCP-соединения (**MaxKeepAliveRequests**) и максимальное время, в течение которого соединение может оставаться неактивным (**KeepAliveTimeout**). При достижении одного из этих ограничений клиентский процесс закрывает TCP-соединение.

ПУЛЫ РЕСУРСОВ

Web-сервер состоит из программных модулей, выполняющих различные задачи, такие как прослушивание новых соединений, синтаксический анализ запросов, создание ответов и передача сообщений-ответов. Подобно любому другому программному обеспечению, эти модули потребляют ресурсы операционной системы. Например, обработка запроса может потребовать от процесса выделения памяти, доступа к файлам, создания дочернего процесса для исполнения сценария, передачи данных через TCP-соединение клиенту. Одна из основных проблем при разработке интенсивно работающего сервера — обеспечить, чтобы эти ресурсы операционной системы использовались эффективно. Проблемы возникают, когда прикладное программное обеспечение не освобождает эти системные ресурсы после завершения задачи. Если объем системных ресурсов со временем убывает, то активные процессы не смогут получить необходимые им ресурсы. Это неприемлемо для Web-сервера, который должен постоянно выполняться в течение долгого периода времени. Система должна возвращать ресурсы после завершения задач.

Сервер Apache решает эту проблему с помощью специальных *пулов*. Пул представляет собой структуру данных, которая отслеживает группу ресурсов операционной системы, создаваемых и уничтожаемых совместно. Допустим, что некорректный запрос привел к ошибке. Процесс, обрабатывающий запрос, может просто уничтожить соответствующий пул, чтобы вернуть ресурсы операционной системе для их использования другими модулями. Концепция пулов позволяет оформить выделение и освобождение ресурсов в виде небольшого, удобного для тестирования фрагмента программного обеспечения сервера. Это защищает сервер от программных ошибок или некорректных ситуаций, возникающих в других фрагментах кода. Концепция пулов основывается на двух основных принципах: процедуры, поддерживающие концепцию пулов, будут работать корректно, а остальное программное обеспечение сервера должно выделять и освобождать ресурсы только с помощью механизма пулов.

Другие части программного обеспечения сервера взаимодействуют с пулами через интерфейс прикладного программирования (API). API содержит базовые

функции для создания, инициализации или уничтожения пулов. Другие функции предоставляют информацию о доступных для пула ресурсах или о размещении ресурсов внутри пула. Например, пул, созданный для обработки клиентского запроса, может выделить память для хранения сообщения HTTP-запроса и для создания сообщения HTTP-ответа. В API также имеются специальные функции для работы со строками. Это особенно полезно при построении заголовков HTTP-ответов. Например, серверному процессу может понадобиться скопировать информацию из клиентского запроса в другой буфер или сцепить две строки. После завершения обработки клиентского запроса процесс может очистить пул, чтобы освободить буфера, закрыть файлы и/или соединения. Завершение процесса также освобождает системные ресурсы, ассоциированные с пулом.

4.6.2. Обработка HTTP-запросов

Сервер Apache выполняет пять основных действий при обработке HTTP-запроса: (1) преобразование запрошенного URL в путь к файлу в файловой системе сервера, (2) определение, имеет ли запрос разрешение на доступ к файлу, (3) идентификация и вызов обработчика для создания ответа, (4) передача ответа клиенту, (5) создание записи о запросе в журнале [Tha96]. Каждая фаза запроса обрабатывается одним или несколькими программными модулями. Сервер Apache включает в себя набор базовых модулей. Для обеспечения альтернативных способов выполнения каждой из фаз обработки запроса сторонними разработчиками могут быть написаны дополнительные модули. Сочетание открытого программного обеспечения и расширяемость модулей стало главным фактором, способствовавшим широкому распространению сервера Apache.

ПРЕОБРАЗОВАНИЕ URL В ПУТЬ В ФАЙЛОВОЙ СИСТЕМЕ СЕРВЕРА

На первом этапе обработки HTTP-запроса сервер преобразует URL в путь в файловой системе, если запрашиваемый файл в ней имеется. Это может представлять собой достаточно сложный процесс, который зависит от настроек сервера. Ядро программного обеспечения сервера выполняет предварительную работу по преобразованию URL. Например, на многих Web-сайтах пользователям разрешено создавать Web-страницы. URL **http://www.bar.com/~peter** может соответствовать пути **/usr/users/peter/public_html/wwwfiles**. Это подразумевает настройку сервера для выполнения функции преобразования для всех URL, содержащих символ "~". Серверу также может потребоваться выполнить предварительную обработку URL для удаления определенных строк. Например, **http://www.bar.com//a.html** должно быть преобразовано в **http://www.bar.com/a.html** путем удаления повторяющегося символа "/". Аналогично, **http://www.bar.com/b/../a.html** должно быть преобразовано в **http://www.bar.com/a.html** при обнаружении подстроки "..".

Apache включает большое число настраиваемых программных модулей для преобразования URL. Каждый сервер имеет настраиваемый основной каталог, в котором размещаются файлы. Рассмотрим Web-сервер **www.bar.com** с корневым каталогом **/www**. URL **http://www.bar.com/foo.html** будет преобразован в путь **/www/foo.html**. Для хранения файлов в других каталогах сервером Apache используются псевдонимы. Псевдонимы предоставляют способ отделить структуру каталогов сервера от URL, видимых извне. В противном случае реорганизация файлов на сервере приводила бы к некорректности существующих URL. Кроме того, псевдонимы могут быть использованы для того, чтобы дать возможность нескольким URL ссылаться на один и тот же ресурс. Например, URL **http://www.bar.com/~bala/kandr.ps** и

http://www.bar.com/~jrex/kandr.ps могут оба соответствовать файлу **/www/book17.ps**. Установка соответствия между URL и именем файла может также зависеть от времени получения запроса сервером. Например, сервер может быть настроен так, чтобы URL **http://www.bar.com/a.html** соответствовал **http://www.bar.com/morning.html** утром и **http://www.bar.com/afternoon.html** днем.

Программное обеспечение Apache включает дополнительный модуль для коррекции типичных ошибок в URL. Модуль может просматривать запрашиваемый каталог с целью выявить, нет ли в ней файла с именем, близким к запрашиваемому (это позволяет найти ошибки, связанные с лишними или отсутствующими символами, использованием одного символа вместо другого, либо использованием неправильного регистра символов). Однако эта функция обусловливает дополнительную нагрузку, связанную со сканированием списка файлов в каталоге и идентификацией возможных совпадений. Кроме того, передача ответов на основе частичного совпадения может непроизвольно «засветить» файл, который не предназначен для клиентских запросов. Сервер также имеет модуль, который переписывает запрашиваемые URL на основе настраиваемого набора правил. Модуль перезаписи использует при синтаксическом анализе регулярные выражения для сопоставления с фрагментами строки URL. Действия могут различаться в зависимости от ряда параметров, включая заголовки HTTP-запросов. Например, сообщение HTTP-запроса может включать заголовок **User-Agent**, который предоставляет информацию о браузере пользователя. Преобразование URL, основанное на этой информации, дает возможность серверу возвращать различные ресурсы пользователю в зависимости от используемого им браузера.

АВТОРИЗАЦИЯ ЗАПРОСА

Стратегии управления доступом зависят от настроек сервера. Например, доступ к ресурсу **http://www.bar.com/books/a.html** может быть ограничен определенным кругом клиентов. Запросы, поступающие от определенных доменных имен или IP-адресов, могут быть отвергнуты, либо пользователям может быть предложено указать имя и пароль. Чтобы избежать задания стратегии управления доступом для каждого ресурса, можно применить одну стратегию для группы ресурсов. Конфигурационные файлы состоят из директив, которые управляют доступом к Web-ресурсу в зависимости от URL, имени файла, каталога или виртуального сервера. Указание

```
<Directory /www/books>
AuthType Basic
AuthName specialuser
AutUserFile /www/let_them_in/users
require valid-user
</Directory>
```

ограничивает доступ к файлам в каталоге **/www/books** теми пользователями, которые укажут корректные имя и пароль для области **specialuser**. Файл **/www/let_them_in/users** содержит список имен пользователей и их паролей (в зашифрованном виде). Указание

```
<Directory /www/cgi-bin>
order deny,allow
deny from all
allow from 10.9.57.188
</Directory>
```

разрешает доступ к ресурсам в каталоге **/www/cgi-bin** только Web-клиенту с IP-адресом 10.9.57.188.

Администратор Web-сайта может задавать эти стратегии в основном конфигурационном файле по одной директиве в строке. Однако использование основного конфигурационного файла имеет два основных недостатка. Во-первых, внесение любого изменения в файл требует перезапуска сервера. Во-вторых, реализация стратегий управления доступом в интересах большого числа Web-страниц различных авторов может стать излишне обременительным делом для администратора сайта. Чтобы справиться с этими проблемами, администратор сайта может разрешить отдельным пользователям замещать настройки по умолчанию в основном файле. В каждом каталоге может иметься файл **.htaccess**, в котором можно задать стратегии управления доступом для этого каталога и всех его подкаталогов. Это дает возможность пользователю с именем Viv, хранящему файлы в каталоге **/www/users/viv**, создать файл **/www/users/viv/.htaccess**, который замещает стратегию доступа по умолчанию, заданную в каталоге **/www/users**. В основном конфигурационном файле можно задать, какие директивы могут быть указаны в других файлах **.htaccess**. Это позволяет администратору сайта ограничивать стратегии управления доступом, применяемые отдельными пользователями.

Выяснив, что URL соответствует определенному файлу, сервер начинает процесс *обхода каталогов*, чтобы определить, какие стратегии управления доступом применить к запрашиваемому файлу. Сервер проверяет основной конфигурационный файл на наличие директив, применяемых к этому файлу. Затем сервер просматривает файлы **.htaccess** во вложенных подкаталогах. Для каталога **/www/users/viv** сервер проверяет основной конфигурационный файл, затем **/www/users/.htaccess** и **/www/users/viv/.htaccess**. На каждом этапе сервер может добавлять дополнительные директивы настройки, которые применяются к запрашиваемому файлу. Поскольку файлы **.htaccess** просматриваются для каждого запроса, настройки могут быть изменены без перезапуска сервера. Однако поиск файла **.htaccess** в каждом подкаталоге приводит к повышению нагрузки на сервер при обработке запроса. Администратор сайта может уменьшить эти затраты, указав в основном конфигурационном файле подкаталоги, для которых разрешается замещать директивы.

СОЗДАНИЕ И ПЕРЕДАЧА ОТВЕТА

При синтаксическом анализе HTTP-запроса сервер создает и заполняет структуру данных *записи запроса*, которая используется различными программными модулями. Запись запроса содержит информацию из самого запроса, включая URL, версию протокола HTTP, тип содержимого и формат кодирования данных, отправляемых клиентом в теле запроса (если таковые имеются). Запись имеет указатель на пул, который был выделен для обработки этого запроса; пул очищается, когда сервер заканчивает обработку запроса. Запись включает информацию, полученную на предыдущих этапах обработки запроса: путь к файлу, ассоциированному с URL, список директив настройки, относящихся к ресурсу. Другие составляющие записи запроса, такие как заголовки HTTP-ответа, указываются, когда сервер генерирует сообщение-ответ.

В Apache имеется набор *обработчиков (handlers)*, которые выполняют операции с запрашиваемым файлом. Эти обработчики представлены в таблице 4.3. Обработчик обычно назначается на основе расширения файла или его местоположения, что определяется настройками сервера. Программное обеспечение Apache содержит несколько встроенных обработчиков. Обработчик по умолчанию передает содержи-

мое файла клиенту, добавляя необходимые HTTP-заголовки. Другой обработчик может посылать файл «как есть», включая HTTP-заголовки, имеющиеся в файле. Это полезно, например, для возврата предварительно созданного HTTP-сообщения, которое перенаправляет клиента в случае, если запрашиваемый ресурс был перемещен в другое место. Apache определяет обработчики для динамического содержания, которые трактуют файл как CGI-сценарий, включение на стороне сервера или *карту изображений*, обрабатываемую на стороне сервера. Например, рассмотрим запрос

```
http://www.germancities.de/imagemap/country?163,83
```

где числа 163 и 83 отражают позицию курсора мыши, при щелчке мышью на карте изображений. Файл **/www/imagemap/country** связывает области изображения с URL. Чтобы обработать запрос, сервер идентифицирует URL, ассоциированный с координатами, и отправляет ответ, который указывает браузеру инициировать другой HTTP-запрос с новым URL. Файл также может быть интерпретирован как *карта типов (type map)*, которая содержит список имен файлов с различными вариантами ресурса (например, на английском или испанском языках). В зависимости от информации в заголовке запроса сервер определяет, какой файл и какие HTTP-заголовки возвращать клиенту. Другие обработчики имеют отношение к управлению сервером, трактуя файл как хранилище настроечной информации.

Таблица 4.3. Встроенные обработчики и файловые расширения по умолчанию для сервера Apache

Обработчик	Назначение (расширение файла)
default-handler	Передает файл как статическое содержимое
send-as-is	Передает файл как сообщение HTTP-ответа (.asis)
cgi-script	Активизирует файл как CGI-сценарий (.cgi)
server-parsed	Трактует файл как включение на стороне сервера (.shtml)
imap-file	Трактует файл как файл карту изображений, обрабатываемую на стороне сервера (.imap)
type-map	Трактует файл как карту типов для альтернативной передачи содержания (.var)
server-info	Получает настроечную информацию о сервере
server-status	Получает отчет о состоянии сервера

Настройки сервера также задают, какие метаданные присутствуют в заголовке HTTP-ответа. Web-сервер ассоциирует определенные файловые расширения с определенными атрибутами ресурса. Сервер может иметь файл конфигурации, который определяет тип MIME (Multipurpose Internet Mail Extensions), ассоциированный с каждым файловым расширением:

```
application/octet-stream        bin exe pax tgz jar
application/pdf                 pdf
application/postscript          ai eps ps
audio/x-pn-realaudio            ram rm
audio/x-realaudio               ra
image/gif                       gif
image/jpeg                      jpeg jpg jpe
```

```
text/html                       html htm
text/plain                      txt
video/mpeg                      mpeg mpg mpe
```

Например, URL, который соответствует файлу **/www/index.html**, будет инициировать обработчик по умолчанию default-handler, который передает содержимое файла клиенту. Файловое расширение **.html** заставит сервер включить в сообщение-ответ заголовок **Content-Type: text/html**. При получении ответа браузер использует заголовок **Content-Type** для выбора соответствующего приложения для отображения ответа, о чем говорилось ранее в главе 2 (раздел 2.4.3).

Интерпретация сообщения-ответа браузером также зависит от других метаданных, таких как формат кодирования и язык. Сервер назначает эти атрибуты на основе команд настройки, таких как

```
AddEncoding x-compress          Z
AddEncoding x-gzip              gz
AddLanguage en                 .en
AddLanguage fr                 .fr
AddLanguage de                 .de
AddLanguage it                 .it
```

Файл может иметь несколько расширений, каждое из которых соответствует различным метаданным. Например, файл **/www/index.en.html.Z** будет соответствовать сжатому HTML-файлу на английском языке. Настройки также определяют, содержит ли заголовок сообщения информацию, связанную с кэшированием. Предположим, что администратор сайта знает, что определенный набор ресурсов изменяется каждый день в полдень. В этом случае сервер может быть настроен так, чтобы включать соответствующее время истечения срока актуальности в заголовок HTTP-ответа. Обратившийся с запросом клиент может воспользоваться этой информацией при принятии решения, как долго сохранять ответ в кэше.

4.7. Резюме

В процессе создания и передачи ответов на клиентские запросы Web-сервер выполняет множество важных задач. Сервер связывает URL в сообщении-запросе с определенным файлом в файловой системе и определяет, как генерировать сообщение-ответ. Кроме того, сервер принимает решение, следует ли разрешить доступ к ресурсам для запроса. Создание ответа на запрос может состоять в извлечении запрашиваемого файла с диска или в активизации сценария, генерирующего ответ. Сценарий может создавать ответ на основе пользовательских cookies и информации, хранимой во внутренней базе данных, без участия Web-сервера. Сервер не обязательно сохраняет какую-либо информацию между последовательными запросами, хотя сохранение информации может уменьшить затраты при обработке будущих запросов. Активно работающий сервер обычно одновременно обрабатывает запросы от большого числа клиентов. При настройке сервера на обслуживание потока клиентских запросов требуется учитывать архитектуру сервера. Ограничения, свойственные архитектурам с управлением по событиям и с управлением по процессам, привели к созданию гибридных схем. Обслуживание большого потока обращений к популярному Web-сайту требует репликации содержания на несколько компьютеров. И наоборот, несколько менее популярных сайтов могут быть размещены на одном компьютере.

Часть III
Web-протоколы

5

Протоколы, связанные с HTTP

Передача информации в Web зависит от набора коммуникационных протоколов. Протокол определяет как синтаксис, так и семантику сообщений, которыми обмениваются отправитель и получатель. Например, протокол передачи гипертекста Hypertext Transfer Protocol (HTTP) определяет формат и назначение запросов, отправляемых Web-клиентами, и ответов, возвращаемых Web-серверами. Сетевые протоколы обычно распределяются по *уровням*, каждый из которых обслуживает определенные составляющие коммуникационного взаимодействия. Набор протоколов Internet состоит из четырех основных уровней, представленных на рис. 5.1:

- **Канальный уровень.** Канальный уровень определяет взаимодействие с физической средой передачи данных, например, Ethernet, Asynchronous Transfer Mode (ATM) или Synchronous Optical Network (SONET).
- **Сетевой уровень.** Сетевой уровень управляет доставкой отдельных *пакетов* данных через сеть. Протоколы сетевого уровня реализуется в маршрутизаторах и оконечных компьютерах.
- **Транспортный уровень.** Транспортный уровень координирует взаимодействие между хостами в интересах прикладного уровня. На практике протоколы транспортного уровня обычно реализуются операционной системой хоста.

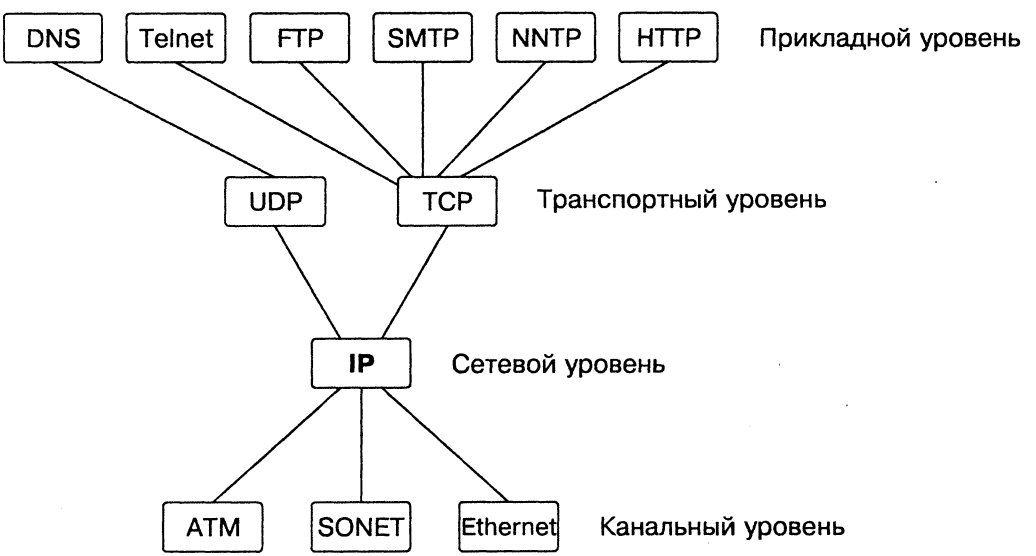


Рис. 5.1. Уровни протоколов

- **Прикладной уровень.** Прикладной уровень определяет специфику конкретных приложений. На практике протокол прикладного уровня обычно реализуется как часть прикладного программного обеспечения, такого как Web-браузер или Web-сервер.

Разделение функций дает возможность каждому протоколу сосредоточиться на выполнении одной задачи с четко определенными интерфейсами с протоколами смежных уровней. Стандартизация протоколов обеспечивает совместимость между компонентами, разработанными различными производителями.

В этой главе представлен обзор трех основных протоколов, вовлеченных в передачу HTTP-сообщений, начиная с сетевого уровня:

- **Internet Protocol (IP).** IP — это протокол сетевого уровня, который координирует доставку отдельных пакетов от одного хоста другому по IP-адресу компьютера-адресата. IP работает поверх различных протоколов канального уровня.
- **Transmission Control Protocol (TCP).** TCP — это протокол транспортного уровня, который координирует передачу IP-пакетов для обеспечения надежного двунаправленного соединения между двумя приложениями. TCP является основным транспортным протоколом в Internet, хотя некоторые приложения используют протокол User Datagram Protocol (UDP).
- **Domain Name System (DNS).** Система именования доменов (DNS) — это протокол прикладного уровня, который управляет преобразованием доменных имен, например, **www.foo.com**, в IP-адреса и наоборот. DNS предоставляет этот общий сервис другим приложениям.

HTTP использует DNS для преобразования доменного имени сервера в IP-адрес, TCP — для отправки HTTP-запроса серверу и HTTP-ответа клиенту, а IP — для доставки отдельных пакетов. После подробного рассмотрения IP, TCP и DNS мы представим краткий обзор четырех протоколов прикладного уровня: Telnet, File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP) и Network News Transfer Protocol (NNTP). На этих протоколах основаны приложения, которые использовались до появления Web и оказали влияние на разработку HTTP. Кроме того, эти протоколы стали составной частью Web, поскольку они поддерживаются Web-браузерами.

5.1. Internet Protocol

Протокол Internet Protocol (IP) представляет собой протокол сетевого уровня, являющийся основой Internet. В этом разделе будет описано, как Internet эволюционировала от экспериментальной сети, объединявшей несколько научно-исследовательских организаций в 60-х годах. Затем будут представлены основные цели разработки, которые нашли воплощение в спецификации IP, и рассказано об использовании IP-адресов для идентификации хостов. Для полноты картины будет также представлен обзор информации, содержащейся в заголовке IP-пакета.

5.1.1. Эволюция архитектуры Internet

В конце 60-х годов в США имелось небольшое число суперкомпьютеров, используемых для научных исследований в различных национальных лабораториях. Эти машины обычно функционировали в пакетном режиме, когда пользователи

сначала вводили в компьютер задания, а позднее забирали результаты. Изначально доступ к этим суперкомпьютерам был возможен только для людей, физически находившихся рядом с ними. Высокая стоимость таких суперкомпьютеров заставила Агентство по перспективным исследовательским проектам (ARPA) Министерства обороны США искать способы совместного использования вычислительных ресурсов национальными лабораториями и университетскими исследовательскими группами. Это привело к созданию сети ARPANET, позволившей исследовательским организациям взаимодействовать друг с другом. Кроме предоставления удаленного доступа к суперкомпьютерам, сеть ARPANET облегчила сотрудничество между вычислительными центрами, дав возможность пользователям совместно использовать файлы и обмениваться сообщениями электронной почты.

Рост ARPANET продолжился в 70-х годах. Все больше университетов и научно-исследовательских организаций по всему миру стали подключаться к сети. Помимо этого, были разработаны коммуникационные протоколы для множества приложений, включая FTP. В 80-х годах официальное принятие открытых, стандартных протоколов для передачи данных через ARPANET создало почву для быстрого роста количества компьютеров, подключенных к сети. Стандарты протоколов IP и TCP имели важное значение для обеспечения взаимодействия между различными типами компьютеров с помощью различных сетевых технологий. Другим важным прорывом стала реализация IP в версии Berkeley Software Distribution (BSD) операционной системы UNIX. BSD UNIX свободно распространялась в высших учебных заведениях, что в свою очередь привело к подключению к ARPANET большого числа пользователей. В ряде случаев большое число компьютеров в локальной сети организации могло иметь доступ к ARPANET. Расширившись, ARPANET стала использоваться для взаимодействия *сетей*, а не компьютеров.

Начиная с 1988 г., ARPANET стала постепенно вытесняться сетью NSFNET, опорной сетью Internet, созданной Национальным научным фондом США (NSF). NSFNET, как и ARPANET, первоначально использовали в основном сотрудники университетов и научно-исследовательских организаций. С начала 90-х годов NSFNET стала использоваться и для передачи коммерческого трафика. В середине 90-х годов Internet стала большой коммерческой сетью, благодаря появлению на свет World Wide Web, или Всемирной паутины, ставшей доминирующим приложением в сети. В США сеть, финансируемая правительством, была преобразована в множество опорных сетей, работу которых обеспечивали крупные сетевые провайдеры. Тем не менее, правительство сыграло важную роль в финансировании создания нескольких пунктов обмена трафиком. К концу 90-х годов сеть Internet состояла из опорных сетей, взаимодействующих между собой через общедоступные пункты обмена трафиком и прямые соединения. Увеличение объема передаваемых данных привело к необходимости увеличению производительности и области охвата, т.е. присутствию в Internet практически каждой стране мира. Подробнее об истории Internet вы можете узнать из других источников [Sal95, LCC'97, Abb99, ISO].

В начале 21-го века Internet состоит из сетей, каждая из которых поддерживается какой-либо организацией, будь то университет, компания, или правительственный орган. Каждая организация имеет свое собственное сетевое оборудование для обмена трафиком между пользователями и остальной частью Internet. Сеть состоит из маршрутизаторов, которые направляют трафик по множеству каналов. Канал может использовать разнообразные средства, начиная от телефонных линий и заканчивая высокоскоростными оптоволоконными линиями. Маршрутизатор представляет собой компьютер специального назначения, который предназначен для перенаправления трафика от входящих к исходящим каналам. Маршрутизаторы

работают под управлением ряда протоколов, стандартизованных Рабочей группой по инженерным проблемам Internet (IETF). Тот факт, что IP не зависит от технологий канального уровня, имеет важное значение, так как дает возможность передавать Internet-трафик по различным каналам, а также между маршрутизаторами, созданными различными производителями оборудования.

После получения трафика из входящего канала маршрутизатор должен выбрать исходящий канал для передачи в направлении его окончательного назначения. Выбор маршрутов передачи IP-трафика определяется рядом протоколов маршрутизации. Внутри одной сети маршрутизаторы взаимодействуют с использованием внутридоменных протоколов, в то время как междоменные протоколы используются для взаимодействия между сетями. Различия внутридоменной и междоменной маршрутизацией сходно с доставкой почты. Сначала письмо направляется из региона, в котором оно было отправлено, в регион, где оно должно быть получено. Затем в регионе, где живет получатель, осуществляются действия по доставке письма непосредственно получателю. Письмо может путешествовать, проходя на своем пути через ряд почтовых отделений в различных странах и городах. Тем не менее, отправителю нужно всего лишь знать *адрес* получателя, а не весь путь, который должно пройти письмо. Фактически любому почтовому отделению также не нужно знать весь путь — достаточно знать лишь о следующем этапе пути. Аналогичным образом маршрутизатор просто пересылает Internet-трафик следующему маршрутизатору на пути следования. Аналогия с обычной почтой весьма полезна для понимания принципов функционирования IP-сетей.

5.1.2. Цели разработки IP

IP унаследовал большинство идей от протоколов, использовавшихся в ARPANET [Cla88]. Основная идея IP — сохранить относительную простоту сети, возложив основные функции на хосты. Это сильно контрастирует с телефонной сетью, где используются очень сложные способы коммутации, но простые оконечные устройства (т.е. телефоны). IP предоставляет интегрированную среду для отправки отдельных *пакетов*. Пакет представляет собой единицу информации — определенное число байтов данных, указанное отправителем. При «путешествии» от отправителя к получателю пакет проходит через ряд маршрутизаторов, которые взаимодействуют по протоколу IP, как показано на рис. 5.2. Вернувшись к аналогии с обычной почтой, можно отметить, что передача IP-пакетов через Internet схожа с отправкой письма по почте. Почта прилагает максимальные усилия, чтобы быстро доставить письмо. Однако письмо может не достичь адресата или находиться в пути слишком долго. Серия писем, отправленных от одного человека к другому, может придти не в том порядке, в котором письма были отправлены.

Аналогичным образом, маршрутизаторы в Internet обрабатывают каждый пакет независимо и не нуждаются в сохранении информации о состоянии между последовательными пакетами. Последовательность IP-пакетов, передаваемых от одного хоста другому, не обязательно проходит в сети одним и тем же путем. Пакеты могут теряться, повреждаться или доставляться не в том порядке. В этом смысле обычная почта и Internet заметно отличаются от традиционной телефонной сети. В телефонной сети устанавливается строго определенное соединение между любыми двумя абонентами, для каждого соединения перед передачей данных выделяется необходимая полоса пропускания. Например, сеть может выделить полосу пропускания в 64 Кбит/с для каждого телефонного соединения. Телефонное соединение не устанавливается, если не будет иметься достаточно ресурсов. Это

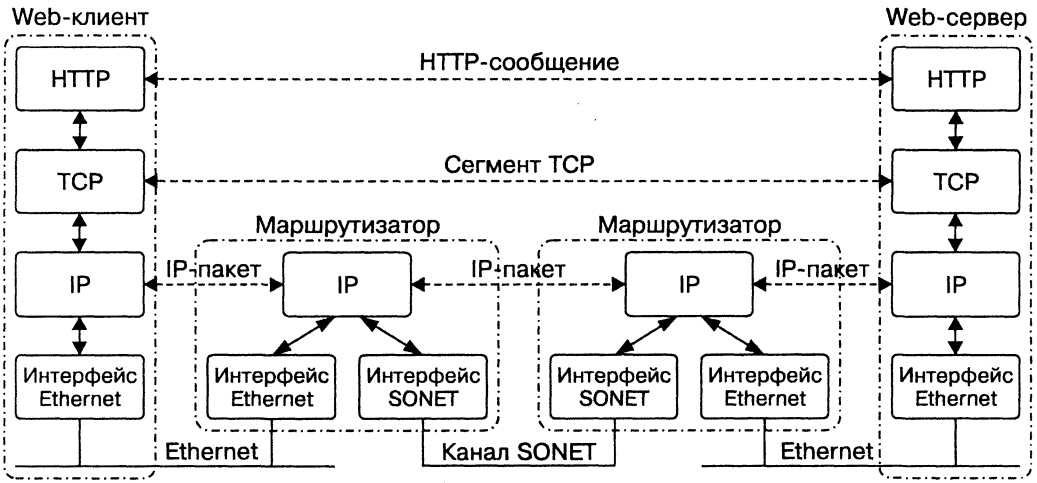


Рис. 5.2. Протоколы, используемые при передаче HTTP-сообщений

гарантирует, что текущие телефонные соединения не будут подвергаться какому-либо воздействию при установлении нового соединения. В противоположность этому обычная почта явным образом не ограничивает число писем, которое может быть отправлено; в периоды наибольшей активности абонентов, например, перед праздниками, производительность системы может снижаться. И Internet, и обычная почта используют *коммутацию пакетов*, в противоположность *коммутации каналов*, используемой в традиционных телефонных сетях.

Различия обусловлены тем фактом, что сеть Internet была разработана совсем для другой цели — поддержки удаленного доступа к совместно используемым ресурсам. Пользователь может провести несколько часов, работая с удаленным компьютером, внутри сеанса могут быть продолжительные периоды бездействия. Наличие выделенного канала для такого трафика будет расточительством. Вместо этого Internet-трафик делится на IP-пакеты, которые передаются по мере необходимости. Поскольку трафик может формироваться в любой момент времени, каждый пакет должен иметь *заголовок*, идентифицирующий место назначения. Заголовок IP-пакета аналогичен конверту с письмом, который передается по почте. Информация, содержащаяся в заголовке, не нужна в сети с коммутацией каналов. Несмотря на затраты, связанные с включением заголовка в каждый отправляемый пакет, пакетная коммутация при передаче данных обычно более эффективна, чем коммутация каналов, поскольку сеть с пакетной коммутацией может чередовать трафик различных пар отправитель-получатель на уровне пакетов. Пакетная коммутация позволяет активным отправителям и получателям использовать доступные сетевые ресурсы.

Если сеть загружена слабо, входящий пакет сразу же передается в исходящий канал любым маршрутизатором на его пути. Однако отсутствие выделенного канала приводит к тому, что сеть не всегда имеет достаточно ресурсов, чтобы передать каждый из пакетов к его месту назначения. Многочисленные пары отправитель–получатель могут быть активными одновременно, что ведет к значительной загрузке определенных каналов в сети. Когда канал перегружен, маршрутизатор временно хранит ожидающие пакеты в очереди. При большой перегрузке маршрутизатор может отвергнуть один или несколько пакетов, чтобы избежать переполне-

ния очереди. Однако большинство Internet-приложений, таких как Web и передача файлов, не могут допускать потери данных. Более строгие требования этих приложений, казалось бы, противоречат основной парадигме ненадежной доставки пакетов. Философия IP не предполагает, что отсутствие гарантии доставки устроит разработчиков приложений. Средства для решения таких проблем реализуются протоколами транспортного уровня, работающими на компьютерах отправителя и получателя, а не маршрутизаторами в сети.

Ограничение функциональных возможностей сетевых маршрутизаторов являлось важной целью разработки при создании ARPANET. Договорившись об относительной простоте сети, разработчики ARPANET смогли сосредоточить больше внимания на разработке приложений. Кроме того, разработчиков беспокоило, как более сложная сеть будет реагировать на сбои. Аппаратные и программные компоненты IP-сетей не отличались надежностью, особенно в сравнении со зрелыми технологиями телефонных сетей. Помимо этого разработчики ARPANET хотели, чтобы сеть продолжала функционировать даже в случае повреждений или злонамеренных атак, приводящих к выходу из строя отдельных компонентов. Позднее, по мере развития ARPANET, а затем и Internet, стало чрезвычайно важно обеспечить подключение новых маршрутизаторов и каналов без нарушения работы сети. IP-маршрутизаторы автоматически приспосабливаются к изменению топологии сети. Когда маршрутизатор или канал связи выходят из строя, оставшиеся маршрутизаторы рассчитывают новые маршруты, и передача трафика продолжается. Отказавший маршрутизатор не хранит какую-либо важную информацию, которая необходима для поддержания связи между отправителем и получателем. Связь может сохраняться до тех пор, пока не произойдет отказ компьютера-отправителя или компьютера-получателя, а также пока в сети имеется крайней мере один путь между ними. В самом худшем случае отказ может привести к потере пакетов.

Простота IP способствует тому, что можно продолжать доставку трафика при наличии временных отказов в сети. Конечные хосты реализуют протоколы транспортного уровня поверх IP, координируя доставку данных между приложениями. Двумя основными транспортными протоколами являются Transmission Control Protocol (TCP) и User Datagram Protocol (UDP). Оба протокола стандартизованы и реализованы практически во всех операционных системах. TCP предоставляет основную абстракцию, необходимую большинству Internet-приложений, — *логическое соединение*, которое осуществляет гарантированную доставку последовательности байтов от отправителя к получателю в упорядоченном виде. Стандартизованный в 1980 г., TCP обеспечивает основу для протоколов Telnet, FTP, SMTP, NNTP и HTTP. TCP является протоколом транспортного уровня, с помощью которого передается основная часть трафика в Internet. Способность TCP адаптироваться к перегрузке сети имеет важное значение в контексте продолжающегося роста Internet. Подробнее о TCP мы поговорим в разделе 5.2.

UDP обеспечивает простую абстракцию ненадежной доставки дейтаграмм. Отправляющее приложение инструктирует операционную систему послать набор байтов удаленному приложению. UDP-дейтаграмма посылается в IP-пакете, направляемом принимающей машине. IP-пакет может быть потерян или задержан в сети. Повторная передача потерянной дейтаграммы, если это требуется, должна быть выполнена отправляющим приложением. UDP хорошо подходит для приложений, которые терпимы к потере пакетов. Например, многие мультимедийные приложения передают аудио и видео как поток UDP-пакетов. Потерянный пакет может ухудшить качество аудио- или видеоинформации, воспринимаемой получателем. Однако приложение будет продолжать функционировать. Предоставление

отправителю возможности решать, передавать ли данные повторно, дает приложению дополнительную гибкость. UDP также используется другими приложениями, которые передают короткие запросы и ответы. Далее в этой главе, в разделе 5.3, будет описано применение UDP для запросов к DNS. Приложение может повторить свой запрос, если ответ не поступил в течение определенного промежутка времени.

IP является центральным в группе протоколов различных уровней, как это показано на рис. 5.1. Протоколы прикладного уровня, такие как FTP и HTTP, основываются на протоколе транспортного уровня при доставке сообщений между связывающимися друг с другом хостами. Протоколы прикладного уровня и протоколы транспортного уровня используются для организации связи между хостами. Протокол сетевого уровня обеспечивает доставку отдельных пакетов. Все Internet-приложения основаны на одном, всеобъемлющем протоколе сетевого уровня — Internet Protocol. Протокол сетевого уровня обеспечивает коммуникационное взаимодействие между отдельными сетевыми компонентами, такими как хосты и маршрутизаторы. IP передает пакеты, используя различные технологии канального уровня, включая Ethernet или SONET. Способность IP согласованно работать с различными протоколами канального и прикладного уровней имеет важное значение для быстрого развития Internet и Web.

Несмотря на преимущества, которые дает наличие простого протокола сетевого уровня, решение реализовать транспортные протоколы на оконечных хостах имеет существенное влияние на производительность. Применение протоколов TCP и UDP совместно с IP подразумевает, что эти протоколы транспортного уровня должны быть терпимы к задержкам и потерям в IP-сети. На первых порах существования ARPANET большинство приложений было не слишком чувствительно к задержкам. Незначительное увеличение задержки при выполнении пакетного задания на суперкомпьютере или при доставке сообщений электронной почты было не особенно ощутимо для пользователей. Хотя задержка оказывает заметное влияние на интерактивные приложения, такие как Telnet, небольшое число пользователей в сети ARPANET не ожидали от сети высокой производительности и надежности. Пользователи стали не столь терпимы к низкой производительности по мере развития Internet. Подобно Telnet, Web является интерактивным приложением. Задержки в доставке Web-содержания раздражают пользователей Web. Кроме того, новые Internet-приложения, такие как Internet-телефония и мультимедийное потоковое вещание, еще более требовательны к производительности сети.

В ответ на эти требования эволюция Internet пошла в сторону совершенствования поддержки приложений, которые требуют предсказуемой производительности при коммуникационных взаимодействиях. Новые IP-маршрутизаторы способны дифференцировать трафик от различных пользователей или приложений, чтобы предоставить лучший сервис для части трафика. Например, маршрутизаторы могут оказывать предпочтение пакетам, передающим аудиоинформацию для приложений IP-телефонии, перед пакетами, передающими сообщение электронной почты. Помимо этого, провайдеры Internet могут предоставлять более высокую производительность для клиентов, которые платят больше, за счет клиентов, которые платят меньше. Со временем Internet в большей степени приблизится к традиционной телефонной сети, которая обеспечивает предсказуемый и надежный сервис. Однако имеется значительное противодействие превращению Internet в сеть с коммутацией каналов, поскольку при этом придется пожертвовать простотой и устойчивостью децентрализованной, некоммутируемой сети. Большинство предлагаемых изменений в Internet сохраняет основную идею доставки пакетов с помощью IP.

5.1.3. IP-адреса

Хосты в Internet идентифицируются числовыми адресами. Вернувшись к аналогии с почтой, можно заметить, что IP-пакет схож с отправленным письмом. Письмо помещается в конверт, на котором написан адрес получателя. Почтовая система пересылает письмо, используя информацию об адресате; содержимое письма не влияет на процесс доставки. Два человека могут обмениваться письмами, общаясь по переписке. Письма могут отправляться через достаточно большие промежутки времени. Почте совершенно нет дела до взаимосвязей между этими письмами. Каждое письмо направляется независимо. Отправитель и получатель ответственны за определение порядка следования писем, относящихся к одной теме, а также за принятие решения, следует ли отправлять ответ, и когда это сделать. Заголовок IP-пакета содержит всю информацию, необходимую маршрутизатору для того, чтобы доставить содержимое в соответствующее место назначения.

Адрес места назначения представляет собой число длиной 32 бита, который относится к конкретному компьютеру[1], например, Web-серверу. В IP-заголовке адрес места назначения представляется 32-битным двоичным числом. Двоичное представление легко интерпретируется маршрутизаторами. 32-битный адрес соответствует 2^{32} местам назначения. Адреса в Internet имеют иерархическую структуру. Снова обратимся к аналогии с обычной почтой. У жителей одного района имеется общий почтовый индекс. Письмо может быть направлено в почтовое отделение, соответствующее данному индексу. Затем это почтовое отделение может использовать оставшуюся часть адреса для пересылки письма в получателю. Аналогично, IP-адрес делится на *сетевую* часть и часть, соответствующую *хосту*. Прокладка маршрута в Internet основывается на сетевой части адреса. Большинству маршрутизаторов в Internet не нужно знать о том, как достичь определенных хостов. После того как пакет достигает назначенной сети, для передачи пакета соответствующему компьютеру-получателю используется часть адреса, определяющая хост.

IP-адреса выделяются организациям в виде смежных блоков различной длины. Изначально IP-адреса относились к трем классам в зависимости от числа 8-битных *октетов*, выделенных для адреса сети и адреса хоста; два других класса адресов резервировались для трафика группового вещания и будущего использования. На рис. 5.3 показано деление 32-битного IP-адреса на адрес сетевую часть и адрес хоста:

- **Класс A.** Адреса класса A начинаются с 0 в первом бите и используют первый октет для адреса сети, оставляя три октета (24 бита) для адреса хоста. Следовательно, первый октет адресов класса A имеет значение в диапазоне от 0 до 127 (что соответствует двоичным числам 00000000 и 01111111, соответственно). Сеть класса A включает в себя 16 777 216 (2^{24}) IP-адресов.
- **Класс B.** Организациям, которым не требуется такое большое число хостов, может быть выделена сеть класса B. Адрес класса B начинается с 10 в первых двух битах и использует первые два октета для сетевого адреса, а последние два октета — для адреса хоста. Сеть класса B включает 65 536 (2^{16}) адресов.
- **Класс C.** Адреса класса C могут быть выделены небольшим организациям. Адрес начинается с 110 в первых трех битах и использует первые три октета для адреса сети и только последний октет для адреса хоста. В Internet имеется большое число сетей класса C, каждая из которых имеет по 256 адресов.

---

[1] Точнее сетевому интерфейсу. — *Прим. ред.*

- **Класс D.** IP-адреса, относящиеся к классу D (начинаются с 1110), используются для трафика группового вещания, направляемого множеству компьютеров.
- **Класс E.** Адреса класса E (начинаются с 11110) зарезервированы для будущего использования.

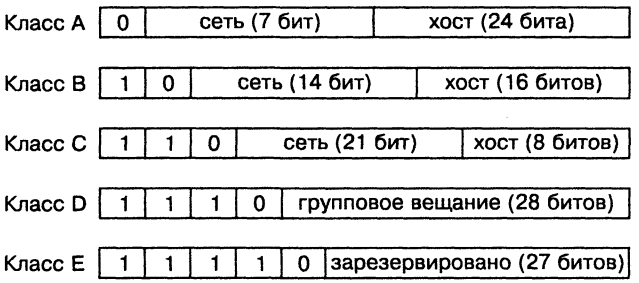


Рис. 5.3. Пять классов IP-адресов

Деление на октеты обусловливает представление IP-адресов в *точечной нотации*, при которой каждый октет записывается в виде десятичного числа в диапазоне от 0 до 255 ($2^8 - 1$) и отделяется от других точкой. Например, запись 122.5.208.78 будет корректна для IP-адреса, а запись 373.57.7.300 — некорректна. Сеть класса A 12.0.0.0 состоит из IP-адресов в диапазоне от 12.0.0.0 до 12.255.255.255.

Ограничение размера IP-адреса 32 битами накладывает серьезное ограничение на количество хостов в Internet. Быстрое увеличение числа компьютеров в Internet в 90-е годы прошлого века привело к истощению доступного адресного пространства. Все больше и больше таких устройств, как телефоны, торговые автоматы и тостеры, получают собственные IP-адреса. Однако первоначально сеть ARPANET была разработана для поддержки относительно небольшого числа хостов. Тогда не предполагалось, что Internet станет общедоступной, всеобъемлющей, глобальной инфраструктурой, поэтому выделение адресов длиной 32-бита и трех классов сетей казалось вполне разумным. Версия IP-протокола с 32-битным адресным пространством получила название версии 4 (IPv4) протокола IP. Проект стандарта RFC 2460 [DH98] для версии 6 протокола Internet Protocol (IPv6) ориентирован на 128-битное адресное пространство [Hui98]. Однако IPv6 требует значительных изменений в инфраструктуре Internet. Во время подготовки этой книги к публикации широкого распространения IPv6 не наблюдалось. Вместо этого используются альтернативные возможности решения проблемы истощения адресного пространства IPv4.

Для ограничения взрывного увеличения требуемого количества IP-адресов были разработаны различные способы. Например, некоторые организации присваивают компьютерам IP-адреса динамически, чтобы избежать выделения IP-адреса машине, которая в данный момент не взаимодействует с Internet. Предположим, что у провайдера Internet имеется 500 модемов и 20000 клиентов. В любой заданный момент времени к сети подключено максимум 2,5% клиентов. Провайдер может оперировать гораздо меньшим пулом IP-адресов, присваивая IP-адрес пользователю, когда устанавливается модемное соединение. Точно так же, компании могут иметь большое число компьютеров, которые взаимодействуют внутри корпоративной сети, и иметь один компьютер (межсетевой экран или прокси-сервер), который координирует взаимодействие с Internet. Компьютеры внутри корпоративной сети невидимы для остальной части Internet; тем самым им не нужно иметь

уникальные IP-адреса. Благодаря этому большое число организаций может присваивать IP-адреса своим компьютерам из одного и того же блока IP-адресов.

Однако подобные механизмы окончательно не решают проблему истощения пространства IP-адресов. Чтобы замедлить этот процесс, инфраструктура Internet была видоизменена с целью обеспечения большей гибкости при выделении блоков IP-адресов. Ограничение на фиксированные размеры блока адресов было снято с появлением бесклассовой междоменной маршрутизации (CIDR — Classless InterDomain Routing) в начале 90-х годов [RL93, FLYV93, FRB93]. Технология CIDR позволила задавать разделение между сетевой частью и частью хоста IP-адреса в *любой* точке 32-разрядного двоичного числа. Таким образом, размер блока IP-адресов может быть любой степенью 2. Сеть CIDR идентифицируется по сетевому адресу и маске, в которой указывается, сколько битов выделяется под сетевую часть адреса. Например, рассмотрим сеть 204.70.2.0/23. Адрес сети длиной 23-бита оставляет 9 из 32 битов для представления 512 (2^9) хостов в этой сети. Последние 9 битов могут варьироваться от 000000000 до 111111111. Младший бит третьего октета может иметь значения 0 или 1, что соответствует диапазону чисел от 00000010 до 00000011. Следовательно, IP-адреса в этом блоке находятся в диапазоне от 204.70.2.0 до 204.70.3.255.

Переход к CIDR дает два важных преимущества. Во-первых, распределение блоков IP-адресов стало более гибким, что позволило эффективнее использовать 32-битное адресное пространство. Например, нужные организации 512 адресов могут быть назначены с помощью блока, имеющего 23-битную маску (что позволяет получить 2^9 = 512 адресов), вместо того, чтобы выделять целую сеть класса B (с 2^{16} = 65536 адресами). Во-вторых, CIDR дал возможность провайдерам объединить их сети в большие блоки, удобные для маршрутизации. Предположим, провайдеру была выделена сеть 12.0.0.0/8. Этот большой блок адресов может быть поделен на меньшие блоки, предоставляемые клиентам данного провайдера. Например, один клиент может иметь сеть 12.45.0.0/16, а другой — сеть 12.194.34.0/23. Выделение адресных блоков может зависеть от «размеров» клиента. Большой адресный блок, например, 12.45.0.0/16, может быть выделен крупному клиенту, в то время как меньший блок, такой как 12.194.34.0/23, может вполне устроить мелкого клиента.

Провайдеру необходимо знать, как осуществить доступ к каждой из этих отдельных сетей. Остальной же части Internet нужно лишь знать, как обратиться к сети провайдера, ответственного за эти адресные блоки. Вместо того чтобы хранить отдельную информацию о маршрутах для блоков 12.45.0.0/16 и 12.194.34.0/23, маршрутизаторы остальной части Internet могут хранить один маршрут для всего блока 12.0.0.0/8. При получении пакета, назначением которого является адрес в этом большом блоке, маршрутизатор должен переслать пакет провайдеру, ответственному за блок 12.0.0.0/8. Маршрутизатору не нужно ничего знать о каждом клиенте, который является конечным получателем пакета. Вернемся к аналогии с обычной почтой, в которой почтовый индекс используется для пересылки письма в нужное почтовое отделение. Далее название улицы, номер дома и квартиры используются для доставки письма получателю. В сетях провайдеров пакеты пересылаются по аналогичному принципу. Главным отличием сетей CIDR является то, что число уровней иерархии и число битов, используемое на каждом уровне, может достаточно гибко меняться. Подробнее об IP-маршрутизации вы можете узнать в других книгах [Hui00, HM00, Ste99].

5.1.4. Заголовок IP

Каждый IP-пакет имеет заголовок, представленный на рис. 5.4. Поля IP-заголовка обычно заполняются операционной системой хоста-отправителя. Хотя IP-маршрутизатор передает пакет, основываясь на адресе получателя, заголовок содержит дополнительные поля, которые важны для успешного взаимодействия между отправителем и получателем:

- **Номер версии (4 бита).** 4-битный номер версии обычно имеет значение 4, что соответствует версии IPv4. Знание номера версии дает возможность маршрутизаторам и хосту-получателю корректно интерпретировать содержимое заголовка. Другие версии протокола, например, IPv6, могут иметь иной формат заголовка.
- **Длина заголовка (4 бита).** 4-битное поле длины заголовка указывает на число 4-байтовых слов в заголовке. Основная часть IP-заголовка состоит из 20 байтов (пять 4-байтовых слов), но при использовании *опций IP* возможно использование более длинных заголовков. Длина IP-заголовка всегда кратна 32 битам.
- **Тип сервиса (8 битов).** 8 бит типа сервиса (TOS — Type Of Service) изначально включались в IP-заголовок, чтобы воздействовать на маршрут передачи пакета по сети. Например, маршруты могут иметь различные характеристики эффективности, такие как малое время задержки, высокая пропускная способность или высокая надежность. Пакет голосового IP-сообщения должен передаваться по маршруту с малой задержкой, поскольку аудиоприложения чувствительны к времени ожидания. В противоположность этому пакет большого файла следует пересылать по маршруту, имеющему высокую пропускную способность, поскольку время передачи файла зависит от пропускной способности. Однако большинство маршрутизаторов не осуществляли выбор маршрута на базе TOS, поэтому эти биты редко использовались. Повышение в 90-х годах интереса к поддержке приложений, требующих прогнозируемой пропускной способности, заставило вновь вернуться к использованию битов TOS с целью воздействовать на способ использования буферов и каналов для различных классов трафика.

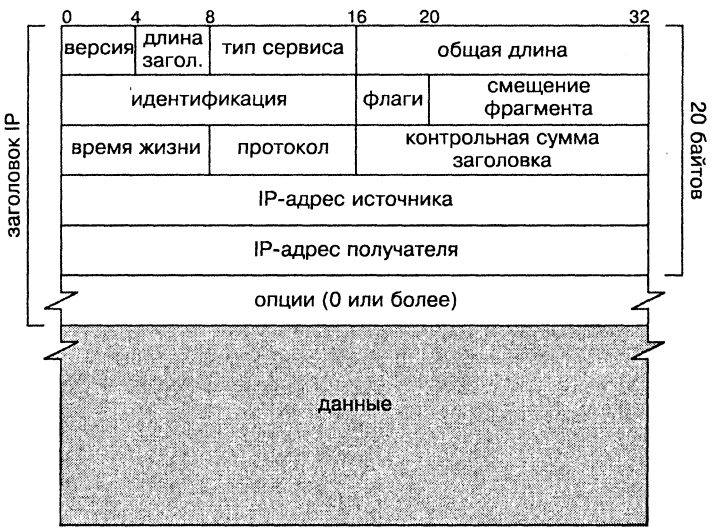


Рис. 5.4. Формат IP-пакета

- **Общая длина (16 битов).** 16-битное поле общей длины указывает на общее число байтов в пакете. IP-пакет может содержать до 65536 (2^{16}) байтов. Однако большинство технологий канального уровня не могут обрабатывать такие большие пакеты и используют пакеты меньшей длины, которые характеризуются *максимальной длиной пакета* (MTU — Maximum Transmission Unit). Например, многие локальные сети используют Ethernet, где MTU составляет 1500 байтов. Ограничение на MTU может реализовываться двумя способами. При первом подходе хост-отправитель избегает посылать IP-пакеты, длина которых превышает MTU в сетях на пути к хосту-получателю. Но приложение обычно не может заранее знать значения MTU. При втором подходе большой IP-пакет делится отправителем или маршрутизатором на два или более *фрагмента*. Фрагментация IP-пакетов поддерживается следующими тремя полями в IP-заголовке: 16-битный идентификатор, 3-битные флаги и 13-битное смещение.
- **Идентификация (16 битов).** 16-битное поле идентификатора содержит уникальное значение для каждого отправляемого IP-пакета. Перед тем, как передать пакет, IP-маршрутизатор проверяет, что размер пакета не превышает значение MTU канала. Слишком большой пакет может быть фрагментирован отправителем или промежуточным маршрутизатором. Каждый фрагмент пакета имеет один и тот же идентификатор. Это дает возможность адресату распознать, что эти фрагменты принадлежат одному пакету. В месте назначения фрагменты вновь объединяются вместе и протоколу верхнего уровня, например, TCP или UDP, передается один IP-пакет.
- **IP-флаги (3 бита).** Два из трех 1-битных флагов относятся к процессу фрагментации; оставшийся бит зарезервирован для дальнейшего использования. Бит «дополнительные фрагменты» устанавливается в 1 для всех фрагментов пакета, кроме последнего. Это гарантирует, что адресат сможет определить, все ли фрагменты получены. Фрагментация может быть запрещена путем установки бита «не использовать фрагменты». При получении пакета излишне большой длины с установленным битом «не использовать фрагменты» маршрутизатор отвергает пакет и уведомляет отправителя. Уведомление отправляется с использованием протокола Internet Control Message Protocol (ICMP) [Pos81]. Этот протокол применяется для передачи сообщений об ошибках и выборе маршрута.
- **Смещение фрагмента (13 битов).** 13-битное поле смещение фрагмента содержит смещение (в 8-байтных блоках) этого фрагмента от начала оригинального IP-пакета. Применение 8-байтных блоков дает возможность использовать 13-битное поле для указания смещения внутри IP-пакета, имеющего длину до 2^{16} (8×2^{13}) байтов, которое является максимальной допустимой длиной, описываемой 16-битным полем длины пакета. Для каждого фрагмента 16-битное поле длины пакета в IP-заголовке устанавливается в соответствии с числом байтов в этом фрагменте. Вместе смещение и длина позволяют получателю определить, какой диапазон в байтах покрывается этим фрагментом. После прибытия всех фрагментов получатель может собрать пакет.
- **Время жизни (8 битов).** 8-битное поле времени жизни (TTL — Time-To-Live) ограничивает число переходов на пути следования пакета. Отправитель устанавливает для поля TTL начальное значение. Затем каждый маршрутизатор на пути уменьшает значение в поле на единицу или на то число секунд, которое пакет провел в маршрутизаторе. Когда значение TTL достигает 0, пакет уничтожается, а отправителю посылается сообщение об ошибке по протоко-

лу ICMP. Эта процедура призвана решать проблему живучести при IP-маршрутизации. В ряде случаев неправильно настроенный маршрутизатор или отказавший канал связи могут вызвать зацикливание. Как результат, некоторые пакеты могут постоянно перемещаться между последовательностью маршрутизаторов, не продвигаясь к месту назначения. Пакеты, застрявшие в таком цикле, потребляют ресурсы сети и маршрутизатора. Пакет, в конце концов, может достигнуть места назначения уже после того, как приложения завершили взаимодействие. Поступление такого пакета может ввести в заблуждение получателя. Уничтожение пакетов с истекшим временем жизни (TTL) позволяет избежать этой проблемы.

- **Протокол (8 битов).** 8-битное поле протокола идентифицирует высокоуровневый протокол, ответственный за отправку IP-пакета. Зарезервированными являются следующие значения: 1 для ICMP, 6 для TCP и 17 для UDP. Информация о протоколе имеет важное значение для интерпретации получателем пакета данных, следующих за IP-заголовком. За IP-заголовком обычно идет другой заголовок, относящийся к высокоуровневому протоколу. Это можно сравнить с упаковкой письма в конверт, который в свою очередь помещается в другой конверт. Первый заголовок используется для того, чтобы достичь нужного компьютера-адресата, а второй заголовок используется для направления IP-пакета соответствующему приложению, работающему на компьютере-получателе. Каждый протокол высокого уровня имеет свой формат заголовка.
- **Контрольная сумма заголовка (16 битов).** 16-битная контрольная сумма дает возможность обнаружить повреждение какого-либо бита в заголовке по мере прохождения пакета через сеть. Так, шумы в линиях связи могут повредить один или несколько битов в заголовке, превратив 0 в 1 или 1 в 0. Повреждение адреса места назначения может привести к доставке пакета не тому хосту. Перед передачей пакета отправитель вычисляет 16-битную сумму битов в IP-заголовке и включает результат в поле контрольной суммы заголовка. IP-пакет с некорректной контрольной суммой отвергается хостом-получателем. Маршрутизаторы на пути обновляют контрольную сумму заголовка при пересылке IP-пакета, чтобы учесть любые изменения в заголовке, например, уменьшение значения поля TTL. Важно заметить, что контрольная сумма применяется только к IP-заголовку, а не к содержимому пакета. IP не обнаруживает повреждения данных. Проверка целостности данных может быть выполнена протоколом более высокого уровня, таким как UDP или TCP, с помощью отдельной контрольной суммы.
- **IP-адрес источника (32 бита).** IP-заголовок включает в себя 32-битный адрес, который идентифицирует отправителя пакета. Включение адреса отправителя в IP-заголовок дает возможность получателю идентифицировать отправителя. Например, приложение-получатель может захотеть отправить ответное сообщение. Кроме того, маршрутизаторы на пути передачи пакета должны знать адрес отправителя, чтобы иметь возможность отправлять сообщения об ошибках с помощью протокола ICMP (например, при истечении времени жизни пакета). Помимо этого многие маршрутизаторы могут быть настроены так, чтобы отвергать трафик нежелательных отправителей. Это предусматривает фильтрацию пакетов на основе поля адреса отправителя.
- **IP-адрес получателя (32 бита).** 32-битный адрес получателя необходим для идентификации получателя пакета. При получении пакета маршрутизатор ис-

пользует IP-адрес получателя для определения «следующего сегмента» на пути к получателю.

- **Опции IP (переменная длина).** Последнее поле IP-заголовка содержит список переменной длины для необязательной информации, общая длина опций кратна 32 битам. Опции IP могут использоваться для дополнительного управления маршрутизацией, в целях безопасности или для добавления маршрутизаторами в пакет отметки времени. Например, опция «исходная маршрутизация» позволяет отправителю задавать маршрут для пакета. Маршрут задается как список IP-адресов маршрутизаторов на пути передачи. Опция «исходная маршрутизация» начинается с 3 байтов, которые содержат код опции, длину опции и указатель на следующий адрес в списке, после чего следует список IP-адресов маршрутизаторов. Поскольку большинство маршрутизаторов оптимизированы для работы с 20-байтными заголовками фиксированного формата, что является типичным случаем, пакеты с опциями IP обычно обрабатываются гораздо медленнее. На практике большинство опций используются редко, а многие хосты и маршрутизаторы не поддерживают все имеющиеся опции.

Отправитель IP-пакета управляет содержимым заголовка. На протяжении многих лет заголовки использовались для целей, не соответствовавших начальному предназначению протокола:

- **Спуфинг адресов источника.** IP-адрес источника обычно соответствует IP-адресу хоста-отправителя. Однако отправитель может преднамеренно помещать в поле адреса источника произвольное значение. Такая подмена адреса источника, или *спуфинг (spoofing)*, обычно делается, когда отправитель хочет атаковать хост-получатель или сеть. Использование некорректного IP-адреса источника затрудняет установление инициатора атаки. Хотя отправитель не получит каких-либо ответных сообщений от получателя, он сможет загрузить хост и сеть получателя нежелательным трафиком. Это называется *атакой отказа от обслуживания (DoS — Denial of Service)*.
- **Определение MTU на пути следования.** Бит «не фрагментировать» в IP-заголовке играет важную роль в определении максимального размера пакета, разрешенного для передачи маршрутизаторами на его пути в сети [MD90]. Фрагментация пакета может быть нежелательной по нескольким причинам [KM87]. Выполнение фрагментации и последующая сборка загружает маршрутизатор и компьютер получателя, соответственно. Кроме того, потеря одного фрагмента (например, вследствие переполнения буфера следующего на пути маршрутизатора) приводит к потере всего IP-пакета, поскольку в IP отсутствует механизм для повторной передачи утерянного фрагмента. При отказе от фрагментации компьютеру-отправителю необходимо знать размер MTU на пути следования к получателю. ICMP-сообщения для пакетов с размером, превышающим допустимый, предоставляет для отправителя возможность получить MTU на пути к получателю. Отправитель может поэкспериментировать с отправкой пакетов различного размера с установленным битом «не фрагментировать», чтобы посмотреть, будет ли какой-либо из маршрутизаторов на пути пакета генерировать сообщение об ошибке. На практике различия в значениях MTU для различных технологий канального уровня не представляет сколько-нибудь существенной проблемы. Коммуникационное программное обеспечение большинства компьютеров используют значением MTU по умолчанию (576 или 1500 байтов) чтобы избежать фрагментации пакетов внутри сети.

- **Идентификация переходов на пути к получателю.** Как и бит «не фрагментировать», поле времени жизни TTL дает отправителю способ получить информацию о маршруте к месту назначения. Чтобы узнать адрес первого маршрутизатора на пути следования, отправитель может передать пакет со значением TTL, равным 1. После этого первый маршрутизатор отправит сообщение об ошибке с помощью протокола ICMP, включающее информацию, идентифицирующую маршрутизатор. Повторение процесса с увеличенными значениями TTL дает возможность отправителю идентифицировать последующие маршрутизаторы на пути. Этот процесс положен в основу популярной утилиты **traceroute**[1] [Jac], которая используется для диагностирования и устранения проблем с маршрутизацией. Однако **traceroute** не всегда дает точный маршрут *от начала и до конца*. Поскольку IP не гарантирует, что последующие пакеты будут проходить по одному и тому же маршруту, пакеты с различными значениями TTL могут идентифицировать маршрутизаторы на различных маршрутах. Аналогично, IP не гарантирует, что пакеты, отправляемые компьютером A компьютеру B будут проходить по тому же пути, по которому следует обратный трафик от B к A. Следовательно, **traceroute** предоставляет информацию только о пути от A к B, но не об обратном пути от B к A. Для того чтобы узнать обратный путь от B к A, необходимо выполнить **traceroute** с компьютера B.

IP предлагает простой и устойчивый сервис по доставке пакетов, который работает с разнообразными технологиями канального уровня и обеспечивает работу протоколов транспортного уровня.

5.2. Transmission Control Protocol

Протокол Transmission Control Protocol (TCP) координирует передачу данных между двумя приложениями. Приложения взаимодействуют, осуществляя чтение и запись с помощью *сокетов (socket)*, которые позволяют представить передаваемые данные как упорядоченный поток байтов с гарантированной доставкой. TCP обеспечивает логическое соединение между двумя конечными пунктами на основе сервиса по доставке пакетов IP. TCP-отправитель делит данные на *сегменты* и передает каждый сегмент в IP-пакете вместе с TCP-заголовком. Перед началом передачи данных конечные пункты должны договориться об установлении TCP-соединения. В процессе передачи данных конечные пункты совместно управляют потоком данных и повторной передачей утерянных IP-пакетов. Кроме того, каждый конечный пункт адаптирует свою скорость передачи к состоянию сети, чтобы избежать ее перегрузки. TCP-заголовок содержит информацию, необходимую для координации действий по упорядоченной, гарантированной доставке сегментов. Подробнее о TCP вы можете узнать из других книг [Ste94, KR00].

5.2.1. Абстракция сокетов

Абстракция сокетов обеспечивает надежное двунаправленное взаимодействие между двумя приложениями, обычно работающими на различных компьютерах. Создание приложений, которые непосредственно передают и получают IP-пакеты, сопряжено с большими сложностями. Вместо этого для координации передачи

[1] В операционных системах семейства Windows эта утилита называется tracert. — *Прим. ред.*

IP-пакетов и предоставления более удобного сервиса применяются протоколы транспортного уровня. Многим приложениям нужно, чтобы они передавали данные в сеть и получали данные из сети точно таким же образом, как осуществляется запись в файл или чтение из файла. Если отправитель передает 5000 байтов, получатель должен получить 5000 байтов. Процесс передачи не должен вызывать утерю, повреждение данных или изменение порядок байтов. Таким образом, передающему и принимающему приложению нужно, чтобы они взаимодействовали через соединение, обеспечивающее *упорядоченный, надежный поток байтов*. IP не предоставляет такого сервиса. Этот сервис предоставляется протоколом Transmission Control Protocol (TCP), который обычно реализуется в операционной системе каждого хоста, подключенного к Internet.

Приложение использует TCP путем создания сокета (двунаправленного канала), что сходно с открытием файла. Оба приложения должны иметь однозначный способ для идентификации сокета. Знать IP-адреса двух компьютеров недостаточно. На одном компьютере могут выполняться несколько приложений, а одно приложение, например, Web-сервер, может работать с несколькими сокетами. Каждый сокет ассоциируется с номером *порта* на каждом конце соединения. Номер порта представляет собой 16-битное целое число в диапазоне от 0 до 65535. Номера, меньшие 1024, являются *известными портами*, зарезервированными для определенных протоколов прикладного уровня. Например, порт 80 предназначен для HTTP. Назначение номеров портов приложениям осуществляется организацией по административному управлению доменами Internet Assigned Number Authority (IANA). Оставшиеся номера портов в диапазоне от 1024 до 65535 могут быть использованы любым приложением. Однако в некоторых случаях вновь разработанный Internet-сервис может не иметь общеизвестного зарезервированного порта. Тогда выбирается номер порта, который де-факто станет стандартным для этого приложения.

По умолчанию Web-клиент, например, браузер создает сокет, который соединяется с портом 80 на компьютере сервера. Однако не возбраняется и выбор другого номера порта. Например, Web-сервер может быть сконфигурирован для прослушивания клиентских запросов на порту 8000. В этом случае клиенту необходимо знать, что следует запрашивать соединение с портом 8000, а не с портом 80. Это достигается путем указания номера порта в URL. Например, если пользователь запрашивает ресурс **http://www.foo.com:8000/bar.html**, клиент должен создать сокет, который соединяется с портом 8000 сервера, а не с портом 80. Клиенту также нужен номер порта на своей стороне соединения. В противном случае сервер не будет знать, как направлять данные клиенту. В процессе создания сокета операционная система на компьютере клиента присваивает клиентскому приложению временный номер порта (от 1024 до 65535). Сокет идентифицируется по пяти информационным составляющим: двум IP-адресам (для компьютеров, выполняющих приложения), двум номерам портов (для двух приложений, выполняющихся на каждой стороне соединения) и протоколу (TCP).

Приложения создают сокеты с помощью системных вызовов, реализуемых операционной системой. Допустим, приложение A (например, Web-клиент) создает сокет для соединения с удаленным приложением B (например, Web-сервером). Приложение A инициирует создание сокета, сделав системный вызов. В операционной системе UNIX для создания нового сокета используется функция *socket()* [Ste98b]. Затем приложение делает системный вызов *connect()* для связывания сокета с IP-адресом и номером порта приложения B. Во время выполнения вызова *connect()* операционная система также выбирает неиспользуемый локальный номер порта (от 1024 до 65535)

для приложения A. На этот момент операционной системе, выполняющей приложение A, известны два IP-адреса и два номера порта, которые уникально идентифицируют двунаправленное соединение между двумя приложениями. Затем операционная система инициирует установку TCP-соединения с приложением B. После того как соединение установлено, вызов *connect()* завершается, а приложение A может начать чтение данных из сокета и запись данных в сокет.

В противоположность этому, приложение B первоначально играет пассивную роль в создании соединения. Приложение B ожидает поступления запросов на определенный порт для установления соединения. В UNIX это реализуется путем создания сокета и вызова *bind()* для назначения локального номера порта (например, порта 80). Затем приложение B делает системный вызов *listen()* для прослушивания запросов на соединение от удаленных приложений. Это заставляет операционную систему отвечать на любые запросы на установление TCP-соединений с этим портом. Приложение узнает об этих новых TCP-соединениях с помощью системного вызова *accept()*. По умолчанию вызов *accept()* ждет запроса на новое соединение. После получения запроса завершается создание сокета. С этого момента приложение B может начать чтение данных из сокета или запись данных в сокет.

После того, как соединение установлено, любое из приложений может читать или записывать данные. Фактически, оба приложения могут читать и записывать одновременно, поскольку сокет предоставляет двунаправленный коммуникационный канал. Web-клиент (приложение A) инициирует взаимодействие. Клиент записывает HTTP-запрос в сокет. Сервер (приложение B) ожидает получения данных через свой сокет. Затем после поступления данных сервер читает HTTP-запрос из сокета. После обработки запроса сервер записывает HTTP-ответ в сокет. В то же самое время клиент ожидает получения данных через сокет. Подобный образ действий типичен для приложений клиент–сервер. TCP предоставляет гораздо более широкие возможности для поддержания двунаправленного взаимодействия между двумя приложениями. В общем случае и A, и B может первым осуществить запись данных, либо оба приложения могут читать и записывать данные одновременно.

Операционная система берет на себя все действия по установке логического соединения между двумя приложениями и по координации передачи IP-пакетов. Для приложения A операционная система выполняет системные вызовы *socket()* и *connect()*. Если удаленный хост не отвечает, операционная система информирует приложение A, что запрос на создание сокета окончился неудачей. Для приложения B операционная система выполняет функции *socket()*, *bind()*, *listen()* и *accept()*. В процессе создания сокета каждый хост выделяет оперативную память для передачи и приема пакетов. Когда приложение записывает в сокет, операционная система направляет данные получателю на удаленный IP-адрес и порт. Точно так же, при поступлении пакетов операционная система направляет данные соответствующему сокету, основываясь на номере порта. Приложение может затем прочитать данные из сокета. На нижнем уровне при взаимодействии приложений операционная система координирует отправку и получение пакетов с целью создания абстракции упорядоченного, надежного потока байтов.

5.2.2. Упорядоченный, надежный поток байтов

TCP-соединение доставляет данные в виде упорядоченного потока, гарантируя доставку данных. Каждый IP-пакет имеет заголовок, информация в котором идентифицирует хосты отправителя и получателя. Информации из IP-заголовка доста-

точно маршрутизаторам в сети для пересылки пакета соответствующему хосту-получателю. Однако IP-заголовок не предоставляет достаточно информации, чтобы ассоциировать входящий IP-пакет с нужным сокетом. Для решения этой проблемы TCP-отправитель создает заголовок внутри IP-пакета, который содержит дополнительную информацию. Если вновь обратиться к аналогии с обычной почтой, это подобно помещению одного конверта внутрь другого; информации на внешнем конверте достаточно, чтобы определить адрес получателя с точностью до почтового отделения. С точки зрения почты, внутренний конверт является просто частью содержимого внешнего конверта. Получатель может открыть внешний конверт и изучить информацию на внутреннем конверте.

Точно так же TCP располагается поверх IP. TCP-заголовок содержится в данных IP-пакета. Маршрутизаторам в сети не нужно просматривать биты в TCP-заголовке. После того, как IP-пакет достиг компьютера-получателя, операционная система изучит TCP-заголовок, чтобы направить данные соответствующему сокету. Сокет идентифицируется двумя 16-битными номерами портов, которые включаются в TCP-заголовок. Разделение функций между IP и TCP очень важно. TCP-заголовок должен содержать достаточно информации для передающего и принимающего хостов, чтобы те могли передать данные через сокет. TCP должен учитывать тот факт, что IP-пакеты могут быть утеряны, повреждены или доставлены не в том порядке. Эти проблемы решаются путем взаимодействия между TCP-отправителем и TCP-получателем.

Рассмотрим передачу сообщения из одного приложения в другое. Операционная система на передающем компьютере делит сообщение на сегменты, каждый из которых представляет собой последовательность байтов, помещенных в IP-пакет. TCP-заголовок идентифицирует соединение, ассоциированное с сегментом. Чтобы обработать пакеты, поступающие не в том порядке, отправитель помечает каждый сегмент номером. Получатель отвечает за упорядочение пакетов, поступивших не в том порядке. Если пакет 2 прибыл до пакета 1, операционная система получающего хоста ожидает пакета 1, прежде чем доставить данные получающему приложению. Отправитель также включает информацию, помогающую получателю определить, были ли данные повреждены в процессе передачи. В частности, TCP-отправитель вычисляет контрольную сумму содержимого пакета и включает контрольную сумму в TCP-заголовок; получатель пересчитывает контрольную сумму и отвергает пакет, если результаты не совпадают.

Отправитель не знает, что пакеты достигли получателя. Чтобы разрешить эту проблему, получатель посылает отправителю подтверждение, указывающее, что пакеты были получены. Например, после получения сегментов 1 и 2 получатель может проинформировать отправителя, что первые два сегмента поступили. Если получатель имеет данные, ожидающие передачи, подтверждение и исходящий сегмент могут быть включены в один пакет. Если подтверждение не получено, отправитель делает предположение, что пакет 1 был утерян. В действительности доставка пакета могла быть задержана, пакет мог быть поврежден, либо утеряно подтверждение. Однако отправитель не может распознать и различить эти ситуации. Вместо этого отправитель просто передает другую копию данных, включая порядковый номер. Если подтверждение не получено, отправитель может передать еще одну копию. При получении по крайней мере одной (неповрежденной) копии пакета получатель посылает отправителю подтверждение. Если поступило более одной копии, получатель может просто отвергнуть лишние копии.

5.2.3. Открытие и закрытие TCP-соединения

Флаги SYN, ACK, FIN и RST в TCP-заголовке используются при открытии и закрытии TCP-соединения. Пакеты с этими флагами отправляются в ответ на системные вызовы, которые открывают или закрывают соответствующий сокет. Когда приложение A создает сокет, операционная система координирует установку TCP-соединения с приложением B на удаленной машине. Установка TCP-соединения состоит в *трехэтапном обмене с квитированием (handshake)*, как показано на рис. 5.5.

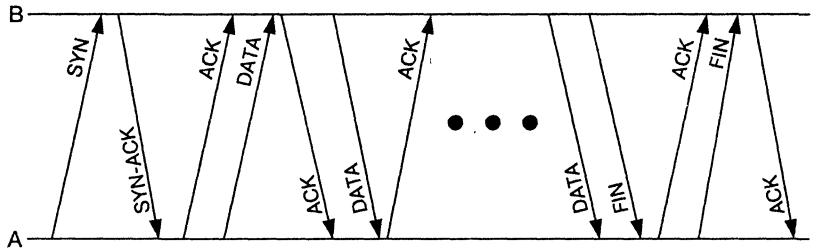


Рис. 5.5. Временная диаграмма TCP-соединения

1. **SYN от A к B.** A инициирует соединение, отправляя пакет с установленным битом SYN. SYN-пакет содержит *начальный порядковый номер* для потока в 32-битном поле порядкового номера. Таким образом, SYN-пакет *синхронизирует* порядковый номер. SYN-пакет сам по себе является частью потока и имеет первый порядковый номер. Предположим, что SYN-пакет имеет порядковый номер, равный 4500. После этого первый пакет данных от A должен иметь порядковый номер 4501, что идентифицирует первый байт данных.
2. **SYN-ACK от B к A.** После передачи SYN-пакета A ожидает ответа от удаленного хоста. Предположим, что приложение B ожидает запрос на установку TCP-соединения на определенном порту. Прибытие SYN-пакета от A обусловливает создание сокета и передачу подтверждающего пакета от B к A. B посылает пакет A, в котором установлены флаги SYN и ACK. Флаг SYN инициирует соединение в обратном направлении (от B к A), а флаг ACK подтверждает получение SYN-пакета от A. Номер подтверждения в TCP-заголовке устанавливается равным значению, на единицу большему, чем начальный порядковый номер в SYN-пакете от A. Как и SYN-пакет от A, пакет SYN-ACK содержит начальный порядковый номер. Этот порядковый номер отмечает начало потока байтов, передающихся от B к A, и не имеет отношения к начальному порядковому номеру для трафика от A к B.
3. **ACK от A к B.** После поступления пакета SYN-ACK установление соединения от A к B завершается, и операционная система информирует приложение A, что соединение было установлено. С этого момента приложение A может начать запись и чтение из сокета. Однако в данный момент приложение B не знает, получило ли приложение A пакет SYN-ACK. Третья часть в трехэтапном взаимодействии состоит в отправке пакета ACK от A к B для подтверждения создания соединения от B к A. ACK-пакет от A имеет номер подтверждения на единицу больший, чем начальный порядковый номер в пакете SYN-ACK от B. После получения этого пакета приложение B может передавать данные через сокет приложению A.

Операционная система выбирает различные начальные порядковые номера для TCP-соединений. Посмотрим, что произойдет, если каждое TCP-соединение использует в качестве начального порядкового номера 0. Предположим, что по TCP-соединению между A и B по сети передается ожидающий обработки пакет, что сопровождается большой задержкой времени. В конце концов, пакет будет повторно передан и доставлен получателю. Предположим также, что A и B закрывают TCP-соединение, как только передача данных заканчивается. После этого A и B могут установить новое TCP-соединение с теми же номерами портов, что и для старого соединения. Допустим, что после установления нового соединения повторный пакет, принадлежащий старому соединению, достигает, наконец, места назначения. В этой ситуации получатель может ошибочно ассоциировать старый пакет с новым соединением и доставить данные приложению. Если новое соединение имеет другой начальный порядковый номер, то получатель сможет распознать, что задержанный пакет не принадлежит этому упорядоченному потоку байтов.

TCP-соединение может быть завершено на любой стороне. Приложение инициирует закрытие TCP-соединения путем закрытия соответствующего сокета. Завершение соединения обычно осуществляется путем *четырехэтапного* обмена. Если приложение B закрывает сокет первым, как показано на рис. 5.5, эти четыре этапа заключаются в следующем:

1. **FIN от B к A.** Приложение B закрывает сокет, что приводит к передаче пакета с установленным флагом FIN. С этого момента B не передает каких-либо новых данных. Тем не менее, B ответственно за гарантированную, упорядоченную доставку предыдущих данных, отправленных приложению A. Так что операционная система, под управлением которой выполняется приложение B, продолжает повторную передачу утерянных пакетов потока. Кроме того, B может продолжать получать и подтверждать получение пакетов от A. Подобно пакету SYN, пакет FIN от B занимает один номер в пространстве порядковых номеров.
2. **ACK от A к B.** После получения этого пакета A передает пакет ACK с номером подтверждения, который на единицу больше, чем порядковый номер в пакете FIN от B. Выделение порядкового номера флагу FIN позволяет A подтвердить получение флага, даже если пакет FIN не содержит каких-либо данных.
3. **FIN от A к B.** После того, как приложение A прочитает все байты из сокета, следующая операция чтения покажет, что был достигнут конец данных. В этот момент приложение A будет знать, что B не предполагает передавать каких-либо дополнительных данных. Раз так, приложение A закрывает свой сокет, что приводит к передаче пакета FIN приложению B.
4. **ACK от B к A.** После получения пакета FIN B передает пакет ACK для подтверждения закрытия соединения от A к B. После получения пакета ACK A становится известно, что B получило пакет FIN. С этого момента соединение закрыто. Передача каких-либо новых данных между A и B потребует открытия нового TCP-соединения.

Следует отметить, что возможна ситуация, когда оба приложения инициируют закрытие соединения одновременно. В этом случае приложения будут отправлять свои FIN-пакеты параллельно. Для полного закрытия соединения оба этих FIN-пакета должны быть подтверждены.

Четырехэтапный обмен пакетами FIN и ACK является обычным способом закрытия TCP-соединения. Однако TCP-заголовок также содержит флаг RST для сброса соединения в особых ситуациях. Предположим, хост получает пакет данных TCP

для соединения, которое не существует, т.е. IP-адреса и номера портов в заголовках IP и TCP не соответствуют какому-либо активному сокету этого хоста. В этом случае хост отвечает пакетом RST. Рассмотрим другую ситуацию. Допустим, что приложение посылает пакет SYN на определенный порт другого хоста. Если ни одно из приложений хоста-получателя не ожидает запросов на этом порту, то хост также отвечает пакетом RST. В качестве примера предположим, что на компьютере **www.foo.com** отсутствует Web-сервер. Если нет программы, прослушивающей запросы на порту 80, то хост отправит RST-пакет отправителю, который пытается открыть TCP-соединение. Операционная система на отправляющей машине должна уведомить приложение, что попытка создать сокет была неудачной.

5.2.4. Управление потоком с помощью скользящего окна

TCP-отправитель ограничивает передачу данных, чтобы избежать переполнения буфера на стороне получателя. Теоретически в TCP можно передавать данные в то время, когда приложение записывает данные в сокет. Однако TCP ограничивает передачу данных по двум существенным причинам. Во-первых, отправитель не должен передавать больше данных, чем получатель может хранить в своих буферах, — передача излишних данных приведет к переполнению буфера на стороне получателя и к утере пакетов. Во-вторых, отправитель не должен передавать данные быстрее, чем сеть может их обработать, — слишком «агрессивная» передача может привести к перегрузке сети и возникновению заторов, которые увеличивают время ожидания и повышают вероятность утери пакетов. Каждый TCP-отправитель ограничивает число ожидающих обработки (неподтвержденных) байтов в сети, используя технологию управления потоком с помощью *скользящих окон*. Чтобы избежать переполнения буфера у получателя, пакеты от B к A содержат в TCP-заголовке *окно приема*.

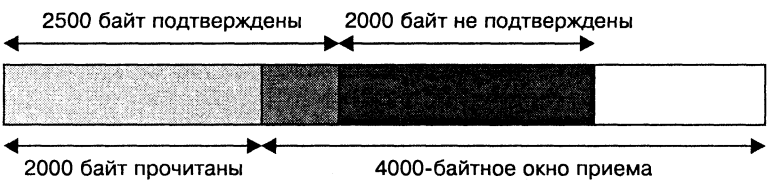


Рис. 5.6. Пример 4000-байтного окна приема

Окно приема определяет число байтов, которые приложение A может отправить после последнего байта, подтвержденного приложением B. Предположим, что операционная система, под управлением которой выполняется приложение B, выделила 4000-байтовый входной буфер для хранения входящих данных для этого TCP-соединения, как показано на рис. 5.6. Предположим также, что приложение B получило и подтвердило прием 2500 байтов от A, и 2000 байтов из этих 2500 байт были прочитаны приложением B. Тогда во входном буфере остается еще 500 байтов данных. Приложение B не может обработать более 3500 дополнительных байтов данных. ACK-пакет приложению A подтверждает получение первых 2500 байт и сообщает о размере окна приема в 3500 байтов. После получения пакета ACK приложение A может продолжить передавать данные. Однако A не может отправить более 3500 дополнительных байтов данных. В противном случае может произойти переполнение входного буфера на стороне приложения B. Допустим, что A отправляет дополнительные 2000 байтов, что составит всего 4500 байт. Пер-

вые 2000 байт были отправлены, получены, подтверждены и прочитаны. Следующие 500 байт были отправлены, получены и подтверждены, но не были прочитаны приложением B. Поскольку следующие 2000 байтов были отправлены, но не подтверждены, A не знает о том, что они были получены.

Следующие 1500 байт могут быть переданы немедленно. Приложение A не может отправлять оставшиеся данные без риска вызвать переполнение входного буфера на стороне B. В любой заданный момент времени A не может передать данные, если они выходят за границу текущего окна приема, которая определяется суммой последнего числа подтверждения и размером окна, полученного от B. Умалчиваемый размер входного буфера TCP является настраиваемым параметром операционной системы. На практике большинство операционных систем выделяют буфер размером 8 Кб, 16 Кб или 32 Кб. Приложения могут с помощью системного вызова увеличить размер буфера приема. Для пары хостов с большим временем доставки (например, если клиентом и сервером находится большое число маршрутизаторов) размер буфера приема может ограничивать производительность. Например, предположим, что пара хостов имеет время доставки в 500 мс, а размер входного буфера на стороне B равен 8 Кбайтам. Отправитель A не может передать более 8 Кбайтов каждые полсекунды. Это соответствует максимальной скорости передачи 128 Кб/с вне зависимости от пропускной способности соединения от A к B. В общем случае производительность TCP обратно пропорциональна времени доставки данных между отправителем и получателем.

5.2.5. Повторная передача утерянных пакетов

Повторная передача утерянных пакетов играет важную роль в обеспечении протоколом TCP надежной доставки потока байтов. IP не информирует TCP-отправителя, когда пакет теряется. Вместо этого отправитель должен предположить, что пакет был потерян, на основе ответа (или отсутствия ответа) получателя. Получатель подтверждает прием данных от отправителя путем передачи пакетов квитирования — пакетов с установленным битом ACK и с числом подтверждения, указывающим следующий байт, ожидаемый в потоке байтов от отправителя. Если получатель имеет данные для передачи, подтверждение может быть возвращено с данными. Отправитель делает вывод, что пакет был потерян, двумя способами: по таймауту повторной передачи или повторному подтверждению. Отправитель устанавливает таймер повторной передачи после передачи данных получателю. Если отсчет времени таймером завершается до прибытия подтверждения, отправитель полагает, что пакет был утерян на пути к получателю. Выбор соответствующего значения таймаута повторной передачи (RTO — Retransmission TimeOut) — достаточно тонкий процесс. Установка слишком малого значения RTO приводит к «ложной тревоге», и отправитель напрасно повторно отправляет пакет, который в действительности не был утерян. Установка слишком большого значения RTO откладывает обнаружение утерянного пакета, что приводит к неоправданной задержке повторной передачи пакета.

Правильное значение таймаута повторной передачи зависит от расстояния между отправителем и получателем, а также от загруженности сети. Компьютеры, задержка при передаче данных между которыми составляет пять секунд, должны использовать большее значение RTO, чем компьютеры, с задержкой 200 мс. Выбор нужного значения RTO определяется конкретной ситуацией. TCP-отправитель может *узнать* нужное значение RTO, исследовав величину задержки при передаче данных получателю. TCP-отправитель оценивает время «прохождения данных

туда и обратно» (RTT — Round Trip Time) — время между передачей пакета и получением подтверждения. На основе этих измерений отправитель может оценить среднее значение RTT, а также его дисперсию. Значение RTO устанавливается, исходя из значения RTT плюс некая дополнительная величина, которая зависит от изменчивости измеренной задержки и позволяет избежать необоснованных повторных передач [PA00]. Оценка RTT становится тем точнее, чем интенсивнее обмен данными между отправителем и получателем.

В некоторых случаях отправитель может сделать вывод, что пакет был утерян, не дожидаясь истечения RTO перед повторной передачей. Допустим, отправитель посылает получателю несколько пакетов. При этом второй пакет был утерян, а третий, четвертый и пятый пакеты достигли получателя. После получения первого пакета получатель отправляет ACK-пакет. Число подтверждения задается равным номеру первого байта, ожидаемого во втором пакете. Однако поскольку второй пакет был утерян, получатель принимает следующим третий пакет. Получатель отправляет еще один ACK-пакет. Число подтверждения по-прежнему указывает на первый байт второго пакета, поскольку это поле устанавливается на основе получения *непрерывного* потока байтов. После поступления четвертого пакета получатель отправляет еще один ACK-пакет с тем же самым числом подтверждения. На этот момент отправитель получил три ACK-пакета с одним и тем же числом подтверждения.

Получение пакетов *повторного подтверждения* дает возможность отправителю сделать вывод, что второй пакет данных был утерян. Тем не менее, отправитель не должен реагировать слишком быстро. Возможно, что второй пакет был задержан, но не утерян, т.е. третий и четвертый пакеты были доставлены не в том порядке. Получение трех повторных ACK-пакетов (четырех идентичных подтверждений) является серьезным аргументом в пользу того, что второй пакет данных действительно был утерян. Вместо того чтобы ожидать завершения работы таймера повторной передачи, отправитель выполняет *быструю повторную передачу* второго пакета. Быстрая повторная передача приводит к гораздо более быстрому восстановлению утерянного пакета. Вероятность быстрой повторной передачи по отношению к времени таймаута повторной передачи зависит от множества факторов, включая длину передаваемых данных и степень загруженности соединения от отправителя к получателю.

По большей части механизм повторной передачи сокращает задержку в восстановлении утерянного пакета без выполнения необоснованных повторных передач. Однако повторные подтверждения имеют место и в тех случаях, когда пакеты от TCP-отправителя доставляются не в том порядке. Такие повторные подтверждения могут склонить TCP-отправителя к предположению, что пакет был утерян. Выпадение пакета из установленного порядка следования может случиться, если IP-пакеты проходят через сеть к получателю по различным маршрутам. Например, предположим, что пакет 7 продвигается медленнее, чем пакеты 8, 9 и 10, в результате чего пакет 7 прибывает после пакета 10. TCP-получатель передает повторные подтверждения при получении пакетов 8, 9 и 10. Третье повторное подтверждение приводит к повторной передаче TCP-отправителем пакета 7. Беспорядочное поступление пакетов снижает производительность TCP. Хотя протокол IP не гарантирует доставку пакетов в строго определенном порядке, пакеты имеют тенденцию прибывать упорядоченно. Однако изменения в маршрутах доставки и использование нескольких маршрутизаторов между парой хостов на практике ведут к изменению порядка следования пакетов.

5.2.6. Адаптация к загруженности сети в TCP

TCP-отправитель приспосабливается к загруженности сети путем уменьшения скорости передачи данных. Такая адаптация имеет важное значение в свете быстрого роста Internet. В противном случае группа агрессивных TCP-соединений может перегрузить сеть и привести к потере большого числа пакетов. Повторная передача этих пакетов только увеличивает загрузку. Однако сама сущность протокола IP, в котором отсутствует понятие соединения, затрудняет управление загруженностью маршрутизаторами в сети. Вместо этого ответственность за управление загруженностью берут на себя оконечные хосты. Поскольку IP не предоставляет явной информации о загруженности сети, TCP-отправитель должен сделать вывод, что сеть перегружена, на основе косвенных наблюдений за производительностью [Jac88]. TCP-отправитель осуществляет передачу данных с помощью *скользящего окна*, размер которого зависит от доступного размера входного буфера получателя и пропускной способности сети, которые представлены *окном приема* и *скользящим окном*, соответственно. Отправитель передает данные на основе минимального из этих двух значений, чтобы избежать переполнения буфера приема и предотвратить перегрузку сети. Перегрузка сети приводят к потере IP-пакетов. Обнаружив, что пакет был утерян, отправитель уменьшает размер скользящего окна, чтобы снизить скорость передачи. При отсутствии потерь пакетов TCP-отправитель постепенно увеличивает размеры скользящего окна для более агрессивной передачи данных.

TCP реализует алгоритм *аддитивного увеличения* и *мультипликативного уменьшения* для изменения размера скользящего окна в соответствии с требованиями документа RFC 2581 [APS99]. При отсутствии потерь пакетов отправитель постепенно (линейно) увеличивает размер скользящего окна. В частности, отправитель увеличивает окно на максимальный размер сегмента (MSS — Maximum Segment Size) для TCP-соединения. Например, при максимальной длине пакета (MTU) в 1500 байтов величина MSS должна составить 1460 байт, чтобы оставить место для двух 20-байтных заголовков IP и TCP. В ответ на утерю одного или нескольких пакетов отправитель быстро (мультипликативно, т.е. в геометрической прогрессии) уменьшает скользящее окно для снижения нагрузки на сеть. После получения третьего повторного ACK-пакета размер скользящего окна устанавливается равным половине величины текущего значения. Аддитивное (в арифметической прогрессии) увеличение и мультипликативное (в геометрической прогрессии) уменьшение размеров скользящего окна приводит к изменениям данной величины во времени, как показано на рис. 5.7. График имеет пилообразную форму с постепенным увеличением скользящего окна, пока соединение не столкнется с утерей пакета.

Процесс увеличения и уменьшения размера скользящего окна позволяет экспериментально определить необходимую скорость передачи в сети. Предположим, что четыре TCP-соединения используют совместно один канал. Для простоты примем, что соединения характеризуются одинаковым RTT, и что окно приема не ограничивает скорость передачи данных. Если соединение прекращает передачу данных, другие три соединения будут увеличивать размеры своих скользящих окон, чтобы использовать в среднем одну третью часть пропускной способности канала. Аналогично, если пятое соединение начинает передавать данные, каждое из соединений будет уменьшать свое скользящее окно, чтобы освободить в среднем одну пятую пропускной способности.

TCP предоставляет эффективный способ настройки скорости передачи для каждого соединения в ответ на изменение загрузки сети. Однако *новое* соединение не обладает какой-либо информацией о состоянии сети. Если начать с большого

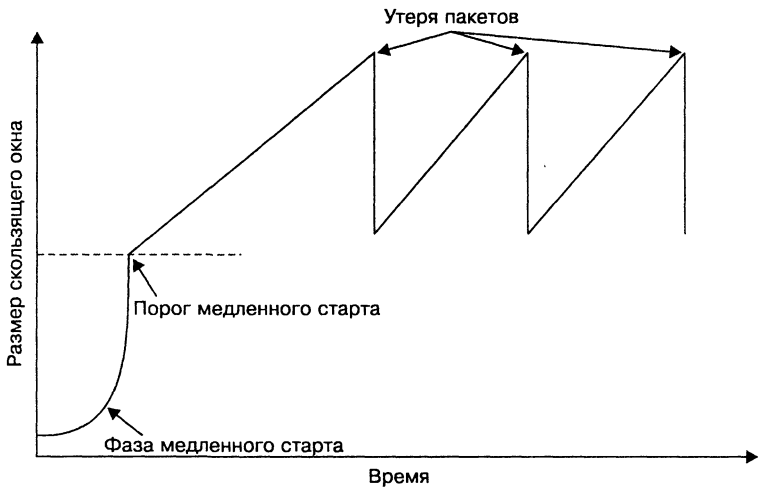


Рис. 5.7. Изменение размера скользящего окна TCP от времени

скользящего окна, то тем самым трафик в сети резко увеличится. Это может привести к перегрузке сети. Вместо этого TCP начинает новые соединения с фазы *медленного старта*, как показано на рис. 5.7. В начале фазы медленного старта TCP-отправитель начинает с небольшого скользящего окна, размер которого равен одному или двум MSS. Таким образом, отправитель может передать получателю только один или два пакета максимального размера. Скользящее окно увеличивается на значение MSS с получением каждого подтверждающего пакета от получателя. Тем самым скользящее окно увеличивается в два раза после получения подтверждений на данные, размер которых равен размеру окна. В результате в процессе фазы медленного старта скользящее окно увеличивается *мультипликативно*, а не линейно. Это позволяет соединению быстрее увеличить размер скользящего окна.

Несмотря на мультипликативное увеличение размера окна, отправитель на первых порах передает данные относительно медленно. В это время TCP-соединение не может полностью воспользоваться имеющейся пропускной способностью пути от отправителя к получателю. Фаза медленного старта завершается, как только скользящее окно достигает *порога медленного старта*. TCP-отправитель регулирует значение порога медленного старта в ответ на утерю пакетов [APS99]. При достижении скользящим окном границы фазы медленного старта TCP-отправитель переходит к фазе *недопущения перегрузки*, в которой размеры скользящего окна линейно увеличиваются. Аддитивное увеличение и мультипликативное уменьшение продолжается, пока у TCP-отправителя не возникнет таймаут повторной передачи. При возникновении таймаута повторной передачи отправитель полагает, что загруженность сети значительно изменилась. В результате TCP-отправитель сбрасывает размер скользящего окна в его начальное значение, равное одному или двум MSS, и повторяет фазу медленного старта.

Механизмы управления скользящим окном TCP достаточно сложны, а детали их различаются для разных версий протокола TCP [FF96]. Совершенствование управления скользящим окном TCP является областью активных исследований [APS99, MMFR96, FF96, AFP98, RF99, Flo94, Flo00]. Большинство предлагаемых расширений TCP пытаются сделать управление скользящим окном менее консервативным с целью повышения производительности приложений. Тем не менее, основные операции

управления скользящим окном остаются теми же. В общем случае каждое новое соединение и соединения, столкнувшиеся с таймаутом повторной передачи, проходят через фазу медленного старта. Фаза медленного старта предполагает небольшой начальный размер скользящего окна и мультипликативное увеличение его размера по мере подтверждения пакетов. После фазы медленного старта соединение входит в фазу недопущения перегрузки, в которой размеры скользящего окна аддитивно увеличиваются (при поступлении ACK-пакета) и мультипликативно уменьшаются (при обнаружении перегрузки). TCP-отправитель осуществляет передачу данных на основе минимальных размеров скользящего окна и окна приема, чтобы избежать перегрузки сети или переполнения входного буфера, соответственно.

5.2.7. Описание TCP-заголовка

TCP-отправитель передает каждый сегмент в одном IP-пакете вместе с TCP-заголовком. Как показано на рис. 5.8, TCP-заголовок состоит из следующих полей.

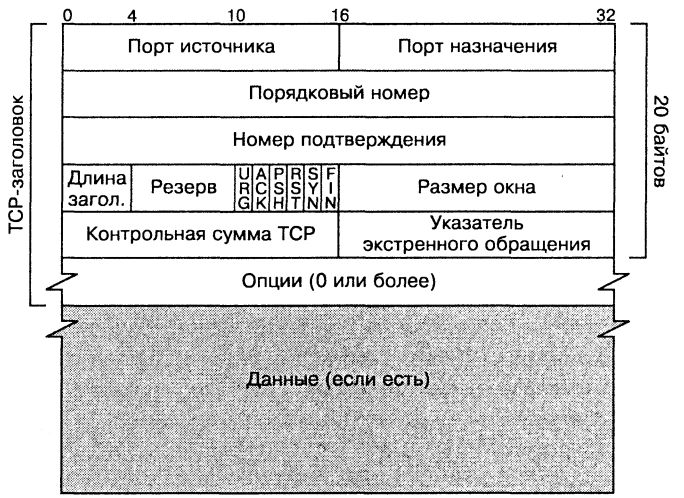


Рис. 5.8. Формат TCP-сегмента

- **Номер порта источника (source port) (16 бит).** 16-битный номер порта, ассоциированный с TCP-отправителем. IP-адрес отправителя содержится в IP-заголовке.
- **Номер порта получателя (destination port) (16 бит).** 16-битный номер порта, ассоциированный с TCP-получателем. IP-адрес получателя содержится в IP-заголовке.
- **Порядковый номер (sequence number) (32 бита).** 32-битный порядковый номер идентифицирует позицию первого байта сегмента, содержащегося в пакете. Получатель использует порядковый номер для идентификации местоположения сегмента в потоке байтов и реорганизации сегментов, поступающих не в том порядке. Перед передачей первого сегмента отправитель выбирает *начальный порядковый номер*, который представляет начало упорядоченного байтового потока. Порядковые номера для всех сегментов строятся относительно этого начального порядкового номера.

- **Номер подтверждения (acknowledgement number) (32 бита).** Для подтверждения получения данных TCP-заголовок содержит 32-битный номер подтверждения. Это поле указывает на следующий байт, который ожидается получателем, и корректно только в том случае, если флаг ACK установлен в 1. Когда приложение A передает данные приложению B, TCP-заголовок содержит порядковый номер каждого сегмента, и пакеты от B к A подтверждают получение этих сегментов.
- **Длина заголовка (header length) (4 бита).** 4-битная длина заголовка указывает на количество 32-битных слов в TCP-заголовке. Заголовок обычно имеет длину 20 байтов, что соответствует пяти 32-битным словам. Более длинным заголовок может быть в случае использования отправителем *опций TCP*, которые содержат дополнительную управляющую информацию.
- **Зарезервировано (6 бит).** 6-битное поле зарезервировано для будущего использования.
- **Флаги TCP (TCP flags) (8 битов).** TCP-заголовок также содержит 8-битное поле с шестью 1-битными флагами. Эти флаги соответствуют различным управляющим действиям:
  - **URG.** Флаг URG (флаг срочности) инструктирует TCP-получателя обратиться к части сегмента, идентифицируемой 16-битным полем указателя срочности в TCP-заголовке.
  - **ACK.** Флаг ACK (подтверждения) устанавливается при отправке подтверждения. Если флаг ACK установлен, 32-битное поле подтверждения указывает, сколько данных было получено отправителем. В начале передачи данных TCP-отправитель почти всегда устанавливает бит ACK.
  - **PSH.** Флаг PSH (форсированной отправки) указывает, что TCP-получатель должен немедленно передать входящие данные сокету приложения.
  - **RST.** Флаг RST (сброс) устанавливается при разрыве TCP-соединения.
  - **SYN.** Флаг SYN (синхронизации) устанавливается при установлении TCP-соединения. Если флаг SYN установлен, значение в поле порядкового номера идентифицирует начальный порядковый номер.
  - **FIN.** Флаг FIN (окончания передачи) устанавливается, когда отправитель закончил передачу данных. При чтении принимающим приложением из сокета FIN преобразуется в символ конца файла.

  Флаги SYN, ACK, FIN и RST будут подробнее обсуждаться далее в этом разделе. На практике большинство операционных систем не предоставляет для отправляющего приложения возможность установки флага PSH, а для TCP-получателя — возможность реакции на этот флаг. Флаг URG применяется для уведомления интерактивных приложений, таких как Telnet, о наличии управляющих символов (например, ctrl-C), которые могут повлиять на обработку предыдущих байтов в потоке.
- **Окно приема (receiver windows) (16 битов).** 16-битное поле окна приема содержит число дополнительных байтов, которые получатель может принять, помимо подтвержденных на данный момент данных. Чтобы избежать переполнения входного буфера, TCP-отправитель не должен передавать данные, объем которых превышает размер окна приема.
- **Контрольная сумма TCP (TCP checksum) (16 битов).** 16-битная контрольная сумма помогает TCP-получателю обнаруживать поврежденные пакеты.

В отличие от контрольной суммы IP-заголовка, контрольная сумма TCP распространяется и на заголовок, и на данные. Контрольная сумма дает возможность получателю выявить, был ли TCP-сегмент поврежден во время передачи по сети. Отправитель вычисляет контрольную сумму массива из заголовка и данных. Получатель пересчитывает контрольную сумму и сравнивает ее со значением в заголовке TCP. Если результаты различаются, получатель отвергает поврежденный пакет. Получатель не подтверждает получение поврежденного пакета. Следовательно, отправитель должен повторно передать не поступившие данные.

- **Указатель срочных данных (urgent pointer) (16 битов).** Если флаг URG установлен, 16-битный указатель срочных данных обращает внимание получателя на определенную часть входных данных (например, символ ctrl-C, который прерывает передачу данных). 16-битный указатель срочных данных идентифицирует последний байт срочных данных как целочисленное смещение от порядкового номера в TCP-заголовке.

IP-заголовок содержит некоторую информацию, необходимую для TCP-соединения, включая IP-адреса отправителя и получателя, а также размер пакета. Размер IP-пакета представляет собой сумму длин IP-заголовка, TCP-заголовка и TCP-сегмента. Длина IP-заголовка включается в IP-заголовок, а длина TCP-заголовка включается в TCP-заголовок.

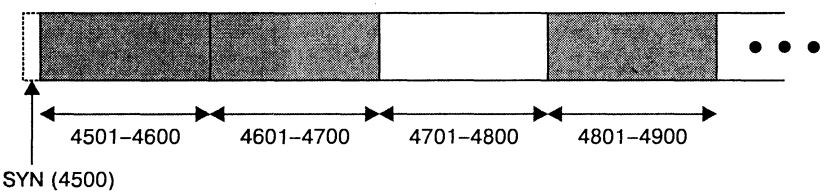


Рис. 5.9. Четыре сегмента в TCP-соединении (третий сегмент утерян)

При поступлении пакета получатель определяет диапазон байтов, занимаемый TCP-сегментом, основываясь на его порядковом номере и длине. Предположим, что отправитель начал с исходного порядкового номера 4500 и передает данные 100-байтными сегментами, как показано на рис. 5.9. Порядковый номер 4500 соответствует открытию TCP-соединения. Первый сегмент имеет порядковый номер 4501 и длину 100, и занимает байты с 4501 по 4600. Второй сегмент имеет порядковый номер 4601 и длину 100, и т.д. Допустим, что B получает первый, второй и четвертый сегменты; третий 100-байтный сегмент был утерян или задержан. Поскольку B было получено только 200 последовательных байтов потока, подтверждающий пакет от B к A будет иметь номер подтверждения, равный 4701 (4501+200). Номер байта 4701 представляет собой порядковый номер следующего байта, который B ожидает получить. Важно отметить, что поле порядкового номера соответствует передаче сегмента, а поле номера подтверждения соответствует получению данных. Приложения A и B могут передавать данные друг другу одновременно. Для любого пакета значения полей порядкового номера и номера подтверждения не взаимосвязаны, поскольку они не относятся к одному и тому же направлению передачи.

5.3. Служба именования доменов (DNS)

Служба именования доменов (DNS — Domain Name System) координирует преобразование доменных имен хостов в IP-адреса и IP-адресов в доменные имена [Moc87b, Pos94]. В этом разделе будет рассмотрено, как приложение взаимодействует с DNS-преобразователем, который выдает запросы локальному DNS-серверу. Далее будет рассмотрена иерархическая и распределенная структура DNS, после чего мы рассмотрим протокол DNS. Далее мы посмотрим, как Web-клиенты, прокси-серверы и Web-серверы выдают DNS-запросы, и как DNS используется для выравнивания нагрузки между несколькими репликами Web-сайта.

5.3.1. DNS-преобразователь

IP-маршрутизаторы пересылают пакеты на основе 32-битного адреса места назначения в IP-заголовках. Однако цифровые адреса неудобны для пользователей и приложений. Компьютеры в Internet обычно имеют доменные имена, например, **ftp.foo.com** или **cs.berkeley.edu**, которые состоят из строк, разделяемых точками. Запомнить доменное имя гораздо проще, чем IP-адрес. Кроме того, IP-адрес, ассоциированный с доменным именем, может со временем меняться. Например, IP-адрес Web-сайта может зависеть от того, какая компания осуществляет хостинг. Если Web-сайт передается другой хостинговой компании, IP-адрес может измениться. Если бы URL Web-сайта включал IP-адрес (например, **http://10.187.56.3/ bar.html**), а не доменное имя (например, **http://www.foo.com/ bar.html**), URL изменялся бы каждый раз при изменении IP-адреса. Любые попытки использовать старый URL (например, из списка закладок браузера) оканчивались бы неудачей. Наконец, один и тот же Web-сайт может быть доступен на нескольких хостах, каждый из которых имеет собственный IP-адрес. Идентификация сайта по доменному имени обеспечивает гибкость при принятии решения, с каким сервером контактировать. Помимо преобразования доменных имен в IP-адреса, приложению может потребоваться установить доменное имя, ассоциированное с определенным IP-адресом. Например, FTP-сервер может проверять, принадлежит ли FTP-клиент домену верхнего уровня **.edu**.

Преобразование доменных имен в IP-адреса и IP-адресов в доменные имена осуществляется DNS. Internet-приложения, например, браузеры, осуществляют доступ к DNS через *преобразователь (resolver)*, представляющий собой программную библиотеку, которая связана с приложением. DNS-преобразователь выполняет две основные функции. Функция *gethostbyname()* преобразовывает доменное имя в IP-адрес, а функция *gethostbyaddr()* преобразовывает IP-адрес в доменное имя. Преобразователь взаимодействует с одним или с несколькими DNS-серверами для выполнения этих функций от имени приложения. Чтобы осуществить этот процесс, преобразователь должен знать, как связаться по крайней мере с одним DNS-сервером. Для компьютера, подключенного к Internet, обычно указывается список IP-адресов, каждый из которых соответствует локальному DNS-серверу. Сетевые администраторы обычно стараются размещать локальные DNS-серверы ближе к клиентам, обращающимся с запросами. Например, факультет университета может иметь сеть Ethernet, объединяющую все компьютеры, в составе которой имеются локальные DNS-серверы. Провайдер может размещать свои DNS-серверы вблизи от модемного пула, через который пользователи взаимодействуют с сетью.

Итак, предположим, что пользователь вводит в Web-браузере URL **http://www.foo.com/a.html**. Программное обеспечение браузера извлекает имя до-

мена **www.foo.com** из URL. Чтобы связаться с Web-сервером, браузер должен перевести **www.foo.com** в IP-адрес. Программное обеспечение браузера вызывает для этого функцию *gethostbyname()*, которая связывается с одним из локальных DNS-серверов. Ответ DNS-сервера содержит IP-адрес Web-сервера, который возвращается функцией *gethostbyname()*. После этого браузер может инициировать взаимодействие с **www.foo.com**. В некоторых случаях приложение вызывает функцию *gethostbyname()* с *полностью заданным именем домена*, которое идентифицирует полное доменное имя, такое как **zippy.bar.com**. В других случаях приложение обращается к хосту по усеченному имени, например, **zippy**. При настройке клиентского компьютера указывается имя домена по умолчанию, используемое для DNS-запросов. Так, предположим, что клиентский компьютер настроен для использования **bar.com** в качестве имени домена по умолчанию. Если пользователь вводит URL **http://zippy** в Web-браузере, функция *gethostbyname()* запрашивает DNS-сервер для определения IP-адреса, ассоциированного с полным именем домена **zippy.bar.com**.

Практически каждое Internet-приложение вызывает функцию *gethostbyname()* перед началом взаимодействия с другим компьютером. Например, при отправке сообщения электронной почты по адресу **buddy@zippy.bar.com** или загрузке файла с **ftp.foo.com** требуется, чтобы приложение знало IP-адрес удаленного компьютера. Приложению, выполняющемуся на удаленном компьютере, может потребоваться отправить ответ. Для этого приложение-получатель должно знать IP-адрес отправителя. IP-адрес отправителя доступен из заголовка IP-пакета. Таким образом, удаленному компьютеру не нужно связываться с DNS-сервером для ответа отправителю. Однако приложению-получателю не известно доменное имя компьютера, отправившего сообщение. Знать доменное имя полезно, если удаленное приложение выполняет регистрацию или аутентификацию. Например, предположим, что администраторы **ftp.bar.com** не хотят предоставлять доступ пользователям с **zippy.bar.com** по Telnet. Для этого Telnet-приложению на **ftp.bar.com** понадобится определить доменное имя компьютера, запрашивающего соединение, перед тем, как разрешить доступ. Telnet-приложение должно преобразовать IP-адрес отправителя в доменное имя, воспользовавшись функцией *gethostbyaddr()*.

5.3.2. Архитектура DNS

На первых порах существования ARPANET преобразование доменных имен в IP-адреса выполнялось централизованно. Один главный файл содержал список IP-адресов, ассоциированных с доменными именами. Файл обновлялся несколько раз в неделю. Каждая организация, подсоединенная к ARPANET, была ответственна за обновление своей локальной копии этого файла. Подобный подход годился, пока сеть была небольшой. Когда сеть разрослась, обслуживание копий главного файла стало вызывать затруднения. Для решения этой проблемы была создана служба DNS, представляющая собой распределенную базу данных с иерархической системой серверов, каждый из которых ответственен за группу доменных имен и IP-адресов. Архитектура DNS отражает иерархию доменных имен и IP-адресов. Преобразование доменных имен в IP-адреса осуществляется на основе составляющих доменных имен. Аналогично, преобразование IP-адресов в доменные имена осуществляется на основе чисел в IP-адресе.

DNS имеет иерархическое пространство имен с безымянной корневой частью, как показано на рис. 5.10. Первый уровень древовидной структуры содержит домены верхнего уровня. Имена доменов верхнего уровня отражают историю эволюции

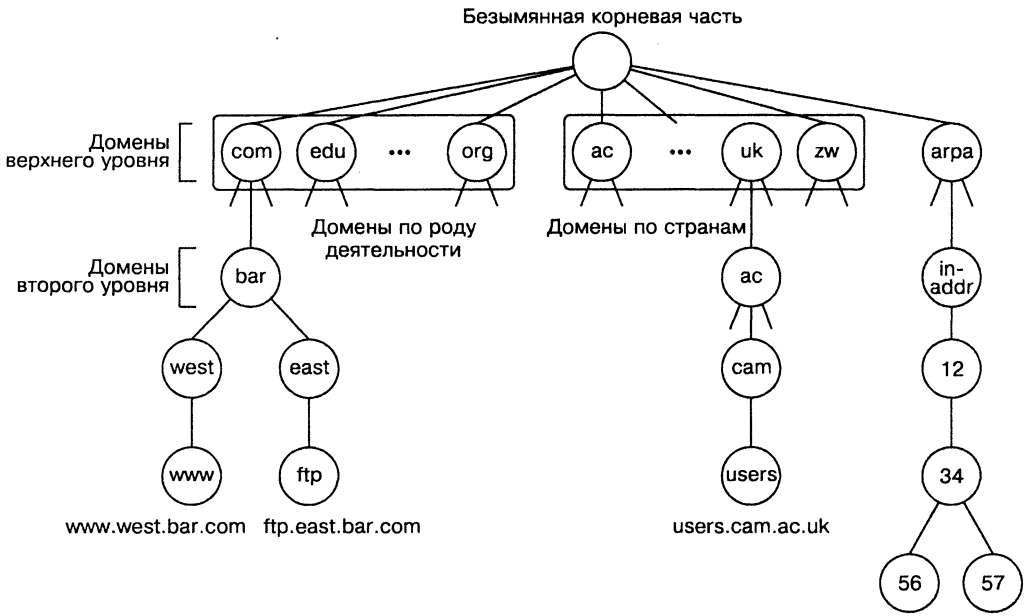


Рис. 5.10. Иерархия DNS

Internet из сети ARPANET, созданной в США. Верхний уровень включает трехсимвольные родовые домены:

- коммерческие организации (**.com**),
- образовательные учреждения (**.edu**),
- правительственные учреждения США (**.gov**),
- военные учреждения США (**.mil**),
- сети (**.net**),
- другие организации (**.org**).

Хотя имена **.edu**, **.gov** и **.mil** зарезервированы для учреждений США, остальные домены верхнего уровня используются за пределами США. В ноябре 2000 г. были введены семь новых доменов верхнего уровня:

- отрасли, связанные с воздушным транспортом (**.aero**),
- организации, связанные с бизнесом (**.biz**),
- некоммерческие организации (**.coop**),
- для неограниченного использования (**.info**),
- музеи (**.museum**),
- отдельные личности (**.name**),
- бухгалтеры, юристы и врачи (**.pro**).

К доменам верхнего уровня также относятся двухсимвольные домены, соответствующие *странам* от острова Вознесения (**.ac**) до Зимбабве (**.zw**). Отдельный домен **.arpa** управляет преобразованием IP-адресов в имена доменов. Этот процесс будет более подробно описан далее в данном разделе.

До конца 90-х годов за управление DNS и присвоение IP-адресов и имен доменов отвечали организации, контролируемые правительством США. В конце 90-х годов эти функции были переданы Корпорации по присвоению имен и номеров (ICANN — Internet Corporation for Assigned Names and Numbers) [ICA]. Домены верхнего уровня обслуживаются группой корневых серверов. Предположим, Web-пользователь вводит URL **http://www.east.bar.com/a.html** в браузере. Преобразователь связывается с локальным сервером имен для преобразования **www.east.bar.com** в IP-адрес. Если информация не была кэширована, локальный сервер имен связывается с корневым сервером, чтобы определить IP-адрес DNS-сервера **.com**. Затем локальный DNS-сервер связывается с DNS-сервером **.com**, чтобы узнать IP-адрес DNS-сервера, ответственного за *зону* **bar.com**. Каждая зона представляет собой часть дерева в иерархии DNS, администрируемую отдельно. Зона обычно имеет первичный сервер и несколько вторичных серверов, которые могут замещать первичный в случае аварии. В зависимости от размера организации домен может быть поделен на множество зон в соответствии, например, с различным географическим положением.

Первоначальный подход, при котором все доменные имена и IP-адреса содержались в одном файле, не мог быть использован для обслуживания множества доменных имен, которое имеется сегодня в Internet. Разделение административной ответственности имело важное значение для поддержки быстрого роста Internet. Организации не нужно уведомлять центральную базу данных при добавлении или удалении хостов в своей сети. Аналогично, организация bar.com может принять решение иметь две отдельные зоны для **east.bar.com** и **west.bar.com**. Компания может иметь два главных офиса в США, один на восточном побережье, а другой — на западном. Деление **bar.com** на две зоны дает возможность каждому офису обновлять соответствия между доменными именами и IP-адресами независимо друг с другом. Хостам же надо лишь знать IP-адрес одного или нескольких DNS-серверов. Им не нужно располагать информацией о добавлении или удалении других хостов. Организации нужно обновлять данные в своей родительской зоне только в том случае, если добавляется или удаляется DNS-сервер. Это случается не слишком часто.

Иерархия имен доменов не соответствует иерархической структуре IP-адресов. Хотя и **www.east.bar.com**, и **www.west.bar.com** имеют в составе имени **bar.com**, их IP-адреса могут быть совершенно различными. Для эффективного установления соответствия IP-адресов и доменных имен требуется *отдельная* иерархия на основе IP-адресов. В начале двадцать первого века выделение IP-адресов осуществляется тремя региональными Internet-регистрами: APNIC (Asia-Pacific Network Information Centre), ARIN (American Registry for Internet Numbers) и RIPE NCC (Reseaux IP Europeens Network Coordination Centre). ICANN распределяет фрагменты пространства IP-адресов между этими организациями, которые, в свою очередь, выделяют адреса организациям внутри их регионов. Выделение фиксированного набора адресов каждому из регионов помогает осуществлять эффективное выделение оставшихся адресов IPv4. Кроме того, выделение больших блоков IP-адресов по географическому принципу облегчает маршрутизацию. В идеале маршрутизатору в одной части земного шара не нужно знать, как достичь каждой из организаций в другой части земного шара.

При получении блока IP-адресов организация становится ответственной за часть пространства имен **in-addr.arpa**. Пространство имен **in-addr.arpa** представляет собой иерархию, основанную на октетах в 32-битном IP-адресе, начиная с верхнего уровня иерархии адресации — крайнего левого октета в адресе. Например,

чтобы найти доменное имя, ассоциированное с IP-адресом 12.34.56.78, необходимо связаться с сервером **12.in-addr.arpa**. Этот домен может быть поделен на составляющие домены, одним из которых является домен **34.12.in-addr.arpa**. Такое разделение ответственности по октетам отражает исторически сложившуюся практику выделения IP-адресов с фиксированной длиной префикса, равной 8, 16 и 24. В соответствии с положениями CIDR, организация может иметь блок IP-адресов, который не совпадает с границами октетов. Так, предположим, что адресный блок 12.34.56.0/23 выделен одной компании. При этом DNS-сервер, ответственный за домен **34.12.in-addr.arpa**, может выделить этой компании как адрес **56.34.12.in-addr.arpa**, так и адрес **57.34.12.in-addr.arpa**. Компания будет отвечать за связь отдельных IP-адресов в пространстве 12.34.56.0/23 с доменными именами.

5.3.3. Протокол DNS

Протокол DNS управляет взаимодействием между DNS-клиентом и DNS-сервером. DNS-клиент посылает *запрос* (например, на IP-адрес, ассоциированный с определенным доменным именем) DNS-серверу, а DNS-сервер возвращает *ответ* с запрошенной информацией (например, IP-адрес). Локальный DNS-сервер посылает ответы преобразователям и выдает запросы другим DNS-серверам. Корневые DNS-серверы посылают ответы на запросы других DNS-серверов, а сами не выдают запросы. Предположим, что приложение вызывает функцию *gethostbyname()* для определения IP-адреса для **www.foo.bar.com**. Преобразователь связывается с локальным DNS-сервером, который обращается к корневому DNS-серверу с целью узнать IP-адрес DNS-сервера **.com**. Затем локальный DNS-сервер посылает запрос DNS-серверу **.com**, чтобы узнать IP-адрес DNS-сервера **bar.com**, после этого локальный DNS-сервер посылает запрос DNS-серверу зоны **bar.com**. Если зона имеет подзоны, то может быть выдан дополнительный запрос домену **foo.bar.com**, который отвечает IP-адресом для **www.foo.bar.com**. Аналогично, установление связи IP-адреса 12.34.56.78 с доменным именем сопровождается серией DNS-запросов к различным DNS-серверам в иерархии **in-arpa**.

DNS-запрос может быть *рекурсивным* или *итеративным*. Рекурсивный запрос требует, чтобы DNS-сервер, принимающий запрос, сам осуществил преобразование. Например, преобразователь выдает рекурсивный запрос локальному серверу имен на преобразование доменного имени в IP-адрес. Как показано на рис. 5.11, на этапе 1 преобразователь активизируется посредством системного вызова из приложения. Затем преобразователь направляет DNS-запрос локальному DNS-серверу (этап 2) и ожидает ответа (этап 9). Локальный DNS-сервер осуществляет действия по обработке запроса преобразователя. Итеративный запрос требует, чтобы принимающий DNS-сервер напрямую ответил DNS-клиенту IP-адресом следующего DNS-сервера в иерархии DNS. Корневые серверы обслуживают только итеративные запросы. Локальный DNS-сервер посылает запрос корневому DNS-серверу (этап 3), чтобы узнать имена и IP-адреса DNS-сервера (серверов) для зоны на следующем уровне иерархии (этап 4). Это помогает разгрузить корневые серверы от выполнения полного процесса преобразования. Затем локальный DNS-сервер может отправить запрос следующему DNS-серверу в цепочке (этапы 5, 6, 7 и 8). Наконец, локальный DNS-сервер отвечает преобразователю (этап 9), а преобразователь предоставляет IP-адрес приложению (этап 10).

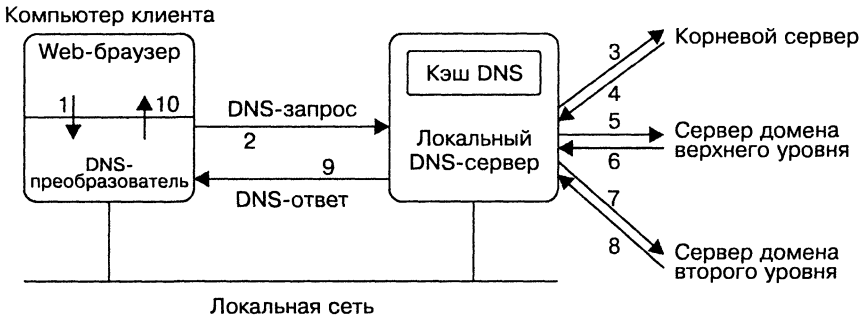


Рис. 5.11. DNS-преобразователь и локальный DNS-сервер

DNS-серверы используют кэширование для уменьшения времени ожидания при ответах на запросы и уменьшения DNS-трафика в Internet [DOK92]. Локальный DNS-сервер имеет кэш, который хранит ответы, посылаемые преобразователям клиентов. Кроме того, кэш может хранить IP-адреса каждого из DNS-серверов, участвующих в обработке запроса. Для будущих запросов локальный DNS-сервер может избежать взаимодействия с корневым сервером, обратившись вместо этого к DNS-серверам первого или второго уровней напрямую, на основе кэшированной информации. Например, при преобразовании **www.foo.com** локальный DNS-сервер может определить IP-адрес DNS-сервера верхнего уровня **.com** и DNS-сервера второго уровня **foo.com** напрямую. Преобразователь не выполняет кэширование, поскольку информация существует только во время работы приложения и не может быть использована совместно другими приложениями, выполняющимися на том же компьютере, или другими компьютерами в этой же сети.

DNS-сервер кэширует ответы на запросы, основываясь на поле TTL. Каждый DNS-ответ содержит TTL, указывающий число секунд, в течение которых ответ может быть кэширован. DNS-кэширование значительно сокращает время на преобразование доменных имен в IP-адреса. Первый запрос на **www.foo.com** может сопровождаться большим временем ожидания на соединение с корневым сервером и сервером зоны, но последующие запросы будут обрабатываться непосредственно локальным DNS-сервером. После истечения TTL информация удаляется из кэша и вновь сохраняется в нем при получении другого запроса на преобразование **www.foo.com** в IP-адрес. DNS-серверы также кэшируют негативную информацию о неудачных запросах. Так, предположим, что пользователь сделал ошибку и ввел URL **http://www.boo.com** вместо **http://www.foo.com**. Если **www.boo.com** не существует, локальный DNS-сервер получает негативный ответ на свой запрос и возвращает сообщение об ошибке преобразователю. Локальный DNS-сервер помнит, что запрос на **www.boo.com** был безуспешным, поэтому он может в дальнейшем быстрее среагировать на ошибку.

DNS в основном использует протокол UDP для передачи запросов и получения ответов, хотя может быть также использован протокол TCP. Применение UDP дает возможность преобразователю и DNS-серверам взаимодействовать с помощью однопакетных сообщений без необходимости тратить ресурсы на установку соединений. Это очень полезно, поскольку большинство DNS-запросов и ответов являются очень короткими. Однако UDP-пакет может быть утерян в сети. Отправитель DNS-запроса повторно передает запрос, если ответ не поступил по истечении некоторого времени. Если отсчет времени таймером завершается до получения ответа, запрос выдается снова. TCP используется вместо UDP для более длинных посы-

лок. Например, организация может иметь вторичный DNS-сервер, который используется в качестве резервного в случае отказа первичного сервера. Вторичный DNS-сервер должен иметь копию информации, доступной на первичном сервере. TCP обычно применяется для копирования этих данных с первичного сервера на вторичный сервер.

5.3.4. DNS-запросы и Web

Web-клиент вызывает *gethostbyname()*, чтобы преобразовать доменное имя в IP-адрес до установления соединения транспортного уровня с Web-сервером. Например, предположим, что пользователь вводит URL **http://www.foo.com/a.html** в браузере. Если ресурс отсутствует в кэше браузера, браузер должен связаться с Web-сервером. В некоторых случаях клиенту не требуется обращаться к DNS для преобразования **www.foo.com**:

- **Запрос, направляемый прокси-серверу.** Клиент может быть настроен для отправки запросов прокси-серверу. Установка TCP-соединения с прокси-сервером требует, чтобы клиенту был известен IP-адрес прокси-сервера, а не исходного сервера. Если клиент настроен с указанием доменного имени прокси-сервера, то вызов *gethostbyname()* может стать необходимым для определения IP-адреса прокси-сервера. Клиенту, который сконфигурирован с указанием IP-адреса прокси-сервера, нет необходимости инициировать вызов *gethostbyname()*.
- **Запрос, удовлетворяемый из кэша клиента.** Перед выдачей HTTP-запроса клиент осуществляет поиск Web-ресурса в своем локальном кэше. Если в кэше имеется актуальная копия запрашиваемого ресурса, клиенту нет нужды связываться с Web-сервером. В противном случае клиент должен связаться с Web-сервером для проверки актуальности кэшированного ресурса и, возможно, загрузки нового ресурса с сервера.
- **Использование результата предыдущего запроса.** Клиент может загрузить несколько ресурсов с одного и того же сервера. При обработке запроса на ресурс **http://www.foo.com/a.html** клиент определяет IP-адрес для **www.foo.com**. Если Web-страница имеет встроенные изображения, клиент может установить дополнительные TCP-соединения с сервером для загрузки этих ресурсов. Чтобы избежать повторных вызовов *gethostbyname()*, клиент может воспользоваться IP-адресом, полученным при предыдущем вызове. Сохранение результатов вызова *gethostbyname()* отличается от DNS-кэширования на локальном DNS-сервере, поскольку вызов *gethostbyname()* не возвращает TTL, ассоциированный с ответом на DNS-запрос. Чтобы уменьшить риск использования устаревшей информации, Web-клиенту не следует хранить IP-адрес для **www.foo.com** слишком долго (т.е. не более нескольких минут).

Web-клиент должен выяснить IP-адрес Web-сервера, Web-серверу IP-адрес клиента становится известным при поступлении запроса, поскольку клиентский IP-адрес содержится в заголовке каждого IP-пакета. Сервер может узнать доменное имя, ассоциированное с этим IP-адресом, вызвав функцию *gethostbyaddr()*. Знание доменного имени клиента может быть полезно для различных применений, в том числе регистрации, аутентификации и целевой рекламы. Например, запись доменного имени в журнал сервера дает возможность администратору сайта определить, какие запросы поступили от пользователей из определенных организаций. Web-сервер может также ограничивать доступ к определенным ресурсам, основываясь на доменном

имени обратившегося с запросом клиента. Или же сервер может настроить HTTP-ответ на основе доменного имени клиента. Наиболее типичный пример такой настройки — целевая реклама, при которой встроенные изображения на Web-странице выбираются на основе доменного имени клиента, обратившегося с запросом.

Установление соответствия между IP-адресом Web-клиента и доменным именем осуществляется DNS-сервером в сети Web-клиента. У Web-клиента может возникнуть соблазн неправильно указать свое доменное имя. Например, предположим, что Web-клиент имеет IP-адрес 10.34.56.78 и доменное имя **a.badclient.com**. В то же время на локальном DNS-сервере Web-клиента IP-адресу 10.34.56.78 соответствует доменное имя **b.goodclient.edu**. Предоставление неверного доменного имени может ввести в заблуждение Web-сервер, и он предоставит доступ не тому клиенту. В качестве меры предосторожности Web-сервер может после вызова *gethostbyaddr()* вызвать *gethostbyname()*. Вызов *gethostbyname()* определяет IP-адрес, ассоциированный с **b.goodclient.edu**. В данном примере соответствие может контролироваться другой организацией, что приведет к возврату другого IP-адреса, скажем, 122.33.205.4. Узнав, что два IP-адреса отличаются друг от друга, Web-сервер может отклонить HTTP-запрос.

Определение по IP-адресу клиента соответствующего доменного имени часто сопровождается значительной задержкой. Вызов функции *gethostbyaddr()* приводит к запросу к локальному DNS-серверу Web-сервера. Этот DNS-сервер вряд ли имеет информацию об IP-адресе клиента в своем кэше, если только клиент недавно не посылал Web-запрос этому же Web-серверу. Следовательно, должен быть выдан запрос к корневому серверу и к различным зональным серверам в домене **in-addr.arpa**. Запрос может оказаться неудачным. Многие клиенты имеют динамически выделяемые IP-адреса или IP-адреса, располагающиеся за корпоративным сетевым экраном. Эти адреса обычно не ассоциируются с доменными именами, подобные запросы приводят к задержкам в обходе DNS-иерархии и не возвращают полезной информации. Даже если запрос был успешным, выполнение последующего вызова *gethostbyname()* для проверки корректности соответствия IP-адреса доменному имени приведет к увеличению времени ожидания.

Помимо прочего, DNS-запросы потребляют ресурсы Web-сервера. Процесс, вызывающий функцию *gethostbyaddr()*, должен ожидать, пока будет сформирован ответ. Это увеличивает продолжительность времени обработки HTTP-запроса, что, в свою очередь, увеличивает число процессов, выполняющихся на сервере. Чтобы избежать перегрузки, Web-сервер, одновременно обрабатывающий большое количество запросов, может пропускать этап преобразования IP-адреса клиента в доменное имя. В зависимости от конкретного Web-сайта, знание доменного имени обратившегося с запросом клиента может быть и необязательным для корректной работы сервера. Однако в некоторых случаях Web-сервер может осуществлять настройку или аутентификацию на основе доменного имени клиента. Для достижения максимальной эффективности сервер может быть настроен таким образом, чтобы ограничить применение функции *gethostbyaddr()* только теми Web-запросами, которые требуют эту информацию.

5.3.5. Выравнивание нагрузки на Web-серверы с помощью DNS

DNS также играет важную роль в перенаправлении HTTP-запросов в группе Web-серверов, которые предоставляют доступ к одному и тому же содержанию. Хотя доменное имя обычно соответствует одному IP-адресу, достаточно активно

используемый Web-сайт может быть реплицирован на несколько компьютеров, имеющих одно и то же доменное имя, о чем говорилось в главе 4 (раздел 4.5.2). Реплики необходимы, чтобы иметь возможность обрабатывать больше запросов. Кроме того, реплики могут быть размещены в различных географических зонах, чтобы предоставить лучший сервис различным клиентам. Например, доменному имени **www.big.com** могут соответствовать четыре компьютера с IP-адресами 10.198.3.47, 10.198.3.48, 10.34.99.1 и 10.34.99.2. Чтобы поддерживать доменные имена с несколькими IP-адресами при ответе на запрос IP-адреса для **www.big.com** локальный DNS-сервер должен знать все четыре IP-адреса. Локальный DNS-сервер выбирает один из этих IP-адресов, чтобы вернуть его преобразователю, сделавшему запрос к DNS-серверу.

В традиционной реализации DNS локальный DNS-сервер будет выбирать первый IP-адрес в списке. Это приведет к большей загруженности HTTP-запросами компьютера с IP-адресом 10.198.3.47. Чтобы избежать этого, DNS-сервер, ответственный за **www.big.com**, может варьировать порядок IP-адресов для каждого запроса. Например, ответ на первый DNS-запрос вернет 10.198.3.47, 10.198.3.48, 10.34.99.1 и 10.34.99.2, в то время как следующий ответ может вернуть 10.198.3.48, 10.34.99.1, 10.34.99.2 и 10.198.3.47. Подобная *карусельная* последовательность помогает распределить нагрузку между репликами Web-содержимого. Однако подобный подход не учитывает текущей нагрузки каждого из компьютеров и сети. Кроме того, он не учитывает расстояний между клиентским компьютером и каждой из реплик. Расширения реализаций DNS-серверов позволяют устранить эти недостатки без внесения изменений в протокол DNS. Эти изменения могут быть применены к DNS-серверу, получающему запрос, или к DNS-серверу, посылающему ответ.

Сначала рассмотрим изменения в программном обеспечении DNS-сервера, *выдающего* запрос. Предположим, что локальный DNS-сервер получает запрос от преобразователя на определение IP-адреса для **www.big.com**. DNS-сервер, ответственный за **www.big.com**, предоставляет список из четырех IP-адресов. Вместо того чтобы вернуть первый IP-адрес преобразователю, обратившемуся с запросом, локальный DNS-сервер может определить, какая из реплик подходит наилучшим образом. Этот процесс может основываться на различных факторах, таких как время передачи в прямом и обратном направлении (RTT) при взаимодействии с компьютером сервера или число маршрутизаторов на пути к серверу. Локальный DNS-сервер может время от времени измерять эти характеристики, чтобы уточнить оценки. Затем преобразователю предоставляется IP-адрес «наилучшей» реплики. Подобный подход не требует внесения изменений в программный код преобразователя или в программное обеспечение, выполняющееся на DNS-сервере, ответственном за **www.big.com**. Нуждается в изменении лишь программное обеспечение локального DNS-сервера. Однако выполнение оценок текущих характеристик времени передачи в обоих направлениях (RTT) обусловливает дополнительную загрузку локального DNS-сервера и сети.

Далее рассмотрим изменения в программном обеспечении DNS-сервера, отвечающего за ответ на запрос. DNS-сервер, ответственный за **www.big.com**, может попытаться выбрать «наилучший» из IP-адресов. Этот DNS-сервер может иметь доступ к актуальной информации о текущей загрузке каждого из четырех компьютеров **www.big.com**. Кроме того, DNS-сервер, ответственный за **www.big.com**, может попытаться оценить, какая из реплик ближе всего к локальному DNS-серверу, выдавшему запрос. DNS-сервер для **www.big.com** не знает IP-адрес Web-клиента, который отправил запрос локальному DNS-серверу. При выборе наилучшей реплики Web-сайта DNS-сервер для **www.big.com** может предположить, что Web-

клиент находится вблизи локального DNS-сервера. Близость Web-клиента сокращает время ожидания при обработке Web-запросов и снижает объем сетевых ресурсов, требующихся для доставки IP-пакетов. Решение, какой IP-адрес возвращать, может основываться на различных факторах, в числе которых близость к клиенту, нагрузка на сеть и нагрузка на реплики сервера.

Как правило, ответ на DNS-запрос может кэшироваться достаточно продолжительное время в зависимости от значения поля TTL. Значения TTL может достигать дней или даже недель. Однако большие значения TTL не дают возможности DNS-серверу **big.com** контролировать, к каким репликам **www.big.com** осуществляется доступ. Чтобы добиться большей степени контроля, DNS-ответы должны использовать меньшие значения TTL, порядка минут. Это гарантирует, что локальный DNS-сервер не будет кэшировать ответ слишком долго. По истечении времени хранения записи в кэше локальный DNS-сервер не сможет обработать следующий запрос от преобразователя, не связываясь с DNS-сервером **big.com** повторно. Это дает DNS-серверу возможность отвечать различными IP-адресами для **www.big.com**. Однако это также требует, чтобы Web-браузер дольше ожидал завершения вызова *gethostbyname()* при истечении времени хранения кэшированной информации на локальном DNS-сервере. Помимо этого, небольшие значения TTL ведут к более высокой загрузке системы в целом. Инфраструктура DNS не была предназначена для поддержки выравнивания нагрузки на серверы. Рост Web и уменьшение значений TTL поднял вопрос об изменении и расширении инфраструктуры DNS. Методы направления клиентских запросов репликам подробнее обсуждаются в главе 11 (раздел 11.12).

5.4. Протоколы прикладного уровня

Приложения выполняются на компьютерах в сети и взаимодействуют через протоколы прикладного уровня. Протоколы прикладного уровня определяют синтаксис и семантику обмена сообщениями между оконечными системами. Синтаксис определяет формат сообщений, тогда как семантика диктует, как приложения должны интерпретировать сообщения и отвечать отправителю. В этом разделе будут описаны протоколы, лежащие в основе четырех Internet-приложений: Telnet, передача файлов, электронная почта и группы новостей, которые были предшественниками Web. Мы обсудим, как особенности каждого из протоколов прикладного уровня соотносятся с назначением приложений, а также поговорим об их взаимодействии с транспортным уровнем.

5.4.1. Протокол Telnet

Протокол Telnet [PR93] дает возможность пользователю взаимодействовать с удаленным компьютером. Клиентская программа, выполняющаяся на компьютере пользователя, взаимодействует по протоколу Telnet с серверной программой, выполняющейся на удаленном компьютере.

ПРИМЕНЕНИЕ TELNET

На первых порах существования ARPANET главным назначением сети было дать возможность пользователю в одном месте получить доступ к компьютеру, находящемуся в другом месте. Пусть у пользователя есть учетная запись на локальном компьютере и учетная запись на удаленном компьютере. Пользователь запускает Telnet-кли-

ент на локальной компьютере для соединения с Telnet-сервером на удаленном компьютере. Предположим, что пользователь вводит "**telnet big.foo.gov**". Telnet-клиент выполняет вызов *gethostbyname()* для определения IP-адреса **big.foo.gov**. Затем клиент создает сокет для взаимодействия с Telnet-сервером. Сервер запрашивает у пользователя идентификатор для входа в систему — имя учетной записи пользователя на удаленной машине — и пароль. В большинстве случаев пользователи Telnet взаимодействуют с удаленным компьютером точно так же, как если бы они взаимодействовали с локальным компьютером. Клиент и сервер Telnet просто передают данные в прямом и обратном направлении.

ПРОТОКОЛ TELNET

Telnet-клиент выполняет две важные функции: взаимодействие с терминалом пользователя на локальном компьютере и обмен сообщениями с Telnet-сервером. По умолчанию Telnet-клиент соединяется с удаленным компьютером по порту 23, поскольку данный номер TCP-порта был зарезервирован для Telnet-серверов. TCP-соединение существует все время сеанса работы. Клиент и сервер поддерживают TCP-соединение, даже когда клиент прерывает передачу данных (например, нажимая ctrl-C или Delete). Для совместимости между различными платформами протокол Telnet предполагает, что оба компьютера поддерживают виртуальный сетевой терминал Network Virtual Terminal (NVT). NVT — это простое символьное устройство с клавиатурой и принтером — данные вводятся пользователем с клавиатуры и передаются серверу, данные, полученные от сервера, выводятся на принтер. NVT-терминалы двух компьютеров обмениваются данными в 7-битном варианте ASCII, в котором каждый символ посылается в виде октета со старшим битом, установленным в 0. Управляющая информация, такая как индикация конца строки, посылается в виде последовательности из двух октетов.

Каждое управляющее сообщение Telnet начинается со специального октета (8 битов, установленных в 1), чтобы гарантировать, что получатель интерпретирует последующие октеты как команду. В противном случае каждый октет интерпретируется как данные (например, введенный пользователем символ). Отправка управляющих сообщений в том же соединении называется *передачей команд управления через то же соединение, что и данные (in-band signaling)*. Для обмена информацией между клиентом и сервером об их возможностях используются начальные управляющие сообщения. Например, клиент может указать тип и скорость работы своего терминала, а также то, как будут передаваться данные: по одному символу или по строкам. После обмена информацией о возможностях сервер инструктирует клиента отправить имя учетной записи для входа в систему и пароль. После завершения аутентификации пользователь напрямую взаимодействует с удаленным компьютером. Клиентское приложение передает введенные пользователем символы серверу, а сервер передает выходные данные обратно клиенту. Однако клиент и сервер должны просматривать каждый полученный байт на предмет выявления команд (начинающихся с октета 11111111). Telnet-отправитель использует поле указателя срочных данных TCP для привлечения внимания получателя к командам Telnet.

5.4.2. File Transfer Protocol

FTP — протокол передачи файлов [PR85] дает возможность пользователю копировать файлы на удаленный компьютер и с него. Клиентская программа посылает команды серверной программе для координации копирования файлов между двумя компьютерами в интересах пользователя.

ПРИМЕНЕНИЕ FTP

FTP-клиент связывается с удаленным компьютером и предлагает пользователю ввести идентификатор для входа и пароль. Однако некоторые пользователи могут не иметь собственных учетных записей на удаленном компьютере. Чтобы предоставить доступ любым пользователям, многие FTP-серверы имеют специальную «анонимную» учетную запись, которая не требует, чтобы пользователь знал пароль. FTP-сервер координирует доступ к набору файлов в различных каталогах. FTP-сервер обычно имеет специальный каталог с одним или несколькими подкаталогами, которые могут быть доступны для FTP-клиентов. Многие FTP-клиенты имеют простой интерфейс командной строки. Пользователь может входить на FTP-сервер, просматривать каталоги, отправлять или получать файлы. Интерфейс может давать возможность отправлять или принимать *несколько* файлов с помощью одной команды. Последние версии FTP-клиентов предоставляют графический пользовательский интерфейс. Web-браузер дает возможность пользователям указывать желаемый файл в виде URL (например, **ftp://ftp.foo.com/bar/neatfile**). Браузер выполняет низкоуровневые задачи по соединению с FTP-сервером от лица анонимного пользователя для отправки последовательности FTP-команд и для загрузки запрошенного файла.

ПРОТОКОЛ FTP

FTP использует отдельные TCP-соединения для передачи команд управления и данных. По умолчанию FTP-клиент соединяется с портом 21[1] на удаленном компьютере. Клиент использует это соединение для отправки команд и получения ответов, при этом соединение продолжает существовать на протяжении выполнения нескольких команд. Спецификация FTP включает в себя более 30 различных команд, которые передаются по управляющему соединению в формате ASCII NVT. Команды нечувствительны к регистру и могут иметь аргументы; каждая команда заканчивается последовательностью из двух символов: возврат каретки (CR) и перевод строки (LF). Эти команды могут отличаться от команд, вводимых пользователем в FTP-клиенте. Передача одного файла осуществляется одной командой пользовательского уровня, которая предписывает клиенту отправить последовательность команд FTP-серверу. FTP-сервер отвечает на каждую команду трехзначным кодом ответа (для FTP-клиента) и необязательным текстовым сообщением (для пользователя). Первые две цифры предоставляют информацию о типе ответа; третья цифра не имеет определенного значения. Например, в коде 200 ("команда успешно выполнена") цифра "2" указывает на успешное завершение команды, а следующий за ней "0" указывает на то, что ответ имеет отношение к синтаксису.

Управляющее соединение продолжает поддерживаться в процессе обмена командами и данными между клиентом и сервером в ходе их диалога. Типичное взаимодействие между клиентом и сервером начинается с команды, которая идентифицирует учетную запись (например, анонимную) на компьютере сервера, вслед за которой идет команда, отправляющая пользовательский пароль (например, адрес электронной почты пользователя для анонимной учетной записи FTP). Параметры этих команд берутся из пользовательского ввода. Сервер использует эту информацию для принятия решения, к каким файлам предоставить доступ данному клиенту. Например, для анонимного пользователя доступ может быть ограничен небольшой группой файлов. В начале сеанса FTP-клиент может увидеть корневой каталог, предоставляемый FTP-сервером. Следующее действие клиента зависит от

---

[1] Это соединение используется для передачи команд управления, соединение для передачи данных устанавливается по 20 порту. — *Прим. ред.*

запроса, инициированного пользователем, например, чтение файла, запись файла, вывод списка файлов в текущем каталоге, переход к другому каталогу, вывод имени текущего каталога и т.д.

Файл передается с использованием отдельного соединения, установленного компьютером, отправившим данные. Если пользователь хочет получить файл **foo.txt**, сервер инициирует создание TCP-соединения. Однако сервер не знает, какой номер порта использовать в качестве порта-адресата для FTP-клиента. Перед отправкой команды на загрузку файла **foo.txt** FTP-клиент просит свою базовую операционную систему выделить номер порта (больше 1023). FTP-клиент использует управляющее соединение, чтобы проинформировать сервер о выбранном номере порта для соединения, используемого для передачи данных. Сервер создает соединение для передачи данных, записывает содержимое файла **foo.txt** и закрывает соединение. FTP-клиент считывает байты из сокета до получения символа конца файла (EOF). На практике некоторые организации имеют межсетевые экраны, которые не позволяют внешним компьютерам инициировать TCP-соединения. Чтобы позволить клиентам за межсетевыми экранами получать данные с FTP-серверов, в FTP имеется команда, которая принуждает сервер играть пассивную роль в установлении соединения для работы с данными.

Файл обычно передается как поток байтов с закрытием TCP-соединения при достижении конца файла (EOF). Спецификация FTP также описывает *блочный режим*, который позволяет отправителю передавать файл как серию блоков данных; каждый блок начинается с одного или нескольких байтов заголовка. Блочный режим дает возможность отправителю пересылать несколько файлов через одно и то же соединение для передачи данных; при этом каждый файл заканчивается на границе блока. Однако блочный режим не получил широкого распространения. На практике каждая пересылка данных требует отдельного TCP-соединения. В противоположность этому управляющее соединение может существовать в ходе нескольких передач данных. В FTP имеется команда для прерывания передачи исходящих данных без завершения управляющего соединения. Это дает возможность пользователю завершить копирование файла без требования к клиенту повторить процесс соединения с сервером и аутентификации пользователя.

В то время как в Telnet передача данных ограничена 7-битными символами ASCII, FTP позволяет использовать более широкий набор типов данных, включая двоичные файлы. Однако FTP-сервер *не* предоставляет информации о типе каждого из файлов. Вместо этого FTP-клиент должен запросить передачу данных в определенной форме. Например, предположим, что пользователь хочет выгрузить файл **picture.gif** с локальной машины на удаленный компьютер. Пользователю нужно запросить двоичную передачу данных. Это приведет к выдаче FTP-клиентом команды FTP-серверу переключиться в двоичный режим. Клиент и сервер не могут указать режим передачи без помощи от пользователя, поскольку локальный файл напрямую не ассоциируется с дополнительными атрибутами. Расширение **.gif** в имени файла не обязательно указывает на тип файла. После копирования файла пользователь может запустить отдельную программу для отображения или модификации содержимого файла, основываясь на знании пользователем типа файла. Например, пользователь может вызвать *ghostview* для отображения файла PostScript.

5.4.3. SMTP — простой протокол передачи электронной почты

Протокол SMTP (Simple Mail Transfer Protocol) [Pos82] используется для передачи сообщений электронной почты от локального почтового сервера удаленному почтовому серверу. Кроме того, SMTP может быть использован для отправки сообщений электронной почты от почтового агента пользователя на локальный почтовый сервер.

ПРИМЕНЕНИЕ ЭЛЕКТРОННОЙ ПОЧТЫ

Доставка сообщения электронной почты от одного пользователя другому вовлекает несколько компонентов. Пользователь запускает почтовый агент для отправки и приема сообщений электронной почты. Агент может также поддерживать множество других функций, таких как распределение сообщений по папкам, создание, редактирование и удаление сообщений. Помимо взаимодействия с пользователем, почтовый агент осуществляет взаимодействие с локальным почтовым сервером. Локальный почтовый сервер обслуживает почтовые ящики пользователей и осуществляет обмен сообщениями с другими почтовыми серверами. В качестве примера предположим, что пользователь с именем Viv имеет собственный компьютер и получает электронную почту по адресу **viv@foo.com**; почтовый сервер обслуживает почтовый ящик **viv**. Разделение обязанностей между агентом пользователя и почтовым сервером весьма важно — почтовый агент предоставляет богатые возможности для одного пользователя, а почтовый сервер обеспечивает надежный сервис для множества пользователей.

В противоположность передаче файлов отправка и прием сообщений электронной почты не является интерактивным приложением. При отправке сообщения электронной почты почтовому агенту не обязательно знать, достигло ли сообщение почтового сервера или почтового агента получателя, и когда это произошло. Фактически, отправляющий сообщение пользователь может завершить работу почтового агента до того, как почтовый сервер закончит доставку сообщения электронной почты удаленному почтовому серверу. На практике многие пользователи отправляют и принимают сообщения электронной почты с помощью своих Web-браузеров. Это осуществляется двумя основными способами. В первом случае Web-браузер может действовать как почтовый агент, который взаимодействует с локальным почтовым сервером и предоставляет интерфейс для чтения и составления сообщений. Во втором случае браузер может использоваться для обращения к Web-сайту, который дает возможность пользователям читать сообщения (путем получения HTML-файла) и отправлять сообщения (путем отправки HTML-формы). В этой ситуации Web-сервер координирует взаимодействие с почтовым сервером. Например, сервер может запускать сценарии для извлечения сообщений электронной почты, предназначенных пользователю, и для отправки сообщений электронной почты, созданных пользователем.

Сообщение электронной почты состоит из заголовка и тела сообщения. Тело сообщения представляет собой текст, отправляемый пользователем. Каждое поле заголовка начинается с новой строки и состоит из одной строки, заканчивающейся точкой с запятой (например, **Date: Sat Oct 28 2000 11:29:32 GMT**). Некоторые из полей, такие как **To (Кому)** и **Subject (Тема)**, зависят от пользовательского ввода. Другие, такие как **Date (Дата)** и **Message-Id (Идентификатор сообщения)**, задаются агентом пользователя или локальным почтовым сервером, который отправляет сообщение. Заголовок и тело сообщения состоят из текстовых строк в 7-битном

формате ASCII. Каждая строка заканчивается последовательностью из двух символов: возврата каретки (CR) и перевода строки (LF). Первоначально сообщения электронной почты могли содержать только текстовые данные. Позднее спецификация Multipurpose Internet Mail Extensions (MIME) [FB96a, FB96b] предоставила стандартный способ для преобразования других типов данных в текстовый формат и включения их в сообщения электронной почты. MIME предусматривает дополнительные заголовки, которые указывают на размер и способ кодирования содержимого сообщения.

ПРОТОКОЛ SMTP

SMTP был разработан в 1982 г., чтобы заменить FTP при передаче сообщений электронной почты от одного почтового сервера другому. В SMTP отправляющий почтовый сервер устанавливает TCP-соединение по порту 25 принимающего почтового сервера. Объединенное с заголовком тело сообщения передается от одного почтового сервера другому с помощью последовательности команд. При передаче сообщений электронной почты почтовые серверы не делают различий между заголовком и телом сообщения. Единственно, что делает почтовый сервер с сообщением, это заполнение дополнительных полей заголовка, например, **Received (Получено)**. Это дает возможность получателю идентифицировать последовательность почтовых серверов, участвующих в передаче сообщения. Аналогично FTP, SMTP ориентирован на передачу текста и основывается на командах. Отправитель выдает последовательность команд, по одной за раз, и получает ответы, состоящие из трехзначного кода ответа и текстовой строки.

Локальный почтовый сервер определяет удаленный почтовый сервер, выполняя особый вид DNS-запроса. Локальный почтовый сервер выдает запрос на информацию о записи MX (Mail Exchanger) для полностью заданного имени домена в правой части адреса электронной почты (например, **users.foo.com** в **viv@users.foo.com**). DNS-ответ состоит из имен одного или нескольких хостов, действующих в качестве почтовых серверов для данного полного имени домена. После выбора удаленного почтового сервера (например, **bigmail.foo.com**) локальный почтовый сервер выполняет дополнительный DNS-запрос для определения IP-адреса удаленного почтового сервера. Затем локальный почтовый сервер устанавливает TCP-соединение с удаленным почтовым сервером. Проблемы могут возникнуть на каждом из этих этапов. Во-первых, DNS-запрос может возвратить ошибку. Например, сообщение может быть направлено домену, который не существует (например, **viv@users.foo.com**, где **users.foo.com** не существует). Это приведет к тому, что локальный почтовый сервер возвратит ошибку отправителю сообщения. Во-вторых, предположим, что локальный почтовый сервер знает IP-адрес удаленного почтового сервера, но не может установить TCP-соединение. Это может произойти, если удаленный сервер временно отключен, или отказ в сети вызван разрывом соединения двух хостов. В таких ситуациях локальный почтовый сервер сохраняет сообщение и пытается снова его передать через некоторое время. Одновременно отправитель может быть проинформирован, что передача сообщения была задержана.

Коммуникационное взаимодействие между двумя серверами начинается с приветственного сообщения от удаленного почтового сервера. Затем локальный почтовый сервер выдает команды для передачи сообщения электронной почты. Типичный обмен включает отдельные команды для:

- идентификации локального почтового сервера;
- идентификации отправителя сообщения электронной почты;

- идентификации каждого получателя сообщения электронной почты;
- отправки данного сообщения электронной почты.

В противоположность FTP, SMTP использует одно TCP-соединение для обмена командами и передачи сообщения электронной почты. Отправитель помечает конец сообщения последовательностью CRLF с последующей точкой (".") и еще одной последовательностью CRLF. Однако сообщение электронной почты может содержать строку, начинающуюся с символа ".", что может ввести в заблуждение получателя. В связи с этим локальный почтовый сервер добавляет второй символ "." к таким строкам, и отправляя ".." вместо ".". Лишний символ "." удаляется получателем перед отображением сообщения пользователю.

Помимо передачи сообщения между почтовыми серверами, доставка сообщения электронной почты требует двух дополнительных действий, выполняемых агентом пользователя: передачи сообщения локальному почтовому серверу и получение сообщения от удаленного почтового сервера. Пользовательский агент отправителя инициирует передачу сообщения электронной почты локальному почтовому серверу. Эта передача может также использовать SMTP, хотя возможно применение и других протоколов. Отправка сообщения электронной почты состоит в передаче данных от почтового агента почтовому серверу. Напротив, извлечение сообщения из почтового ящика состоит в получении данных от почтового сервера. При получении сообщений электронной почты SMTP обычно не используется. Для доступа к почтовому ящику в Web может быть использован HTTP. Другие протоколы, такие как Post Office Protocol (POP3) и Internet Message Access Protocol (IMAP), были специально разработаны для получения сообщений электронной почты с почтового сервера. Более подробный обзор протоколов, используемых для обмена сообщениями электронной почты, можно найти в соответствующей литературе [Jox00].

5.4.4. Network News Transfer Protocol

Сетевой протокол передачи новостей (NNTP – Network News Transfer Protocol) [KL86] поддерживает передачу статей электронных групп новостей. Агент пользователя использует NNTP для коммуникационного взаимодействия с локальным сервером новостей, который использует NNTP для взаимодействия с центральным хранилищем статей новостей.

ПРИМЕНЕНИЕ СЕТЕВЫХ НОВОСТЕЙ

Помимо обмена сообщениями между парой пользователей, электронная почта иногда используется также для распространения сообщений для групп пользователей в течение определенного периода времени. Например, организация может создать список рассылки с целью распространения сообщений по определенной теме. Заинтересованные пользователи могут подписаться на список. Однако списки рассылки не слишком эффективны при наличии больших, меняющихся групп пользователей. Каждый член списка получает отдельную копию сообщения, которая потребляет сетевые ресурсы и требует места для хранения. Кроме того, менеджер списка рассылки должен обновлять перечень получателей по мере прихода и ухода пользователей. На практике этот процесс может быть автоматизирован, если пользователи будут отправлять по электронной почте на специальную учетную запись сообщения для подписки и аннулирования подписки на список рассылки, которые обрабатываются специальной программой. Почтовый сервер списка рассылки может также испытывать затруднения при обращении ко всем получателям, особенно

если пользователи забывают аннулировать подписку на список рассылки перед удалением или изменением своих учетных записей электронной почты.

Затруднения, связанные со списками рассылки, стали толчком для создания системы групп новостей USENET, разработанной в 1979 г. и стандартизованной в 1983 г. [Hor83]. Главная идея состоит в хранении сообщений в централизованной базе данных вместо помещения отдельных копий в почтовые ящики каждого из подписчиков. База данных состоит из набора *групп новостей* (например, **soc.culture.indian**), каждая из которых соответствует упорядоченному списку сообщений. Имена групп новостей состоят из строк, разделенных точками. Схема именования иерархическая. Например, множество связанных между собой групп новостей начинается с **soc** или **rec**. База данных поддерживает запросы для извлечения статей из групп новостей, а также индексирование, перекрестные ссылки и контроль «возраста» статей. Системный администратор может задавать стратегии по удалению устаревших статей после истечения определенного периода времени с целью освобождения пространства на сервере.

Подобно сообщению электронной почты, статья в группе новостей содержит несколько строк заголовка. Например, статья включает следующие строки заголовка:

- адрес электронной почты отправителя статьи,
- тема статьи,
- дата/время создания статьи,
- число строк текста в статье,
- уникальный идентификатор сообщения статьи,
- список групп новостей, являющихся получателями статьи.

Группы новостей могут быть реплицированы на различные серверы в Internet. Например, компания или университет может иметь свой сервер новостей для поддержки определенного круга пользователей. Локальный сервер новостей может не реплицировать все группы новостей, а поддерживать особые группы новостей, предназначенные для сообщества его пользователей. К примеру, университет может исключить группы новостей по ряду тем, но поддерживать дополнительные группы новостей по различным учебным курсам, по которым ведется обучение.

Пользователи читают и размещают статьи с помощью агента пользователя, который координирует взаимодействие с локальным сервером новостей. Агент пользователя предоставляет интерфейс, который отображает краткое содержание статей, которое включается в строки заголовка, и поддерживает чтение и публикацию статей. Большинство Web-браузеров способно связываться с локальным сервером новостей. В Web группа новостей идентифицируется по URL, который содержит имя группы (например, **news:soc.culture.indian**). Помимо отображения текста некоторые пользовательские агенты могут осуществлять синтаксический анализ и отображение форматированных данных. Кроме того, пользовательский агент запоминает, к какой группе новостей обращается пользователь, а также какие сообщения прочитаны или удалены. Эта информация обычно хранится в локальном файле, который может быть прочитан при следующем вызове пользователем агента. Некоторые Web-сайты предоставляют доступ к статьям групп новостей и разрешают пользователям читать статьи (путем загрузки HTML-файлов), а также публиковать статьи (путем отправки содержимого HTML-форм). В этом случае Web-браузер связывается непосредственно с Web-сервером, который координирует взаимодействие с сервером групп новостей.

ПРОТОКОЛ NNTP

NNTP появился в 1986 г., заменив протокол UNIX-to-UNIX Copy Protocol (UUCP), использовавшийся для передачи статей групп новостей. NNTP координирует передачу сообщений между локальным сервером новостей и централизованным хранилищем. NNTP также может использоваться для взаимодействия между агентом пользователя и локальным сервером новостей. NNTP-клиент устанавливает одно TCP-соединение с портом 119 NNTP-сервера. Подобно FTP и SMTP, NNTP-клиент выдает команды серверу, который возвращает трехзначный код ответа и необязательное текстовое сообщение. В зависимости от кода ответа, отклик NNTP может содержать дополнительные параметры. Количество и тип этих параметров фиксирован для каждого кода ответа, чтобы упростить интерпретацию ответа NNTP-клиентом. При обработке команд NNTP-сервер отслеживает текущее имя группы новостей и номер статьи, к которой осуществляет доступ NNTP-клиент.

В NNTP имеется множество команд, которые выполняют различные функции. Один набор команд предоставляет общую информацию о NNTP-сервере и различных группах новостей, в том числе:

- перечень команд, воспринимаемых NNTP-сервером,
- список имеющихся групп новостей, включая информацию о первой и последней статье в каждой группе, а также сведения, разрешена ли публикация статей в данной группе новостей,
- список групп новостей, созданных после определенной даты/времени,
- список статей, опубликованных в определенной группе новостей после определенной даты/времени.

Предоставление информации об изменениях, произошедших после определенной даты/времени, дает возможность NNTP-клиенту обновлять свое представление групп новостей и статей без предъявления к NNTP-серверу требования хранить информацию в интересах каждого клиента. Другие команды используются для перехода к другой группе новостей (ссылка на которую осуществляется по ее имени), перехода к другой статье (ссылка на которую осуществляется по идентификатору сообщения или по ее номеру в группе новостей), перехода на одну статью назад в пределах текущей группы новостей и перехода на одну статью вперед в пределах текущей группы новостей.

Другие команды возвращают всю статью, только ее заголовки или только содержание. Для каждой из этих команд статья идентифицируется по ее идентификатору или по числу, указывающему ее позицию в группе новостей. В NNTP также имеются команды, связанные с созданием новых статей. Клиент отправляет статью через существующее TCP-соединение, следуя тем же соглашениям, которые действуют для SMTP. Конец статьи идентифицируется по одиночной точке "." в строке. После получения статьи централизованное хранилище создает уникальный идентификатор сообщения и добавляет статью в группу новостей. Другие команды поддерживают репликацию сообщений между двумя NNTP-серверами. В NNTP также предусмотрена команда для завершения сеанса клиентом. После получения ответа на команду завершения NNTP-сервер закрывает TCP-соединение.

5.4.5. Свойства протоколов прикладного уровня

Для поддержки новых способов коммуникационного взаимодействия через Internet создаются новые протоколы прикладного уровня. Telnet был создан для

поддержки взаимодействия пользователей с удаленными компьютерами, а FTP решал задачу передачи файлов между различными компьютерами. По мере роста популярности электронной почты SMTP пришел на смену ранее использующемуся для передачи сообщений электронной почты протоколу FTP. Аналогичным образом NNTP возник с целью поддержки групп новостей в качестве альтернативы спискам рассылки. В результате такой преемственности Telnet, FTP, SMTP и NNTP имеют черты сходства и различия:

- **Команда/ответ.** Клиенты и серверы Telnet отправляют команды в двоичном формате, начинающиеся со специального октета (11111111). В противоположность этому в FTP, SMTP и NNTP команды являются текстовыми и отправляются клиентом. Команды имеют четко определенный, фиксированный формат, а сервер отвечает трехзначным кодом ответа и необязательным текстовым сообщением. Некоторые NNTP-ответы содержат дополнительную информацию в фиксированном формате в зависимости от кода ответа.
- **Типы данных.** Telnet, FTP, SMTP и NNTP передают текстовые данные в 7-битном формате ASCII. FTP также поддерживает передачу данных в двоичной форме, когда это указывается клиентом. SMTP и NNTP поддерживают MIME для преобразования нетекстовых данных в формат ASCII и доставки информации о типе данных получателю.
- **Транспорт.** Все четыре протокола основаны на надежном транспортном протоколе TCP. Telnet, SMTP и NNTP используют одно TCP-соединение для передачи команд/ответов и данных, а соединение сохраняется на протяжении сеанса. В противоположность этому, FTP использует отдельные соединения для управления и для данных. Управляющее соединение поддерживается в процессе обмена командами/ответами; при последовательных передачах данных обычно используются различные соединения.
- **Направление передачи.** FTP и NNTP могут передавать данные в обоих направлениях — копирование данных или публикация статей клиентом и получение файлов или статей с сервера. SMTP используется для передачи сообщений электронной почты от клиента серверу. В Telnet и FTP клиентом обычно является агент, который взаимодействует непосредственно с пользователем. В SMTP и NNTP клиентом может быть локальный сервер, который обменивается данными (сообщениями электронной почты или статьями групп новостей) с удаленным сервером.
- **Сохранение информации о сеансе.** Во всех четырех протоколах сервер сохраняет информацию о сеансе с клиентом. Например, Telnet-сервер сохраняет информацию о параметрах терминала NVT. FTP-сервер запоминает текущий каталог клиента и текущий режим передачи данных (двоичный или текстовый). SMTP-сервер запоминает информацию об отправителе и получателях сообщения электронной почты в процессе ожидания команды **DATA** для передачи содержимого сообщения. NNTP-сервер сохраняет для клиента текущую группу новостей и номер статьи.

Telnet, FTP, SMTP и NNTP до появления Web представляли собой устоявшиеся протоколы прикладного уровня. В связи с этим некоторые ключевые элементы HTTP тесно связаны с этими более ранними протоколами. Тем не менее, у HTTP также имеется большое число существенных отличий, о чем пойдет речь в следующих двух главах.

5.5. Резюме

Протокол Internet Protocol обеспечивает базовый сервис по доставке пакетов через различные физические среды передачи данных. Наличие простого, открытого стандарта способствовало быстрому превращению Internet во всемирную коммуникационную инфраструктуру. Более сложные сервисы поддерживаются протоколами транспортного и прикладного уровней, которые реализуются на оконечных хостах. Каждый уровень выполняет определенную задачу вне зависимости от деталей реализации на других уровнях. Протокол Transmission Control Protocol предоставляет абстракцию сокетов, которые обеспечивают надежную доставку упорядоченных последовательностей байтов между двумя хостами. Сокеты служат основными строительными блоками для разнообразных Internet-приложений. Хосты могут идентифицироваться по доменным именам и по IP-адресам. Система именования доменов Domain Name System предоставляет простой, расширяемый способ преобразования одного из этих представлений в другое. Все вместе IP, TCP и DNS поддерживают множество Internet-приложений, включая Telnet, передачу файлов, электронную почту и группы новостей. Эти приложения основываются на протоколах Telnet, FTP, SMTP и NNTP, которые были предшественниками Web и оказали влияние на HTTP.

6

Структура и описание протокола HTTP

Любой протокол — это язык со своей грамматикой, синтаксической структурой, включающей в себя форматы сообщений, и семантическими правилами, указывающими, как следует интерпретировать поля сообщений. Hypertext Transfer Protocol (HTTP) представляет собой протокол типа запрос–ответ, составляющий основу Всемирной паутины. Это протокол прикладного уровня, подобно File Transfer Protocol (FTP) и Telnet. Однако в отличие от этих протоколов, HTTP не сохраняет своего состояния. HTTP появился в 1990 г., а с 1995 г стал протоколом, с помощью которого передается значительная часть трафика Internet.

Важно понимать разницу между HTTP и Всемирной паутиной. Как упоминалось в первой главе, Web состоит из трех семантических частей: протокола HTTP, языка гипертекстовой разметки (HTML) и схемы именования с помощью унифицированных идентификаторов ресурсов (URI). В Web используются несколько важных программных компонентов (браузеры с графическим пользовательским интерфейсом, прокси-серверы и собственно Web-серверы), которые взаимодействуют между собой с помощью протокола HTTP. Web-компоненты также используют HTTP как механизм коммуникационного взаимодействия для обращения к ресурсам, доступным через другие протоколы, например, FTP или Telnet.

HTTP используется для передачи информации в различных форматах, на различных языках и с различными наборами символов. Синтаксис HTTP-сообщения основан на стандарте MIME — Multipurpose Internet Mail Extensions [FB96a, FB96b]. Содержимое HTTP-сообщения слепо воспринимается протоколом — никакой интерпретации при этом не производится

Материал, изложенный в этой главе, основывается на документе RFC 1945 [BLFF96] и предваряется описанием ключевых понятий и терминологии, которые существенны для понимания HTTP. Документ RFC 1945 отражает общие принципы использования HTTP в середине 90-х годов. Текущей версией протокола HTTP на момент публикации данной книги была версия HTTP/1.1. Следующая глава будет посвящена HTTP/1.1 и отличиям между HTTP/1.0 и HTTP/1.1. Наше решение представить описание HTTP/1.0 и HTTP/1.1 в отдельных главах является не случайным. Предварительное рассмотрение HTTP/1.0 дает читателю возможность познакомиться с основными принципами протокола и его преимуществами по сравнению с другими конкурировавшими на момент его создания протоколами. Дальнейшее рассмотрение HTTP/1.1 позволяет познакомиться с эволюцией протокола.

В этой главе будет рассмотрено, как HTTP развивался с первых дней возникновения Web. Мы начнем с обзора протокола HTTP, познакомимся с его истоками, основными возможностями и его влиянием на другие технологии. Протокол HTTP

разрабатывался на основе других приложений, популярных во времена создания Web, и использовал некоторые их идеи.

В следующем разделе будут рассмотрены «строительные блоки» протокола: элементы языка (объекты, методы запросов, заголовки, используемые в HTTP-сообщениях) и различные классы ответов. Имеющийся на настоящий момент набор популярных Web-приложений, таких как поиск и удаленное выполнение команд, представляют лишь небольшое подмножество операций, осуществляемых с использованием HTTP. Понимание основных элементов протокола поможет оценить все его возможности и с успехом применять его в различных приложениях.

После рассмотрения вопросов, связанных с возможностями расширения протокола HTTP, мы поговорим о проблемах безопасности при взаимодействиях с использованием HTTP. Будет рассмотрен протокол Secure Socket Layer (SSL) и его использование для обмена информацией в Web. Как и в других коммуникационных протоколах, в HTTP имеется несколько правил, которые помогают при построении совместимых реализаций. Мы рассмотрим правила совместимости для протокола HTTP и поговорим о значении номера версии протокола. Завершается глава обсуждением принципов и практики применения HTTP/1.0.

6.1. Обзор HTTP

Обзор HTTP удобно осуществлять в историческом контексте. HTTP эволюционировал вместе со Всемирной паутиной и ее двумя другими компонентами: URI и HTML. В процессе эволюции HTTP можно выделить два этапа:

- от появления первой спецификации HTTP/0.9 до версии HTTP/1.0 (этот период занял четыре года);
- от появления HTTP/1.0 до появления HTTP/1.1 (еще четыре года).

В таблице 6.1 приведена частичная хронология публикации проектов и окончательных версий стандартов.

Таблица 6.1. Хронология публикации документов, относящихся к HTTP

Дата	Документ
Март 1990 г.	Документ CERN с предложениями по созданию Web
Январь 1992 г.	Спецификация HTTP/0.9
Февраль 1992 г.	W3 и WAIS/X.500
Декабрь 1992 г.	Предложения по добавлению MIME в HTTP
Февраль 1993 г.	Универсальные идентификаторы документов (UDI — Universal Document Identifier) для сетевого использования
Март 1993 г.	Первая версия рабочего проекта HTTP/1.0
Июнь 1993 г.	HTML (спецификация 1.0)
Октябрь 1993 г.	Спецификация URL
Ноябрь 1993 г.	Вторая версия рабочего проекта HTTP/1.0
Март 1994 г.	URI в WWW
Май 1996 г.	Информационный материал по HTTP/1.0 (RFC 1945)

Дата	Документ
Январь 1997 г.	Предложение по стандарту HTTP/1.1 (Proposed Standard, RFC 2068)
Июнь 1999 г.	Рабочий проект стандарта HTTP/1.1 (Draft Standard, RFC 2616)
2001 г.	Официальный стандарт HTTP/1.1 (Formal Standard)

HTTP был предложен в марте 1990 г. Тимом Бернерс-Ли, работавшим тогда в CERN, как механизм для доступа к документам в Internet и облегчения навигации посредством *гиперссылок* [Nel67]. Самая ранняя версия протокола HTTP/0.9[1] была впервые опубликована в январе 1992 г. в списке рассылки **www-talk** [BL92a], хотя его реализация датируется 1990 годом. Спецификация HTTP/0.9 привела к упорядочению правил взаимодействия между клиентами и серверами, а также разделению функций между этими двумя компонентами. Были задокументированы основные синтаксические и семантические положения, а HTTP/0.9 получил статус подмножества более детализированного протокола, разработка которого планировалась в то время. Основное внимание тогда было сосредоточено на поисковых возможностях, поскольку поиск был главной функцией, предоставляемой конкурирующими системами, такими как Gopher и Archie.

Гипертекст приобрел популярность в 1980-е годы в качестве средства навигации в среде взаимосвязанных документов. В главе 1 мы определили гипертекст как нелинейную запись информации, т.е. текст, на который не накладывается ограничение линейности. Читатель может перемещаться от одной части документа к другой части того же документа или к совершенно другому документу. Подобная нелинейность становится легко достижимой с помощью гипертекста. Гипертекстовый документ может состоять всего лишь из набора гиперссылок на другие гипертекстовые документы.

В декабре 1992 г. в списке рассылки **www-talk** завязалась дискуссия о возможности использования MIME в глобальных информационно-поисковых системах, таких как Wide Area Information Servers (WAIS) и Web. Развивавшийся в то время стандарт Internet MIME упорядочивал форматы данных и обеспечивал пересылку сообщений, состоящих из нескольких разнородных частей по электронной почте. HTTP-ответы могли возвращаться в виде MIME-сообщений. Первая предварительная спецификация HTTP/1.0 появилась в марте 1993 г. В июне 1993 г. в проекте спецификации [BLC93] было предложено сделать HTML одним из типов содержимого MIME. Позже, в октябре 1993 г., после дискуссии в рабочей группе URI была опубликована первая предварительная спецификация унифицированных указателей ресурсов (URL — Uniform Resource Locators) [BL93b].

После этого вышло несколько рабочих проектов документов HTTP/1.0. В мае 1996 г. для практической реализации HTTP был выпущен информационный документ RFC 1945 [BLFF96], что послужило основой для реализации большинства компонентов HTTP/1.0. В следующие три года наблюдалась значительная активность в попытках доработать HTTP/1.0. В январе 1997 г. появился документ RFC 2068 с предложением по стандарту (Proposed Standard), а в июне 1999 г. был опубликован документ RFC 2616 [FGM+99] в качестве проекта стандарта (Draft Standard) HTTP/1.1. Официальный стандарт[2] HTTP должен появиться в 2001 г.

---

1 По сложившейся традиции начальной версии протокола до проведения широкого обсуждения, доработки и распространения присваивают номер, меньший 1.

2 На время редактирования книги официальный стандарт не был принят. — *Прим. ред.*

Обзор протокола HTTP в этом разделе осуществляется в двух аспектах:

- **Свойства протокола.** Здесь рассматриваются взаимосвязь HTTP с инфраструктурой URI, система функционирования HTTP по принципу запрос-ответ, отсутствие сохранения информации о состоянии между запросами и понятие метаданных (информация о ресурсе).
- **Влияние других протоколов.** Здесь рассматривается влияние стандарта MIME на HTTP, а также некоторых других протоколов, популярных на момент разработки HTTP. Мы также сравним HTTP с другими протоколами прикладного уровня.

6.1.1. Свойства протокола

В этом разделе будут рассмотрены некоторые основные свойства HTTP. К ним относятся:

- **Глобальные URI.** HTTP основывается на механизме именования URI. HTTP использует URI во всех транзакциях для идентификации ресурсов в Web.
- **Обмен по схеме запрос–ответ.** HTTP-запросы отправляются клиентами, получая затем ответы от серверов. Направление потока — от клиента к серверу; сервер не инициирует Web-трафик.
- **Отсутствие сохранения состояния.** Состояние между запросами и ответами клиентами и серверами не сохраняется. Каждая пара запрос-ответ трактуется как независимый обмен сообщениями.
- **Метаданные ресурсов.** Информация о ресурсах часто включается в Web-транзакции и может быть использована различными способами.

ПРИМЕНЕНИЕ ГЛОБАЛЬНЫХ URI

HTTP основывается на идее унифицированного идентификатора ресурса (URI — Uniform Resource Identifier) [BLFM98]. Механизм именования дает возможность размещать ресурсы в любой точке Internet и разделить понятия ресурса и ответа. Ресурс может иметь связанный с ним URI, хотя представление ресурса и его информационное содержание может многократно меняться за время существования ресурса. С точки зрения протокола URI представляет собой форматированную строку. URI просто указывает на ресурс вне зависимости от его текущего местоположения или имени, по которому он известен. В этом смысле URI является комбинацией унифицированного указателя ресурса (URL — Uniform Resource Locator) [BLMM94, Fie95] и унифицированного имени ресурса (URN — Uniform Resource Name) [SM94]. URI представляет собой надмножество URL, URN и может быть выражен одной из этих составляющих, либо обеими. Наиболее популярной формой URI является URL.

В качестве примера, поясняющего различия между URI, URL и URN, возьмем эту книгу. Американское издание книги имеет ISBN 0-201-71088-9, присвоенный Библиотекой Конгресса США. Номер ISBN гарантированно уникален, отличается от всех номеров уже опубликованных книг и останется неизменным даже после публикации миллиона других книг. Только Библиотека Конгресса США решает, как присвоить уникальный номер этой книге. Номер ISBN представляет собой URN для этой книги; он именует книгу, но не предоставляет информацию о том, где книга может быть получена. URI для этой книги может представлять ее содержание, если все содержание доступно через компьютерный протокол. Предположим, что содержание

этой книги было помещено на Web-сервер и стало доступным по HTTP. Местонахождение книги может быть указано с помощью URL **http://www.research.att.com/books/kandr.ps.gz**. Книга также может быть доступной через анонимный FTP-сервер в каталоге **pub/bala** на компьютере **ftp.research.att.com**. Таким образом, она имеет альтернативный URL **ftp://ftp.research.att.com/pub/bala/kandr.ps.gz**. Этот URL задает местонахождение копии ресурса. Таким образом, URI книги может быть представлен любым из упомянутых выше URL или URN (номером ISBN).

URI считается *абсолютным*, если строка начинается со *схемы*, за которой следует строка, представляющая ресурс, который может быть получен через схему. *Схема* просто указывает протокол, который будет использоваться для доступа к ресурсу. *Относительный* URI не начинается с имени схемы. В самой первой спецификации протокола HTTP, известной позднее как **HTTP/0.9** [BL92a], было перечислено пять схем: **file**, **news**, **http**, **telnet**, **gopher**. Он также предвосхитил использование по меньшей мере еще двух (**WAIS** и **X-500**) и зарезервировал схемы для них.

Хотя имя схемы связано с определенным протоколом, для доступа к ресурсу по URL может быть задействовано более одного протокола. Для обработки Web-запроса на ресурс обычно требуется несколько протоколов, таких как Domain Name System (DNS) для определения IP-адреса хоста, на котором размещен ресурс, по его доменному имени, и Transmission Control Protocol (TCP) для загрузки ресурса по соединению транспортного уровня. Точно так же для идентификации одного ресурса может быть использовано более одной схемы.

Наиболее часто в Web используется схема **http**. Каждая схема имеет свой собственный синтаксис, и для всех схем предусмотрены механизмы для именования ресурсов. За счет отделения схемы от внутреннего синтаксиса конкретного протокола механизм именования в Web дает возможность легко адаптировать себя для других систем. Поскольку доступ к одному и тому же ресурсу может быть осуществлен с использованием различных протоколов, другие системы могут быть наложены поверх HTTP. Возможность использовать не-HTTP URI, т.е. работать с ресурсами, доступ к которым осуществляется через протоколы, *отличные* от HTTP, была необходима, поскольку подобная практика была вполне типичной на момент создания Web. Отделение схемы от синтаксиса привело к тому, что Web превратилась во внешний интерфейс для доступа к ресурсам, для работы с которыми обычно используются другие протоколы, например, **ftp**. В качестве примеров не-HTTP URI можно привести **rtsp://clips.foo.com/preview/audio**, **telnet://ox.aciri.org** и т.д. Столкнувшись с не-HTTP URI, браузер осуществляет синтаксический анализ имени схемы до символа ":" и активизирует соответствующий обработчик протокола — RTSP (Real Time Streaming Protocol) и Telnet, соответственно.

ОБМЕН ЗАПРОСАМИ-ОТВЕТАМИ

Протокол HTTP задает синтаксис и семантику, в соответствии с которыми компоненты Web, такие как клиенты и серверы, взаимодействуют друг с другом. HTTP-сообщение структурировано и обладает определенным синтаксисом. HTTP является запрос–ответным протоколом, в котором *запрос* — это сообщение, посылаемое клиентом принимающему серверу. Принимающим может быть исходный сервер — сервер, на котором размещаются или генерируются ресурсы, или промежуточное звено, такое как прокси-сервер. Сервер-получатель отправляет обратно *сообщение-ответ*. Клиентом может выступать *агент пользователя*, нечто, инициировавшее запрос, или какой-либо компонент на пути между инициатором и конечным сервером-получателем. Протокол определяет набор расширяемых *методов* запроса, которые используются клиентом для выполнения операций, таких как полу-

чение, изменение, создание или удаление *ресурса*. Ресурсом является объект, сервис или коллекция элементарных сущностей, которые могут быть четко идентифицированы и размещены в любом месте сети [BLFM98].

Рассмотрим следующий HTTP-запрос:

```
GET /foo.html HTTP/1.0
```

Строка **/foo.html** идентифицирует запрашиваемый с помощью метода **GET** ресурс. Номер версии протокола HTTP, в данном случае 1.0, используется для идентификации возможностей отправителя. Цель запроса — получить текущее содержимое ресурса, идентифицируемого строкой **/foo.html**, с сервера. Если быть более точным, запрос приводит к применению метода **GET** к ресурсу, идентифицируемому **/foo.html**. В общем случае сообщение-запрос состоит из метода, URI, идентификатора версии протокола, необязательных полей заголовка запроса и необязательных данных, отправляемых с запросом.

После получения сообщения-запроса сервер осуществляет его синтаксический анализ и применяет метод к ресурсу. Сформированный ответ передается обратно клиенту в качестве сообщения-ответа. Например,

```
HTTP/1.0 200 OK
Date: Wed, 22 Mar 2000 08:01:01 GMT
Last-Modified: Wed, 22 Mar 2000 02:16:33 GMT
Content-Length: 3913
...
<3913 байтов текущего содержимого /foo.html>
```

Сообщение-ответ состоит из числового кода ответа (например, указывающего на успех или неудачу), фразы, поясняющей числовой код ответа, необязательных полей заголовка и необязательного тела сообщения. В примере первая строка, известная как *строка состояния*, содержит номер версии HTTP и код ответа **200 OK**, указывающий, что запрос выполнен успешно. Заголовок ответа **Date** указывает время создания ответа. Заголовки **Last-Modified** и **Content-Length** указывают время последней модификации ресурса и длину содержимого, включенного в сообщение-ответ (3913 байта), соответственно. В версии HTTP/0.9 протокола не предусмотрена строка состояния и какие-либо заголовки.

Не все поля заголовка являются информационными. Некоторые поля заголовка запроса могут модифицировать запрос, ограничивая применение метода запроса в зависимости от обстоятельств. Такие поля заголовка называются *модификаторами запроса*. Например, клиент может не захотеть получать ответ, если он не изменился с момента последней его загрузки с сервера.

Чтобы лучше понять протокол, можно вообразить, что исходный сервер представляет собой черный ящик, содержащий ресурсы, идентифицируемые URI. Исходный сервер применяет метод запроса к ресурсу, идентифицируемому URI, и генерирует ответ. Операции по чтению ресурса из файла и записи ответа обратно клиенту «скрыты» внутри черного ящика. Подобное представление обобщает содержимое ресурса и отделяет его от ответа, посылаемого клиенту. Различные запросы с одним и тем же URI могут порождать различные ответы в зависимости от ряда факторов: полей заголовка запроса, времени запроса или имевших место изменений в ресурсе.

HTTP является протоколом прикладного уровня, подобно FTP, Simple Mail Transfer Protocol (SMTP) и Network News Transfer Protocol (NNTP), рассмотренных в пятой главе. HTTP может использовать любой транспортный протокол для передачи сообщения от отправителя получателю. В действительности практически

все известные реализации HTTP используют в качестве протокола транспортного уровня протокол Transmission Control Protocol (TCP).

Сервер не может ответить *до* того, как будет получен запрос. Направление обмена сообщениями от клиента к серверу, а затем от сервера к клиенту. HTTP-сервер может *не отвечать* на запрос. В этом смысле HTTP схож с другими протоколами прикладного уровня, такими как Gopher [AAL+92] и WAIS [KM91].

Мы обрисовали содержимое HTTP-взаимодействия между клиентом и сервером; на практике же имеется еще несколько компонентов, которые могут участвовать в обмене сообщениями, внося свои изменения в способ представления и содержимое сообщений. Наличие других компонентов, таких как посредники, шлюзы или туннели, не меняет основной сути HTTP как механизма для обмена сообщениями между отправителем и получателем. Тем не менее, посредники играют важную роль в HTTP, являясь неким разделительным звеном между клиентом и окончечным сервером-получателем. Посредники дают возможность масштабирования: несколько клиентов могут участвовать в HTTP-взаимодействии, не вызывая излишней загрузки сети. Посредники подробно рассматривались в главе 3.

HTTP НЕ СОХРАНЯЕТ СВОЕГО СОСТОЯНИЯ

HTTP представляет собой протокол, *не сохраняющий своего состояния (stateless)*. Это означает отсутствие сохранения промежуточного состояния между парами запрос–ответ. Каждый новый запрос на ресурс инициирует отдельное применение метода запроса к URI ресурса и создание нового ответа. Компоненты, использующие HTTP, могут и осуществляют сохранение информации о состоянии, связанной с последними запросами и ответами. Браузер, посылающий несколько запросов подряд, может отслеживать задержки ответов. Сервер может хранить информацию об IP-адресе клиента, отправившего последние десять запросов. Однако сам протокол не осведомлен о предыдущих запросах и ответах. В протоколе не предусмотрена внутренняя поддержка состояния, к нему не предъявляются такие требования. В отличие от HTTP, NNTP и FTP частично осуществляют сохранение состояния (см. главу 5).

Решение не поддерживать сохранение состояния в HTTP было принято на стадии разработки протокола с целью обеспечить его расширяемость. На момент появления Web существовало несколько конкурирующих систем, в которых были реализованы протоколы, не сохраняющие состояние. Также предполагалось осуществить поддержку использования конкурирующих систем для доступа к ряду ресурсов в Internet. Добавление сохранения состояния в HTTP создало бы помеху для расширяемости Web. Протокол, требующий сохранения состояния между несколькими, не связанными друг с другом соединениями, также привел бы к необходимости хранения значительных объемов информации на сервере.

По мере развития Web отсутствие сохранения состояния в HTTP стало представлять проблему для некоторых приложений. Например, приложения электронной коммерции требуют сохранения состояния между HTTP-запросами. Транзакция, состоящая из последовательности запросов и ответов, должна быть повторена полностью, если один из запросов был прерван в ходе его выполнения. Управление состоянием HTTP стало очевидной проблемой, что привело к появлению cookies (см. главу 2, раздел 2.6).

МЕТАДАННЫЕ РЕСУРСА

Метаданные — это информация, относящаяся к ресурсу, но не являющаяся частью самого ресурса. Концепция метаданных возникла при разработке протокола

Simple Mail Transfer Protocol (SMTP) [Pos82], в котором имелась возможность определения характеристик полезной нагрузки. Метаданные могут быть включены и в запрос, и в ответ HTTP. Примерами метаданных являются: размер ресурса; тип содержания, например, **text/html**; время последней модификации ресурса и т.д. В зависимости от конкретного ресурса в сообщение могут включаться различные метаданные. Например, для динамически генерируемых ответов длину содержимого включать нежелательно, поскольку определение длины приведет к увеличению времени ожидания ответа. Для статических же ресурсов эта информация может быть легко получена. Точно так же включать время последней модификации имеет смысл для статического ресурса, но не для динамического ресурса. Метаданные ресурса возвращаются в ответе с помощью ряда заголовков, которые будут рассмотрены далее в этой главе. Присутствие некоторых метаданных для определенных запросов и ответов может быть обязательным.

Включение метаданных расширяет возможности взаимодействия между отправителями и получателями. Вот несколько вариантов применения метаданных:

- Знание информации о кодировании содержимого может облегчить ее обработку.
- Метаданные о длине содержимого могут быть использованы получателем, чтобы убедиться в получении именно того, что ожидалось.
- Сервер может указать время последней модификации ресурса, что окажет помощь при кэшировании. Кэш при этом сможет принять решение, нужно ли кэшировать ответ. Эта информация также позволит определить, насколько устарел ресурс.

6.1.2. Влияние других протоколов

Как упоминалось в первой главе, несколько систем конкурировали с Web в предоставлении доступа к ресурсам в Internet. На развитие протокола HTTP оказали влияние конкурирующие системы, такие как Gopher и WAIS. В этом разделе мы исследуем внешние влияния, оказанные на HTTP, и сравним HTTP с другими протоколами прикладного уровня.

РОЛЬ MIME

Разработанный в 1992 г. стандарт Multipurpose Internet Mail Extensions (MIME) стал расширением протокола Internet Mail Protocol (документ RFC 822), призванным облегчить отправку нескольких *объектов*, как текстовых, так и нетекстовых, в одном сообщении. MIME может быть использован для представления текста с помощью наборов не-ASCII символов. MIME определяет объекты мультимедийных данных и создает предпосылки для регистрации организацией Internet Assigned Numbers Authority (IANA) новых форматов данных.

В июне 1992 г. было высказано предложение [Con92] добавить возможности MIME в Web, WAIS и Gopher. В идеале представлялось, что все ресурсы могут быть инкапсулированы с помощью MIME, а протоколы, такие как Gopher и HTTP, будут работать только с MIME-совместимыми данными. Хотя эта идея не была реализована, некоторые концепции MIME [BL92b] оказались полезными для Web:

- **Классификация форматов данных.** Главной концепцией MIME, получившей признание в Web, является формат данных — данные в различных форматах могут пересылаться между отправителями и получателями различными способами. В HTTP формат данных определен как MIME-тип. HTTP использует преимущества механизма расширяемости MIME.

- **Форматы для сообщений, состоящих из нескольких частей.** Способность MIME включать несколько элементарных объектов в одно тело сообщения (этот тип MIME называется «multipart») был с некоторыми расширениями принят в HTTP.

Другие концепции MIME не были приняты:

- Механизм «обогащенного текста (rich text)» для разметки текста MIME был в чем-то схожим с языком разметки Standard Generic Markup Language (SGML), от которого произошел HTML. Возможности представления информации в HTML оказались такими же или даже лучшими.
- Способ обращения к внешним документам. Формат MIME для ссылок на документы мог быть синтаксически отображен на URI (хотя в тот момент они назывались UDI (от Universal Document Identifier)).

Различия между MIME и HTTP можно обобщить следующим образом:

- MIME был предназначен для обмена сообщениями электронной почты. Главное назначение HTTP — обеспечить высокую производительность при любых соединениях.
- Использование и интерпретация определенных полей заголовка в MIME и в HTTP различаются. Например, в HTTP заголовок **Content-Length** используется для указания размера *тела содержимого*, которое следует за полями заголовка в запросе или ответе. В MIME соответствующее поле заголовка является необязательным. При его наличии оно представляет собой метаданные, которые *не* включаются в сообщение. **Content-Length** в MIME не является обязательным полем заголовка, однако в HTTP рекомендуется по возможности включать это поле заголовка.
- В отличие от MIME, в HTTP нет ограничений на длину строки сообщения. Строки в заголовке и в теле HTTP-сообщения могут иметь сколь угодно большую длину.
- HTTP не является совместимым с MIME. В MIME отсутствует понятие кодирования содержания, определенного в протоколах HTTP/1.0 и HTTP/1.1. Заголовок **Content-Encoding** в HTTP (см. раздел 6.2.3) содержит список модификаций представления ресурса. В HTTP не используется заголовок **Content-Transfer-Encoding** MIME.
- Понятие содержимого (entity) в MIME и в HTTP различается. MIME был предназначен для почтовых сообщений, и содержимое включалась в почтовое сообщение. В MIME существуют два типа данных: содержимое и сообщение. При этом не делается различий между сообщениями, отправляемыми между отправителем и получателем напрямую, или же через промежуточные звенья. Первоначально делались попытки адаптировать для HTTP модель содержимого MIME, однако при этом возникли трудности с преобразованиями, применяемыми к содержимому.

Определенное несоответствие имеется и между заявленной универсальностью Web [BL] и зависимостью от регистрации MIME-типов организациями, такими как International Corporation for Assigned Numbers and Names (ICANN). Наличие централизованного официального хранилища требует, чтобы все реализации, ссылающиеся на типы, были согласованными. Создание онлайнового реестра, который может быть доступен через сеть, для добавления нового типа считается наилучшей альтернативой.

Введение терминологии MIME затруднило разделение ролей полей заголовка, используемых в различных контекстах, например, при передаче от клиента к серверу или наоборот. Набор полей HTTP-заголовка стал трактоваться, как произвольный набор строк ASCII, который может игнорироваться Web-компонентами, если последние их не понимают. HTTP-заголовки, как мы увидим в разделе 6.2.3 и в главе 7 (раздел 7.2.2), используются как для предоставления метаданных, так и для модификации поведения и интерпретации методов HTTP.

HTTP И ПРОТОКОЛЫ ARCHIE, GOPHER И WAIS

На разработку HTTP оказали влияние протоколы, которые были популярны во время создания Web. В главе 1 (раздел 1.1.1) обсуждалась роль, которую сыграли системы Archie, Gopher и WAIS в эволюции Web в целом. Различные аспекты, относящиеся к протоколам таких систем, повлияли на принципы, положенные в основу HTTP. В частности, в Gopher [AML+93] имелся протокол клиент-сервер, не сохраняющий состояние и схожий с тем, который был реализован в HTTP. Все серверы используют фиксированные TCP-порты: 70 в Gopher, 210 в WAIS и 80 в HTTP. Идея типа документа в зачаточном состоянии использовалась в Gopher. В HTTP понятие типа содержания было значительно расширено.

Протокол Prospero [NA93], который создал условия для единообразного обращения к множеству файлов в Internet, был применен в Archie. Prospero отличался главным образом своей распределенной файловой системой, которая давала возможность пользователям создавать собственные представления наборов файлов, расположенных в любой точке Internet. Prospero был достаточно простым в смысле разделения доступа к данным и интерпретации данных. Однако он не имел специального протокола для извлечения файлов, для этого в системе использовались другие протоколы, такие как FTP и Gopher. Сам Prospero использовал протокол UDP и дополнительный уровень для обеспечения надежности передачи данных поверх UDP. В противоположность этому, HTTP, Gopher и WAIS обычно используют TCP. В Prospero, в отличие от Gopher, не предусмотрены различные уровни аутентификации. В последних версиях Archie планировалось использовать URI.

Главным назначением протокола Gopher было удаленная выборка документов. Основным различием между Gopher и HTTP является использование гипертекста в HTTP, в Gopher использовались текстовые файлы и меню. HTTP трактует меню и текстовые файлы как особый вид гипертекста [BL92a]. Было бы неправильно сравнивать Gopher с более поздними версиями HTTP, поскольку протокол Gopher не был подвержен значительным усовершенствованиям с того времени, как Web стала доминировать. Хотя использование виртуальных каталогов давало возможность клиентам Gopher кэшировать последние извлеченные ресурсы, в протоколе Gopher отсутствовал какой-либо механизм для идентификации несоответствий между кэшированными элементами и текущим содержанием. Кэширование выполнялось на уровне приложений, вне протокола Gopher. Протокол Gopher был специально разработан, чтобы воспользоваться интеллектуальными возможностями серверов, а не реализовывать их в протоколе. Необходимость предоставления более широкого набора сервисов требовало усовершенствования сервера без внесения изменений в протокол. Хотя при этом протокол является гораздо более простым, чем HTTP, различные реализации серверов могут вести себя по-разному. Gopher определяет набор символов, указывающих, является ли ресурс, к которому осуществляется доступ, текстовым файлом, каталогом, двоичным файлом и т.д. Выбор одного символа для идентификации типа ресурса естественно ограничивает количество различных типов, которые могут быть представлены. HTTP в дальнейшем перешел

к расширенному набору MIME-типов, а различные программные компоненты HTTP получили возможность воспринимать произвольные типы ресурсов. В протоколе Gopher не затрагиваются вопросы безопасности.

HTTP И ДРУГИЕ ПРОТОКОЛЫ ПРИКЛАДНОГО УРОВНЯ

На HTTP оказали влияние другие протоколы прикладного уровня, которые предшествовали Archie, Gopher и WAIS. Сам HTTP тоже повлиял на протоколы прикладного уровня. Ниже мы рассмотрим по одному примеру каждого из таких взаимовлияний.

HTTP перенял идею классов ответов (раздел 6.2.4) от протокола прикладного уровня SMTP [Pos82]. Вместо того чтобы попытаться изобрести новую схему для классификации различных ответов, была использована концепция *функциональной группы* кодов ответов SMTP. Функциональная группа классифицировала ответы на небольшое число категорий, таких как успех, неудача или ошибка. По мере развития HTTP список классов расширялся, но базовая схема осталась неизменной.

HTTP представляет собой только один из протоколов прикладного уровня, наиболее популярный на текущий момент. Существуют другие протоколы прикладного уровня, которые переняли синтаксис и часть семантики HTTP, например, Real Time Streaming Protocol (RTSP) [SRL98], о котором пойдет речь в главе 12 (раздел 12.4).

6.2. Элементы протокола HTTP

В этом разделе мы рассмотрим различные элементы протокола HTTP/1.0. Спецификация HTTP/1.0 (RFC 1945) была гораздо более подробной, нежели спецификация HTTP/0.9. Спецификация включает раздел, подробно описывающий терминологию, грамматику, а также компоненты HTTP-сообщений. Сначала мы познакомимся с несколькими основными понятиями, используемыми в спецификации протокола, а затем обсудим различные элементы. Как упоминалось ранее, протокол — это язык, а ключевыми элементами протокола HTTP являются методы, заголовки и классы ответов.

6.2.1 Термины, относящиеся к HTTP

Наши определения различных терминов HTTP идентичны тем, которые содержатся в двух документах RFC, описывающих HTTP/1.0 и HTTP/1.1 (соответственно RFC 1945 [BLFF96] и RFC 2616 [FGM+99]). Где это необходимо, мы предоставим дополнительные пояснения и примеры. Большинство терминов было определено в главе 1. Здесь мы более подробно остановимся на четырех важных понятиях: сообщение, содержимое, ресурс и агент пользователя.

СООБЩЕНИЕ

HTTP-сообщение представляет собой последовательность байтов, передаваемых по соединению транспортного уровня. Сообщение — это основная единица коммуникационного взаимодействия в HTTP. HTTP-сообщение может быть запросом, передаваемым от клиента серверу, или ответом, отправляемым сервером клиенту. Сообщение-запрос начинается со строки *запроса*, в то время как сообщение-ответ начинается со строки *состояния*. Сообщения запроса и ответа могут иметь нуль или более заголовков, отделенных от необязательного тела сообщения двумя символами: возврата каретки (CR) и перевода строки (LF).

Сообщение-запрос HTTP, показанное на рис. 6.1, имеет следующий синтаксис:

```
Строка запроса
Заголовок (заголовки) общий/запроса/содержимого
CRLF
Необязательное тело сообщения
```

Сообщение-запрос начинается со строки запроса, за которой следует ряд необязательных заголовков и необязательное тело заголовка. Строка запроса содержит метод запроса, запрашиваемый URI и версию протокола клиента. Например, в HTTP-запросе

```
GET /motd HTTP/1.0
Date: Wed, 22 Mar 2000 08:09:01 GMT
Pragma: No-cache
From: gorby@moskvax.com
User-Agent: Mozilla/4.03
CRLF
```

также представленным на рис. 6.1, методом запроса является **GET**, запрашивается ресурс **/motd**, а клиентской версией протокола является HTTP/1.0. Заголовки **Date, Pragma** являются *общими* заголовками, такие заголовки могут присутствовать в запросах и ответах. Заголовки **From** и **User-Agent** являются заголовками запроса и могут присутствовать только в сообщениях-запросах. Общие заголовки, заголовки запроса и заголовки содержимого подробнее будут рассмотрены в разделе 6.2.3. Это сообщение-запрос заканчивается последовательностью CRLF, состоящей из символов возврата каретки и перевода строки. В этом HTTP-сообщении нет информационного содержания (тела содержимого).

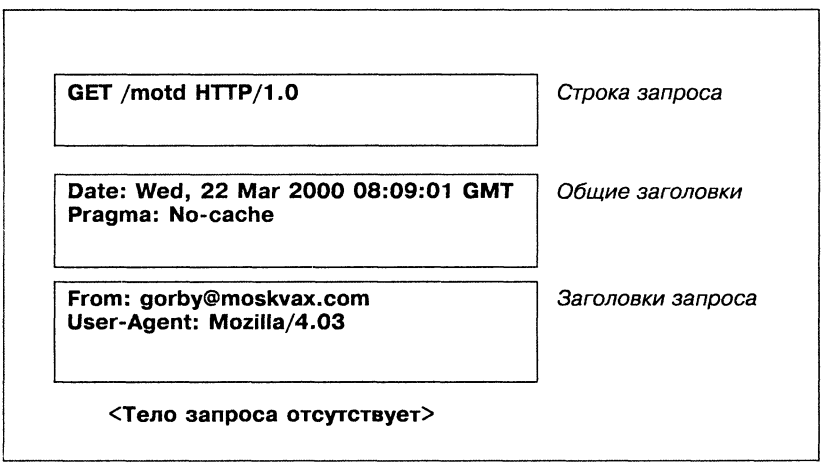


Рис. 6.1. Сообщение-запрос HTTP

На рис. 6.2 показано сообщение-запрос с телом сообщения. Для создания ресурса **/motd** используется метод **PUT**. За методом запроса следует один общий заголовок **(Date)** и два заголовка запроса **(From** и **User-Agent)**. Сообщение-запрос имеет два заголовка содержимого **(Content-Length** и **Allow)**. Первый из них указывает длину содержимого, а второй задает методы, которые могут быть применены к ресурсу **/motd**. Тело содержимого состоит из строки *Welcome to Comer's Vax.*

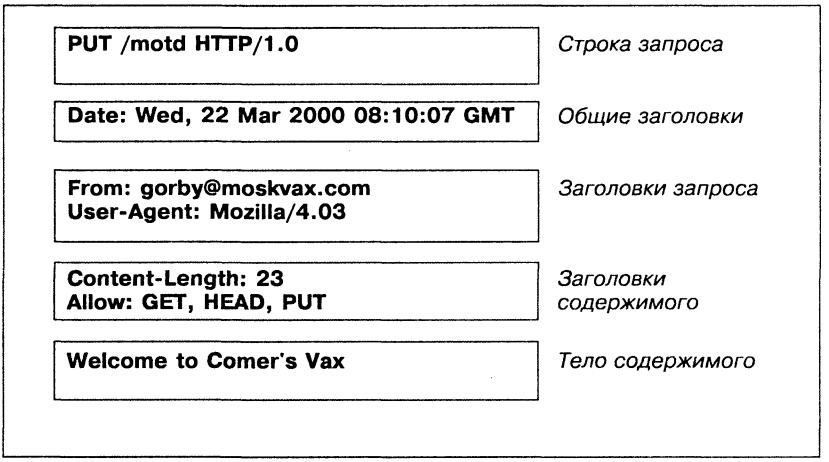


Рис. 6.2. Другое сообщение-запрос HTTP

Сообщение-ответ имеет следующий синтаксис:

```
Строка состояния
Заголовок (заголовки) общий/ответа/содержимого
CRLF
Необязательное тело сообщения
```

Сообщение-ответ начинается со строки состояния, которая содержит номер версии HTTP сервера и код ответа. Далее следуют необязательный общий заголовок, заголовок ответа, а также необязательное тело сообщения. Следует заметить, что из-за наличия промежуточных звеньев окончательное полученное сообщение не обязательно будет отражать номер версии протокола исходного сервера. На рис. 6.3 показано сообщение-ответ для описанного выше запроса **GET**.

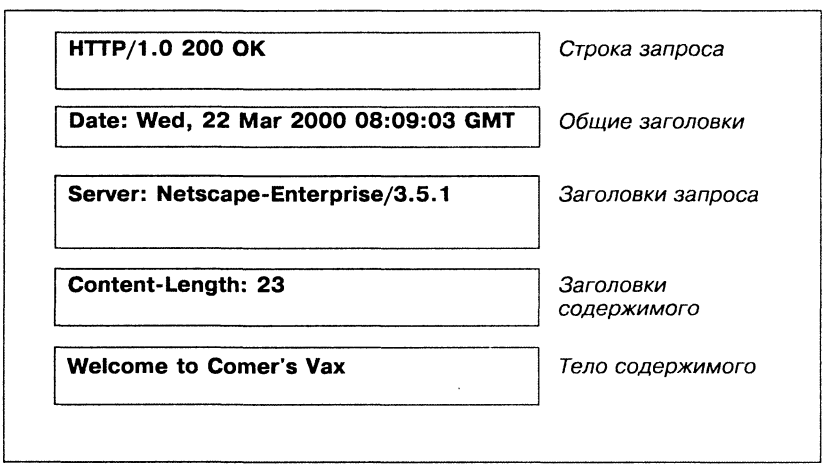


Рис. 6.3. Сообщение-ответ HTTP

```
HTTP/1.0 200 OK
Date: Wed, 22 Mar 2000 08:09:03 GMT
Server: Netscape-Enterprise/3.5.1
Content-Length: 23
CRLF
Welcome to Comer's Vax
```

Строка состояния в сообщении-ответе указывает, что сервер поддерживает HTTP/1.0, а код ответа **200 OK** указывает на успешное выполнение запроса. Сообщение-ответ содержит общий заголовок **Date** и заголовок ответа **Server**. Заголовок содержимого **Content-Length** отражает длину тела содержимого. За последовательностью CRLF следует тело содержимого, которое в этом примере представляет собой строку *Welcome to Comer's Vax*.

Сообщение не имеет тела содержимого, если отсутствует тело сообщения. Тело сообщения в запросе и ответе также называют *телом запроса* и *телом ответа*, соответственно. Не всем сообщениям разрешено иметь тело сообщения.

СОДЕРЖИМОЕ

HTTP/1.0 определяет содержимое как представление ресурса, которое заключено в сообщении запроса или ответа. Содержимое состоит из заголовков содержимого и необязательного тела содержимого. Заголовки содержимого состоят из метаданных о содержимом. Длина тела содержимого может быть задана в заголовке **Content-Length**. Очевидно, что ответ, у которого значение **Content-Length** равно нулю, не будет иметь тела содержимого. Определение содержимого несколько отличается в протоколах HTTP/1.0 и HTTP/1.1 главным образом из-за введения механизма согласования содержимого (описанного в главе 7, разделе 7.9).

Тело содержимого, если оно присутствует, в известном смысле является наиболее важной частью HTTP-сообщения. Для сообщения-запроса телом содержимого могут быть данные, введенные пользователем в HTML-форму. В сообщении-ответе телом содержимого является тело сообщения, т.е. содержимое ответа без заголовков ответа.

РЕСУРС

HTTP/1.0 [BLFF96] определяет ресурс как «сетевой объект данных или сервис, который может быть идентифицирован унифицированным идентификатором ресурса (URI)». Термин «сетевой» означает, что объект данных или сервис могут располагаться в любой точке сети, а доступ к нему осуществляется через сетевое соединение. Решение расширить определение ресурса сервисом имело важное значение. Объект данных может быть статическим или генерироваться динамически, тогда как сервис может представлять собой любое приложение, которое просто использует Web в качестве транспортного средства для инициирования сервиса и предоставления ответа. Например, рассмотрим Web-сайт, который возвращает текущие котировки акций. Цены на акции постоянно меняются, и ресурс **http://www.cnnfn.com/quote=T?** будет выдавать текущую цену акций AT&T. Сервис предоставления котировок может использовать любые внутренние механизмы для получения текущей цены акций. Он просто использует Web как механизм для передачи запроса (определенного набора акций) и ответа (текущей цены каждой из акций).

АГЕНТ ПОЛЬЗОВАТЕЛЯ

Агент пользователя — это клиент, который *инициирует* запрос, и может быть браузером, спайдером или любым другим средством, участвующим в формирова-

нии запроса. Разница между агентом пользователя и клиентом очень важна. Обычно к инициирующему компоненту предъявляются дополнительные требования. Агент пользователя является единственной стороной, непосредственно связывающейся с пользователем, если запрос был инициирован пользователем. Агент пользователя может игнорировать сообщения об ошибке или передать их пользователю, предлагая пользователю сделать выбор: повторить запрос после неудачной аутентификации, переадресовать запрос или указать определенное альтернативное место назначения, либо изменить представление ответа. Например, агент пользователя должен определять, что запрос не содержит необходимой аутентификационной информации, на основе ответа, полученного от сервера. Агент пользователя может затем запросить у пользователя соответствующие данные и повторно отправить запрос, содержащий аутентификационную информацию.

Часто информация об агенте пользователя посылается как часть запроса в заголовке **User-Agent** (описанном более подробно в разделе 6.2.3). Заголовок содержит такую информацию, как названия браузера и операционной системы компьютера. В качестве примера приведем: **Mozilla/3.01 (X11; I; SunOS 5.5 sun4m)** и **Mozilla/2.0 (compatible; MSIE 2.1; AOL 3.0; Mac)**. Имеется возможность идентифицировать версию браузера (Mozilla/3.01 или MSIE 2.1) и операционной системы, применяемой пользователем (например, SunOS 5.5).

6.2.2. Методы запроса HTTP/1.0

Метод запроса уведомляет HTTP-сервер, какое действие следует выполнить над ресурсом, идентифицируемым URI запроса (т.е. URI, указываемом в строке запроса). Наиболее часто используется метод **GET**, который осуществляет выборку текущего содержимого ресурса, идентифицируемого URI. Хотя в HTTP/1.0 определены только три метода (**GET**, **HEAD**, **POST**), в некоторых версиях клиентов и серверов, поддерживающих HTTP/1.0, были реализованы и другие методы, а именно: **PUT**, **DELETE**, **LINK** и **UNLINK**. Все семь методов уже применялись в различных реализациях на момент появления спецификации HTTP/1.0. Поскольку методы **LINK** и **UNLINK** не входят в стандарт HTTP/1.1, они не будут подробно рассматриваться.

Достаточно сложно последовательно описывать методы, поля заголовков и коды ответов, поскольку любое рассмотрение одного аспекта неизбежно приводит к необходимости обращения к двум другим. На данный момент достаточно будет знать, из каких элементов состоит транзакция запрос-ответ:

- Метод запроса включается в клиентский запрос вместе с несколькими заголовками и URI.
- Метод применяется к ресурсу исходным сервером, после чего генерируется ответ. Ответ состоит из кода ответа, метаданных о ресурсе и других заголовков ответа.

Имейте в виду, что промежуточное звено, такое как кэширующий прокси-сервер, получающий запрос и возвращающий кэшированный ответ, *не* применяет метод.

Двумя важнейшими характеристиками метода являются безопасность и идемпотентность. Метод запроса, который лишь просматривает состояние ресурса (например, получает текущее содержимое), считается безопасным методом. Метод, который способен изменить состояние ресурса, не является безопасным. Идемпотентный метод, с другой стороны, имеет то свойство, что побочные эффекты для запроса являются точно такими же, что и для множества идентичных запросов. Другими словами, если последовательно выдается несколько идентичных запросов,

применение метода к ресурсу либо не дает никаких побочных эффектов, либо дает одинаковый побочный эффект во всех случаях. Предсказуемость воздействия метода на ресурс является полезной для определения, должен ли ресурс поддерживать метод. Кроме того, если последовательность запросов идемпотентна, и если соединение было неожиданно закрыто, для клиента будет иметь смысл повторить набор запросов.

Ниже описываются методы HTTP/1.0.

GET

Наиболее популярным на сегодняшний день является метод **GET**. Запрос **GET** применяется к ресурсу, задаваемому URI, а генерируемым ответом является текущее значение ресурса. Этот ответ возвращается обратившемуся с запросом клиенту. Если URI указывает на статический файл, запрос **GET** обычно приводит к чтению файла и возврату его содержимого. Если URI указывает на программу, то в теле ответа возвращаются данные (если они имеются). Метод **GET** является безопасным и идемпотентным. Запрос на CGI-ресурс, например, может вызвать изменение ресурса при применении к нему метода **GET**. Поскольку такой побочный эффект соответствует *намерениям* пользователя, метод считается безопасным.

Запрос **GET** может содержать параметры, которые формируются на основе данных, введенных пользователем. Это часто имеет место при запросе CGI-ресурса. Например,

```
GET http://www.altavista.com/cgi-bin/query?q=foo
```

передает пользовательскую строку запроса ("foo") ресурсу **http://www.altavista.com/cgi-bin**.

Запрос **GET**, содержащий модификатор запроса, может привести к выполнению другого действия. Например, методу **GET** может быть предписано выбирать ресурс только в том случае, если время последней модификации запрашиваемого ресурса больше, чем значение, указанное в заголовке **If-Modified-Since**. Таким образом, запрос **GET** на ресурс **/foo.html** от клиента HTTP/1.0

```
GET /foo.html HTTP/1.0
```

может дать результат, отличный от того, который дает запрос

```
GET /foo.html HTTP/1.0
If-Modified-Since: Sun, 12 Nov 2000 11:12:23 GMT
```

в зависимости от того, когда ресурс **/foo.html** был в последний раз модифицирован на исходным сервере. Модификатор запроса **If-Modified-Since** используется для снижения числа обращений к сети и уменьшения времени ожидания пользователем ответа, генерируемого и отправляемого исходным сервером. Если ресурс не был модифицирован с момента времени, указанного в заголовке **If-Modified-Since**, сервер отправляет код ответа, указывающий на это, не сопровождая ответ телом содержимого. Отметим, что знание времени последней модификации упрощает работу с периодически изменяемыми файловыми ресурсами. Для динамически генерируемого ресурса знание времени последней модификации особого значения не имеет. В HTTP/1.0 имеется несколько полезных модификаторов запроса. В HTTP/1.1 был добавлен еще ряд модификаторов, как мы увидим в следующей главе.

Метод запроса **GET** не имеет тела запроса. Если тело запроса присутствует, серверами оно игнорируется.

HEAD

Метод **HEAD** был задуман для получения метаданных, ассоциированных с ресурсом. В результате выполнения запроса **HEAD** тело ответа не возвращается. Однако метаданные, возвращаемые сервером, могут оказаться теми же метаданными, которые были бы возвращены, если бы методом запроса был **GET**. Например, запрос **HEAD** для получения метаданных, ассоциированных с ресурсом **/foo.html**,

```
HEAD /foo.html HTTP/1.0
```

может вернуть

```
HTTP/1.0 200 OK
Content-Length: 3219
Last-Modified: Sun, 12 Nov 2000 11:12:23 GMT
Content-Type: text/html
```

Ответ состоит из строки состояния (**HTTP/1.0 200 OK**), указывающей на успешное выполнение запроса, и группы заголовков, представляющих метаданные для ресурса **/foo.html**. В этом примере метаданные содержат информацию о длине содержимого, времени последней модификации ресурса и типе ресурса. Метод **HEAD** безопасен и идемпотентен.

Метод **HEAD** в основном используется при отладке для серверов с относительно малой загруженностью, а также для определения, был ли ресурс изменен с момента последней его загрузки. Метаданные могут кэшироваться или использоваться для обновления имеющихся кэшированных данных, если было установлено изменение ресурса. В HTTP/1.0 изменение может быть обнаружено путем анализа значений полей заголовков **Last-Modified** или **Content-Length**. Однако в связи с тем, что ресурс может быть изменен без изменения его длины, анализ одного только поля заголовка **Content-Length** не гарантирует обнаружения изменений.

В HTTP/1.0 метод **HEAD** нельзя использовать с модификаторами запроса, такими как **If-Modified-Since**. Это ограничение было снято в HTTP/1.1.

Метод запроса **HEAD** также не имеет тела запроса. Если тело запроса присутствует, серверами оно игнорируется.

POST

В отличие от методов **GET** и **HEAD**, которые используются для *извлечения* информации, метод **POST** применяется главным образом для модификации имеющегося ресурса или передачи данных обрабатывающему их процессу. Тело запроса содержит данные. Исходный сервер в зависимости от URI запроса, разрешает выполнение определенных действий. Метод **POST** может изменять содержимое ресурса, поэтому не может считаться безопасным методом. Поскольку побочные эффекты множества идентичных запросов могут отличаться, метод **POST** не является идемпотентным методом.

Чтобы модифицировать ресурс, пользователь должен иметь необходимые полномочия. Не все пользователи могут обладать правами на изменение ресурса. Если пользователь имеет право на изменение ресурса, исходный сервер примет новую версию ресурса от клиента. Другое применение метода **POST** состоит в получении тела запроса сервером и использовании его в качестве входных данных для программы, идентифицируемой URI запроса. Такой программой может быть почтовая служба или менеджер доски объявлений, который создает файл, доступный другим приложениям, таким как программа для чтения электронной почты или новостей. Бывает также, что в результате выполнения запроса **POST** ресурсы не изменяются

и не создаются. В этих случаях тело запроса трактуется как входные данные для программы, которая использует эти данные.

Рассмотрим пример:

```
POST /foo/bar.cfm HTTP/1.0
Content-Length: 143

<тело содержимого>
```

Если принимающий сервер может успешно применить метод к ресурсу, он возвращает ответ, указывающий на успешное выполнение запроса. Предположим, что **/foo/bar.cfm** представляет собой ресурс, который не существует на исходном сервере. Сервер создаст ресурс и отправит ответ, указывающий на то, что ресурс был создан. Если, с другой стороны, **/foo/bar.cfm** является программой, ожидающей входных данных, то 143-байтное тело содержимого трактуется как входные данные для программы. Любой выход, генерируемый программой, отправляется обратно пользователю как тело ответа. Следует иметь в виду, что заголовок **Content-Length** для запроса **POST** является обязательным, поскольку он дает возможность принимающему серверу HTTP/1.0 определить, что получен весь запрос.

Метод **GET** также может быть использован для отправки входных данных программе. Однако в использовании для этих целей методов **GET** и **POST** имеются различия. В запросе **GET** входные данные включаются в URI запроса. Предположим, что пользователь заполнил два поля в форме для поиска: искомая строка и база данных, в которой следует вести поиск. Например, запрос

```
GET /search.cgi?string=iktinos&db=greek-architects HTTP/1.0
```

демонстрирует, как данные, введенные пользователем в форме, могут быть включены в URI запроса. Здесь запрашиваемым с помощью метода **GET** ресурсом является **search.cgi**, которому передаются значения двух полей (**string** и **db**). При использовании метода **POST** этот же запрос выглядел бы следующим образом:

```
POST /search.cgi HTTP/1.0
Content-Length: 34
CRLF
query iktinos
db greek-architects
```

Оба запроса выдадут один и тот же ответ. Предположим, однако, что на пути между клиентом и исходным сервером имеются посредники. Посредник обычно регистрирует проходящие через него запросы. Но в то время как URI запроса скорее всего будет занесен в журнал регистрации, тело содержимого вряд ли будет зарегистрировано. Таким образом, в случае запроса **GET** искомая строка будет занесена в журнал, а в случае запроса **POST** — нет. Некоторые посредники и серверы ограничивают длину подлежащего обработке URI, и это может стать еще одной причиной, чтобы предпочесть метод **POST** методу **GET** при передаче данных форм.

PUT

Метод **PUT** схож с методом **POST** в том, что выполнение метода обычно приводит к изменению ресурса, идентифицируемого URI запроса. Если запрашиваемый через URI ресурс не существует, он создается, а если ресурс существует, то модифицируется. При использовании метода **PUT** в результате выполнения запроса изменяется *сам* идентифицируемый URI ресурс.

В HTTP/1.0 метод **PUT** официально не определен. На момент выхода RFC 1945 несколько реализаций клиентов и серверов уже начали использовать этот метод, поэтому метод **PUT** (наряду с методами **DELETE**, **LINK**, **UNLINK**) был вкратце упомянут в приложении RFC 1945. В главе 7 (раздел 7.12.1) мы подробнее поговорим о различиях между методами **PUT** и **POST**. Метод **PUT** не является безопасным методом. Метод **PUT** является идемпотентным, поскольку последовательность идентичных запросов **PUT** будет в каждом случае давать одно и то же содержимое, а побочные эффекты каждый раз будут одинаковыми.

DELETE

Метод **DELETE** используется для удаления ресурса, идентифицируемого URI запроса. Метод предоставляет возможность дистанционного удаления ресурсов. Однако принимая во внимание суть этого действия, исходные серверы контролируют, было ли в действительности выполнено запрашиваемое действие, и когда это произошло. Сервер может отправить ответ об успешном выполнении, в действительности не удалив ресурса. Имеется два вида ответов для успешного выполнения: один указывает на приемлемость запроса для последующей обработки, а другой указывает на реальное выполнение запроса. Подобная гибкость важна для исходных серверов при принятии решения, когда и как планировать действие, а также чтобы не приходилось держать открытым соединение с клиентом до тех пор, пока действие реально не будет завершено. Метод **DELETE** не является безопасным методом. Подобно методу **PUT**, метод **DELETE** является идемпотентным.

LINK и UNLINK

Метод **LINK** позволяет создавать связи между запрашиваемым URI и другими ресурсами. После того, как такая связь создана, можно запрашивать ресурсы по одному и тому же URI запроса. Метод **UNLINK** используется для удаления связей, созданных посредством метода **LINK**.

Хотя эти методы были определены в приложении HTTP/1.0, они не получили широкого распространения и в HTTP/1.1 отсутствуют.

6.2.3. Заголовки HTTP/1.0

Заголовок (или, точнее говоря, поле заголовка) — это ASCII-строка в свободном формате, в котором задано имя, а часто и значение. Заголовки играют важную роль в протоколе HTTP и являются основным средством для указания способа обработки запроса. Заголовки могут использоваться для предоставления метаданных о ресурсе, таких как его длина, формат кодирования и язык. Заголовки можно считать описателями ответа или запроса. Заголовок ответа может указывать, допустимо ли кэширование ответа, либо каким образом декодировать сообщение для получения исходного содержания (например, какой алгоритм сжатия был применен для его преобразования).

Как мы выяснили в главе 5, каждый из протоколов нижних уровней имеет заголовки. В противоположность фиксированному формату заголовков IP и TCP, протоколы прикладного уровня имеют более свободный формат представления заголовков. В протоколах нижних уровней размер пакетов часто ограничивается из соображений производительности. Фиксированный формат заголовков протоколов нижних уровней гарантирует, что сообщения не будут произвольно увеличиваться в результате добавления заголовков. Для протоколов прикладного уровня такой проблемы не существует, добавление новых заголовков является типичным способом добавления новых функций.

В HTTP могут быть определены новые заголовки, которые могут иметь произвольную длину. Механизм расширяемости протокола (вкратце об этом речь пойдет в разделе 6.3) дает возможность отражать новые идеи в виде новых заголовков. Некоторые реализации могут использовать эти новые возможности, а другие — игнорировать нераспознанные заголовки. Однако неограниченный рост числа полей заголовков и длины заголовков увеличивает общий размер сообщений и ведет к увеличению времени ожидания на стороне пользователя, а также к возрастанию загрузки сети. Большое число новых заголовков может также увеличить сложность протокола. Взаимодействие между функциями, которые зависят от различных заголовков, — существенный фактор в увеличении сложности.

HTTP-сообщение может иметь любое число заголовков, отделяемых символами CR и LF. Существуют определенные правила, связанные с передачей и интерпретацией заголовков сообщения по мере прохождения запроса или ответа в сети через серверы-посредники. Определенные типы сообщений запросов-ответов HTTP могут иметь обязательные заголовки. Большинство заголовков не являются обязательными, и Web-компоненты добавляют их по определенным причинам. Web-компоненты могут игнорировать и игнорируют заголовки, которые не являются обязательными. Заголовки, описанные в спецификации, должны быть понятными для всех Web-компонентов: клиентов, серверов-посредников и исходных серверов.

Заголовок имеет общий синтаксис, в соответствии с которым имя и значение отделяются друг от друга символом двоеточия ":". Например, чтобы указать время создания сообщения, заголовок **Date** должен быть включен в сообщение примерно следующим образом:

```
Date: Thu, 23 Dec 1999 08:12:31 GMT
```

"Date" представляет собой поле имени заголовка, а строка "Thu, 23 Dec 1999 08:12:31 GMT" — поле значения. Вот пример заголовка с несколькими полями значений:

```
Accept-Language: de-CH, en-US
```

Здесь поле имени — **Accept-Language**, а поля значений — *de-CH* и *en-US*.

В HTTP имеется иерархия заголовков. Заголовок сообщения в HTTP/1.0 является родовым именем для заголовка и может быть:

- *Общим* заголовком, используемым в сообщениях запросов и ответов.
- Заголовком *запроса*, присутствующим в сообщении-запросе для выражения предпочтительного варианта ответа, для включения дополнительной информации в запрос или для указания ограничений на обработку запроса сервером.
- Заголовком *ответа*, присутствующим в сообщении-ответе для предоставления дополнительной информации об ответе или для запроса дополнительной информации от пользователя.
- Заголовком *содержимого*, присутствующим в сообщениях запроса и ответа. Заголовки содержимого используются для предоставления информации о содержимом, например, время последнего изменения. Если поле общего заголовка или поле заголовка запроса-ответа не распознается, оно трактуется как поле заголовка содержимого.

Если заголовок не распознается получателем сообщения, он просто игнорируется. Однако если получателем является посредник, посредник должен переслать заголовок.

Хотя порядок следования различных полей заголовков не имеет значения, обычно сначала идут поля общих заголовков, потом следуют поля заголовков за-

проса или заголовков ответа, а затем поля заголовков содержимого. Если для заданного поля заголовка указано несколько полей значений, их порядок не должен изменяться посредниками, пересылающими сообщение.

ОБЩИЕ ЗАГОЛОВКИ

Заголовки, которые могут присутствовать в сообщениях запросов и ответов, называются *общими* заголовками. В HTTP/1.0 определены только два поля общих заголовков: **Date** и **Pragma**. Общие заголовки значимы только для самого сообщения, но *не* для содержимого, являющегося частью сообщения. Так, предположим, что запрашивается ресурс **/foo.html**. Заголовок **Date** в соответствующем сообщении-ответе указывает, что сообщение было сформировано в указанное время и не связано с тем, когда было создано или в последний раз модифицировано соответствующее содержимое.

- **Date.** Общий заголовок **Date** указывает на дату и время создания сообщения. Заголовок **Date** имеет такой же синтаксис, что и строка даты/времени в стандарте для текстовых сообщений Internet (документ RFC 822, позднее обновленный документом RFC 1123). Дата в этом формате имеет следующий вид:

  ```
  Date: Tue 16 May 2000 11:29:32 GMT
  ```

  Хотя это наиболее предпочтительный формат, HTTP/1.0 разрешает использование двух других форматов (определенном в документе RFC 1036 и в формате *asctime()* стандарта ANSI C). Формат RFC 1036 выглядит следующим образом:

  ```
  Date: Tuesday, 16-May-00 11:29:32 GMT
  ```

  Формату RFC 1036 присуща проблема, связанная с представлением года двумя цифрами. Как видно из примера, значение года трактуется как 00 вместо 2000. Неоднозначность, связанная с разрешением использования второго формата на основе RFC 1036, была впоследствии устранена в новой версии протокола (HTTP/1.1) в результате усилий по решению проблемы двухтысячного года. В соответствии с форматом **asctime()** стандарта ANSI C дата записывается следующим образом:

  ```
  Date: Tue May 16 11:29:32 2000
  ```

  HTTP/1.0 требует, чтобы клиенты и серверы генерировали строку данных либо в первом, либо во втором формате. Однако клиенты и серверы HTTP могут воспринимать все три формата, что необходимо по той причине, что некоторыми отправителями могут выступать приложения, не отвечающие спецификации HTTP. Спецификация протокола устанавливает правила для взаимодействия Web-компонентов друг с другом. Однако поскольку компоненты могут взаимодействовать и с не-HTTP приложениями, то должен быть определен интерфейс компонентов с ними.

- **Pragma.** Заголовок **Pragma** дает возможность отправлять директивы получателю сообщения. Директива — это способ указать для компонентов определенный вариант обработки запроса или ответа. Вообще директивы не являются обязательными для протокола, хотя в реальности есть несколько специфических директив, которым подчиняются большинство компонентов. Эта разница очень важна, поскольку, хотя протокол требует от компонентов пересылать директивы, он не обязывает компоненты следовать им, если они не являются уместными. Единственной директивой, определенной в протоколе, является

  ```
  Pragma: no-cache
  ```

которая информирует серверы-посредники на маршруте следования сообщения не возвращать кэшированную копию; т.е. отправитель заинтересован в получении ответа непосредственно от исходного сервера. HTTP/1.0 не определяет назначение директивы **Pragma: no-cache** в сообщении-ответе.

ЗАГОЛОВКИ ЗАПРОСА

Заголовок запроса может использоваться клиентом для отправки информации с запросом или задания ограничений при обслуживании запросов сервером. Передаваемая информация может содержать дополнительные сведения о клиенте, например, идентификационные данные пользователя или агента пользователя, либо информацию для авторизации, необходимую, чтобы запрос был обработан исходным сервером. В HTTP/1.0 определены пять заголовков запроса:

- **Authorization.** Заголовок **Authorization** используется агентом пользователя для включения соответствующих полномочий, необходимых для доступа к ресурсу. Для некоторых ресурсов сервера доступ разрешается только при наличии соответствующих полномочий. Вот пример заголовка **Authorization**:

  ```
  Authorization: Basic YXZpYXRpS29IDizM1NA==
  ```

  Здесь *Basic (обычная)* обозначает схему аутентификации, согласно которой данные представляются в виде идентификатора пользователя и пароля. Строка **YXZpYXRpS29IDizM1NA==** представляет собой кодированные идентификатор пользователя и пароль в формате Base64 [FB96a]. Формат использует простой алгоритм кодирования и декодирования, а закодированные данные не намного длиннее исходных данных. Другие допустимые в HTTP схемы аутентификации предусматривают шифрование на транспортном уровне.

- **From.** Заголовок запроса **From** дает возможность пользователю включать в свои идентификационные данные адрес электронной почты. Это полезно для клиентских программ, работающих в качестве агентов (например, агент-робот в спайдере), для идентификации пользователя, в интересах которого функционирует программа. Вот пример заголовка **From**:

  ```
  From: gorby@moskvax.com
  ```

  Следует заметить, что использование заголовка **From** нежелательно, поскольку он нарушает конфиденциальность пользователя, особенно если это делается без его ведома.

- **If-Modified-Since.** Заголовок **If-Modified-Since** является примером условного заголовка, указывающего, что запрос может быть обработан различными способами в зависимости от значения, заданного в поле заголовка. Если предыдущий ответ от сервера был кэширован клиентом или посредником, значение, заданное в заголовке *ответа* **Last-Modified**, используется в последующем запросе GET в качестве аргумента в заголовке **If-Modified-Since**. Например, для следующего запроса:

  ```
  GET /foo.html HTTP/1.0
  If-Modified-Since: Sun, 21 May 2000 07:00:25 GMT
  ```

  сервер будет сравнивать значение, заданное в заголовке **If-Modified-Since**, с текущим значением времени последней модификации ресурса. Время последней модификации ресурса может быть доступно на уровне приложения и зависит от типа сервера. Во многих серверах это значение может быть получено с помощью системного вызова операционной системы (например, *stat()*

или *fstat()* в UNIX). Если ресурс не изменился с указанного времени **Sun, 21 May 2000 07:00:25 GMT**, сервер просто вернет ответ **304 Not Modified**. Для резко меняющихся ресурсов это поможет избежать ненужных пересылок данных. Серверу не нужно повторно генерировать ресурс, и время ожидания на стороне пользователя снижается, поскольку клиент может получить содержимое локально.

- **Referer.** Поле заголовка **Referer**[1] дает возможность клиенту включать URI ресурса, от которого был получен запрашиваемый URI. Например, предположим, что пользователь посещает Web-страницу **http://www.cnn.com** и щелкает на гиперссылке на ресурс **http://www.disasterrelief.org/Disasters/worldglance.html** на этой странице. Заголовок **Referer** в запросе, передаваемом на **http://www.disasterrelief.org**, будет содержать строку **http://www.cnn.com**:

```
GET /Disasters/worldglance.html HTTP/1.0
Referer: http://www.cnn.com
```

Польза от применения поля **Referer** состоит в выявлении устаревших гиперссылок. Однако чаще всего оно становится источником нарушения конфиденциальности пользователя. Ранее (глава 2, раздел 2.6.4 и глава 3, раздел 3.7) мы говорили о проблемах нарушения конфиденциальности, связанных с применением cookies и посредников. Поле **Referer** представляет собой заголовок, который может быть использован исходным сервером для отслеживания действий пользователя через журнал регистрации.

Есть и худшие ситуации. Предположим, что с помощью формы, использующей метод GET (см. раздел 6.2.2) передается номер кредитной карты пользователя. Допустим, что сама форма передается безопасным способом (скажем, с применением SSL), а полученная после выполнения запроса с формой страница содержит ссылку на другую страницу, возможно, находящуюся на сервере S. Теперь, если пользователь переходит к этой странице, журнал регистрации сервера S будет содержать запись с полем **Referer**, содержащим номер кредитной карты пользователя.

Кроме того, сервер может осуществлять проверку поля **Referer** с целью отклонения запросов на определенные ресурсы, если ссылки на ресурсы находились на страницах, не контролируемых сервером. Например, предположим, что ссылка на встроенное изображение **onlymine.gif**, изначально принадлежащее ресурсу A, было скопировано в другой документ, B. Теперь, если браузер попытается отобразить документ B, он должен сформировать запрос на ресурс **onlymine.gif** и включить в запрос заголовок **Referer** с полем значения, соответствующим ресурсу B. Сервер может отказать в обработке запроса, поскольку поле **Referer** не содержит A.

- **User-Agent.** Поле **User-Agent** может быть использовано для включения информации о версии программного обеспечения браузера, версии операционной системы компьютера клиента и, возможно, каких-либо сведений об аппаратной конфигурации. Вот примеры заголовков **User-Agent**:

```
User-Agent: Mozilla/4.03 (Macintosh; I; 68K, Nav)
User-Agent: Mozilla/4.04 [en] C-WorldNet (Win95; I)
```

---

[1] К ряду забавных моментов, связанных с HTTP, можно отнести орфографические ошибки, например **Referer** вместо *Referrer*, с которыми приходится мириться, поскольку они уже получили широкое распространение.

Эта информация достаточно полезна. Например, по ней можно получить статистику по используемым браузерам. Сервер может отправить альтернативную версию ресурса, если ему известно, что программное обеспечение определенного браузера не сможет отобразить версию, используемую по умолчанию. Сомнительность использования этого заголовка состоит в отслеживании действий пользователя и потенциальном вторжении в его частную жизнь. Например, предположим, что несколько пользователей, применяющих различные версии браузеров в большой системе с разделением времени, посылают запросы на исходный сервер. В этом случае поле **User-Agent** может быть использовано для выявления пользователя, пославшего определенные запросы.

ЗАГОЛОВКИ ОТВЕТА

Так же как заголовки запроса используются для отправки дополнительной информации о запросах, заголовки ответа применяются для отправки дополнительной информации об ответе и о сервере, создавшем ответ. Синтаксис строки состояния в заголовке строго фиксирован и не дает возможности включения дополнительной информации. Если заголовок ответа не распознан, он считается заголовком содержимого.

В HTTP/1.0 определены три заголовка ответа:

- **Location.** Заголовок **Location** используется для направления запроса в место расположения ресурса и полезен при переадресации ответов (см. раздел 6.2.4). Заголовок **Location** имеет следующий синтаксис:

  ```
  Location: http://www.foo.com/level1/twosdown/Location.html
  ```

  Поскольку в ответ на запрос могут быть выбраны различные варианты ресурса, заголовок **Location** предоставляет возможность идентификации местонахождения выбранного варианта. Если группа ресурсов реплицирована на нескольких зеркальных сайтах, заголовок **Location** может быть использован для указания на нужный сайт, с которого клиент должен получить ресурс. Если в результате выполнения запроса (например, **POST**) был создан новый ресурс, заголовок **Location** идентифицирует созданный ресурс.

- **Server.** Заголовок ответа **Server** аналогичен заголовку запроса **User-Agent**. Заголовок **Server** несет информацию о версии программного обеспечения исходного сервера и любую другую информацию, связанную с его конфигурацией. Вот несколько типичных примеров заголовка **Server**:

  ```
  Server: Apache/1.2.6 Red Hat
  Server: Netscape-Enterprise/3.5.1
  Server: Apache/1.x.y mod_perl mod_ssl mod_foo mod_bar
  ```

  Значение, указанное в заголовке **Server**, полезно для выявления и решения проблем в ответе, а также для сбора статистической информации, позволяющей определить наиболее популярные версии серверов. Минусом здесь является то, что, данная информация облегчает атаку на сервер. Подобно заголовку **User-Agent**, заголовок **Server** не является обязательным и может не включаться в ответы.

- **WWW-Authenticate.** Заголовок **WWW-Authenticate** используется для указания клиенту, что ресурс требует аутентификации. Клиенту возвращается ответ **401 Unauthorized**, и он может повторно обратиться с запросом, указав соответствующие данные в заголовке **Authorization**. Например, сервер может отправить такой ответ:

```
WWW-Authenticate: Basic realm="ChaseChem"
```

и клиенту следует включить в свой запрос необходимую аутентификационную информацию, как разъяснялось ранее в этом разделе.

ЗАГОЛОВКИ СОДЕРЖИМОГО

Заголовок *содержимого* используется для включения информации о теле содержимого или о ресурсе при отсутствии тела содержимого. Заголовок содержимого является принадлежностью не запроса или ответа, а конкретного ресурса, который запрашивается или отправляется. Заголовки содержимого могут иметься как в запросах, так и в ответах. Нераспознанный общий заголовок, заголовок запроса или заголовок ответа трактуется как заголовок содержимого. Другими словами, протокол задает иерархию для интерпретации полей заголовка. В сообщение запроса или ответа может быть включен новый заголовок, добавляющий новые метаданные о содержимом без опасения, что он будет интерпретирован как общий заголовок, заголовок ответа или заголовок запроса.

В HTTP/1.0 определено шесть заголовков содержимого:

- **Allow.** Заголовок содержимого **Allow** используется для указания списка доступных методов, которые могут быть применены к ресурсу. Он может использоваться и в запросах, и в ответах. Например, метод **PUT** может содержать заголовок **Allow** в запросе, перечисляющий методы, которые могут применяться к ресурсу. При получении запроса на применение некорректного метода к ресурсу исходный сервер отправит в ответ заголовок **Allow** со списком допустимых для этого ресурса методов. Заголовок **Allow** также может быть применен в успешном ответе для указания полного списка приемлемых для данного ресурса методов.

  Рассмотрим пример использования заголовка содержимого в сообщении-запросе

  ```
  PUT /foo.html HTTP/1.0
  Allow: HEAD, GET, PUT
  ```

  Запрос информирует сервер, где будет сохранен ресурс **/foo.html**, что к этому ресурсу разрешается применять методы **HEAD**, **GET** и **PUT**. Другие методы запроса, которые могут попытаться применить клиенты, не должны допускаться исходным сервером.

- **Content-Type.** Заголовок **Content-Type** указывает на тип представления тела содержимого, например, **image/gif**, **text/html** или **application/x-javascript**. Метод **POST** может включать форму со следующим заголовком **Content-Type**:

  ```
  POST /chat/chatroom.cgi HTTP/1.0
  User-Agent: Mozilla/3.0C
  Content-Type: application/x-www-form-urlencoded
  ```

  Различные типы представления содержимого, используемые в **Content-Type**, должны быть зарегистрированы организацией IANA.

- **Content-Encoding.** Заголовок **Content-Encoding** указывает, как было модифицировано представление ресурса, и как оно может быть декодировано в формат, указанный в заголовке содержимого **Content-Type**. Кодирование содержимого выполняется над документами для преобразования их без потерь информации. Типичным примером такого преобразования является

сжатие. Чтобы преобразовать ответ обратно к его исходному виду, получатель должен располагать соответствующей технологией декодирования. Сообщение, содержимое которого было сжато с использованием *gzip*, может содержать заголовок

```
Content-Encoding: x-gzip
```

По мере регистрации IANA могут быть определены новые типы содержания, согласно документу RFC 1590.

- **Content-Length.** Заголовок **Content-Length** указывает на длину тела содержимого в байтах. Длина содержимого важна для обеспечения полной доставки отправленного тела содержимого получателю. Значение **Content-Length** может использоваться в качестве *валидатора* для сравнения кэшированного содержания с текущей его версией. Без заголовка **Content-Length** в запросе клиенту пришлось бы закрывать соединение, чтобы указать на завершение передачи запроса. В разделе 6.2.2 мы рассмотрели пример запроса **POST**, использующего заголовок **Content-Length**. Если ответ генерируется динамически, весь ответ должен быть сформирован до того, как станет известна длина содержимого. Поскольку длина содержимого **Content-Length** является значением поля заголовка и должна быть вставлена в сообщение-ответ перед телом ответа, время ожидания на стороне получателя увеличивается. Обычно для динамических ответов заголовок **Content-Length** опускается. Если динамически сгенерированный ответ был буферизирован перед отправкой, длина содержимого может быть известна. В HTTP/1.0 в качестве индикатора конца содержимого используется закрытие соединения.
- **Expires.** Заголовок **Expires** предоставляет отправителю возможность заявить, что содержимое должно считаться устаревшим после истечения времени, указанного в заголовке. Клиент не должен кэшировать ответ позже даты, заданной в заголовке **Expires**. Разница между *хранением* ресурса в кэше и *кэшированием* весьма существенна. Ответ может храниться (т.е. удерживаться) в кэше и по истечении срока хранения, но он не может быть *возвращен* как ответ без проверки его актуальности на исходным сервере. **Expires** является заголовком содержимого, а не заголовком ответа по той причине, что истечение времени хранения ассоциируется с ресурсом, а не с сообщением. Сервер может отправить сообщение

```
HTTP/1.0 200 OK
Server: Microsoft-IIS/4.0
Date: Mon, 04 Dec 2000 18:16:45 GMT
Expires: Tue, 05 Dec 2000 18:16:45 GMT
```

указывающее, что время хранения ресурса составляет один день.

- **Last-Modified.** Заголовок **Last-Modified** указывает время последнего изменения ресурса. Если ресурсом является статический файл, то этим значением может быть дата последнего изменения ресурса в файловой системе. Динамически генерируемый ресурс может иметь время **Last-Modified**, совпадающее со временем создания сообщения-ответа сервером. Фактически время **Last-Modified** никогда не может быть большим, чем время создания сообщения. Протокол не определяет правила для вычисления времени истечения срока хранения. Заголовок **Last-Modified** в ответе подсказывает получателю сравнить это значение со значением **Last-Modified** кэшированной версии, чтобы проверить, не устарел ли кэшированный ресурс. Наличие заголовка **Last-**

Modified трактуется некоторыми кэшами в HTTP/1.0 как указание, что содержимое *не* было сгенерировано динамически, поскольку заголовок **Last-Modified** для динамически сформированного содержимого не имеет особого смысла. В главе 7 (раздел 7.3.2) мы более подробно обсудим назначение заголовков, связанных с кэшированием.

Сервер может отправить сообщение, подобное следующему:

```
HTTP/1.0 200 OK
Date: Sun, 21 May 2000 08:09:12 GMT
Last-Modified: Sun, 21 May 2000 07:00:25 GMT
```

6.2.4. Классы ответов HTTP/1.0

Каждое сообщение-ответ HTTP начинается со строки состояния, которая имеет три поля: номер версии протокола сервера, код ответа и поясняющая фраза на естественном языке. Запрос на сервере может быть успешно обработан, вызвать отказ в обработке, либо быть переадресованным на другой сервер. В сообщении-запросе могут содержаться синтаксические ошибки, могут возникнуть проблемы при обработке запроса сервером. Различные виды ответов группируются в *классы ответов*. В HTTP каждый из пяти классов ответов имеет несколько кодов ответов, каждый из которых выражается трехзначным целым числом. К пяти классам ответов относятся: информационный класс, коды ответов для которого начинаются с 1 (записываются как 1xx); класс успешного выполнения запроса (2xx); класс переадресации (3xx); класс ошибок клиента (4xx); класс ошибок сервера (5xx).

Идея классов ответов была заимствована у почтового протокола SMTP [Pos82]. Протокол FTP [PR85] имеет схожий механизм кодов ответов. Изначально в HTTP имелось только четыре класса, однако позднее был добавлен информационный класс (1xx). Выбор первоначальных четырех классов ответов был в определенном смысле произвольным, поскольку протокол HTTP отличается от почтового протокола. Однако адаптация хорошо известного и понятного механизма послужила хорошей отправной точкой. Хотя теоретически протокол допускает создание дополнительных классов, на практике имеющиеся реализации могут недостаточно четко использовать новые классы ответов. Правильно написанная реализация будет трактовать код ответа, принадлежащий неизвестному классу ответов, как ошибку.

Помимо числовых кодов ответов в HTTP предусмотрена короткая описательная фраза на естественном языке, называемая *строкой описания (reason phrase)*, которая облегчает отладку и разработку приложений. Строка описания похожа на текст, ассоциированный с кодом ответа в SMTP. Некоторые браузеры отображают строку описания. Однако между кодами ответов в SMTP и в HTTP имеются различия. Каждая из трех цифр в коде ответа SMTP имеет специальное значение, как указано в Приложении E документа RFC 821. Первая цифра обозначает класс ответа, вторая цифра предоставляет несколько более подробную информацию об ответе, а третья цифра дает окончательную степень детализации. В отличие от первой и второй цифр, третья цифра не имеет специального смысла, связанного с ее значением. Вторая цифра в коде ответа SMTP имеет значение 0 для синтаксической информации, 1 для ответов, несущих дополнительную информацию, 2 для информации, относящейся к коммуникационному каналу, и 5 для почтовой информации. В HTTP же вторая и третья цифры не несут какой-либо дополнительной информации. Коды ответов в каждом классе получали последовательные номера.

В то время как коды ответов стандартизованы, строки состояния *не* являются стандартизованными. В различных программах могут использоваться различные

строки, что не оказывает влияния на интерпретацию кода ответа. Например, ответ **404 Not Found** (Не найдено) означает то же самое, что **404 Missing Resource** (Ресурс отсутствует) или **404 Kaanavillai**.

Следует заметить, что хотя не все *коды* ответов понятны для всех HTTP-приложений, *класс* ответа, к которому принадлежит код, должен быть понятным. Класс определяется первой цифрой кода. Каждый класс ответа x имеет x00 в качестве ответа по умолчанию. Если приложение получает код ответа, который ему непонятен, оно ведет себя так, как если бы получило код ответа по умолчанию. Предположим, что сервер-посредник HTTP/1.0 получает код ответа **206 Partial Content**, который был определен в HTTP/1.1. Сервер-посредник HTTP/1.0 может не знать, как должным образом интерпретировать такой ответ. Однако посредник должен трактовать его как код ответа **200 OK** и не должен кэшировать ответ. HTTP-приложение, такое как браузер или сервер-посредник, должно вести себя *предсказуемо*, даже если получает неопознанный код ответа. Заметим, что такой образ поведения неявным образом запрещает добавление новых классов кодов ответов, поскольку старые HTTP-приложения не будут знать, как интерпретировать коды ответов для нового класса. Далее мы подробно рассмотрим все пять классов ответов.

ИНФОРМАЦИОННЫЙ КЛАСС ОТВЕТОВ

Хотя информационный класс ответов был определен в HTTP/1.0, в действительности коды ответов не выделялись до появления HTTP/1.1. Это является примером заботы о будущих нуждах и расширяемости, что является признаком хорошего стиля разработки протокола. Этот класс ответов был явно предназначен для экспериментальных приложений. Ко времени формализации HTTP/1.1 информационный класс кодов ответов, которые были полезными для Web-транзакций, был определен и в настоящее время применяется. Класс 1xx кодов состояния ответов подробнее рассматривается в главе 7 (раздел 7.2.3).

КЛАСС ОТВЕТОВ УСПЕШНОГО ВЫПОЛНЕНИЯ

После того как сервер получил и принял к выполнению HTTP-запрос, он генерирует ответ, принадлежащий к классу успешного выполнения. Это не означает, что результат будет соответствовать ожиданиям клиента. Сервер принял запрос и включает соответствующий ответ на основе метода запроса. Код успешного выполнения никак не связан с наличием или отсутствием тела сообщения. Например, код ответа **200 OK** вполне применим к сообщению с пустым телом, поскольку тело сообщения может иметь нулевую длину.

В HTTP/1.0 имеются следующие четыре кода ответов 2xx:

- **200 OK.** Ответ **200 OK** возвращается, если запрос выполнен успешно. В зависимости от метода запроса в ответ включается различное количество деталей. Например, запрос **GET** приводит к выдаче ответа исходным сервером, применяющим метод **GET** к ресурсу. В случае запроса **HEAD** ответ **200 OK** будет содержать только метаданные.
- **201 Created.** Этот ответ возвращается, если ресурс был успешно создан в результате выполнения запроса **POST**. Хотя это официально и не определено в HTTP/1.0, метод **PUT** может также возвращать ответ **201 Created**, если запрашиваемый URI не существовал и был успешно создан.
- **202 Accepted.** Ответ **202 Accepted** информирует клиента, что запрос был получен, но пока не обработан полностью. Хотя позднее запрос может закончиться неудачей, главное назначение этого кода ответа — дать возможность

агенту пользователя продолжить выполнение своей задачи, не ожидая завершения выполнения действия исходным сервером. Код **202 Accepted** был специально предусмотрен для ситуаций, когда исходный сервер знает, что ответ не будет создан немедленно. Например, запрос может инициировать программу, выполнение которой на исходном сервере занимает много времени, или процесс откладывается для последующего выполнения. Вместе с кодом ответа исходный сервер может предоставить сведения, когда действия могут реально завершиться. Если могут быть сформированы достоверные оценки, то они на естественным языке включаются в тело ответа.

- **204 No Content.** Код **204 No Content** является сигналом от исходного сервера, подтверждающим завершение обработки запроса, и не требует реально видимых пользователем изменений. Предположим, пользователь щелкает мышью на чувствительной области карты изображений, обрабатываемой на исходном сервере; т.е. щелчок на активных участках может инициировать запрос. Если пользователь щелкает на *других* участках изображения карты, отображаемой пользователю, не произойдет никаких изменений. Это достигается за счет отправки сервером ответа **204 No Content**, который браузер интерпретирует как «изменений не требуются».

КЛАСС ОТВЕТОВ, СВЯЗАННЫХ С ПЕРЕАДРЕСАЦИЕЙ

Коды класса ответов, связанные с переадресацией, используются для информирования агента пользователя, что для завершения запроса необходимо дополнительное действие. Код ответа для переадресации может затем привести к успешному ответу. Здесь также имеется риск возникновения бесконечного цикла из последовательных переадресаций. Для предотвращения этой возможности количество переадресаций ограничивается. Ограничения определяются той стратегией, которую реализует сервер. Имеется четыре кода ответов, связанных с переадресацией:

- **300 Multiple Choices.** Код **300 Multiple Choices** главным образом выполняет роль кода ответа по умолчанию. В действительности приложения не используют этот код ответа. Информация о доступности ресурса в другом месте включается в ответ **300 Multiple Choices** с помощью заголовка ответа **Location**. Агент пользователя использует информацию о местонахождении и автоматически пытается извлечь ресурс оттуда.
- **301 Moved Permanently.** Код **301 Moved Permanently** используется для указания, что запрашиваемый ресурс имеет новое местонахождение. Это полезно для автоматической переадресации запросов **GET** и **HEAD** (но не **POST**) другому сайту. Метод **POST**, в отличие от методов **GET** и **HEAD**, может потребовать явного подтверждения от пользователя, поскольку он потенциально изменяет содержимое на исходном сервере.
- **302 Moved Temporarily.** Код **302 Moved Temporarily** является разновидностью ответа **301 Moved Permanently** в том смысле, что запрашиваемый ресурс также был перемещен, но не постоянно. В отличие от ответа **301 Moved Permanently**, клиенты будут продолжать использовать в будущих запросах старый URL. В то время как ответы **301 Moved Permanently** по умолчанию являются кэшируемыми, ответы **302 Moved Temporarily** не кэшируются.
- **304 Not Modified.** Код ответа **304 Not Modified** возвращается, если время последней модификации проверяемого ресурса не изменилось. Это имеет особое значение для кэширования, о чем подробнее рассказывается в главе 11.

КЛАСС ОТВЕТОВ, СВЯЗАННЫЙ С ОШИБКАМИ КЛИЕНТА

Класс 4xx кодов ответов используется для идентификации ошибок, которые могли иметь место на клиенте. Агент пользователя отображает коды ошибок и строки описания пользователю, чтобы он предпринял необходимые действия по исправлению ошибок. В HTTP/1.0 имеются следующие четыре кода ошибок клиента:

- **400 Bad Request.** Этот ответ указывает, что синтаксис запроса был либо некорректен, либо не смог быть распознан исходным сервером.
- **401 Unauthorized.** Если в запросе отсутствует необходимая информация о правах доступа, сервер возвращает код ошибки **401 Unauthorized**. Это может случиться, если запрос либо вообще не содержит какой-либо информации о полномочиях доступа, либо эта информация неверна. Вопросы, связанные с аутентификацией в HTTP/1.0, рассматриваются в разделе 6.4.
- **403 Forbidden.** Код ответа **403 Forbidden** возвращается, если сервер не будет принимать запрос. При этом запрос был понят сервером, но сервер умышленно отказывается его обработать. Причина отказа может включаться сервером в тело содержимого.
- **404 Not Found.** Принимая во внимание большое количество запросов на несуществующие ресурсы, следует признать наиболее широко распространенным кодом ошибки клиента код **404 Not Found**. Этот код возвращается, если исходный сервер не может найти запрашиваемый ресурс. Ошибка может быть обусловлена тем фактом, что ресурс временно недоступен. В HTTP/1.0 не предусмотрено возможностей для указания, когда ресурс снова может стать доступным.

КЛАСС ОТВЕТОВ, СВЯЗАННЫЙ С ОШИБКАМИ СЕРВЕРА

Класс кодов ответов 5xx используется для ошибок, связанных с сервером, или же в случаях, когда сервер знает, что он не может обработать запрос в данный момент. Отличие от класса кодов 4xx состоит в том, что ошибка имеет место на сервере, а клиент не может решить проблему путем отправки альтернативных запросов.

В классе 5xx имеются следующие четыре кода ответов:

- **500 Internal Server Error**. Умалчиваемым для класса 5xx является код **500 Internal Server Error**, который возвращается, если сервер не может точно определить характер ошибки.
- **501 Not Implemented.** Если сервер знает, что у него нет возможности работать с определенным методом запроса, он может вернуть код **501 Not Implemented**. Например, сервер, получивший запрос с методом, определенным в более ранней версии протокола, возвратит этот код.
- **502 Bad Gateway.** Код ответа **502 Bad Gateway** используется, если сервер, действующий в данный момент как посредник или шлюз, не способен обработать ответ, полученный от другого сервера. Это указывает, что отвечающий сервер не был источником ошибки.
- **503 Service Unavailable.** Если сервер временно не может ответить, но надеется ответить позднее, он использует код ответа **503 Service Unavailable**. Сервер может временно быть занят. Этот ответ указывает клиенту, что ему следует попытаться повторить запрос позднее.

6.3. Расширяемость HTTP

Одним из главных принципов при разработке HTTP была идея расширяемости. Заранее предопределенного списка допустимых методов здесь не существует. По мере эволюции протокола могут вводиться новые методы запросов, новые классы ответов и коды ответов. Как мы увидим в главе 7, в HTTP/1.1 было добавлено несколько новых методов, заголовков и кодов ответов. Ресурсы могут иметь любое представление и тип содержания. Отсутствие ненужных ограничений в протоколе дает возможность адаптировать к протоколу новые приложения, которых не было на момент создания протокола. В качестве примера можно привести появление в Web потокового мультимедиа (см. главу 12).

На первых порах существования Web популярностью пользовались несколько браузеров, периодически появлялись их новые версии. Новые пользователи Web начинают с последней версии браузера. Это позволило вводить новые функции, которые являются зависимыми от браузера. Поскольку число пользователей в Web стало очень большим, трудно ожидать, что все пользователи перейдут к последней версии браузера. Было бы непрактично вводить новые функции, зависимые от возможностей последней версии браузера. Серьезным требованием стала совместимость со старыми версиями браузеров.

Расширяемость не означает, что протокол совершенен, или что реализации компонентов не определяют некоторых ограничений. Принцип *разделения ответственности* гласит, что протокол не должен принимать во внимание специфику своей реализации. Реальность рынка диктует принципы эволюции протокола. В главе 7 мы рассмотрим, как протокол эволюционирует в соответствии с требованиями практики.

6.4. SSL и безопасность

Проблема безопасности возникла с того момента, как компьютерные сети стали взаимодействовать друг с другом. Все больше информации доступно через Web, все больше коммерческих организаций стало взаимодействовать через Web. Проблеме безопасности в Web с самого начала было уделено серьезное внимание. Известной проблемой явилось то, что сложные программы труднее защитить, а браузеры и Web-серверы стали достаточно сложными. Исходный код современных браузеров и серверов включает в себя сотни тысяч строк. В типовой Web-транзакции принимают участие несколько программных компонентов. При наличии нескольких посредников безопасного Web-взаимодействия невозможно достичь только защитой клиентского и серверного приложений. Любое из промежуточных звеньев может изменить сообщение или быть уязвимым для перехвата.

Протокол HTTP/1.0 имеет свой собственный механизм безопасности. Однако вместо того, чтобы сделать протокол HTTP зависимым от Web-приложений в обеспечении безопасности, на самом раннем этапе предпринимались попытки обеспечить безопасность Web-взаимодействий на более низком, чем прикладной, уровне. Таков механизм протокола Secure Socket Layer, ставшего широко известным как SSL [SSL95]. В этом разделе мы обсудим SSL и его применение в связке с HTTP. Мы также рассмотрим собственный механизм безопасности HTTP/1.0.

6.4.1. SSL

SSL был разработан Netscape в 1994 г. и стал основой для стандарта IETF Transport Layer Security (TLS) [DA98]. SSL — это протокол между транспортным и прикладным уровнями. Все протоколы прикладного уровня, такие как HTTP, LDAP и Internet Message Access Protocol (IMAP [IMA]), могут использовать SSL. SSL версии 3.0 [SSL], который появился после выявления и устранения недостатков более ранних версий, в настоящее время является текущей версией SSL и составляет основу TLS. SSL поддерживается большинством известных на сегодняшний день браузеров.

Главная цель SSL — обеспечить для клиента, использующего TCP/IP в качестве соединения транспортного уровня, безопасную передачу данных серверу, воспринимающему SSL. Безопасность определяется шифрованием данных, передаваемых между аутентифицированными клиентом и сервером. Естественная форма сообщения преобразуется путем шифрования в форму, которая может быть прочитана только тем, кто способен осуществить расшифровку. Преобразование открытого текста в *зашифрованный текст* (криптографическую версию текста, которую невозможно прочесть) выполняется с помощью алгоритма и *ключа*. Существует множество математических алгоритмов шифрования, обеспечивающих различную степень криптографической устойчивости (т.е. степени сложности для постороннего лица извлечь открытый текст из зашифрованного). Чтобы осуществить шифрование, необходимо применить определенный алгоритм *и* выбрать ключ. Ключом часто является пароль, известный участвующему в коммуникационном взаимодействии программному компоненту, и не являющийся частью сообщения. Надежность ключа определяется количеством битов, необходимых для его представления, поскольку взломщик может попытаться подобрать ключ, испробовав все возможные комбинации. Чрезвычайно сложный алгоритм с небольшим ключом, представленным дюжиной битов, требует рассмотрения 2^{12} (т.е. 4096) значений, что легко может быть реализовано любым компьютером. Однако мощному компьютеру стоимостью несколько миллионов долларов потребуется 10^{25} лет, чтобы рассмотреть все возможные значения 128-битного ключа, использовав метод прямого перебора. Длительность процесса шифрования/дешифрования напрямую не зависит от длины ключа. Следует заметить, что простой алгоритм с длинным ключом более уязвим, чем сложный алгоритм с коротким ключом.

Зашифрованное сообщение, таким образом, может быть передано через сеть, которая является уязвимой для перехвата сообщения. Даже если по перехваченным фрагментам удастся получить копию зашифрованного текста, и известен алгоритм, использованный для шифрования, без ключа преобразование криптограммы в открытый текст будет невозможно. Различные алгоритмы шифрования и их возможности не входят в сферу рассмотрения этой книги. Читателям можно порекомендовать обратиться к другим источникам, затрагивающим эту тему (например, [Ste98a, RGR97]).

Криптография на основе открытого ключа и сертификатов используется сервером для собственной аутентификации. Сертификат в своей основе представляет собой файл, идентифицирующий личность или организацию. Клиент может убедиться, что сервер действительно является тем сервером, который заявлен, проверив сертификат доверенной сторонней организации. Сервер, в свою очередь, может гарантировать тождество клиента, проверив клиентский сертификат. На практике сервер редко проверяет клиентский сертификат. И клиент, и сервер могут иметь небольшой список таких доверенных сторонних организаций, выдающих сертификаты.

Протокол SSL состоит из двух составных частей: установления SSL-соединения и защищенного взаимодействия. Протокол установления SSL-соединения схож

с соответствующим протоколом, используемым в TCP. На фазе установления SSL-соединения протоколы нижнего уровня используются для настройки защищенного взаимодействия и для аутентификации сторон, участвующих в обмене.

Формат передачи данных согласуется на уровне защищенного взаимодействия, которое осуществляет необходимые шифрование, сжатие и реорганизацию данных. Главным назначением защищенного взаимодействия является гарантия для обеих участвующих в обмене сторон, что данные будут зашифрованы при сохранении целостности сообщения. Протокол защищенного взаимодействия разбивает информацию на фрагменты размером не более 16 Кб, при необходимости осуществляя сжатие перед вычислительными операциями по проверке целостности, выполняет шифрование результата и отправляет его вместе с заголовком. При получении сообщения с Web-сервера действия выполняются в обратном порядке: проверка целостности, декомпрессия (необязательно) и доставка, скажем, браузеру.

Отметим, что протокол защищенного взаимодействия никоим образом не связан с тем, что происходит на верхнем (прикладном) уровне. Например, выбор сертификата осуществляется независимо от аутентификации HTTP/1.1.

Первым шагом в коммуникационном взаимодействии с использованием SSL является выбор метода шифрования. Не все методы шифрования одинаково стойкие, и не все пары взаимодействующих между собой клиентов и серверов поддерживают все методы. Таким образом, клиент и сервер сначала выбирают максимально стойкий метод, поддерживаемый обеими сторонами. Практически все клиенты и серверы SSL используют аутентификацию на основе сертификатов, поэтому сервер отправляет подписанный сертификат для аутентификации себя перед клиентом. В некоторых случаях сервер может потребовать, чтобы клиент сделал то же самое, хотя это бывает достаточно редко. Сайты электронной коммерции обычно осуществляют авторизацию пользователя на основе предоставленной им информации о кредитной карте. Сайт электронной коммерции обычно взаимодействует со сторонней компанией (банком-эмитентом кредитной карты), действующей в качестве сертифицирующей организации, и не слишком заботится об идентичности пользователя. Однако в ряде случаев могут иметься причины для того, чтобы потребовать у клиента сертификат. Например, сайт, который хочет максимально обезопасить себя, может настоять на проверке сертификата клиента.

Далее клиент и сервер обмениваются сообщениями, схожими с сообщениями SYN, используемыми в TCP, после чего они готовы начать обмен информационными сообщениями. С целью снизить риск компрометации обменов они также обмениваются хэшами сообщений, используемыми при установлении соединения. С этого момента начинается передача потока данных, зашифрованного с помощью выбранного метода шифрования.

Вопросы, связанные со стойкостью протокола шифрования и органами сертификации, находятся вне сферы рассмотрения этой книги. Читателям можно посоветовать обратиться к другим книгам (например, [Ste98a]), посвященным проблемам безопасности в Web.

6.4.2 HTTPS. Использование SSL при обмене данными в Web

HTTPS — это протокол, который использует SSL для передачи HTTP-сообщений. Вся информация в коммуникационном взаимодействии между клиентом и сервером зашифровывается, включая и URL запроса. Сетевые анализаторы, перехватывающие передаваемый трафик, не смогут определить, к какому ресурсу осу-

ществляется доступ (и с какими значениями параметров, если таковые имеются). Таким образом, запросы защищены от нарушения безопасности и постороннего вмешательства. В отличие от HTTP, в котором обычно используется порт 80 TCP, в HTTPS чаще всего используется порт 443. Необходимость отдельного порта обусловлена наличием выделенного сокета TCP/IP для обмена HTTPS-сообщениями.

Наиболее часто SSL используется совместно с HTTP, хотя могут применяться и другие протоколы прикладного уровня. Пользователи часто даже не замечают, что браузер использует SSL. Главные отличия состоят в том, что URL начинаются с **https:** вместо **http:**, а если процедура установления SSL-соединения заканчивается неудачно, никакие реальные данные между клиентом и Web-сервером не передаются.

Следует отметить, что хотя HTTP является протоколом, не сохраняющим состояния, и HTTP-соединение может быть закрыто путем закрытия TCP-соединения, относящаяся к SSL информация может быть сохранена и повторно использована. Другими словами, пара клиент-сервер может продолжать безопасным образом взаимодействовать, если этим же браузером клиента осуществляется доступ к другому ресурсу на том же сайте. Новый сеанс SSL при этом устанавливать не нужно. В настоящее время ключи могут применяться в течение одного дня.

Многие версии популярных клиентских браузеров, таких как Netscape и Internet Explorer, а также многие популярные Web-серверы поддерживают SSL. Браузеры разрешают пользователям выбирать один из нескольких криптографических протоколов (например, SSL 2.0 или SSL 3.0). После выбора версии протокола возможна дополнительная настройка, такая как выбор метода шифрования. Например, браузер Netscape предоставляет несколько вариантов выбора метода шифрования для SSL 3.0, включая стандарт Data Encryption Standard (DES), принятый Национальным институтом стандартизации США (NIST), и ряд методов, разработанных фирмой RSA Company [RSA], таких как RC2 и RC4.

Является спорным, достаточно ли подготовлено большинство пользователей браузеров, чтобы понимать различия между версиями протокола SSL. Еще менее осознанно они смогут сделать выбор метода шифрования. В связи с этим настройки по умолчанию для сайта имеют важное значение, в конечном итоге они определяют реальный уровень безопасности для пользователей этого сайта. Браузеры также имеют возможность предупреждать пользователей о сайтах, использующих сертификаты, которые не вызывают достаточного доверия, или у которых истек срок действия сертификатов.

Обычно браузеры отображают значок (закрытый замок), если соединение использует SSL.

6.4.3. Безопасность в HTTP/1.0

Поскольку не все транзакции используют SSL, в HTTP/1.0 предусмотрен собственный механизм безопасности. Соглашения по безопасности в HTTP/1.0 главным образом связаны с безопасностью различных методов, с аутентификацией клиента и с защищенной передачей данных. Меньшая часть проблем связана с атаками, нацеленными на пути ресурсов в виртуальной файловой системе сайта с целью доступа к файлам, которые могут быть явным образом не связаны с URL, видимыми с сайта. Например, хотя на Web-странице сайта может иметься гиперссылка на URL **http://www.1729.org/unit1/f1**, пользователь может попытаться осуществить доступ по URL **http://www.1729.org/unit1/f2**. Если такой URL существует, внешний пользователь может осуществить доступ к нему, несмотря на то, что владелец сайта **www.1729.org** мог не предусмотреть такой возможности. Подобные попытки

основаны на известном факте, что ресурсы организованы в файловой системе единообразно, и многие пользователи применяют схожие пути в файловой системе для взаимосвязанных URL. Также имеются более изощренные возможности предположить наличие URL, которые официально недоступны извне. Многие пользователи полагают, что для доступа по URL должна существовать хотя бы одна гиперссылка на Web-странице. Конечно, можно требовать аутентификации для каждого ресурса, предотвращая, таким образом, свободный доступ к ресурсам.

Спецификация протокола учитывает возможность неподобающего использования информации, представленной в журналах регистрации серверов (подробнее об этом говорится в главе 9), что создает условия для вторжения в личную жизнь пользователя за счет раскрытия круга интересующей его информации, частоты обращений и т.д. Заголовки **Referer** и **From** (см. раздел 6.2.3) являются примерами заголовков, неподобающее использование которых способствует нарушению безопасности пользователя. Заголовок **Referer** может быть использован для определения образа действий пользователя, а через заголовок **From** могут быть переданы идентификационные данные пользователя без его согласия. HTTP/1.0 настоятельно рекомендует, чтобы реализации предоставляли способ, дающий возможность пользователям избегать такого нарушения их безопасности.

Одна из основных проблем безопасности, связанная с HTTP, обусловлена использованием CGI-сценариев (глава 4, раздел 4.2.3). CGI служит в качестве основного средства для удаленного вызова сценариев, а с ними сопряжены существенные проблемы, связанные с безопасностью. К таким проблемам относятся побочные эффекты при вызове сценариев, например, злонамеренные пользователи могут указать параметры, которые приведут к переполнению буфера.

Метод считается безопасным, если он приводит только к извлечению ресурса и не приводит к возникновению каких-либо побочных эффектов на исходном сервере. На первом месте по безопасности идут методы **GET** и **HEAD**. Однако вполне можно предположить, что запрос, активизировавший сценарий на исходном сервере (например, CGI-сценарий), приведет к созданию, модификации или удалению ресурса. Суть безопасности состоит в том, что *пользователь* не отвечает за побочные эффекты. Следует заметить, что нет ограничений на использование популярных методов, которые могут вызвать побочные эффекты.

Аутентификация клиента в HTTP/1.0 основывается на типовой схеме вызов-ответ, известной как схема *простой (basic) аутентификации*. Клиенту, желающему осуществить доступ к ресурсу, сервером направляется вызов (в заголовке ответа **WWW-Authenticate**). Клиент должен ответить корректным набором *полномочий*, обычно это пользовательское имя и пароль. Полномочия возвращаются серверу в заголовке **Authorization**, и, если они принимаются сервером, посылается ответ. Однако если полномочия не принимаются, сервер обычно передает ответ **403 Forbidden** или **401 Unauthorized**. К несчастью, в HTTP/1.0 полномочия передаются открытым текстом, что делает их незащищенными.

На одном из первых совещаний IETF, посвященных обсуждению протокола HTTP/1.0, была сформирована отдельная группа по проблемам безопасности [HTT94]. Хотя нельзя сказать, что безопасность вообще не была учтена в HTTP/1.0, в спецификации отсутствовали какие-либо обязательные требования к безопасности. Как мы увидим в следующей главе, частично это было исправлено в HTTP/1.1.

6.5 Совместимость и взаимодействие протоколов

Протокол описывает синтаксис и семантику коммуникационного взаимодействия между двумя или более участниками. Правила, ассоциированные с различными сообщениями, заголовки, которые могут и должны включаться в сообщения, а также коды ответов, ожидаемые в качестве отклика, являются частью спецификации протокола. Совместимость является одной из главных целей при разработке протокола. Компонент с определенной версией протокола (например, HTTP/1.0) подразумевает наличие определенных возможностей по взаимодействию с другими компонентами. При создании новых версий этого протокола различные компоненты должны сохранять обратную совместимость.

Требование совместимости к протоколу может быть простым, например, чтобы строка с номером версии протокола не имела начальных пробелов, это является чисто синтаксическим требованием. Однако требования могут быть и более сложными: сервер, обслуживающий несколько организаций, не доверяющих друг другу, должен проверять значения определенных заголовков, чтобы убедиться, что идентификационные данные пользователя или адрес не были изменены. Не все правила, определенные в протоколе, столь же насущны, как остальные, например, обязательно, чтобы каждый HTTP-запрос содержал метод, не столь существенно, чтобы запрос содержал данные, идентифицирующие пользователя или агента пользователя.

Сначала мы рассмотрим, как интерпретируются номера версии в заголовках сообщений, а затем остановимся на различных уровнях требований к совместимости. Заметим, что требования к совместимости присутствуют только в документах RFC, считающихся стандартами. Например, RFC 1945, описывающий HTTP/1.0, является информационным документом, и в связи с этим не содержит требований к совместимости. RFC 2616, описывающий HTTP/1.1, является проектом стандарта. О принципах совместимости для протокола HTTP/1.1 мы поговорим в главе 15 (раздел 15.3).

6.5.1. Номер версии и совместимость

Когда HTTP-компоненты взаимодействуют друг с другом, номер версии в сообщении указывает на возможности отправителя и получателя. HTTP-сервер должен быть способен интерпретировать все сообщения, имеющие номер версии, меньший или равный его собственному. Требования к интерпретации сообщения затрагивают как синтаксис (формат сообщений), так и семантику (возможности и особенности этой версии протокола). Ответ сервера должен быть представлен в версии протокола, которая понятна для клиента. HTTP-клиент должен правильно указать собственный номер версии и интерпретировать сообщения с сервера в формате версии, равной или меньшей, чем его собственная.

Номер версии HTTP состоит из старшей и младшей части. Например, в HTTP/1.0 старшей частью является 1, а младшей частью — 0. Правила для старшей и младшей частей относительно просты. Предположим, что возможности отправителя расширены путем добавления в HTTP-сообщение семантики, но алгоритм синтаксического анализа сообщений остался неизменным. Младший номер может быть изменен (изменение номера версии предполагает добавление единицы к текущему значению) без изменения старшего номера. Однако если требуется изменение формата сообщения, то должен быть также изменен старший номер. Изменение старшего номера существенно влияет на совместимость.

Детали интерпретации номера версии поясняются в документе RFC 2145 [MFGF97]. Требования к надежности коммуникационного взаимодействия требуют, чтобы компоненты четко представляли, как интерпретировать номер версии в сообщении. Как мы увидим в следующей главе, серверы-посредники вызывают значительные проблемы в интерпретации номера версии.

6.5.2. Уровни требований MUST (обязательный), SHOULD (желательный) и MAY (возможный)

Чтобы гарантировать, что реализации протокола следуют правилам, задаются различные *уровни требований*. Это свойственно многим протоколам, и язык для задания уровней требований был формализован в документе RFC 2119 [Bra96a]. RFC 2119 определяет уровень требований, используемых в спецификациях протоколов Internet. Имеется три уровня требований: **MUST** (обязательный), **SHOULD** (желательный) и **MAY** (возможный). Имеются также отрицательные требования **MUST NOT** (не должен) и **SHOULD NOT** (нежелательно).

Уровни требований к совместимости являются важной составной частью спецификации любого протокола. Без них возможно появление произвольных реализаций, не способных взаимодействовать друг с другом. Разработчики компонентов, таких как клиенты, серверы и посредники, могут без труда идентифицировать те функции протокола, которые должны быть реализованы, и те, которые являются необязательными. Деление на различные уровни по совместимости упрощает задачу разработчикам, поскольку они могут сосредоточиться на реализации только тех составляющих протокола, которые являются строго обязательными.

Обязательный уровень требований (**MUST**) означает, что совместимость абсолютно необходима. Реализация протокола, не обладающая этой характеристикой, *не соответствует* спецификации протокола. Если хотя бы одно из требований уровня **MUST** не соблюдается, реализация *не* является совместимой. Требование уровня **SHOULD** трактуется как рекомендация, и реализация может быть *условно* совместимой со спецификацией, если она не содержит данную функцию. Однако рекомендация подразумевает, что функция по возможности должна быть реализована. Реализация, удовлетворяющая всем требованиям уровней **MUST** и **SHOULD**, считается *безусловно* совместимой.

Требования уровня **MAY** являются дополнительными, и хотя подобные функции могут быть представлены в некоторых реализациях, их наличие не является обязательным. Реализация, отвечающая требованиям уровней **MUST** и **SHOULD**, будет считаться совместимой, даже если в ней не выполнены требования уровня **MAY**.

Различные уровни требований к совместимости оказывают сильное влияние на набор функций, которые могут присутствовать в различных реализациях Web-сервера, посредника или клиента. Хотя некоторые функции являются необязательными, разработчики Web-компонентов должны уделить особое внимание требованиям уровней **MUST** и **SHOULD**. Отсутствие совместимости со стороны одного компонента является основной проблемой для сложных систем, таких как Web, поскольку это может привести к некорректной интерпретации результата другими компонентами. Например, предположим, что сервер игнорирует требование уровня **MUST** и генерирует некорректный ответ. Расположенный на пути сообщения прокси-сервер может предположить, что он получил корректный ответ и кэшировать его. Некорректный ответ будет переправлен как верный ответ многим клиентам, которые запросят этот же ресурс в дальнейшем.

Имеется несколько общих правил совместимости для клиентов и серверов HTTP, посылающих номера версий. Например, HTTP-сервер не должен (MUST NOT) отправлять номер версии, для которой не имеет места по меньшей мере условная совместимость. Если серверу известно о наличии программных ошибок на клиенте, он может (MAY) отправить меньший номер версии. Точно так же, если клиент предполагает, что взаимодействующий с ним сервер содержит ошибки, он может (MAY) отправить меньший номер версии.

6.6. Резюме

Хотя текущей версией HTTP является версия HTTP/1.1 (она будет рассмотрена в следующей главе), полезно начать изучение эволюции протокола HTTP с первых дней возникновения World Wide Web. Что еще более важно, имеется ряд клиентов, серверов-посредников и Web-серверов, которые все еще используют HTTP/1.0. Значительная часть трафика в Web приходится на HTTP/1.0. По-прежнему еще создаются встроенные клиенты HTTP/1.0. Некоторые решения при разработке HTTP/1.0 определялись популярностью других систем, которые широко использовались в конце 80-х и в начале 90-х годов прошлого века. Введение гипертекстовых ссылок способствовало взрывному росту Web и массовому применению протокола HTTP. Систематизация имеющихся методов, заголовков и кодов ответов помогает Web-компонентам без проблем взаимодействовать друг с другом.

Первая версия протокола редко бывает лучшей, в результате анализа эксплуатации клиентов, серверов-посредников и Web-серверов, использующих HTTP/1.0, было извлечено много уроков. К счастью, ключевые идеи, такие как классы кодов ответов, были заимствованы из более устоявшихся протоколов, что помогло избежать ряда проблем. Решение сделать HTTP протоколом без сохранения состояния привело к необходимости введения множества разнообразных заголовков и способствовало расширению протокола в различных направлениях, как мы увидим далее в этой книге.

Применение HTTP/1.0 на практике привело к ряду усовершенствований, от долговременных соединений до безопасности. Эти усовершенствования стали побудительной причиной появления HTTP/1.1. Хотя некоторые популярные Web-браузеры и многие Web-серверы, поддерживающие функционирование популярных Web-сайтов, перешли на HTTP/1.1, версия HTTP/1.0 по-прежнему широко используется.

7

HTTP/1.1

В предыдущей главе мы рассмотрели происхождение и начальный этап эволюции протокола HTTP, приведший к появлению версии, известной как HTTP/1.0. В этой главе мы подробно познакомимся с актуальной на момент публикации версией HTTP, являющийся проектом стандарта. Хотя возможно внесение отдельных незначительных изменений до выхода официального стандарта, мы не думаем, что вероятны сколько-нибудь существенные изменения. Основными источниками информации, на которых основана эта глава, являются RFC 2616 [FGM+99], RFC 2617 [FHBH+99] и архив, включающий тысячи сообщений электронной почты списка рассылки HTTP Working Group [WG-99] с сентября 1994 до момента публикации данной книги.

Эта глава начинается с обзора эволюции протокола HTTP/1.1. Проблемы, связанные с HTTP/1.0, и промежуточные нестандартные реализации привели к тому, что было предложено нескольких усовершенствований. Мы перечислим проблемы, связанные с HTTP/1.0, и сопоставим их с основными семантическими изменениями в HTTP/1.1, представленными в остальной части главы. Перечисление синтаксических отличий между HTTP/1.0 и HTTP/1.1, сгруппированных по методам, заголовкам и кодам ответов, необходимо для понимания семантических отличий. Затем обсуждаются основные семантические изменения, каждое в отдельности, совместно с целями внесения этих изменений. Для большей ясности изложения приводятся примеры, иллюстрирующие новые или подвергшиеся изменениям возможности. Там, где возможно, обсуждаются также и последствия, к которым приведут внесенные изменения, когда получат широкое распространение реализации компонентов, соответствующих HTTP/1.1. Завершается глава описанием роли прокси-серверов, которые должны удовлетворять требованиям HTTP/1.1, предъявляемым как со стороны клиентов, так и серверов.

7.1. Эволюция протокола HTTP/1.1

К тому времени, когда с целью зафиксировать актуальную для того времени практику применения HTTP/1.0 был опубликован документ RFC 1945, уже существовали сотни тысяч Web-сайтов, миллионы пользователей и терабайты передаваемых через Internet гипертекстовых данных. HTTP уже стал основным протоколом Internet по числу передаваемых пакетов и байтов. Повсеместное преобладание Web-документов со встроенной графикой привело к увеличению пользовательского трафика. Доля текста в общем объеме передаваемых данных стремительно снижалась. HTML-документы ссылаются на десятки встроенных изображений от небольших картинок в формате Graphics Interchange Format (GIF), таких как кнопки и значки, до больших иллюстраций и фотографий, дополняющих текст. Всемирная

паутина стала частью образа жизни миллионов людей, быстро развивалась электронная торговля.

К сожалению, некоторые из архитектурных решений, принятых в HTTP/1.0, имели неблагоприятные побочные эффекты. Например, одна из проблем заключалась в том, что протокол прикладного уровня HTTP работал поверх наиболее распространенного протокола транспортного уровня TCP. Хотя TCP считали одним из лучших транспортных протоколов еще на начальной стадии развития Web, даже первоначальная версия HTTP, HTTP/0.9, не требовала использования TCP. Статья 1992 г., в которой описывалось то, что позже стало известно как HTTP/0.9 [BL92a], включала, в частности, следующий фрагмент:

> Примечание. В настоящее время HTTP работает поверх TCP, но сможет работать поверх любого транспортного протокола, ориентированного на соединения.

Так как TCP являлся основным транспортным протоколом с начала 1980-х годов, не удивительно, что фактически все реализации HTTP работали поверх TCP. Однако TCP *не был оптимизирован* для передачи коротких сообщений; большинство Web-данных на ранней стадии развития Web (около 1994 г.) представляли собой файлы объемом менее 10 Кб [BC94]. В этом разделе мы познакомимся с полным списком такого рода проблем в HTTP/1.0.

Мы начнем с истории эволюции HTTP/1.1 и рассмотрения процесса стандартизации протоколов организацией Internet Engineering Task Force (IETF). Затем мы рассмотрим проблемы, связанные с HTTP/1.0, которые привели к появлению HTTP/1.1. Далее исследуется новые концепции, внесенные в HTTP/1.1.

7.1.1. История развития HTTP/1.0

С января 1995 г. по июнь 1999 г. HTTP Working Group прилагала значительные усилия для решения проблем, связанных с HTTP/1.0. За четыре с лишним года эволюции от HTTP/1.0 до HTTP/1.1 вышло несколько версий документов для этого протокола. Дискуссии в HTTP Working Group за этот период доступны в онлайновом архиве [WG-99]. Некоторые из основных изменений протокола, сделанные в результате этих дискуссий, обсуждаются в статье [KMK99]. Движущей силой развития протокола было влияние, оказанное HTTP/1.0 на Internet на начальном этапе, и желание реализовать новые возможности, основываясь на опыте разработчиков различных Web-компонентов. Взрыв популярности Web пришелся именно на этот период.

Переход от HTTP/1.0 к HTTP/1.1 не был ни гладким, ни простым. Некоторые разработчики браузеров и серверов включали в свои продукты возможности, которые не соответствовали общепринятой практике применения HTTP 1.0, по ним не было достигнуто согласия в процессе дискуссий в HTTP Working Group. Как только новые возможности становились частью повседневной практики, на повестке дня в Working Group вставал вопрос об обратной совместимости. Источником многих известных проблем с HTTP/1.1 являлось как раз обеспечение обратной совместимости и ограниченные возможности расширения HTTP/1.0. Тем не менее, протокол все же проделал значительный путь эволюции, на котором было разрешено большое число проблем.

Многие браузеры и прокси-серверы остаются по-прежнему несовместимыми с HTTP/1.1. На группу разработки стандарта HTTP оказывалось значительное давление, связанное с коммерческими интересами, а не с техническими вопросами.

Несколько известных компаний являлись владельцами популярных продуктов. Корпорации, которые были заинтересованы в сохранении определенных возможностей или в продвижении своего подхода, открыто заявляли о своих особых интересах. И все же процесс совершенствования протокола был завершен таким образом, что ни одна из частных компаний не смогла навязать свою особую точку зрения всему Web-сообществу. Даже в тех случаях, когда большая часть рынка принадлежит одному продукту, окончательная версия протокола не выглядит попавшей под влияние решений, реализованных в этом продукте. Одной из причин этого является процедура работы над протоколами в IETF, в основе которой лежит достижение общего согласия. Другая причина — это усилия HTTP Working Group. Однако существующие реализации оказывали влияние на введение некоторых новых возможностей. Одним из примеров является возможность осуществлять запросы на диапазоны данных, что будет рассмотрено в разделе 7.4.1. Существующие реализации оказали также существенное влияние на проблему управления состоянием в HTTP [KM00].

Попытки стандартизации или модернизации любого протокола проходят через ряд последовательных этапов; детальное описание этого процесса представлено в RFC 2026 [Bra96b].

1. Проект документа (Internet Draft) готовится и затем обсуждается с использованием электронных средств информации и на совещаниях, проводимых IETF три раза в год. Проект документа публикуется в электронном виде на Web-сайте IETF.
2. Исполнительный комитет IETF (IESG — Internet Engineering Steering Group) может отнести проект документа к одной из трех категорий: информационной, экспериментальной или стандартам. Если IESG одобряет проект документа, то последний становится официальным документом RFC (Request For Comment) и ему присваивается номер. Информационный документ не должен рассматриваться ни в качестве руководства при реализации, ни в качестве стандарта. Например, документ RFC 1945, в котором описана структура HTTP/1.0, относится к категории информационных документов. Аналогичным образом, спецификация нового протокола Internet Printing Protocol (IPP), описанная в RFC 2565 [HBMT99], относится к категории экспериментальных документов, а не стандарта. Обычно процесс стандартизации требует нескольких версий проекта документа.
3. Проект документа в процессе разработки стандарта проходит через три этапа: предложение по стандарту (Proposed Standard), проект стандарта (Draft Standard) и стандарт Internet (Internet Standard). Предложение по стандарту — это первая ступенька на пути к спецификации протокола, предложение должно иметь ясную структуру, апробированную сообществом, но ни реализация, ни практическое применение не являются на этом этапе обязательными. Проект стандарта должен иметь как минимум две совместимые между собой реализации, разработанные независимо друг от друга. Эти реализации должны включать в себя различные возможности спецификации с тем, чтобы только зрелые и стабильные возможности спецификации достигли уровня проекта стандарта. Наконец, спецификация достигает статуса стандарта Internet после продолжительного тестирования и общего признания полезности этой спецификации для Internet-сообщества. Такой спецификации присваивается и номер RFC, и номер в отдельной серии документов, называемых STD (сокращение от Standard). На каждом этапе стандартизации отводится некоторое время на получение отзывов и комментариев от Internet-сообщест-

ва. Предложение по стандарту должно минимум шесть месяцев оставаться на данной стадии, прежде чем осуществляется переход на уровень проекта стандарта, хотя этот период может продолжаться значительно дольше. На уровне проекта стандарта документ должен находиться четыре месяца или, по меньшей мере, до следующего заседания IETF (независимо от срока), прежде чем можно будет перейти к этапу стандарта Internet.

Процесс эволюции протокола HTTP был осложнен появлением промежуточного документа — предложения по стандарту RFC 2068 [FGM+97], в котором было зафиксировано состояние дискуссий по HTTP/1.1 на то время. К сожалению, документ, представленный на рассмотрение в качестве предложения по стандарту HTTP/1.1, обсуждался дольше, чем обычно, и был принят в качестве предложения по стандарту только в январе 1997 г. Частично из-за неудачного стечения обстоятельств, появление большинства новых версий браузеров совпало по времени с появлением RFC 2068. Вскоре некоторые реализации клиентов и серверов в общем и целом совместимые со спецификацией, описанной в RFC 2068, стали претендовать на то, чтобы считаться совместимыми HTTP/1.1, хотя спецификация протокола HTTP/1.1 была всего лишь на этапе предложения по стандарту, а не стандарта Internet. Этот номер версии быстро стал притчей во языцех, поскольку Web-серверы претендовали на то, что они реализуют «стандарт» HTTP/1.1, который в то время не существовал (и не существует на момент публикации этой книги). Таким образом, хотя спецификация вскрыла существующие проблемы (и зафиксировала их в разделе 19.6.3 RFC 2616), программное обеспечение, реализующее спецификацию, изменялось не так быстро. Тем не менее, появление RFC 2068 послужило поводом для тестирования различных программных компонентов, реализующих спецификацию.

Таким образом, усилия по стандартизации протокола HTTP/1.1 усугублялись дополнительными проблемами, связанными с обеспечением совместимости с браузерами и серверами, реализующими RFC 2068. Проект стандарта RFC 2616 [FGM+99] вышел в июне 1999 г. Предполагалось, что этот проект станет официальным стандартом Internet в 2001. Одна из причин задержки заключается в том, что стандарт IETF не может быть оформлен официально до тех пор, пока из него не будут удалены «непормативные» ссылки. В случае с HTTP/1.1 это ссылка на MIME E-mail Encapsulation (MHTML, RFC 2110 [PH97]), документ который впоследствии был заменен документом RFC 2557 [PH99].

7.1.2. Проблемы, связанные с HTTP/1.0

Считалось необходимым изменить HTTP/1.0 в различных областях. В RFC 2616 (в разделе 19.6.1) перечислено несколько изменений в HTTP/1.1 по сравнению с HTTP/1.0. Более обширный список отличий был представлен в работе [KMK99], здесь используется расширенная версия этой систематики. Полный список проблем, связанных с HTTP/1.0, можно представить следующим образом:

- Отсутствие возможностей управления продолжительностью кэширования, местом кэширования и вариантами специальных запросов и ответов для кэширования.
- Загрузка всего содержимого ресурса, в то время как нужна только небольшая его часть и отсутствие возможности возобновления прерванной передачи данных.
- Недостатки использования в TCP для коротких ответов, типичных для Web.
- Отсутствие гарантии полного получения динамически генерируемых ответов.

- Проблемы расширяемости и отсутствие возможности получать информацию о промежуточных серверах.
- Сокращение резерва IP-адресов из-за привлекательности высокоуровневых доменных имен для Web-бизнеса.
- Неспособность принимать во внимание предпочтения клиентов и серверов и соответственно изменять сообщения запросов и ответов.
- Недостаточность средств защиты при быстро растущих потребностях в защищенных взаимодействиях в Web, а также использование в основной схеме аутентификации, принятой в HTTP/1.0, передачи паролей открытым текстом.
- Неопределенность правил взаимодействия с прокси-серверами и кэшами.
- Разнообразные проблемы, связанные с различными методами, заголовками и кодами ответов.

Значительное увеличение общих задержек в сети, увеличение воспринимаемой пользователем задержки отклика и повышение интереса к безопасности выявили некоторые из проблем, связанных с HTTP/1.0. В некоторых случаях стали использоваться альтернативные средства (например, протокол Secure Socket Layer [SSL] обеспечивает защищенную передачу данных). Однако важные проблемы оставались нерешенными. Группа HTTP Working Group реализовала несколько усовершенствований, уделив внимание правильному применению уровней требований (согласно RFC 2119 [Bra96a]), устранив специфические зависимости от TCP как единственного для HTTP протокола транспортного уровня и удалив практически не используемые методы (рассмотренные в приложении к RFC 1945).

7.1.3. Новые концепции в HTTP/1.1

Кроме попыток решить известные проблемы HTTP/1.0, в HTTP/1.1 были предложены и некоторые новые концепции. Основными концепциями, предложенными для усовершенствования протокола, были следующие:

- Механизм промежуточных передач (hop-by-hop).
- Кодирование при передаче.
- Виртуальный хостинг.
- Обеспечение семантической прозрачности кэширования.
- Поддержка ресурсов, представленных в нескольких вариантах.

В этом разделе мы рассмотрим первые две концепции: механизм промежуточных передач и кодирования при передаче. Эти две идеи широко используются при обмене сообщениями по протоколу HTTP. Другие возможности имеют более ограниченный характер и обсуждаются в последующих разделах. Виртуальный хостинг, обсуждаемый в разделе 7.8, предоставляет возможность нескольким Web-серверам функционировать на одном компьютере, не требуя при этом отдельного IP-адреса для каждого сервера. Поддержка семантической прозрачности стала в HTTP/1.1 приоритетным вопросом. Прокси-сервер, осуществляющий кэширование, при получении запроса от клиента не может вернуть ответ из своего кэша, не проверив его актуальности. Поскольку ресурсы, размещенные на Web-серверах, могут быть представлены в различных вариантах для конкретных клиентов (например, на различных языках, с использованием специфических наборов символов), необходимо было обеспечить выборку из кэша подходящего варианта. Под-

держка семантической прозрачности и управление вариантами содержания были добавлены в HTTP/1.1 и подробно обсуждаются в разделе 7.3.3.

МЕХАНИЗМ ПРОМЕЖУТОЧНЫХ ПЕРЕДАЧ

С точки зрения пользователя Web-серверу посылается запрос, а агенту пользователя возвращается ответ. Это обобщенное представление Web-транзакции. На самом деле это не всегда так, из-за наличия кэшей и прокси-серверов на пути транзакции. HTTP-сообщение может проходить через несколько промежуточных серверов. Заголовки HTTP/1.0 посылаются от отправителя получателю и от получателя отправителю. Предполагается, что промежуточные прокси-серверы, если они не воспринимают поля заголовков, просто пересылают сообщения дальше (этот момент особо выделен в разделе 7.1 RFC 1945 и в RFC 2616).

В условиях широкого распространения прокси-серверов в Web возможно, что два соседних промежуточных звена могли бы использовать альтернативные способы обмена HTTP-сообщениями. Например, они могут использовать алгоритмы сжатия, недоступные другим компонентам в цепочке обмена сообщениями. Эти соседние звенья должны будут включать в заголовки специальные метаданные, отражающие их особые возможности. Однако такие заголовки не смогут быть переданы дальше по цепочке, так как другие получатели на пути от источника к получателю могут не воспринимать те специфические возможности, которыми располагает указанная пара. Становится необходимым механизм, посредством которого произвольный набор заголовков можно было бы надежно пересылать только определенным промежуточным звеньям. Это приводит к введению заголовков промежуточных передач, которые действуют только в одном соединении транспортного уровня. Заголовок **Transfer-Encoding,** введенный в HTTP/1.1, первоначально появился, чтобы обеспечить сжатие данных на промежуточных участках передачи [Fie97]. Заголовки промежуточных передач не могут сохраняться в кэше или передаваться по цепочке прокси-серверами. Получатель заголовков промежуточных передач изымает их, прежде чем передавать сообщение далее. Промежуточные серверы удаляют заголовки промежуточных передач и могут добавлять новые. HTTP/1.1 использует новый заголовок **Connection** для перечисления заголовков, которые должны восприниматься как заголовки промежуточных передач, то есть заголовки, предназначенные только для одного этого соединения, а не для передачи по цепочке. Например,

```
Connection: header 1, header2
```

означает, что сервер, получивший данное сообщение, должен удалить заголовки **header1** и **header2**, прежде чем передавать это сообщение дальше.

Понимание механизма промежуточных передач необходимо для ответа на вопрос о том, какая доля сетевого трафика использует HTTP/1.1. Недостаточно проверить, что клиент и Web-сервер поддерживают HTTP/1.1. Если промежуточные серверы между клиентом и Web-сервером не используют HTTP/1.1, то версия сообщения обычно понижается до предыдущей версии HTTP. Таким образом, сквозной трафик может и не соответствовать HTTP/1.1.

КОДИРОВАНИЕ ПРИ ПЕРЕДАЧЕ

В HTTP/1.1 понятие *содержимого* (entity) было более четко отделено от понятия *сообщения* (message), следующим образом:

- **Сообщение.** Сообщение представляет собой единицу передачи данных в HTTP и содержит заголовки и (необязательно) тело.

- **Содержимое.** То, что реально передается с помощью сообщения. Содержимое подразделяется на заголовки содержимого и тело содержимого.

В HTTP/1.1 было введено представление о *кодировании при передаче* (*transfer-coding*), в отличие от кодирования содержания (content coding) в HTTP/1.0. В HTTP/1.0 не было способа отличить кодирующее преобразование, применяемое к содержимому, от преобразования, применяемого к телу содержимого. Кодирование содержания применяется ко всему содержимому, тогда как кодирование при передаче относится к кодированию, применяемому к телу содержимого. Вид преобразования указываются с помощью значения кодирования при передаче.

Документ с кодированным содержанием часто передается от отправителя к получателю без каких-либо изменений на промежуточных шагах передачи. Получатель декодирует содержание. Например, типичным примером кодирования содержания является сжатие; отправитель сжимает документ, а получатель восстанавливает его. С другой стороны, кодирование при передаче преобразует тело содержимого для того, чтобы надежно транспортировать его по сети. Кодирование при передаче является свойством сообщения, а не исходного содержимого. Поскольку кодирование при передаче является свойством сообщения, оно может добавляться или удаляться любыми промежуточными звеньями. В действительности, кодирование при передаче может применяться к сообщению, содержание которого является закодированным. Механизму кодирования при передаче ничего не известно о семантике сообщения. Выбор конкретного алгоритма кодирования содержания, например, алгоритма сжатия, может зависеть от конкретного типа содержания. Алгоритм кодирования при передаче, наоборот, преобразует сообщение механически, независимо от содержания.

В терминах синтаксиса HTTP-сообщения удаление заголовков сообщения оставляет нам тело сообщения. Но если удалить заголовки сообщения и выполнить декодирование при передаче, то будет получено тело содержимого. В HTTP/1.0 удаление заголовков сообщения непосредственно дает тело содержимого.

На рис. 7.1 показан процесс кодирования сообщения при передаче. Заметьте, что это кодирование не зависит от семантики содержания. Содержание уже могло быть закодировано до того, как было осуществлено кодирование для передачи.

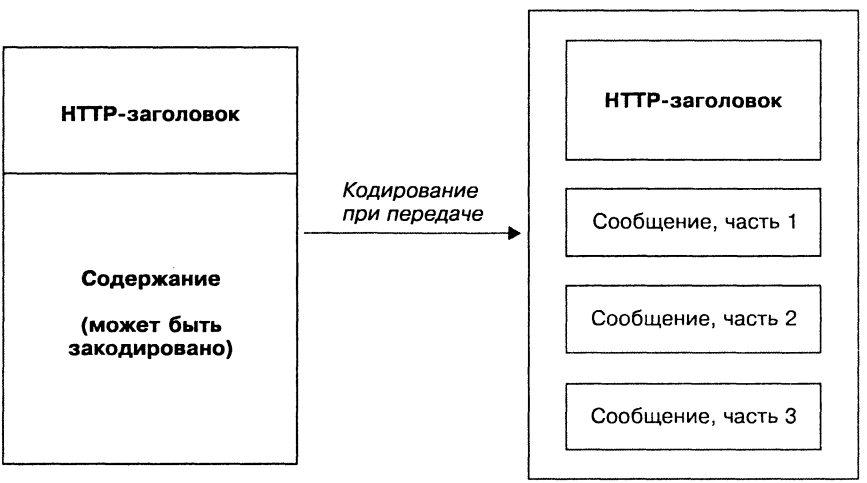


Рис. 7.1. Кодирование сообщения при передаче

На рис. 7.2 показано как сообщение, которое было закодировано при передаче, может быть восстановлено для получения исходного сообщения. Сначала осуществляется преобразование, обратное преобразованию, которое применялось для кодирования при передаче. Затем, если содержание кодировалось, оно декодируется для получения исходного сообщения.

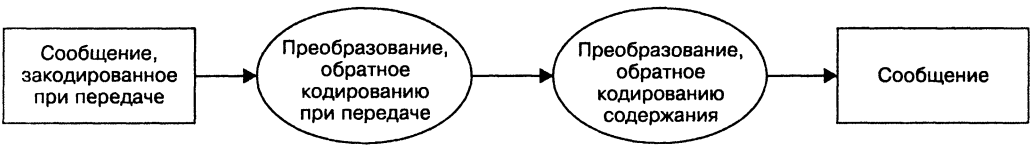


Рис. 7.2. Восстановление исходного сообщения

То, что сообщение закодировано при передаче, определяется в HTTP/1.1 с помощью следующих двух новых заголовков:

- Заголовок запроса **TE** используется отправителем для указания кодирования при передаче, которое приемлемо для него в ответах.
- Общий заголовок **Transfer-Encoding,** который используется для указания кодирования при передаче, примененного к данному сообщению.

Сам механизм кодирования при передаче является механизмом промежуточных передач, так как промежуточные средства могут осуществлять его декодирование и добавлять свое собственное. И **TE**, и **Transfer-Encoding** являются заголовками промежуточных передач. Применение **TE** и **Transfer-Encoding** описано в разделах 7.4.3 и 7.12.2, соответственно.

7.2. Методы, заголовки, коды ответов в 1.0 и 1.1

В этом разделе мы перечислим отличия между HTTP/1.0 и HTTP/1.1. Отличия разделены на три части и относятся к старым и новым методам запросов, заголовкам и кодам ответов. Методы, заголовки и коды ответов составляют большую часть синтаксиса протокола. Анализ новых методов запросов, заголовков, кодов ответов дает общее представление о масштабе отличий между HTTP/1.0 и HTTP/1.1. Добавление заголовков, снижение неопределенности при выборе правильного заголовка и установление правил для задания нескольких заголовков — все это увеличивает и объем, и сложность протокола. Семантика этих изменений обсуждается в следующих разделах. Таблицы служат указателями разделов, в которых обсуждаются семантические изменения.

7.2.1. Старые и новые методы запросов

Как показано в таблице 7.1, в HTTP/1.0 было только три метода запросов (**GET, HEAD** и **POST**). Остальные четыре метода (**PUT, DELETE, LINK, UNLINK**) использовались только в некоторых реализациях HTTP/1.0 и описаны в приложении к RFC 1945. Однако во многих реализациях эти методы отсутствуют. На самом деле, даже в тех продуктах, в которых эти методы есть, они не были реализованы единообразно. Два из этих четырех методов (**PUT** и **DELETE**, помеченные в таблице 7.1 звездочкой) были сохранены и формально определены в HTTP/1.1. Другие два метода (**LINK** и **UNLINK**) в HTTP/1.1 отсутствуют. В HTTP/1.1 введены три

новых метода запросов (**OPTIONS, TRACE** и **CONNECT**). Изменения в методах HTTP/1.0 настолько значительны, что заслуживают отдельного обсуждения, которое мы оставляем до последующих разделов этой главы.

Являясь информационным, RFC 1945 не определял уровни требований к методам. Состояние требований, связанных с различными методами в HTTP/1.1, документировано в RFC 2616.

Таблица 7.1. Методы запросов в HTTP/1.1.
Числа в третьем столбце указывают номера разделов в данной главе. С помощью '*' помечены методы HTTP/1.0, а с помощью '†' обозначены нестандартные методы HTTP/1.0.

Метод	Новое/изменение в HTTP/1.1	Раздел
GET*	Запрос частей содержимого	7.4.1
HEAD*	Разрешены заголовки условных запросов	7.12.1
POST*	Управление соединением, пересылка сообщений	7.5, 7.6
PUT†	Метод формализован	7.12.1
DELETE†	Метод формализован	7.12.1
OPTIONS	Новый метод, расширяемость	7.7
TRACE	Новый метод, расширяемость	7.7
CONNECT	Новый метод, для будущих применений	7.12.1

В соответствии с требованиями только методы **GET** и **HEAD** должны быть реализованы обязательно (уровень требований MUST). Иначе говоря, браузер, корректно реализующий эти два метода, сможет взаимодействовать с Web-сервером (также реализующим только эти два метода) и оба смогут претендовать на совместимость с HTTP/1.1. Конечно, сервер должен возвращать корректные коды ошибок для методов, которые в нем не реализованы. На деле в большинстве браузеров и серверов реализовано значительно больше, чем только эти два обязательных метода.

7.2.2. Старые и новые заголовки

Заголовки — это основное средство изменения режима запроса и предоставления метаданных о ресурсе. В следующих разделах мы рассмотрим заголовки: общие, запросов, ответов, содержимого и промежуточных передач.

ОБЩИЕ ЗАГОЛОВКИ

В HTTP/1.0 определены только два общих заголовка (**Date** и **Pragma**). В HTTP/1.1 имеются семь дополнительных общих заголовков. Общие заголовки могут использоваться в запросах, в ответах и обычно относятся к более общим понятиям, чем заголовки запросов и ответов. В таблице 7.2 перечислены общие заголовки и связанные с ними основные понятия. Мы обсудим новые общие заголовки позже в этой главе.

Заголовки **Date** и **Pragma**, о которых шла речь в главе 6 (раздел 6.2.3), присутствовали и в HTTP/1.0. Список допустимых форматов даты и времени был изменен, из него были удалены ранее допустимые в RFC 1036 форматы даты. В RFC 1036 допускалось указание года в виде строки из двух цифр, что очевидно неприемлемо в свете проблемы двухтысячного года (для представления года требуются четыре, а не две цифры). Таким образом, хотя клиенты и серверы по-прежне-

му могут принимать данные во всех трех форматах (RFC 822/1123, RFC 850/1036 и формат *asctime()*, принятый в ANSI C), генерировать они могут только строки дат в формате RFC 1123.

Таблица 7.2. Общие заголовки в HTTP/1.1.
С помощью '*' помечены общие заголовки HTTP/1.0.

Заголовок	Определение/семантическое понятие(ия)	Раздел
Date*	Дата/время создания сообщения	6.2.3
Pragma*	Директива сообщения	6.2.3
Cache-Control	Директива кэширования	7.3.3
Connection	Управление соединением при промежуточных передачах	7.1.3, 7.5
Trailer	Список заголовков в конце сообщения	7.6
Transfer-Encoding	Преобразование тела сообщения	7.6
Upgrade	Переход на другие протоколы	7.7
Via	Получение информации о промежуточных серверах	7.7
Warning	Уведомление об ошибке	7.12.2

В семантику общего заголовка **Pragma** не внесено никаких изменений, к нему не добавлено никаких новых директив. Протоколом не выдвигается требований к использованию директивы **Pragma**. В HTTP/1.0 **Pragma: no-cache** определено только для сообщений-запросов, в HTTP/1.1 действие директивы не расширено на сообщения-ответы. HTTP/1.1 разрешает использование **Pragma** для обеспечения обратной совместимости и вводит более общий механизм управления кэшами посредством общего заголовка **Cache-Control**. Заголовок **Cache-Control** используется, чтобы передавать директивы в запросах и ответах, а также управлять поведением компонентов кэша.

ЗАГОЛОВКИ ЗАПРОСОВ

Как видно из таблицы 7.3, в HTTP/1.0 было 5 заголовков запросов, а в HTTP/1.1 их 19. Все 5 заголовков запросов из HTTP/1.0 (**Authorization, From, If-Modified-Since, Referer** и **User-Agent**) оставлены в HTTP/1.1, хотя семантика некоторых из них изменена. В частности, возможности аутентификации доступа, описываемые заголовком **Authorization**, в HTTP/1.1 значительно расширены. Существует отдельный RFC, в котором описаны обычная (Basic) аутентификация и аутентификация с помощью дайджестов [FHBH+99]. В HTTP/1.0 модификатор запроса **If-Modified-Since** можно было использовать только с методом **GET**, а в HTTP/1.1 его можно использовать с любым методом.

Мы подразделяем заголовки запросов на четыре класса, следующим образом:

- **Предпочтения в ответе.** Заголовки, используемые для передачи дополнительной информации о предпочтениях в ответах, например, о языке или наборе символов.
- **Информация, передаваемая вместе с запросом.** Заголовки, используемые для передачи в сообщении дополнительной информации, например, информации идентифицирующей пользователя и/или браузер.

- **Условные запросы.** Заголовки, используемые для условных запросов, которые меняют интерпретацию сервером данного метода запроса.
- **Ограничения, накладываемые на сервер.** Заголовки, используемые для получения определенной реакции сервера.

14 новых заголовков запросов в HTTP/1.1 обсуждаются в соответствующих разделах позже в данной главе.

Таблица 7.3. Заголовки запросов HTTP/1.1.
Числа в последнем столбце указывают номер раздела в данной или в предыдущей главах. С помощью '*' помечены заголовки запросов HTTP/1.0.

Класс	Заголовок	Краткое описание (Раздел)
Предпочтения в ответе	**Accept**	Предпочтительные типы содержания (7.9)
	Accept-Charset	Предпочтительные наборы символов (7.9)
	Accept-Encoding	Предпочтительные схемы кодирования содержания (7.9)
	Accept-Language	Предпочтительные языки (7.9)
	TE	Предпочтительные схемы кодирования при передаче (7.4.3)
Информация, передаваемая вместе с запросом	**Authorization***	Полномочия агента пользователя (6.2.3)
	From*	Адрес электронной почты пользователя (6.2.3)
	Referer*	URI, от которого получен URI запроса (6.2.3)
	User-Agent*	Информация о программном обеспечении агента пользователя (6.2.3)
	Proxy-Authorization	Авторизация клиента прокси-сервером (7.11.3)
Условный запрос	**If-Modified-Since***	Сравнение с временем последнего изменения (6.2.3)
	If-Match	Сравнение атрибутов содержимого (7.3.3)
	If-None-Match	Сравнение атрибутов содержимого (7.3.3)
	If-Unmodified-Since	Сравнение с временем последнего изменения (7.4.1)
	If-Range	Отправить диапазон, только если содержимое не изменено (7.4.1)
Ограничения, накладываемые на сервер	**Expect**	Реакция сервера, ожидаемая клиентом (7.4.2)
	Host	Хост запрошенного ресурса (7.8)
	Max-Forwards	Допустимое число переходов при передаче данных (7.7.1)
	Range	Запрос диапазона содержимого (7.4.1)

ЗАГОЛОВКИ ОТВЕТОВ

Заголовки ответов используются для предоставления дополнительной информации о сервере или о самом ответе. Заголовки ответов были изменены в HTTP/1.1 двумя способами: существующие в HTTP/1.0 заголовки ответов (**Location, Server** и **WWW-Authenticate**) были уточнены, кроме того, были добавлены шесть новых заголовков ответов. Проведено различие между заголовком ответов **Location** и новым заголовком содержимого **Content-Location**.

Полный список заголовков ответов в HTTP/1.1 приведен в таблице 7.4.

Таблица 7.4. Заголовки ответов HTTP/1.1.
С помощью '*' помечены заголовки ответов HTTP/1.0.

Понятие	Заголовок	Краткое описание	Раздел
Переадресация	**Location***	Альтернативное местоположение для URI	6.2.3
Дополнительная информация	**Server***	Идентификация сервера	6.2.3
	Retry-After	Величина задержки перед повторной попыткой запроса	7.12.2
	Accept-Ranges	Частичный запрос	7.4.1
Безопасность	**WWW-Authenticate***	Запрос информации для аутентификации	6.2.3
	Proxy-Authenticate	Запрос информации для аутентификации	7.11.3
Кэширование	**ETag**	Проверка непрозрачности	7.3.3
	Age	Время с момента создания ответа	7.3.3
	Vary	Выбор варианта ресурса	7.3.3

ЗАГОЛОВКИ СОДЕРЖИМОГО

Заголовки содержимого используются для предоставления дополнительной информации о ресурсе или о теле содержимого и имеют ту же семантику, что и в HTTP/1.0. В дополнение к шести уже имевшимся в HTTP/1.0 заголовкам содержимого (**Allow, Content-Encoding, Content-Length, Content-Type, Expires, Last-Modified**) в HTTP/1.1 добавлены четыре новых заголовка. Хотя протокол не нужно менять для того, чтобы определить новые поля заголовка, получатели могут не воспринимать новые поля. Нераспознанные поля заголовка содержимого (как и полностью нераспознанные заголовки) получателями игнорируются. Прокси-серверы могут их игнорировать, но, по-прежнему, обязаны передавать нераспознанные заголовки дальше. В таблице 7.5 приведен список заголовков содержимого в HTTP/1.1.

Таблица 7.5. Заголовки содержимого в HTTP/1.1.
С помощью '*' помечены заголовки содержимого в HTTP/1.0.

Заголовок	Краткое описание	Раздел
Allow*	Методы, применимые к ресурсу	6.2.3
Content-Encoding*	Кодирование содержания, примененное к телу содержимого	6.2.3
Content-Length*	Длина тела содержимого	6.2.3
Content-Type*	Тип тела содержимого	6.2.3
Expires*	Продолжительность актуальности тела содержимого	6.2.3
Last-Modified*	Время последней модификации ресурса	6.2.3
Content-Language	Язык содержимого	7.9
Content-Location	Альтернативный адрес ресурса	7.9
Content-MD5	Контроль целостности тела содержимого	7.10
Content-Range	Положение диапазона в теле содержимого	7.4.1

ЗАГОЛОВКИ ПРОМЕЖУТОЧНЫХ ПЕРЕДАЧ

В HTTP/1.1 существует восемь заголовков промежуточных передач: **Connection, Keep-Alive, Proxy-Authenticate, Proxy-Authorization, Trailers, TE, Transfer-Encoding** и **Upgrade**. Остальные заголовки являются сквозными и *не могут быть преобразованы* в заголовки промежуточных передач простым включением их имен в заголовок **Connection** в расчете на то, что при следующем соединении они будут изъяты. Новые заголовки могут быть введены и преобразованы в заголовки промежуточных передач включением их в заголовок **Connection**. Например, если в сообщении определить новый заголовок **Volume**, то он будет трактоваться как сквозной и будет передаваться промежуточными серверами далее по цепочке, даже если они распознают этот заголовок. Однако если добавить заголовок,

```
Connection: Volume
```

то этот заголовок будет изъят, как только он достигнет следующего получателя.

7.2.3. Старые и новые коды ответов

Смысл 16 кодов состояния ответов из HTTP/1.0 (об этом шла речь в главе 6, раздел 6.2.4) остается неизменным в HTTP/1.1. Всего в HTTP/1.1 существует 41 код ответа, как показано в таблице 7.6.

Таблица 7.6. Коды состояний ответов HTTP/1.1.
С помощью '*' помечены коды ответов HTTP/1.0.

100 Continue	404 Not Found*
101 Switching Protocols	405 Method Not Allowed
200 OK*	406 Not Acceptable
201 Created*	407 Proxy Authentication Required
202 Accepted*	408 Request Timeout
203 Non-Authoritative Information	409 Conflict
204 No Content*	410 Gone
205 Reset Content	411 Length Required
206 Partial Content	412 Precondition Failed
300 Multiple Choices*	413 Request Entity Too Large
301 Moved Permanently*	414 Request-URI Too Long
302 Found*	415 Unsupported Media Type
303 See Other	416 Requested Range Not Satisfiable
304 Not Modified*	417 Expectation Failed
305 Use Proxy	500 Internal Server Error*
306 (Unused)	501 Not Implemented*
307 Temporary Redirect	502 Bad Gateway*
400 Bad Request*	503 Service Unavailable*
401 Unauthorized*	504 Gateway Timeout
402 Payment Required	505 HTTP Version Not Supported
403 Forbidden*	

В HTTP/1.1 в каждом классе зарезервировано несколько новых кодов состояния. Мы рассмотрим некоторые из самых важных новых кодов состояния и объясним, для чего их потребовалось ввести в HTTP/1.1. Вместе с тем, не было определено ни одного нового класса кодов состояния — новые классы не воспринимались бы клиентами и прокси-серверами, работающими по протоколу HTTP/1.0. Клиенты, работающие по протоколу HTTP/0.9, не воспринимают коды ответов, поскольку они получают только тело содержимого.

В одной и той же ситуации коды состояния ответов не всегда одни и те же: сервер может посылать различные коды состояния, основываясь на текущем состоянии и на политике обработки запросов данного сервера. Выбор кода состояния, осуществляемый сервером, дает не слишком много информации для понимания конкретной причины сообщения об ошибке.

Трудно равномерно распределить коды ответов по конкретным семантическим категориям. HTTP/1.0 не дает особых указаний, какой код ответа следует вернуть для данной ситуации. В основном этого не делает и HTTP/1.1. Таким образом, хотя и существует некоторая свобода в том, какой код в пределах конкретного класса следует вернуть, не может быть сомнений относительно класса кода ответов, подходящего в данной ситуации. Например, коды ответов **200 OK, 201 Created** или **204 No Content** могут быть переданы в ответ на запрос **PUT**. Подобным же образом сервер, который не хочет, чтобы клиент имел возможность различать ситуации, когда ресурса не существует и когда у данного клиента нет прав доступа к нему, может использовать вместо ответа **403 Forbidden** ответ **404 Not Found**. Если известна информация об особенностях функционирования агента пользователя, сервер может посылать разные коды ответов. Например, некоторые агенты пользователя, созданные до появления HTTP/1.1, не воспринимают ответ **303 See Other**, но при получении ответа **302 Found**, они реагируют точно так же, как должны были бы в соответствии со спецификацией реагировать на ответ **303 See Other**. Эту информацию можно использовать и передавать такому агенту пользователя ответ **302 Found** вместо ответа **303 See Other**.

КЛАСС (1XX) КОДОВ ОТВЕТОВ HTTP/1.1. ИНФОРМАЦИОННЫЕ СООБЩЕНИЯ

В HTTP/1.0 (RFC 1945), не определено никаких кодов информационного класса (1xx), тогда как в HTTP/1.1, как показано в таблице 7.7, зарезервировано два новых кода ответов этого класса. В HTTP/1.0 была правильно предугадана необходимость этого класса ответов. Фактически документ RFC 1945 даже включал ограничения на ответы класса (1xx) (ответы не могли иметь тел), хотя сами коды состояния класса (1xx) в то время отсутствовали.

Таблица 7.7. Класс (1xx) кодов состояния ответов (информационные сообщения)

Код состояния и строка сообщения	Краткое описание	Раздел
100 Continue	Передача запроса разрешена	7.4.2
101 Switching Protocols	Переключение на другой протокол	7.7

КЛАСС (2XX) КОДОВ ОТВЕТОВ HTTP/1.1. УСПЕШНОЕ ЗАВЕРШЕНИЕ

Класс 2xx объединяет семь кодов ответов, четыре из которых перешли из HTTP/1.0. В таблице 7.8 приведены старые и новые коды, а также разделы, в которых обсуждается назначение новых кодов ответов.

Таблица 7.8. Класс (2xx) кодов состояний ответов (успешное завершение). '*' обозначены коды ответов HTTP/1.0.

Код состояния и строка сообщения	Краткое описание	Раздел
200 OK*	Запрос принят	6.2.4
201 Created*	Добавление вариантов	7.3.3
202 Accepted*	Временный ответ	6.2.4
203 Non-Authoritative Information	Метаданные не полны	7.12.3
204 No Content*	Не изменено пользовательское представление	7.12.3
205 Reset Content	Изменено пользовательское представление	7.12.3
206 Partial Content	Успешный запрос на диапазон	7.4.1, 7.12.3

КЛАСС (3XX) КОДОВ ОТВЕТОВ HTTP/1.1. ПЕРЕАДРЕСАЦИЯ

Коды ответов класса 3xx используются для определения альтернативных действий, которые могут быть предприняты агентом пользователя для выполнения данного запроса. Старый код состояния ответов, связанный с переадресацией в HTTP/1.0, **300 Multiple Choices** имеет в HTTP/1.1 несколько иное значение. Значение другого кода состояния HTTP/1.0, **302 Found**, было также изменено. Кроме того, в HTTP/1.1 добавлены четыре новых кода состояния, относящихся к переадресации. В таблице 7.9 показаны все коды ответов класса 3xx и указаны разделы, в которых обсуждаются изменения.

Таблица 7.9. Класс (3xx) кодов состояний ответов (переадресация). С помощью '*' помечены коды ответов HTTP/1.0

Код состояния и строка сообщения	Краткое описание	Раздел
300 Multiple Choices*	Основной код ответа	7.9
301 Moved Permanently*	Новое местоположение ресурса	6.2.4
302 Found*	Изменена строка описания кода состояния	6.2.4
303 See Other	Некорректная обработка ответа 302	7.12.3
304 Not Modified*	Подтверждение актуальности ответа	6.2.4
305 Use Proxy	Повторить запрос через прокси-сервер	7.12.3
306 (Unused)	Код вышедший из употребления	7.12.3
307 Temporary Redirect	Временное изменение URI	7.12.3

Хотя агент пользователя может не отображать строку описания (фраза на естественном языке, поясняющая код состояния), ассоциированную с ответом, эти строки являются хорошим способом дать описание конкретных кодов ответов. Строка сообщения для кода состояния 302 была изменена при переходе от HTTP/1.0 к HTTP/1.1. Исходная строка сообщения в первых дискуссиях и реализациях была **Found**, но в RFC 1945 эта строка была из-за небрежности заменена на **Moved Temporarily**. Новый вариант этой строки сообщения в HTTP/1.1 — **Found**.

КЛАСС (4XX) КОДОВ ОТВЕТОВ HTTP/1.1. ОШИБКИ НА СТОРОНЕ КЛИЕНТА

В HTTP/1.1 появилось 14 новых кодов ответов, относящихся к классу ошибок на стороне клиента. Новые коды отражают значительно расширенный диапазон откликов, направляемых клиентам, если предполагается, что они совершили ошибку в данном запросе. Изменения подразделяются на пять обширных категорий, как показано в таблице 7.10. Первая категория включает четыре кода ответов HTTP/1.0. Добавились четыре новые категории:

- **Коды уточнения состояния.** Данные четыре кода ответов уточняют некоторые из старых кодов ответов, имевшихся в HTTP/1.0. Изменения относительно невелики, а новые коды ответов были созданы для обработки отдельных ситуаций.
- **Коды согласования.** Так как некоторые Web-ресурсы доступны в нескольких вариантах (язык, набор символов и т.д.), в HTTP/1.0 был введен механизм согласования содержания, посредством которого клиент может проинформировать сервер о языках и/или наборах символов, приемлемых для данного пользователя. Ключевое изменение в HTTP/1.1 — это расширение возможностей согласования между источником и получателем: процесс согласования может осуществляться как агентами пользователя, так и серверами.
- **Коды, связанные с размером.** В этой категории два кода. Понятие размера возникает в двух контекстах: размер содержания сообщения и размер URI запроса. Серверы могут требовать принятия некоторых мер, основанных на отсутствии или присутствии информации о размере.
- **Коды состояний, связанные с новыми требованиями.** Этот класс состоит из четырех кодов ответов, представляющих результат существенного расширения возможностей в HTTP/1.1. Они отличаются от предыдущих двух категорий, в которых HTTP/1.0 не предлагал никаких возможностей, требующих этих кодов.

Таблица 7.10. Класс кодов ответов (4xx). Ошибки на стороне клиента. С помощью '*' помечены коды ответов HTTP/1.0

Классификация	Код состояния и строка сообщения	Раздел
Коды ошибок HTTP/1.0	400 Bad Request*	6.2.4
	401 Unauthorized*	6.2.4
	403 Forbidden*	6.2.4
	404 Not Found*	6.2.4
Уточнение кодов HTTP/1.0	405 Method Not Allowed	7.12.3
	407 Proxy Authentication Required	7.11
	408 Request Timeout	7.12.3
	410 Gone	7.12.3
Использование возможности согласования HTTP/1.1	406 Not Acceptable	7.9
	415 Unsupported Media Type	7.12.3
	413 Request Entity Too Large	7.12.3
Коды, относящиеся к размеру	411 Length Required	7.10
	414 Request-URI Too Long	7.10

Классификация	Код состояния и строка сообщения	Раздел
Новые коды для других возможностей HTTP/1.1	402 Payment Required	7.12.3
	409 Conflict	7.12.3
	412 Precondition Failed	7.3.3
	416 Requested Range Not Satisfiable	7.4.1
	417 Expectation Failed	7.4.2

КЛАСС (5XX) КОДОВ ОТВЕТОВ HTTP/1.1. ОШИБКИ НА СТОРОНЕ СЕРВЕРА

Коды ответов класса 5xx возвращаются в том случае, когда серверу не удается обработать данный запрос или он обнаруживает ошибку. В HTTP/1.0 было четыре кода ответов класса 5xx, тогда как в HTTP/1.1 их стало шесть. Коды ответов класса 5xx приведены в таблице 7.11.

Таблица 7.11. Класс (5xx) кодов ответов. Ошибки на стороне сервера. С помощью '*' помечены коды ответов HTTP/1.0.

Код состояния и строка сообщения	Краткое описание	Раздел
500 Internal Server Error*	Непредвиденное состояние сервера	6.2.4
501 Not Implemented*	Отсутствие функции	6.2.4
502 Bad Gateway*	Ошибка расположенного на пути сообщения сервера	6.2.4
503 Service Unavailable*	Временно невозможно обработать запрос	6.2.4
504 Gateway Timeout	Расположенный на пути сообщения сервер превысил лимит времени на ответ	7.3.3 7.12.3
505 HTTP Version Not Supported	Версия протокола не поддерживается	

В HTTP/1.1 было определено использование заголовка **Retry-After** с ответом **503 Service Unavailable** в HTTP/1.0. Было введено два новых кода состояния: **504 Gateway Timeout** и **505 HTTP Version Not Supported**.

В последующих разделах подробно рассматриваются различные семантические изменения с учетом приведенного выше обзора изменений методов, заголовков и кодов ответов в HTTP/1.1.

7.3. Кэширование

Кэш — это память для хранения ответов с целью их последующего повторного использования. Спецификация HTTP включает несколько важных правил относящихся к кэшированию. Эти правила относятся к возможности кэширования (cacheability), проверке актуальности (validation) и согласованности (coherence) ответов, хранящихся в кэше. Однако стратегические решения, такие как замещение элементов кэша и эвристики для удаления потенциально неактуальных ответов, выходят за рамки протокола. В этом разделе мы ограничиваемся обсуждением вопросов кэширования, относящихся к компетенции протокола. Более углубленное обсуждение общих вопросов кэширования можно найти в главе 11. Мы начнем с краткого обзора терминологии относящейся к кэшированию вообще и в связи

с протоколом, затем рассмотрим механизм кэширования в HTTP/1.0. Далее обсудим новые заголовки в HTTP/1.1, связанные с кэшированием.

7.3.1. Термины, относящиеся к кэшированию

В спецификации протокола используются несколько терминов, связанных с кэшированием. Ответ, который был получен от Web-сервера, может быть сохранен в кэше на некоторый период времени. В зависимости от нескольких критериев ответ, хранящийся в кэше, может быть возвращен при получении последующего запроса на тот же ресурс. Как правило, в связи с кэшированием используются следующие термины:

- **Возраст (age).** Возраст ответа — это или время с момента создания его содержимого сервером-источником, или время с момента подтверждения кэшем его актуальности. Возраст измеряется в секундах.
- **Срок годности (expiration time).** Срок годности помещенного в кэш содержимого устанавливается сервером-источником как время, после которого кэш должен подтвердить актуальность содержимого, прежде чем возвращать его в качестве ответа. Если сервер-источник не устанавливает срока годности, кэш может назначить помещенному в кэш содержимому свой собственный, определяемый на основе *эвристик*, срок годности. Срок годности также измеряется в секундах.
- **Период актуальности (freshness lifetime) и устаревание (staleness).** Ответ генерируется Web-сервером в определенное время. Сервер-источник вправе решить, как долго этот ответ можно считать *актуальным*, то есть когда становится некорректным трактовать данный ответ как неустаревший. Как только возраст ответа превышает его период актуальности, такой ответ рассматривается как *устаревший*, независимо от того изменился ли на самом деле этот ресурс на сервере-источнике или нет. Таким образом, период актуальности— это период времени от того момента, когда ответ был создан, и до истечения срока его годности. Актуальность — это период, тогда как срок годности — конкретный момент времени.
- **Актуальность (validity).** Кэш может проверить на сервере-источнике, является ли копия конкретного ресурса, хранящегося в кэше, актуальной. Такая проверка называется *проверкой актуальности* (*revalidation*). Проверка актуальности может осуществляться по отношению не только к серверу-источнику, но и прокси-серверу.
- **Возможность кэширования (cacheability).** Кэш должен принять решение, сохранять ответ в кэше или нет. Возможность сохранения ответа в кэше зависит от множества факторов, например, от наличия разрешения сервера-источника, сохранять данный ответ в кэше и от отсутствия явного указания срока годности. Если даже какое-то содержимое помещается в кэш, может потребоваться проверка актуальности. Возможность кэширования и актуальность ответа являются взаимосвязанными, но разными понятиями. Не все ответы являются кэшируемыми, и не все ответы, хранимые в кэше, являются актуальными.
- **Работа с кэшем (cache maintenance).** Хранящийся в кэше ответ, может быть позднее возвращен. Работа с кэшем требует решения проблем, включая возможность кэширования ответа, определения периода времени хранения ответа в кэше, принятие решения о проверки актуальности кэшированного ответа.

7.3.2. Кэширование в HTTP/1.0

Кэширование стало предметом обсуждения уже в первых дискуссиях по HTTP/1.0. Ко времени завершения работы над RFC 1945, в HTTP/1.0 была введена минимальная поддержка кэширования. Потребность в кэшировании возникла из-за проблем с задержками, связанными с низкоскоростными соединениями и большим количеством запросов на популярные ресурсы. Если ресурс не изменялся между запросами, то загрузка того же самого ресурса приводила к дополнительной и непроизводительной нагрузке на сеть и сервер.

HTTP/1.0 обеспечивал управление кэшированием следующими тремя способами:

- Директива запроса (**Pragma: no-cache**).
- Модификатор запроса **GET** (**If-Modified-Since**).
- Заголовок ответа (**Expires**).

В HTTP/1.0 можно было использовать директиву запроса **Pragma: no-cache**, посредством которой клиенты могли запрашивать получение ответа непосредственно от сервера-источника. Директива **Pragma** могла быть проигнорирована, но, как правило, получатели ее выполняли. Клиенты имели возможность обойти любые кэши и получить текущую копию данного ресурса от сервера-источника.

В HTTP/1.0 определен также заголовок содержимого **Expires**, чтобы дать возможность серверам-источникам указывать срок годности, до достижения которого данный ресурс можно было хранить в кэше и считать актуальным.

Еще одним механизмом кэширования в HTTP/1.0 являлся условный запрос **GET** с использованием модификатора запроса **If-Modified-Since** (обсуждался в главе 6, в разделе 6.2.3). Если копия данного ответа, хранящаяся в кэше по-прежнему актуальна, то сервер возвращает в качестве ответа **304 Not Modified** без тела ответа. Отправитель запроса вместо этого мог использовать копию ответа из кэша. Если время последней модификации ресурса было более поздним по сравнению с указанным в запросе, сервер возвращает соответствующий код состояния (чаще всего **200 OK**) вместе с полным телом ответа. Заметьте, что ответы, возвращенные с кодами состояний, отличными от **200 OK**, могут также сохраняться в кэше. Предположим, что прокси-сервер послал запрос и получил ответ **304 Not Modified**. Прокси-сервер вернет клиенту или ответ из своего кэша, или **304 Not Modified**, в зависимости от того был ли в переданном клиентом запросе заголовок **If-Modified-Since**.

Некоторые из реализаций HTTP/1.0 использовали заголовок **Expires** некорректно, задавая значения, которые тут же приводили к истечению срока годности этого ресурса (препятствуя, таким образом, его сохранению в кэше). Это известно как *аннулирование кэша* (*cache busting*). В решении сервера-источника аннулировать кэш есть и плюсы и минусы. Если у сервера-источника есть повод предполагать, что кэш, возможно, возвращает устаревшие ресурсы, сервер-источник мог бы попытаться убедиться, что ресурс не сохраняется в кэше. *Попадание в кэш* (*cache hit*) (успешное обнаружение запрашиваемого ресурса при обращении к кэшу) результата запроса клиента к прокси-серверу, осуществляющему кэширование, будет «невидимо» для сервера-источника. Сервер-источник может генерировать страницы, которые включают рекламу в виде отдельных GIF-изображений. С целью увеличения дохода от рекламы, серверу может потребоваться сделать так, чтобы реклама передавалась клиентам при каждом посещении ими страницы. Сервер-источник окажется не способным выяснить, что реклама, размещенная на его страницах, была действительно доставлена пользователям. Если же ресурс не был сохранен в кэше, то запрос клиентом страницы приведет к тому, что GIF-изображения с рек-

ламой будут загружаться с сервера-источника, увеличивая, таким образом, число показов рекламы сервером-источником.

Заголовок **Last-Modified** обычно означает, что данный ресурс не был сгенерирован динамически, и некоторые кэши, реализующие HTTP/1.0, использовали это как критерий при принятии решения о том, можно ли данный ответ размещать в кэше. Однако такой логический вывод не всегда справедлив. Некоторые динамически генерируемые отклики могут повторяться каждый раз, когда они генерируются. CGI-сценарий, который генерирует n-ую цифру числа π, генерирует каждый раз один и тот же результат. Другими словами, в зависимости от запроса и его параметров, динамически генерируемые отклики также могут размещаться в кэше. Но прокси-сервер, осуществляющий кэширование, может оказаться не в состоянии оценить характер выполняемого сценария.

Все сравнения интервалов времени в HTTP/1.0 привязывались к отсчету абсолютного времени. Как мы увидим в дальнейшем, это оказалось недостатком, потребовавшим доработки в HTTP/1.1.

7.3.3. Кэширование в HTTP/1.1

Популярность Web, высокая популярность небольшого числа ресурсов, похожесть запросов отдельных групп пользователей, все это вело к значительному росту потребности в кэшировании. В то же время повсеместно ощущалась потребность в *семантически прозрачном кэшировании.*

Кэш должен обеспечивать возможность возвращения ответов без опасения, что сервер-источник вернул бы другой ответ.

Некоторые ключевые правила, которым следовали при внесении изменений в HTTP/1.1, относящихся к кэшированию, были следующими:

- Разделение вопросов о том, можно ли размещать ответ в кэше и можно ли хранящийся в кэше ответ использовать без опасений. Задача протокола состоит в формулировании правил, а не выборе стратегии, как долго объект может храниться в кэше, когда и как содержание кэша следует обновлять.
- Обеспечение корректности, даже если это связано с дополнительными издержками. Кэш не должен «*бездумно*» возвращать устаревшие значения. Спецификация допускает ослабление семантической прозрачности в некоторых обстоятельствах. Клиент или сервер-источник могут явным образом снизить уровень прозрачности. Кэш может предупреждать пользователя, если ответ не соответствует требуемому уровню прозрачности (с помощью предупреждающего заголовка, как мы увидим позже в разделе 7.12.2).
- Предоставление серверу возможности давать больше информации о возможности кэширования ресурса, а пользователям — контроля за тем, что хранится в кэше.
- Независимость от абсолютного времени или от требования синхронизации времени на клиенте и на сервере.
- Обеспечение поддержки согласования кэширования ответов.

Рассматривалось несколько других менее значительных вопросов, таких как разделение механизмов кэширования и регистрации предыстории браузером (обсуждалось в главе 2, раздел 2.3). Механизм регистрации предыстории позволяет пользователю обращаться к предшествующим запросам независимо от актуальности ресурса, представленного пользователю. Протокол все же позволяет агентам

пользователя давать пользователю информацию об устаревании конкретного ресурса. Некоторые браузеры на самом деле перезагружают страницы, срок годности которых мог истечь с того времени, когда страница была сохранена в кэше.

Так как в HTTP/1.0 уже существовало понятие об истечении срока годности (заголовок **Expires**), то изменения, относящиеся к механизму кэширования, должны были быть совместимы с использованием времени окончания срока годности. В HTTP/1.1 были введены механизмы для решения вопросов о возможности кэширования и обеспечения семантической прозрачности. Кроме того, с добавлением к механизму кэширования новых возможностей, должны были быть созданы ясные правила для обработки сочетаний заголовков. В протоколе правила для обработки любой возможной комбинации заголовков не определены явно. Комбинаторный взрыв, являющийся результатом перебора всех возможных комбинаций пар заголовков, делает это невозможным. Столкнувшись с обработкой комбинаций заголовков, реализации должны уметь адаптироваться и следовать набору основных правил. Часто используется так называемый принцип устойчивости, рассмотренный в RFC 791 [Jon81]. Принцип устойчивости — положение часто цитируемое в различных контекстах, звучит так

> Компоненты должны быть либеральными при получении сообщений, и консервативными при их передаче.

Применительно к HTTP этот афоризм можно истолковать так, что компоненты, получающие сообщения, должны быть невосприимчивы к неточностям, путанице или непредусмотренным комбинациям заголовков, но сами должны стремиться передавать только осмысленные сочетания заголовков.

HTTP/1.1 устанавливает некоторые особые правила сочетания заголовков для надлежащей генерации ответов из кэша. При получении ответа от сервера-источника прокси-сервер, осуществляющий кэширование, должен решить, что послать клиенту, от которого исходит запрос. Если сервер-источник послал ответ **200 OK**, возможно, с новой копией ресурса, кэш может сохранить новый ответ и передать его клиенту. Однако предположим, что получен ответ **304 Not Modified**, то есть кэшу разрешается использовать хранящийся в нем ответ для передачи клиенту. Кэш *не может* использовать сквозные заголовки в ответе, хранящемся в кэше. Но кэш *обязан* использовать сквозные заголовки в ответе, который им только что получен от сервера-источника. Это важно, так как сервер-источник может послать ответ **304 Not Modified** с дополнительными заголовками, например, **Expires** и **Cache-Control**.

В HTTP/1.1 введены четыре новых заголовка, относящиеся к кэшированию: **Age, Cache-Control, ETag** и **Vary**. Далее обсуждается их семантика.

ЗАГОЛОВОК AGE

Сервер использует заголовок **Age**, чтобы указать, как давно ответ был сгенерирован Web-сервером. Если сервер — это кэширующий прокси-сервер (т.е. не сервер-источник), то это значение определяет, как давно проверялась актуальность сохраненного в кэше ответа. Прокси-серверы обязаны генерировать заголовок **Age**. Значение заголовка **Age** указывается в секундах и не может быть отрицательным.

ЗАГОЛОВОК CACHE-CONTROL

В отличие от модели кэширования, принятой в HTTP/1.0, HTTP/1.1 предлагает более изощренный механизм управления кэшем. Для обеспечения управление процессом кэширования и со стороны отправителя, и со стороны получателя

в HTTP/1.1 добавлен новый общий заголовок **Cache-Control**. И в запросах, и в ответах можно использовать несколько директив управления кэшем, состав которых можно расширять. Web-серверы могут потребовать, чтобы ответ был приватным и предназначался единственному пользователю. Можно также заставить кэш проверять по прошествии определенного времени актуальность сохраненных ответов перед тем, как возвращать их инициаторам запроса. Промежуточные серверы должны передавать далее директивы управления кэшем, так как они могут относиться ко всем прокси-серверам и шлюзам в цепочке запрос-ответ. Заметим, что эти директивы относятся к обязательным требованиям (уровень MUST) и все кэши должны подчиняться им. При этом важно помнить, что требование совместимости, предъявляемое протоколом, не означает, что кэши на самом деле *будут* выполнять эти директивы. В главе 15 (раздел 15.3), мы рассмотрим последствия несовместимости Web-компонентов.

Хотя некоторые директивы могут применяться и в запросах, и в ответах, наличие директивы в запросе не требует присутствия или отсутствия аналогичной директивы в ответе. Далее мы рассмотрим директивы запросов и ответов, относящиеся к управлению кэшированием.

Таблица 7.12. Директивы запросов, относящиеся к управлению кэшированием

Директива	Краткое описание
no-cache	Принудительная загрузка с Web-сервера
only-if-cached	Получение ресурса только из кэша
no-store	Кэшам не разрешено сохранять запрос, ответ
max-age	Возраст ответа должен быть не больше данного значения
max-stale	Возможно использование устаревшего ответа, но не старше указанного значения
min-fresh	Ответ должен оставаться актуальным по меньшей мере указанное время
no-transform	Прокси-сервер не должен изменять тип данных ресурса
extension tokens	Новые лексемы, представляющие новые директивы запросов

Директивы запросов, относящиеся к управлению кэшированием. Директивы запросов, относящиеся к управлению кэшированием, являются для клиента средством изменить способ обработки запросов, используемый кэшем по умолчанию. В таблице 7.12 представлен набор директив запросов управления кэшированием. Клиентам бывает иногда необходимо, чтобы их запросы не кэшировались для сохранения конфиденциальности запросов. Клиентам иногда также бывает нужно, чтобы ответ приходил только из кэша и чтобы промежуточные средства не изменяли ответ. Заметим, что хотя существует ряд директив запроса, реализации браузеров могут не предоставлять пользователям возможности использовать эти директивы. Далее рассматриваются различные директивы запросов.

- **no-cache**. Директива запроса **no-cache** выполняет ту же функцию, что и директива HTTP/1.0 **Pragma: no-cache**; то есть любому прокси-серверу на пути между отправителем и сервером-источником не разрешено возвращать ответ из кэша. Вместо этого прокси-сервер обязан пересылать запрос дальше серверу-источнику. Этот процесс известен как «сквозная перезагрузка (end-to-end reload)». Например, когда пользователь щелкает мышью на кнопке **Reload** браузера Netscape (при нажатой клавише **Shift**), браузер посылает директиву

Pragma: no-cache. Пользователь получает текущую версию ресурса, независимо от того, что находится в любом из промежуточных кэшей.

- **only-if-cached.** Зеркальным отражением директивы **no-cache** является директива **only-if-cached**. Вместо уверенности в том, что ответ на запрос будет получен от сервера-источника, а не из кэша, в этом случае, отправителю нужен ответ *только* из кэша. Может быть, отправитель озабочен задержкой ответа или его не беспокоит актуальность кэшированного ответа. Кэш может послать ответ, если это соответствует условиям запроса, но в других обстоятельствах может вернуть код ответа **504 Gateway Timeout**. Прокси-сервер может вернуть заведомо устаревший ответ, но только если запрос не содержит других ограничений, таких как заголовка **If-Modified-Since**. Условия семантической прозрачности требуют только, чтобы прокси-серверы не возвращали устаревший ответ, не зная о том, что он устарел.
- **no-store**. Директива запроса **no-store** не допускает сохранения запроса и ответа в кэше. Возможно, пользователь не хочет, чтобы запрос был сохранен даже на короткое время. Директива **no-store** полезное средство, отвечающее требованиям конфиденциальности. Эта директива запрещает сохранять в кэше запрос, а также любой ответ на него. Однако не существует гарантии, что запрос (или ответ) не будут случайно где-либо сохранены. Возможно, было бы лучше для пользователя зашифровать этот запрос.
- **max-age.** Чтобы решить, является ли ответ в кэше устаревшим, агент пользователя может изменить действующий по умолчанию механизм истечения срока годности с помощью директивы управления кэшированием **max-age**. Ответ должен иметь «возраст» меньше, чем значение времени (в секундах), определенное в директиве **max-age**. Таким образом, директива **max-age=0** форсирует сквозную проверку актуальности.
- **max-stale**. Директива **max-stale** выражает готовность клиента принимать ответы из кэша, срок годности которых возможно истек. Однако время истечения срока годности ответа не должно превышать значение, указанное в значении параметра директивы (если оно имеется). Если значение параметра не задано, клиент готов принять любой ответ. Задание **max-stale** позволяет клиентам обойти правила, которыми руководствуется прокси-сервер. Директива **max-stale=60** означает, что агент пользователя готов принять ответ, который может быть устарел, но не более чем на минуту.
- **min-fresh.** Директива **min-fresh** выражает желание клиента принять ответ, который будет актуальным по меньшей мере заданное число секунд. Если в качестве значения параметра директивы **min-fresh** задано число 60, ожидаемый результат состоит в том, что ответ не будет устаревшим через 60 секунд.
- **no-transform.** Директива **no-transform** не допускает, чтобы кэш изменял тип (например, **text/html, image/gif**) тела содержимого. Преобразование тела содержимого может включать изменение алгоритма кодирования содержания (например, сжатие тела содержимого), типа содержания (изменение формата изображения GIF на формат JPEG) или проверку целостности (контрольная сумма). Одной из причин преобразования тела содержимого является эффективность — сжатие ответа ускоряет его передачу. Однако в некоторых случаях, ответ не должен изменяться; так изменение степени детализации медицинского снимка может привести к потере критически важной информации. Если получатель тела содержимого полагается на свою контрольную сумму при

проверке целостности, даже незначительное изменение приведет к тому, что проверка целостности закончится неудачей.

Промежуточный кэш не имеет права отменять сквозные соглашения отправителя и получателя. Директива запроса **no-transform** возвращает управление отправителю (и получателю в случае директивы ответа). Однако кэши могут игнорировать и игнорируют сквозные соглашения, принимая локально-оптимальные решения. Одним из примеров, когда у кэша может быть законное основание проигнорировать директиву, является академический сайт, игнорирующий запросы на загрузку обновленных копий ресурсов сомнительного характера. Но, если кэш игнорирует любые явно заданные директивы запроса (например, **no-cache**), он должен включить в ответ предупреждение, указывая, что ответ, возможно, устарел.

- *лексемы расширений.* Наличие возможностей расширения набора директив управления механизмом кэширования позволяет добавлять новые директивы запросов. Однако если кэши не воспринимают или не намерены выполнять новые директивы, они могут благополучно их игнорировать. Протокол требует (требование уровня MUST), чтобы нераспознанные директивы управления кэшем игнорировались. В любых ситуациях, семантическая прозрачность кэша является главным фактором.

Директивы ответов, относящиеся к управлению кэшированием. Директивы ответов, относящиеся к управлению кэшированием, являются для сервера средством обеспечения особого характера функционирования кэшей, расположенных на пути следования ответа. Список различных директив ответов, относящихся к управлению кэшированием, приведен в таблице 7.13. Директивы ответов предоставляют серверу возможность ограничивать кэширование ответов. Можно управлять кэшированием в диапазоне от разрешения кэшировать ответ до определения того, какие части ответа можно кэшировать и кто имеет доступ к кэшированному ответу.

Таблица 7.13. Директивы ответов, относящиеся к управлению кэшированием

Директива	Краткое описание
public	Разрешение кэшировать ответ, где угодно
private	Ответ только для конкретного пользователя
no-store	Не разрешено кэшировать ответ, запрос
no-cache	Не обслуживать из кэша без предварительной проверки актуальности
no-transform	Прокси-сервер не должен изменять тип содержания
must-revalidate	Можно кэшировать, но проверять актуальность
proxy-revalidate	Форсирует проверку ответа в кэшах, совместно используемых агентами пользователей
max-age	Возраст ответа не должен превышать заданного значения
s-maxage	Максимальный возраст ответа в совместно используемых кэшах
лексемы расширений	Лексемы, представляющие новые директивы ответов

Далее мы рассматриваем различные директивы ответов, относящиеся к управлению кэшированием.

- **public**. Управление кэшированием в HTTP/1.1 по большей части является симметричным. И серверы, и клиенты имеют одинаковые возможности управлять кэшированием ответов. Сервер может установить режим кэширования конкретного ответа, и эти установки будут иметь приоритет над любыми установками, используемыми кэшами по умолчанию. Например, принимая во внимание, что кэш, возможно, решит не сохранять определенные ответы из опасения нарушить их конфиденциальность, сервер, включая заголовок ответа **Cache-Control: public**, может показать, что данный конкретный ответ можно сохранять в кэше.
- **private.** Директива ответа **private** ограничивает возможность кэширования ответа только одним пользователем. Слово private (частный, личный, приватный) вероятно было не лучшим вариантом, так как эта директива ни коим образом не влияет на конфиденциальность сообщения. Разработчики искали антоним к слову shared (совместно используемый), так как директива **private** препятствует сохранению ответа в совместно используемом кэше. Кроме того, область действия директивы **private** может быть сужена со всего сообщения до его частей путем добавления параметра с именем поля. Например, включение в ответ

  ```
  Cache-Control: private = "CustomerID"
  ```

  приведет к тому, что только содержание поля **CustomerID** не будет сохраняться в совместно используемом кэше.
- **no-store.** Директива ответа **no-store** аналогична такой же директиве запроса и используется, чтобы не допустить сохранения ответа в кэше. Если эта директива присутствует в ответе, ни ответ, ни соответствующий запрос не должны сохраняться в кэше.
- **no-cache**. В отличие от директивы **no-store**, директива **no-cache** не запрещает сохранять ответ в кэше. Тем не менее, директива **no-cache** эффективно препятствует такому сохранению, вынуждая кэш проверить актуальность ответа на сервере-источнике до того, как отвечать на запрос. Основное назначение этой директивы — существенно снизить вероятность возврата некоторыми кэшами устаревших ответов. Ответ по-прежнему может храниться в кэше, но кэш должен гарантировать его актуальность путем дополнительной проверки актуальности. Хотя дополнительная проверка увеличивает задержку, если проверка оказывается успешной, можно все же снизить издержки на загрузку ресурса с исходного сервера.

  Директиву **no-cache** можно использовать и с именами полей заголовков и без них. Если директива включает одно или несколько имен полей заголовков, то ее действие распространяется на указанные поля заголовков. Например, предположим, что запрос выглядит следующим образом:

  ```
  HTTP/1.1 200 OK
  Server: Apache/1.2.6 Red Hat
  Date: Thu, 24 Feb 2000 18:13:36 GMT
  Content-Type: text/html
  Location: http://kinghascome.com/ads/elvis_has_left.gif
  Cache-Control: no-cache=Location
  ...
  ```

  Смысл директивы ответа **no-cache** смягчен присутствием в ней дополнительных полей заголовков. Остальное содержание ответа, за исключение заголов-

ка **Location**, может использоваться повторно, снижая, таким образом, неэффективную нагрузку на сеть и воспринимаемую пользователем задержку. Но существует предел контролю Web-сервера над ответом. Кэш должен удалять поле заголовка **Location**, прежде чем отправлять ответ клиенту. Исходный сервер, отказавшись при управлении кэшированием ответа от концепции все или ничего, может сохранить частичный контроль за ресурсами. Аналогично, с помощью директивы **Cache-Control: no-cache=set-cookie2** исходный сервер может запретить сохранять в кэше заголовок **Set-Cookie2** [KM00].

- **no-transform.** Директива ответа **no-transform** аналогична такой же директиве запроса и используется, чтобы не допустить изменения ответа (например, уменьшение разрешения изображения).
- **must-revalidate.** Директива **must-revalidate** позволяет Web-серверу обратить внимание на то, что для некоторых ответов может оказаться слишком рискованным для кэша возвращать возможно устаревшие копии. Эта директива вынуждает кэш провести дополнительную проверку актуальности ресурса на сервере-источнике. Более важно то, что если прокси-сервер не может осуществить проверку актуальности на сервере-источнике (например, если прокси-сервер не может связаться с Web-сервером), то кэш должен вернуть клиенту не устаревший ответ, а сообщение об ошибке. Различие между директивой **must-revalidate** и похожей на первый взгляд директивой **no-cache** заключается в том, что кэши вряд ли сохранят ответы, которые включают директиву **no-cache**. Если ответ включает директиву **must-revalidate**, то он может быть сохранен в кэше, но актуальность его должна быть проверена. Директива **no-cache** вынуждает кэш провести проверку независимо от того, когда этот ответ устарел. На практике, так как кэши вряд ли сохранят ответы, которые включают директиву **no-cache**, запрос на такое содержимое приведет к загрузке данных с сервера-источника, а не просто к их проверке. Использование директивы **must-revalidate** по сравнению с использованием директивы **no-cache** экономнее в отношении нагрузки на сеть. Однако пользователь, пользующийся «магазинной тележкой» в приложениях электронной коммерции, предпочтет либо точно знать ее содержимое, либо получить сообщение об ошибке, но не устаревшие данные.

Предположим, что кэш ждет ответа от предыдущего сервера на пути сообщения и не получает этот ответ в течение некоторого времени. Если это вынудит кэш отказаться от выполнения директивы **must-revalidate**, то кэш обязан передать новый код ответа HTTP/1.1 **504 Gateway Timeout** (как и в случае директивы **only-if-cached**). Сервер может выбрать конкретное время для таймаута. Ответ **504 Gateway Timeout** выглядит более содержательным, чем ответ общего характера **500 Internal Server Error.**

Директива управления кэшем **no-store** гарантирует, что запрос-ответ не будет сохранен, тогда как **no-cache** допускает сохранение, но гарантирует, что не будут возвращаться устаревшие ответы. Директива **must-revalidate** допускает сохранение ответа, но гарантирует, что устаревшие ответы будут обновляться. Однако ответ, который кэш считает обновленным, может быть уже изменен на сервере-источнике. Это различие между директивами является ключевым для понимания необходимости обеих директив: **no-cache** и **must-revalidate**. Директива **no-store** вообще исключает кэширование. В случае **no-cache** существует возможность кэширования, но отсутствует риск возвращения ответа, который уже изменен. В случае **must-revalidate** существует ре-

альная возможность кэширования и не существует возможности устаревания, хотя остается возможность возвращения ответа, который уже был изменен.

- **proxy-revalidate**. Директива ответа **proxy-revalidate** является менее ограничительной, чем директива **must-revalidate**. В то время как последняя относится ко всем кэшам, **proxy-revalidate** не относится к не используемым совместно кэшам агентов пользователей. Кэш агента пользователя (в отличие от кэша прокси-сервера) может возвращать ответы из кэша без перепроверки. Некоторые действия (например, аутентификация) могут без риска выполняться кэшем агента пользователя однократно и запоминаться. Однако совместно используемый кэш используется многими пользователями и, следовательно, действия предпринятые от лица одного пользователя могут не подходить другому. Поэтому перепроверка ответов в совместно используемых кэшах является необходимой. Детализированное управление кэшированием типично для усовершенствований в HTTP/1.1.
- **max-age**. Директива ответа **max-age** является предпочтительным способом задать срок годности содержимого. Наличие директивы ответа **Cache-Control: max-age** отменяет любое значение заголовка **Expires** при вычислении периода актуальности кэшированного ответа. Это пример нового свойства HTTP/1.1, обеспечивающего большую гибкость вновь разрабатываемым компонентам: кэш, использующий HTTP/1.1, получая директиву **max-age,** будет придерживаться указанного там срока годности, тогда как кэш, использующий HTTP/1.0, обычно игнорирует директиву **max-age**.
- **s-maxage.** Заголовок **s-maxage** имеет отношение только к совместно используемым кэшам и применяется для отмены значений директивы **Expires** и директивы **max-age**. Как уже обсуждалось ранее, совместно используемые кэши обязаны перепроверять ответы с истекшим сроком **s-maxage**.
- *лексемы расширений.* Директивы ответов, относящиеся к управлению кэшированием, могут также использовать новые лексемы, определенные использующими их кэшами.

ЗАГОЛОВОК ETAG

Атрибут содержимого (**ETag**) был сначала введен в HTTP/1.1 в качестве индикатора непрозрачности [Mog95] для сравнения содержимого кэша с возможно более новой версией. Данный ресурс может иметь разные версии, и каждая версия будет иметь индивидуальный атрибут содержимого. Атрибут содержимого всегда связан с конкретным ресурсом и не должен использоваться для сравнения разных ресурсов. Структура атрибута содержимого сознательно сделана непрозрачной в том смысле, что от получателей не требуется делать никаких умозаключений, исследуя строку атрибута содержимого.

Атрибут содержимого гарантирует следующее: если существует различное содержимое, сопоставленное с одним и тем же ресурсом, то атрибуты содержимого этих версий различаются. Следовательно, для одного и того же самого ресурса, если атрибуты содержимого идентичны, соответствующие версии ресурсов должны рассматриваться как идентичные.

Атрибуты содержимого отделяют проверку кэша (подтверждение того, что кэшированный ресурс остается актуальным) от срока хранения (с помощью которого сервер явно задает время, после истечения которого ресурс не может рассматриваться как актуальный). Кроме того, ресурсы, когда они изменяются, могут не всегда иметь совершенно новое значение: некоторые версии ресурса возможно суще-

ствовали в прошлом. До введения атрибутов содержимого не уделялось серьезного внимания проблеме асинхронности часов (на компьютерах с кэшами и серверах) и тому факту, что некоторые ресурсы могут возвращаться к прежней версии. Еще одним стимулирующим фактором была возможность надежной идентификации версий ресурсов с шагом квантования менее одной секунды.

Ответ для ресурса **foo.html** может включать атрибут содержимого следующим образом:

```
HTTP/1.1 200 OK
Date: Sun, 26 Dec 26 1999 18:12:26 GMT
Server: Apache/1.2.6 Red Hat
Last-Modified: Fri, 24 Dec 1999 06:21:42 GMT
ETag: cc678-12d12-66394036
Content-Length: 15044
Content-Type: text/html
...
<содержание foo.html>
```

HTTP/1.1 принимает во внимание тот факт, что существуют многочисленные версии ресурсов, и запрос мог быть сделан на любую из этих версий. Новые модификаторы запроса в HTTP/1.1 **If-Match** и **If-None-Match** используют атрибуты содержимого. В частности эти модификаторы запроса могут быть использованы вместе с атрибутами содержимого, чтобы указать на конкретную версию ресурса. Модификатор запроса **If-Match** позволяет клиенту задать один и более атрибутов содержимого, чтобы проверить, совпадают ли они с текущим вариантом на сервере. Например, отправитель, использующий метод **PUT** для обновления ресурса, может использовать условный **PUT**, задавая ту версию ресурса, которую он хочет изменить, чтобы убедиться, что ресурс за это время не изменился. Если у ресурса другая версия, попытка окажется неудачной и сервер вернет код состояния **412 Precondition Failed**.

Например, предположим, что клиент посылает следующий запрос:

```
PUT /big.html HTTP/1.1
If-Match: "er82s1poy", "weoi2re"

<новое содержание для big.html>
```

Сервер сравнит атрибут содержимого текущей версии **big.html**, чтобы определить совпадает ли он с **"er82s1poy"** или с **"weoi2re"**. Если атрибут содержимого совпал с любым из них, сервер выполнит запрашиваемое действие. В этом случае, выполнение запроса приведет к обновлению той версии **big.html**, которая соответствует атрибуту содержимого. Если ни один из атрибутов содержимого в запросе не совпадает с текущей версией, сервер не будет выполнять этот запрос. Вместо этого он вернет код ответа **412 Precondition Failed**.

Новый заголовок запроса **If-None-Match** также можно использовать с атрибутами содержимого. Хорошее применение этого заголовка — это страховка от случайного изменения существующего ресурса. Например, использование заголовка **If-None-Match: *** с методом **PUT** гарантирует, что запрос не будет успешно завершен, если ресурс существует.

Спецификация HTTP/1.1 проводит различие между *строгими* и *слабыми* атрибутами содержимого. Строгий атрибут содержимого изменяется всякий раз, когда одно содержимое отличается от другого хотя бы на один байт. Однако Web-сервер

может изменить строгий атрибут содержимого, даже когда содержимое вариантов не различается. Слабый атрибут содержимого изменяется только тогда, когда существует семантическое отличие. Предполагается, что слабый атрибут содержимого не изменяется, если Web-сервер не считает, что между вариантами существует семантическое отличие.

В отличие от HTTP/1.0 ответы **201 Created** в HTTP/1.1 могут включать атрибут содержимого. Заголовок ответа **ETag**, включенный в ответ **201 Created**, содержит атрибут содержимого того варианта, который был создан (часто в качестве результата запроса **POST**).

ЗАГОЛОВОК VARY

В HTTP/1.0 URI используется в качестве ключа при сохранении и извлечении ответа из кэша. Однако различные варианты представления одного и того же ресурса делают URI неадекватным ключом. Согласование содержания допускает выбор ответа, основанный не только на хранящемся в кэше содержании, но и на вариантах его представления (таких как язык и набор символов). Если кэшируются ответы с согласованным содержимым, кэш должен отбирать соответствующие варианты. Заголовок ответа **Vary** используется для перечисления набора заголовков запросов, которые должны использоваться для выбора соответствующих вариантов из набора, хранящегося в кэше.

Сервер определяет, какие критерии должны быть использованы для выбора подходящего варианта. Например, рассмотрим следующий ответ, который посылает Web-сервер:

```
HTTP/1.1 200 OK
Date: Sun, 26 Dec 26 1999 18:12:26 GMT
Server: Apache/1.2.6 Red Hat
Last-Modified: Fri, 24 Dec 1999 06:21:42 GMT
ETag: cc678-12d12-66394036
Content-Length: 15044
Content-Type: text/html
Vary: Accept-Language
```

Кэш, сохраняющий данный ответ, должен также сохранить значение заголовков запросов перечисленных в заголовке ответа **Vary** (в данном примере **Accept-Language**). Если последующий запрос, сделанный к кэшу, касается получения кэшированного ответа, кэш должен убедиться, что все заголовки в этом запросе, которые перечислены в заголовке **Vary**, идентичны тем, которые были в исходном запросе. Таким образом, если в исходном запросе, результатом которого стал данный ответ, было

```
Accept-Language: en-us
```

тогда для того, чтобы кэш вернул кэшированный ответ на последующий запрос, в нем должно быть **Accept-Language: en-us**. Если в последующем запросе имеется

```
Accept-Language: en-cockney
```

тогда кэш не сможет вернуть кэшированный ответ и вместо этого должен переслать этот запрос Web-серверу.

7.4. Оптимизация загрузки сети

Сразу после того, как популярность Web стала расти, следующие факторы привели к драматическому росту загрузки сети:

- Размеры ресурсов стали увеличиваться и стали гораздо чаще использоваться изображения. Размеры графических ресурсов часто превышали размеры текстовых.
- Изображения часто встраиваются в текстовый документ. Типовая Web-страница включает в среднем около десятка встроенных изображений.
- Пользователей Web становится все больше и больше, совершенствуются способы подключения к сети.
- Netscape Navigator, популярный браузер середины 90-х годов, использовал множественные соединения для параллельной загрузки ресурсов.

Необходимость оптимизации загрузки сети привела к исследованию различных способов, с помощью которых ресурс можно было преобразовать, сократить его размер или не передавать его по сети совсем. В HTTP были введены три категории изменений, относящиеся к оптимальному использованию полосы пропускания сети:

- Если не нужно передавать весь ресурс (например, если пользователь заинтересован только в части ресурса), то это можно использовать для сокращения загрузки сети. В протокол необходимо было включить возможность задавать нужные фрагменты ресурса и передавать их.
- Если отправителю *заранее* известно, что получатель не сможет обработать тело сообщения, не следует передавать такой ресурс совсем. Обмен служебной информацией, которая могла бы подтвердить это, помогла бы избежать передачи ненужных данных.
- Одним из способов снижения трафика является преобразование ресурса перед передачей и полное его восстановление на стороне получателя. Преобразование должно по возможности снижать объем ресурса перед передачей.

Новый механизм *запроса на диапазон* (*range request*) был введен для решения первой из трех категорий описанных выше проблем. Механизм *Expect/Continue* решает вторую, а сжатие данных решает третью. Мы рассмотрим каждую из этих трех возможностей, с помощью которых HTTP/1.1 пытается содействовать оптимизации трафика в сети.

7.4.1. Запросы на диапазоны

Возможности подключения пользователей к Internet не росли наравне с ростом популярности Web. Многие пользователи имели (и продолжают иметь) низкоскоростные соединения с Internet. Медленная загрузка ресурсов приводит к недовольству пользователей, которые часто разрывают установленные соединения, загрузив только часть ресурсов. Если на пути между Web-сервером и клиентом был кэширующий прокси-сервер или если кэширование осуществлялось на уровне браузера, то уже загруженная часть ресурса обычно находится в кэше. Однако новый запрос того же ресурса обычно приводит к полной перезагрузке ресурса. При этом впустую увеличивается трафик и общая задержка.

Первоначальное предложение [Din95] по запросам на диапазоны было сделано в начале 1995 г. и относилось к развитию FTP, при этом приводились два харак-

терных довода: (1) устранение полных перезагрузок при возобновлении прерванных передач данных большого объема и (2) возможно, клиентам нужна только часть ресурса. Возобновление прерванных передач в FTP было уже возможно. Более интересное различие между FTP и HTTP заключалось в том, что в FTP редко осуществлялась передача динамически генерируемых ресурсов. Если прерывалась передача динамически генерируемого ответа, то этот ответ приходилось генерировать снова, увеличивая задержку. В последнем случае отдельные файлы могли извлекаться из файловых архивов (в Internet это обычное дело). В экспериментальных серверах такие возможности уже были реализованы до появления данного предложения [Fra95].

Запрос отдельных частей ресурса может быть полезным и в других ситуациях. Например, агент пользователя может избирательно запрашивать конкретный фрагмент в середине ресурса большого объема, не загружая его начала и конца. Пользователям не всегда нужен весь ресурс. Например, если ресурс постоянно обновляется, пользователей, возможно, интересует только обновленная часть. Или если ресурс представляет собой структурированный документ (например, книга с главами, разделами, страницами и т.д.), избирательный запрос на часть ресурса является естественным желанием пользователя.

Некоторые типы ресурсов позволяют эффективно и легко использовать запрос на диапазон, например, в формате Portable Document Format (PDF) компании Adobe указывается положение начального и конечного байта каждой страницы документа. Клиент может установить несколько параллельных соединений, осуществлять выборку различных страниц одновременно и восстановить исходный ресурс.

Фактически, необходимость извлекать части документов в формате Adobe PDF в браузерах была одной из движущих сил введения запросов на диапазоны в их окончательной форме [FL95].

Эти проблемы могут быть решены, если инициаторы запросов (например, агенты пользователя или прокси-серверы) способны указывать отдельные части ресурса, представляющие для них интерес. Способность формулировать запрос к частям ресурса обеспечивается в HTTP/1.1 механизмом *запроса на диапазон*.

Например, используя заголовок **Range**, клиент может указать, что его интересуют байты в диапазоне 2000–3999, даже если весь ресурс имеет объем 100 000 байт, только 2000 байт будут отправлены сервером. Например:

```
GET bigfile.html HTTP/1.1
Host: www.justwhatiwant.com
Range: 2000-3999
```

Метод **GET** изменен таким образом, чтобы запросить только часть ресурса.

Указанный запрос приведет к следующему ответу сервера, способного обрабатывать запросы на диапазоны:

```
HTTP/1.1 206 Partial Content
Date: Thu, 10 Feb 2000 20:02:06 GMT
Last-Modified: Wed, 9 Feb 2000 14:58:08 GMT
Content-Range: bytes 2000-3999/100000
Content-Length: 2000
Content-Type: text/html
```

Заголовок содержимого **Content-Range** (новый заголовок HTTP/1.1) помогает получателю разместить загружаемую часть данных в нужное место внутри содержимого ресурса.

Код ответа **206 Partial Content** указывает клиенту на то, что полученный ответ не полон. Клиент может также запросить только последние N байтов ресурса, указав, что его интересует *конечная часть — суффикс* ресурса. Например, следующий запрос касается только последних 1000 байтов ресурса:

```
GET bigfile.html HTTP/1.1
Host: www.justwhatiwant.com
Range: -1000
```

Если клиенту не известен текущий размер ресурса, а его начальная часть уже находится в кэше, то и в этом случае можно использовать запрос на диапазон. Например, чтобы получить все байты ресурса, начиная с 9120, клиент может отправить следующий запрос:

```
GET bigfile.html HTTP/1.1
Host: www.justwhatiwant.com
Range: 9120-
```

Можно запрашивать не один диапазон, а сразу несколько. Мотив запрашивать несколько диапазонов очевиден: раз уж клиенту известны отдельные и независимые части ресурса, он может запросить их все в одном запросе, а сервер может отправить их в виде составного ответа. Сервер использует особый тип для такого типа ответа — *multipart/byteranges*. Этот тип включает две и более части, каждая из частей включает свой собственный заголовок **Content-Range**, указывающий диапазон для этой части.

Синтаксис запросов на получение нескольких диапазонов является весьма гибким. Набор диапазонов задается с помощью разделяющих эти диапазоны запятых. Клиент, которому нужно несколько диапазонов, обычно посылает запрос следующего вида:

```
GET bigfile.html HTTP/1.1
Host: www.justwhatiwant.com
Range: 0-100, 2000-2400, 9600-
```

Ответом будут первые 101 байт, 401 байт из середины ресурса и все байты, начиная с байта с номером 9600. Ответ обычно выглядит следующим образом:

```
HTTP/1.1 206 Partial Content
Date: Thu, 10 Feb 2000 20:25:23 GMT
Server: Apache/1.2.6 Red Hat
Last-Modified: Fri, 24 Dec 1999 06:21:42 GMT
ETag: cc678-12dl2-66394036
Content-type: multipart/byteranges; boundary=----ROPE----

----ROPE----
Content-Type: text/html
Content-Range: bytes 0-100/15044
...первые 101 байтов ресурса...
----ROPE----
Content-Type: text/html
Content-Range: bytes 2000-2400/15044
...401 байт из середины ресурса...
----ROPE----
Content-Type: text/html
```

```
Content-Range: bytes 9600-15043/15044
...все байты ресурса, начиная с байта с номером 9600...
----ROPE------
```

Строка-разделитель (**----ROPE----**) называется граничной строкой и используется для разделения разных диапазонов. Сообщение заканчивается граничной строкой с двумя дополнительными тире. Тип **multipart/byteranges** является саморазграничительным и не требует, чтобы сервер предоставлял размер содержимого.

В HTTP/1.1 введен ответ **416 Requested Range Not Satisfiable** для обработки запросов на диапазон, который не может быть возвращен из данного ресурса. Например, если клиент сделал запрос

```
GET /small.html HTTP/1.1
Host: www.justwhatiwant.com
Range: 10000-12000, 14000-19000
```

на ресурс **small.html**, размер которого меньше 10000 байтов, то будет возвращен код состояния **416 Requested Range Not Satisfiable**. Вместе с этим кодом ответа возвращается текущий размер запрашиваемого ресурса в следующем виде:

```
HTTP/1.1 416 Requested Range Not Satisfiable
Content-Length: 9600
```

HTTP/1.1 не требует от серверов способности обрабатывать запросы на диапазон, несмотря на то, что в нем подразумевается, что они должны это делать, если могут. Сервер способный принимать запросы на диапазоны может объявить о своих возможностях с помощью заголовка ответа

```
Accept-Ranges: bytes
```

Тем не менее, клиенты могут делать запросы на диапазоны, не получая такого заголовка ответа от сервера. Если сервер не может обрабатывать запросы на диапазоны, он просто возвращает полный ответ и код ответа **200 OK**. Серверы могут использовать заголовок **Accept-Ranges**, чтобы сообщить о неготовности принимать запросы на любые виды диапазонов для ресурса, указав

```
Accept-Ranges: none
```

Клиенты будут знать, что посылать запросы на части содержимого ресурсов не следует.

Синтаксис запроса на диапазон со временем менялся. Сначала был предложен синтаксис для определения диапазона байтов как части URL. Например, запрос

```
http://www.halfway-svc.com/partial/firstpage;byterange=0-49
```

являлся способом запросить первые 50 байтов ресурса, определяемого ссылкой **http://www.halfway-svc.com/partial/firstpage,** а запрос

```
http://www.halfway-svc.com/partial/firstpage;byterange=1500-
```

мог быть использован, чтобы запросить весь ресурс, за исключением первых 1500 байтов.

Так как получение частей ресурса касается *транспортировки* байтов, а не *именования* ресурса, то данный синтаксис не слишком подходит для механизма частичных запросов. К тому же, некоторые кэширующие прокси-серверы использовали URL в качестве ключа для хранящихся ответов, и наличие информации о диапазоне, встроенной в URL, вызвало бы проблемы. Использование URL означает, что серверы, которые не могут обрабатывать запросы на диапазон, должны будут

возвращать сообщение об ошибке, а не полный ответ. Если же добавить новый заголовок, то серверы, которые не воспринимают запросы на диапазоны, могут проигнорировать этот заголовок, но все же отправить код успешного завершения и полный ответ. Та тщательность, с которой создаются расширения протокола, отражается на способности улаживать такого рода проблемы.

Запрос на диапазон можно разумно комбинировать с условными запросами. Предположим, что запрос на диапазон был сделан для ресурса, находящегося в кэше, но не весь запрошенный диапазон входит в содержимое из кэша. Возможно, что данный ресурс на исходном сервере не изменился, тогда для кэша достаточно получить недостающую часть. Однако если ресурс на Web-сервере изменился, то следует получить весь ресурс. Обычно такая ситуация требует двух запросов: один, чтобы убедиться, что ресурс не изменился, а если ресурс не изменился, то выдается второй с запросом диапазона. Чтобы решить эту задачу одним запросом, HTTP/1.1 предоставляет новое поле заголовка — **If-Range**. Заголовок **If-Range** имеет следующую семантику: если ресурс изменен, то в качестве ответа возвращается весь ресурс; в противном случае возвращаются только запрошенные диапазоны. Заголовок **If-Range** определяет тот вариант ресурса, по отношению к которому выполняется условный запрос.

Например, рассмотрим следующий запрос:

```
GET bigfile.html HTTP/1.1
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Range: bytes=1300-1500
If-Range: cc678-12dl2-66394036
User-Agent: Mozilla
Host: www.notonline.com
```

Если атрибут содержимого ресурса на сервере-источнике совпадает с атрибутом содержимого в заголовке **If-Range**, сервер вернет только 201 байт из запрошенного диапазона 1300–1500, иначе он вернет весь ресурс.

Новый заголовок ответа **If-Unmodified-Since** в HTTP/1.1 можно использовать в сочетании с запросами на диапазоны, чтобы получить части ресурса, *только* если ресурс не изменялся. Например, запрос клиента

```
GET /ads/game03.gif?OCYES HTTP/1.1
Host: www.theads.com
Range: bytes=8353-
If-Unmodified-Since: Tue, 26 Oct 1999 10:54:20 GMT
```

используется, чтобы получить все байты ресурса **/ads/game03.gif?OCYES**, начиная с байта номер 8353, если ресурс не изменялся с указанной даты. Если ресурс изменился, клиенту требуется весь ресурс. Если ресурс не изменился с указанной даты, сервер может ответить следующим образом:

```
HTTP/1.1 206 Partial content
Server: Microsoft-IIS/3.0
Date: Wed, 27 Get 1999 18:17:29 GMT
Content-Type: image/gif
Last-Modified: Tue, 26 Oct 1999 09:54:20 GMT
Content-Length: 14255
Content-Range: bytes 8353-22607/22608
```

В таблице 7.14 представлен ряд примеров запросов на диапазоны, которые могли быть сделаны для ресурса объемом в 1001 байт. Синтаксис запросов на диапазон довольно гибок. В таблице 7.14 показано, как запрашивать один диапазон, несколько диапазонов, все байты после байта с заданным номером, определенное число байтов, начиная с конца, и так далее. Представлены также примеры неправильного задания диапазонов, когда сервер не сможет извлечь запрашиваемые байты.

Таблица 7.14. Примеры запросов на диапазоны для 1001-байтного ресурса

Синтаксис диапазона	Интерпретация
100–300	201 байт от 100 до 300
10–30, 50–100, 300–600	373 байта от 10–30, 50–100 и 300–600
920–950, 951–1000	81 байт от 920 до 1000
920–970, 951–1000	81 байт от 920 до 1000
920–	81 байт от 920 до 1000
–450	последние 450 байтов от 551 до 1000
0–0, 1000–1000	Первый и последний байты
0–0, –1	Первый и последний байты
950–1100	Неправильный диапазон (превышает 1000)
950–900	Неправильный диапазон (номер первого байта превышает номер второго)

Первоначальная идея диапазонов [FL95] вышла за пределы просто диапазонов байтов и подсказала способы запроса строк документов или глав книги, то есть того, что диапазоны должны быть независимыми от типа документа. Хотя текущий стандарт HTTP/1.1 разрешает только запросы на диапазоны байтов, в будущем это может быть расширено на семантически более значимые диапазоны. Например, можно себе представить запрос разделов книги, фрагментов аудиоресурсов или видеоклипов из кинофильмов. Протокол оставляет возможность для таких расширений в будущем открытой. Фактически, протокол Real Time Streaming Protocol (RTSP) [SRL98] допускает запросы на диапазоны по времени (см. главу 12, раздел 12.4).

7.4.2. Механизм Expect/Continue

Если HTTP-сервер не может обрабатывать запросы большого объема, было бы полезно для клиента знать об этом *до того*, как посылается запрос. Клиент может извлечь выгоду, зная, что его ожидания соответствуют действительности до отправки запросов на большие ресурсы. Рассмотрим пример обработки Web-сервером больших форм с помощью методов **PUT** или **POST**. Хотя протокол не накладывает ограничений на размер тела содержимого, у Web-серверов есть свой собственный внутренний предел, основанный на ограничениях, связанных с обработкой и хранением. Однако к тому времени, когда агент пользователя узнает, что сервер отклонит его запрос, будет уже поздно. Пользователь, возможно, затратил значительные усилия и время на заполнение формы. У агентов пользователя могут быть средства для оценки размеров заполненных форм. Кроме того, раз уж запрос был послан, напрасно загружалась сеть. Механизм, посредством которого агент пользователя может узнать о возможных ограничениях сервера до отправки большого запроса, полезен для клиента и сокращает бесполезный трафик.

Механизм **Expect** обеспечивает именно такое решение. Он позволяет клиенту узнать, способен ли сервер удовлетворить ожидания клиента в том, что касается конкретного ресурса. Если сервер готов соответствовать заявленным ожиданиям и обработать данный запрос, он может сообщить об этом с помощью ответа **100 Continue**. Сервер, неспособный обработать такой запрос, может послать соответствующий код ответа, основанный на конкретной причине неспособности обработать данный запрос. Если запрос имеет слишком большой объем, сервер может послать **413 Request Entity Too Large**. Если данному клиенту не было разрешено сделать такой запрос, сервер может послать **403 Forbidden**. Если сервер получает неизвестную лексему в **Expect** или если ему известно, что сервер, расположенный выше в цепочке на пути к исходному серверу, не сможет работать с механизмом **Expect**, он посылает ответ в виде кода состояния **417 Expectation Failed**. Ответ посылается до того, как на исходный сервер реально посылает тело запроса.

Например, рассмотрим следующий запрос, посланный с заголовком **Expect**. Клиент планирует послать тело запроса размером 23248 байтов.

```
POST /foo/bar HTTP/1.1
Content-Length: 23248
Expect: 100-Continue
```

Сервер может ответить так:

```
HTTP/1.1 100 Continue
```

указывая, что он сможет обработать этот запрос и, следовательно, клиент может продолжать и отправить большое тело содержимого по уже открытому соединению. Если сервер не сможет обработать такой запрос, он отправит следующий ответ и, возможно, закроет соединение:

```
HTTP/1.1 417 Expectation Failed
```

Клиенты могут включать дополнительные значения в заголовок **Expect**. Например, клиент может послать аутентификационную информацию пользователя, инициировавшего этот запрос. Если сервер принимает эту информацию, он обычно посылает ответ **100 Continue**, и клиент может отправлять тело сообщения-запроса в следующем виде:

```
POST /secure.txt HTTP/1.1
Host: www.topsecret.org
Authorization: Basic ZGufaWP6J29atWU=
Expect: 100-Continue
```

Если сервер возвращает промежуточный код ответа

```
HTTP/1.1 100 Continue
```

клиент может продолжать запрос. Если вместо этого сервер возвращает

```
HTTP/1.1 401 Authorization Failed
```

оставшаяся часть запроса не передается. Клиенту придется или посылать новые данные для авторизации, или отказаться от запроса.

Изучение того, как протокол применяется на практике, — это один из способов дальнейшего развития протокола его разработчиками. Полезность некоторых свойств протокола может оставаться неясной, поэтому реальный учет данных о ранних версиях протокола может помочь его совершенствованию. К сожалению, уже существует несколько несовместимых версий так называемых HTTP/1.1-реализаций

компонентов, что вынуждает спецификацию протокола относиться более терпимо к операциям с механизмом **Expect**. Может случиться так, что клиент, который посылает заголовок **Expect:100-Continue**, будет ждать ответа **417 Expectation Failed** или **100 Continue** неопределенное время. Чтобы исключить такую ситуацию, клиент должен уметь приостанавливать свою работу на определенное время, а после этого посылать тело запроса (фактическая продолжительность таймаута может зависеть от реализации клиента). Протокол запрещает клиенту ждать неопределенное время, хотя и не определяет продолжительность таймаута. Поэтому клиент использует для ожидания не детерминированный, а случайный период времени.

Механизм **Expect** является механизмом промежуточных передач. Ожидание определенного результата распространяется только на ближайшего соседа, если следующий сервер не соответствует ожиданиям, то этот сервер должен вернуть код ошибки **417 Expectation Failed**. Однако заголовок запроса **Expect** является заголовком сквозного типа. Если следующий сервер (например, прокси-сервер) пересылает запрос дальше, поскольку сам он не может его обработать, он должен переслать вместе с ним также и заголовок запроса **Expect**.

Хотя механизм **Expect** был предложен для обработки запросов большого объема, связанных с методами **PUT** и **POST**, он может использоваться в будущем и для других расширений. Серверы, которые пока еще не воспринимают новых расширений, обязаны возвращать код состояния **417 Expectation Failed**. Возможным расширением механизма **Expect** является совершенствование схемы согласований HTTP/1.1.

Интересно отметить, что заголовок **Expect** был введен не как управляемый клиентом механизм согласования ожидаемого результата, а для решения проблемы, связанной с ответом **100 Continue.** Ответ **100 Continue** был предложен почти за два года до введения заголовка **Expect**. Ответ **100 Continue** был введен, чтобы сервер мог посылать промежуточный код ответа, если он был намерен продолжать принимать входные данные от клиента. Клиент имеет разумное основание не посылать содержимое своего запроса до тех пор, пока не убедится, что Web-сервер примет его запрос. Если Web-сервер не намерен продолжать чтение входных данных, он может послать сообщение об ошибке. Например, если запрос клиента **POST**, включает заголовок **Content-Length,** и сервер знает, что не сможет обработать запрос такого размера, он может вернуть код ошибки. Если сервер может обработать такой запрос, он может послать ответ **100 Continue**. Такое решение требует, чтобы клиент сделал паузу, прежде чем передавать содержимое запроса. Однако отправитель может прождать очень долго, прежде чем выяснится, что сервер не собирается посылать ни **100 Continue**, ни сообщение об ошибке. Первым предложением было ввести пятисекундный таймаут, но скоро стало ясно, что произвольно выбранное время ожидания клиента не является удовлетворительным решением во всех случаях. Еще важнее было то, чтобы клиенты, использующие HTTP/1.0, не могли воспринимать ответ **100 Continue**, так как он был определен только в HTTP/1.1.

Чтобы избежать такого рода проблем, был добавлен заголовок **Expect**, с таким расчетом, чтобы ответ **100 Continue** был возможен *только* в том случае, если в запросе присутствовал заголовок **Expect**. Не все клиенты должны ждать ответ **100 Continue**. Окончательная формулировка правил для клиентов и серверов в HTTP/1.1 содержит следующее: клиент должен послать заголовок **Expect**, если он намерен ждать ответ **100 Continue**, а сервер должен вернуть или **100 Continue**, или заключительное сообщение об ошибке. Если клиент, отправивший заголовок **Expect**, так и не получил от Web-сервера ответ **100 Continue**, он не должен ждать неопределенно долго, а может продолжать свою работу и отправить содержимое запроса.

7.4.3. Сжатие

Хороший способ снижения загрузки сети — это сжать ответ так, чтобы это было приемлемо для получателя. Сжатие используется во многих протоколах прикладного уровня и входит в HTTP/1.0. Некоторые популярные форматы графических данных являются предварительно сжатыми. Например, популярными форматами изображений являются GIF и JPEG, в которые включено сжатие данных. От повторного сжатия таких ресурсов нет особой пользы. Однако многие другие ответы, которые обычно хранятся в несжатом виде (например, обычный текст) могут быть сжаты Web-сервером перед их передачей. Допустимо даже сжатие заголовков запросов и ответов, но выгода от такого сжатия не слишком ясна, если учесть наличие промежуточных серверов. В HTTP заголовки не сжимаются.

Кодирование содержания было возможно в HTTP/1.0 (с помощью заголовка содержимого **Content-Encoding**) и указывало на способ преобразования ресурса по сквозному принципу. Поскольку HTTP/1.0 не делал различий между механизмом промежуточных передач и сквозной передачей данных, невозможно было реализовать сжатие на основе механизма промежуточных передач. Предположим, что двум промежуточным серверам известен хороший алгоритм сжатия, который требует небольшого времени на сжатие, и обеспечивает лучшее сжатие, чем другие алгоритмы. Эти серверы могут стремиться использовать этот алгоритм для кодирования сообщений при обмене между собой. Другие серверы на пути от отправителя к получателю не обязаны ничего знать об этом алгоритме. Кроме того, если имеется выбор между разными алгоритмами сжатия, серверу может потребоваться указать на предпочтительный алгоритм. И то, и другое возможно в HTTP/1.1.

Механизм *кодирования при передаче* является механизмом промежуточных передач для указания преобразований, примененных к телу содержимого. Клиент может включить заголовок **Accept-Encoding** для указания приемлемых схем кодирования содержания, он может также использовать заголовок запроса **TE** для указания предпочтительных схем кодирования при передаче. Например,

```
TE: vdcomp; q=0.9, compress; q=0.1
```

означает, что отправитель готов принять кодирование при передаче **vdcomp** с высоким коэффициентом качества 0.9 и **compress** с низким коэффициентом качества 0.1. Клиент при выборе коэффициента качества (0.9 для **vdcomp** и 0.1 для **compress**) может руководствоваться несколькими доводами, такими как наличие соответствующих программ и быстродействие этих алгоритмов. Сервер, получив такой запрос, попытается выбрать наибольший коэффициент качества, указанный в запросе, и откажется от выбора любого алгоритма, для которого клиент задал значение коэффициента качества, равное 0 (то есть рассматриваемый как неприемлемый для клиента). Вариант, приведенный в запросе, является предпочтительным, но нет гарантии, что сервер может его удовлетворить. В примере, приведенном выше, сервер может выбрать кэшированное значение в формате **compress**, игнорируя тот факт, что инициатор запроса может предпочитать формат **vdcomp**. Сервер, который или не может, или не готов выполнить специфическое кодирование при передаче, не осуществляет преобразование ответа. Заголовок **TE** применим только к прямому соединению, и, следовательно, любые два сервера, воспринимающие алгоритм сжатия **vdcomp**, могут извлечь из этого выгоду, включив этот заголовок.

7.5. Управление соединениями

Фактически все известные реализации HTTP используют TCP в качестве транспортного протокола. Однако TCP не был оптимизирован для краткосрочных соединений, типичных для обмена сообщениями в HTTP. Использование TCP в качестве транспортного протокола требует последовательности из трех квитирующих пакетов для установления соединения и еще четырех для закрытия открытого соединения (глава 5, раздел 5.2.3). Типичное HTTP-сообщение состоит из 10 пакетов. Таким образом 7 из 17 пакетов представляют собой накладные расходы. Это также означает, что ни одна Web-транзакция не обходится без медленной начальной фазы установления TCP-соединения [Jac88] (обсуждалось в главе 5, раздел 5.2). Прежде чем размер скользящего окна TCP сможет существенно увеличиться, соединение закроется. Это означает, что доступная полоса пропускания никогда не будет использована полностью.

По мере роста популярности Web, в Web-страницы стали включать встроенные изображения. Загрузка *составного* документа, то есть совокупности текста и изображений требовала нескольких HTTP-транзакций и, следовательно, нескольких TCP-соединений. Прежде чем в браузере отображался весь документ, пользователь ощущал задержку, являющуюся результатом последовательности соединений. Подобным образом был реализован Mosaic — один из первых популярных браузеров. Одним из способов снизить задержку было введение параллельных HTTP-соединений. Впервые это было реализовано в Netscape, до четырех соединений можно было открыть параллельно для загрузки изображений, ускоряя таким образом загрузку всего документа. Однако это увеличивало общую загрузку сети, на сервер ложилась дополнительная нагрузка по поддержанию множества TCP-соединений.

Другим решением данной проблемы было использование одного TCP-соединение для нескольких HTTP-транзакций. Это привело к введению в HTTP/1.1 *долговременных соединений* (*persistent connections*). Основной идеей, лежащей в основе долговременных соединений, было сокращение числа открываемых и закрываемых TCP-соединений. Вместе со снижением числа открытий и закрытий TCP-соединений, уменьшалось число пакетов, передаваемых по сети, что в свою очередь снижало ее загрузку. Увеличение времени существования TCP-соединений позволяет серверу получить информацию о загруженности сети. Если загрузка меньше, можно послать больше пакетов, и отправитель может гарантировать, что он чрезмерно не увеличивает загруженность сети, посылая слишком много пакетов. Выигрыш при последовательной передаче ответов по одному соединению получается за счет снижения загруженности сети. К тому же значительно снижается воспринимаемая пользователем задержка, так как передаваемые последовательно по одному соединению запросы не должны ожидать закрытия предыдущего соединения, установления нового и фазы медленного старта TCP-соединения. Сообщения об ошибках запросов могут быть переданы по тому же соединению.

В этом разделе мы проследим эволюцию механизма долговременных соединений в HTTP до их теперешнего стандартизованного состояния.

- В разделе 7.5.1 рассматривается старый механизм **Keep-Alive**, существовавший в некоторых реализациях HTTP/1.0.
- Раздел 7.5.2 посвящен нескольким первым предложениям по долговременным соединениям в HTTP/1.1 и множественным параллельным соединениям, которые к тому времени уже были популярны. Затем рассматриваются задачи механизма долговременных соединений, который был введен в HTTP/1.1.

- В разделе 7.5.3 анализируется роль заголовка **Connection** в долговременных соединениях HTTP/1.1, особенно той роли, которую играет этот заголовок при сохранении открытых соединений и при их закрытии.
- В разделе 7.5.4 рассматривается роль конвейеризации в долговременных соединениях и вопросы, относящиеся к неожиданным разрывам соединений в процессе обработки последовательности запросов.
- В разделе 7.5.5 анализируется решения, которые должны принимать клиенты и серверы перед закрытием долговременного соединения. Протокол не дает особых указаний о том, как долго долговременное соединение может оставаться открытым.

Существенные изменения, относящиеся к управлению соединениями, обсуждаются подробно из-за их важности.

7.5.1. Заголовок Connection. Механизм Keep-Alive в HTTP/1.0

Возможность продлевать http-соединение за пределы одной транзакции запрос-ответ в HTTP/1.0 первоначально отсутствовала. Однако по мере роста популярности Web некоторые реализации HTTP/1.0 ввели разновидность долговременных соединений. Некоторые браузеры включали заголовок запроса **Keep-Alive** для запроса установления соединения, которое остается открытым и после выполнения текущего запроса. Серверы, готовые оставить соединение открытым, принимали данный заголовок и не закрывали соединение после передачи ответа.

Идея, лежащая в основе **Keep-Alive**, та же, что и у долговременных соединений в HTTP/1.1. Клиент, заинтересованный в сохранении соединения открытым, просил, чтобы Web-сервер не закрывал данное соединение. Это делалось следующим образом:

```
GET /home.html HTTP/1.0
...
Connection: Keep-Alive
```

Если сервер также был заинтересован в сохранении открытого соединения, он отправлял следующий ответ:

```
HTTP/1.0 200 OK
   ...
Connection: Keep-Alive
   ...
<тело ответа>
```

Однако если сервер, использующий HTTP/1.0, передает динамическое содержание, у клиента, принимающего его, нет способа определить окончание ответа без закрытия этого соединения сервером. Динамически генерируемое содержание обычно не включает заголовок **Content-Length**, поскольку ожидание вычисления длины после завершения формирования содержания увеличивало задержку для клиента.

Простые расширения позволяли серверу указывать (с помощью полей заголовков), как долго соединение может оставаться открытым после последнего запроса или максимальное количество запросов, которые будут переданы с помощью данного соединения. Клиент мог проверить поля заголовков и принять решение не посылать по этому долговременному соединению более определенного числа запросов.

7.5.2. Эволюция механизма долговременных соединений в HTTP/1.1

Еще до введения современного механизма долговременных соединений в HTTP/1.1 предпринимались разнообразные попытки сохранять HTTP-соединения за пределами одной транзакции запрос-ответ. Их можно подразделить на два больших класса: (1) новые методы HTTP, которые можно было использовать для получения более чем одного ресурса и (2) использование нескольких параллельных соединений для выборки ресурсов. Первый подход включал вариации механизма многократного метода GET, позволяющего выполнить многократные запросы поверх одного транспортного соединения. Второй подход использовал параллельно нескольких транспортных соединений для выборки группы ресурсов. Разработка механизма, который был стандартизован в качестве механизма долговременных соединений в HTTP/1.1, начиналась с попыток избежать необходимости создавать несколько параллельных соединений.

Подход, связанный с новыми методами HTTP: MGET, GETLIST, GETALL. Первый подход предлагал новые методы HTTP для передачи нескольких запросов по одному и тому же транспортному соединению. Было предложено три метода: **MGET, GETLIST** и **GETALL**. Ни одно из этих предложений не сохранилось в процессе эволюции протокола и не вошло в HTTP/1.1. Однако рассмотреть предлагавшиеся альтернативные решения представляется полезным.

Предложение по введению нового метода **MGET**, было сделано в [Fra94a, Fra94b]. Метод **MGET** похож на команду FTP **mget**, посредством которой можно было получить с помощью одной команды несколько файлов, соответствующих определенному шаблону. В FTP, команда *mget* требовала нескольких соединений для передачи данных, то есть отдельных TCP-соединений. Предложенный метод HTTP **MGET** не требовал установления нескольких TCP-соединений, путем перечисления набора ресурсов в самом запросе. Весь запрос должен был обрабатываться Web-сервером последовательно. В следующем примере запроса **MGET**, заимствованном из [Fra94a], приводится список нескольких URI вместе с модификаторами запроса:

```
MGET HTTP/1.0
URI: /image1.gif
URI: /image2.gif
If-Modified-Since: Saturday 29-Oct-94 20:04:01 GMT
URI: /image3.gif
CRLF
```

Сервер откликается на запрос **MGET**, посылая ответы для каждого из запрошенных ресурсов с предварительным указанием размера ресурса и кодировки содержания, что позволяет клиенту правильно обработать ответ:

```
HTTP/1.0 200 OK
URI: /image1.gif
Content-Type: image/gif CRLF
2200
... 2200 байтов данных изображения
CRLF
HTTP/1.0 304 Not Modified
URI: /image2.gif
```

```
CRLF
HTTP/1.0 200 OK
URI: /image3.gif
Content-Type: image/gif
CRLF
7180
... 7180 байтов данных изображения
CRLF
```

Клиент просматривает каждый ответ до **CRLF** и извлекает отдельные ответы. Очевидно, что устранение необходимости установления нескольких последовательных соединений для отправки ответов приводит к уменьшению времени ожидания и загрузки сети.

Предложение **MGET** возникло в контексте дискуссий о получении страницы вместе с встроенными в нее изображениями. Имея информацию о размерах изображений, встроенных в HTML-страницу, браузер мог осуществлять ее отображение, хотя изображения еще находились в процессе загрузки. Браузер может получить данную информацию либо средствами HTML, либо от сервера. Например, информация о встроенных изображениях могла быть включена в исходный HTML-текст документа-контейнера. Однако это потребовало бы от протокола сведений о содержимом конкретного типа (HTML) документов, что нежелательно. В качестве альтернативы, сервер мог бы включать эту информацию в заголовок ответа. Однако зависимость от Web-сервера не слишком желательна, так как сервер не всегда в состоянии предоставить нужную информацию. Например, не все изображения могут находиться на том же Web-сервере, что и документ-контейнер.

GETLIST и **GETALL** [PM95] по сути похожи на **MGET**. Используя **GETLIST**, клиент может запросить конкретный список ресурсов. Чтобы запросить все ресурсы (документ-контейнер и все его встроенные ресурсы), следует использовать метод **GETALL**. Единственный запрос с методом **GETALL** автоматически передаст все составляющие документа, не требуя отдельных запросов **GET**. Преимущество **GETLIST** по сравнению с **GETALL** заключалось в том, что клиент имеет возможность выбирать, какие из встроенных ресурсов запрашивать. Один и тот же ресурс может быть встроен в документ-контейнер несколько раз, и не имеет смысла запрашивать такой ресурс более одного раза.

Подход, связанный с параллельными соединениями. Альтернативным подходом к использованию одного и того же соединения для выборки нескольких ресурсов являлось использование нескольких параллельных соединений. Браузер мог открыть *несколько* параллельных соединений и загружать встроенные изображения параллельно. Параллельные соединения были реализованы в 1994 г. в одной из ранних версий браузера Netscape. Такой подход может быть оправдан, если принять ограниченную точку зрения на опыт эксплуатации Web исключительно с позиции сокращения времени ожидания пользователя. Хотя пользователь, возможно, почувствует, что данные загружаются довольно быстро, у этого подхода есть существенный недостаток: открытие нескольких параллельных соединений значительно нагружает сеть. Множество пакетов, передаваемых для установления и закрытия TCP-соединений, увеличивают загрузку сети. Если слишком много клиентов начнут использовать параллельные соединения, результатом будет перегрузка сети. Кроме того, серверы будут испытывать пиковые нагрузки. Сервер может испытывать недопустимую нагрузку, если множество клиентов запрашивают ресурсы со встроенными изображениями, используя параллельные соединения. Основная про-

блема заключается в том, что в условиях ограниченной пропускной способности сети, множество параллельных соединений одного клиента сокращает полосу пропускания, выделяемую другому клиенту. Пытаясь занять как можно большую часть полосы пропускания, множественные соединения на самом деле снижают реальную пропускную способность.

Однако у метода параллельных соединений есть и более серьезный недостаток, связанный с отменой запросов. Рассмотрим следующий сценарий: на Web-сервере имеется ресурс с множеством встроенных изображений, браузер устанавливает несколько параллельных соединений, чтобы начать его загрузку. Если пользователь решает прекратить загрузку ресурса, все параллельные соединения должны быть закрыты. Значительные накладные расходы на установление соединений при этом были затрачены напрасно.

Даже если не учитывать отмены запросов, параллельные соединения не всегда сокращают время ожидания пользователя при загрузке исходного документа. Каждое из параллельных соединений является независимым от другого, и каждое соединение должно отдельно «платить свою плату» за установление TCP-соединения и за прохождение фазы медленного старта (согласно изложенному в главе 5, раздел 5.2.6).

Долговременные соединения в HTTP/1.1. Наряду с множеством предлагавшихся решений, был разработан и реализован подход, который обеспечивал долговременные соединений за счет того, что давал возможность и клиенту, и серверу указывать те соединения, которые могут быть сохранены [PM95]. Это предложение может быть разделено на две части: повторное использование существующего транспортного соединения и внесение некоторых небольших изменений на прикладном уровне (в протоколе HTTP). Идея повторного использования транспортного соединения была похожа на предложения, рассмотренные ранее, и преследовала три цели:

- Снижение «стоимости» TCP-соединения (меньше открытий и закрытий).
- Уменьшение задержки путем сокращения фаз медленного старта TCP-соединений.
- Сокращение потерь полосы пропускания и снижение общей загрузки.

Предложенные изменения на уровне HTTP были незначительными. Можно было добиться дополнительного уменьшения задержки, если посылать по долговременному соединению несколько запросов, не дожидаясь ответов от сервера. Этот способ известен как *конвейеризация* (*pipelining*) и излагается в разделе 7.5.4. Это может повлиять на снижение общего числа соединений, если прокси-серверы используют долговременные соединения. Потребности нескольких конкретных клиентов могут быть удовлетворены меньшим количеством соединений между прокси-сервером и Web-сервером, если прокси-сервер поддерживает долговременное соединение с данным Web-сервером.

Значительным шагом, обеспечивающим широкое использование долговременных соединений, было введение в HTTP/1.1 данного типа соединения в качестве *умалчиваемого*. Соединение **Keep-Alive** было необязательным в реализациях HTTP/1.0. В спецификации HTTP/1.1 реализация долговременных соединений относится к *рекомендуемым* требованиям (требования уровня SHOULD). Хотя ничто не мешает администратору Web-сайта отключить установку долговременных соединений в качестве соединений по умолчанию, требование протокола уровня SHOULD повышает вероятность того, что администраторы совместимых реализаций серверов будут оставлять долговременные соединения в качестве соединений по умолчанию. Адми-

нистратор Web-сервера, использующего HTTP/1.1, для отключения долговременных соединений в качестве установки по умолчанию в настройках сервера должны сделать это в явном виде. Поскольку в большинстве случаев установки по умолчанию оставляются «как есть», преимущества долговременных соединений вероятно должны стать доступны большинству пользователей.

Предложения **GETALL** и **GETLIST** преобразовались в механизм долговременных соединений, который стал частью стандарта HTTP/1.1. Ни один из предложенных методов (**MGET, GETALL** и т.д.) не сохранился в окончательном варианте протокола. Хотя множественные параллельные соединения довольно распространены, общепризнанно, что долговременное соединение с конвейеризацией обеспечивают наиболее эффективное использование сетевых ресурсов и минимальное время ожидания для пользователей.

Еще одна тема, касающаяся долговременных соединений, — это наличие прокси-серверов и кэшей. Должно ли долговременное соединение осуществляться по сквозному принципу (между клиентом, цепочкой прокси-серверов и Web-сервером)? Необходимо позаботиться о прокси-серверах, которые работают по протоколу HTTP/1.0 и не воспринимают долговременных соединений. Эта тема обсуждается более подробно в разделе 7.11.

7.5.3. Заголовок Connection

Некоторые реализации HTTP/1.0 используют заголовок **Connection** чтобы оставить соединение открытым и после завершения одиночной транзакции запрос-ответ. Однако в HTTP/1.1 в соответствии с тенденцией к обеспечению управления соединениями и отправителем, и получателем, заголовок **Connection** может быть использован для указания, что одна из сторон намерена *закрыть* соединение. Возможно, сервер не хочет поддерживать слишком много открытых долговременных соединений, или отправитель (например, прокси-сервер) решит, что у него больше нет оснований сохранять открытое соединение с данным конкретным сервером. Обе стороны могут включить заголовок **Connection:close**, чтобы сообщить о намерении закрыть существующее долговременное соединение независимо от намерений другой стороны.

Общий заголовок **Connection** был описан выше, как имеющий более широкое назначение: перечислять набор заголовков, которые имеют смысл только для текущего соединения транспортного уровня с соседним сервером. Иначе говоря, сервер, получающий заголовок **Connection**, анализирует этот заголовок и удаляет все заголовки перечисленные в нем. Вариант **Connection: close** был введен в качестве способа обеспечения совместимости с прежней (HTTP/1.0) формой долговременных соединений. Некоторые реализации HTTP/1.0 поддерживают заголовок **Connection: Keep-Alive**. Вместо того чтобы изобретать новый заголовок, семантика которого будет пересекаться с семантикой заголовка **Connection**, механизм **Connection** в HTTP/1.1 использовал **Keep-Alive** в качестве одной из многих возможных лексем заголовка **Connection**. Защита заголовка (т.е. обеспечение того, чтобы данный заголовок не будет пересылался прокси-серверами далее) также была согласована с существующей лексемой **Keep-Alive**. Так как по умолчанию в HTTP/1.1 реализуется долговременное соединение, то это выглядит, как если бы заголовок **Connection: Keep-Alive** включался в каждое сообщение. Проблема, связанная с поддержанием долговременного соединения с прокси-серверами, которые не воспринимают заголовки **Connection**, обсуждается в разделе 7.11.3.

7.5.4. Конвейеризация в долговременных соединениях

Установление долговременных соединений снижает количество пакетов, необходимых для создания и закрытия TCP-соединений. Клиент, посылающий запрос и ждущий на него ответ до того, как послать следующий запрос по тому же долговременному соединению, вносит дополнительную задержку. Вместо этого клиент может послать совокупность запросов, *не ожидая* соответствующих ответов (исходя из предположения, что запросы будут обрабатываться последовательно), при этом задержка, связанная с ожиданием приема каждого ответа, исключается. На самом деле основной выигрыш от механизма долговременных соединений получается благодаря конвейеризации [Pad95].

Рассмотрим, например, четыре запроса на изображения по одному и тому же долговременному соединению:

```
GET /foo1.jpg HTTP/1.1
CRLF
GET /foo2.jpg HTTP/1.1
CRLF
GET /foo3.jpg HTTP/1.1
CRLF
GET /foo4.jpg HTTP/1.1
CRLF
```

В примере присутствуют запросы четырех ресурсов: **fool.jpg, foo2.jpg, foo3.jpg** и **foo4.jpg**, объединенных в конвейер в одном HTTP-соединении. Клиент извлекает выгоду из последовательных запросов четырех ресурсов, не ожидая возвращения каждого ответа до отправки следующего запроса.

Запросы, передаваемые по долговременному соединению с конвейером или без него, должны обрабатываться сервером в том порядке, в котором они были получены. Если запросы были обработаны не в том порядке, отправитель может не получить актуальное содержание ресурса. Спецификация протокола предполагает, что только идемпотентные методы (глава 6, раздел 6.2.2) должны использоваться последовательно. Например, если запрос **GET** следует за предшествующим ему запросом **PUT**, ожидаемым результатом должно быть получение последнего содержимого ресурса (возможно измененного запросом **PUT**). Однако это предполагает, что оба конвейеризированных запроса были обработаны без перерыва. Если соответствующее транспортное соединение было преждевременно закрыто (например, по причине аварийного завершения), ожидаемый результат может быть и не получен.

БЛОКИРОВКА ТИПА «ПЕРВЫЙ В ОЧЕРЕДИ»

Если один из предшествующих запросов ресурса в конвейере занимает значительное время, другие запросы того же соединения должны быть задержаны. Эта проблема обостряется, когда соединение осуществляется с прокси-сервером, который передает запросы конвейера *разным* Web-серверам. Возможно, что первый в очереди конвейеризированный запрос получает ресурс очень большого объема, и хотя следующий запрос может адресоваться другому Web-серверу и запрашивать небольшой ресурс, он не может быть обработан пока обрабатывается первый (большой) запрос. Эта ситуация известна как блокировка «первый в очереди (head of line)» [Get97]. Таким образом, если отправителю известно, что предыдущие запросы могут занять много времени, ему вероятно не следует включать в конвейер запросы, следующие за продолжительными запросами. Отправителю, тем не менее, не всегда известно, сколько времени потребует обработка конкретного запроса.

Хотя сервер и может попытаться предложить более эффективную последовательность передачи ответов, избегая в некоторых случаях блокировки «первый в очереди», протокол требует, чтобы ответы посылались в том порядке, в котором были получены запросы. Это самое простое, что может сделать сервер. Спецификации протоколов предпочитают ошибаться «по простоте своей».

СЛУЧАИ НЕПРЕДВИДЕННОГО ЗАКРЫТИЯ ПОТОКА КОНВЕЙЕРИЗИРОВАННЫХ ЗАПРОСОВ

Существует множество причин, по которым HTTP-соединение может завершиться аварийно. Пользователь может намеренно прервать соединение, нажав кнопку браузера **Stop** или щелкнув мышью на гиперссылке, когда текущая страница продолжает загружаться. Сервер может прервать соединение, если решит, что его ресурсы пытаются использовать неправильно. В сети может произойти сбой, ведущий к разрыву текущего соединения.

Долговременные соединения страдают от всех помех, связанных с аварийными завершениями соединений, как и простые запросы, но добавляются и дополнительные проблемы. В случае, когда аварийно завершается одиночный запрос, пользователь может просто повторить его (если он сам не инициировал этого завершения) или проигнорировать неудачный запрос (особенно если сам вызвал его завершение). В этом случае аварийное завершение не порождает проблем сохранения состояния для таких методов как **GET** и **HEAD**. Однако в случае методов **PUT** и **POST** может осуществляться попытка обновления содержания на сервере. Пусть последовательность запросов представляет собой конвейер, и пользователь завершает эти запросы до того, как будут получены все ответы. Агент пользователя должен установить, на какие запросы получены ответы, и повторно передать все остальные запросы.

Протокол предписывает, что клиенты должны иметь возможность восстанавливать закрытое соединение, независимо от причины закрытия. Например, возможно, что клиент должен вновь открыть соединение и отправить все запросы, которые не были обработаны. Обычно это не является проблемой. Однако если сама последовательность запросов имеет существенное значение (то есть существует зависимость от того порядка, в котором обрабатывались запросы), агенту пользователя возможно потребуется согласовать свои действия с пользователем и повторно передать прерванные запросы. Например, обработанная часть потока запросов возможно требует обязательной повторной передачи всей совокупности запросов.

Отсутствие взаимодействия между транспортным и прикладным уровнями является причиной неудовлетворительного аварийного завершения соединения в процессе обработки конвейера запросов. Если бы в HTTP имелось бы средство зафиксировать разрыв, не закрывая соединения транспортного уровня, то тогда эту проблему удалось бы преодолеть. Однако зависимость протокола прикладного уровня от протоколов нижних уровней является некорректным проектным решением. Эта тема обсуждается более подробно в главе 8 (раздел 8.2.1).

7.5.5. Закрытие долговременных соединений

При установлении долговременного соединения рождается естественный вопрос, когда это соединение закрывать. Протокол в принятии такого решения не играет никакой роли, оставляя его на усмотрение компонентов, участвующих в соединении. Единственная рекомендация протокола гласит, что соединения по умолчанию должно быть долговременным. Это вынуждает клиента или сервер явно

формулировать отказ от сохранения соединения открытым. Однако не существует рекомендаций относящихся к тому, сколько времени должно существовать соединение или сколько запросов должен обработать сервер с помощью открытого долговременного соединения, прежде чем закрыть его.

Должны быть согласованы следующие противоречивые интересы:

- Серверу нужно обслуживать множество различных клиентов. Сохранение открытого долговременного соединения с каким-то одним клиентом может затруднить обслуживание других клиентов и обеспечение всем клиентам равных прав.
- Сервер может принять решение обработать больше запросов от какого-то клиента по существующему соединению, поскольку если соединение закрыть, то этот клиент, возможно, просто попробует установить новое соединение.
- Серверу может потребоваться закрыть соединение при достижении некоторого лимита времени, чтобы обеспечить равноправие клиентов. Лимит может относиться к общему времени существования соединения в открытом состоянии или ко времени простоя соединения. Это также препятствует попыткам какого-либо клиента получить большую, чем другие, часть ресурсов Web-сервера и полосы пропускания сети.
- Сервер может быть подключен к особому клиенту (или прокси-серверу), может быть нужным, чтобы соединение с таким клиентом продолжало существовать дольше, чем другие; например, в случае различных дисциплин обслуживания клиентов.

Эти и другие темы обсуждались в разумных пределах до стандартизации протокола HTTP/1.1. Был сделан ряд предложений, которые подразумевали использование параметров долговременных соединений (и в запросе, и в ответе). Среди предложений были:

- **timeout**. Данный параметр задает время существования долговременного соединения.
- **max**. Параметр задает максимальное число запросов, которое будут обработано по данному долговременному соединению.
- **state**. Данный параметр используется для указания заголовков, для которых требуется сохранения их состояний.

Однако перечисленные дополнительные параметры создавали проблемы. Параметр **timeout** не имел смысла там, где невозможно было сделать оценки среднего времени прохождения запроса-ответа по сети туда и обратно, или где промежуточные прокси-серверы могли использовать собственные значения параметров. Использование таймаута, даже если он необязателен, лишает сервер возможности управления закрытием соединения. Например, клиент может использовать большое значение таймаута, рассчитывая сохранить соединение открытым в течение длительного периода времени. Однако если затем клиент переходит в режим ожидания и не использует соединение, сервер должен сам принять решение, когда его закрыть. Параметры **timeout** и **max** были реализованы в одной из ранних версий сервера NCSA [Sin95]. Поскольку запросы должны появляться последовательно, знание максимального числа запросов было бесполезным (это было до того, как была реализована конвейеризация). Параметр **state** был предложен, чтобы не нужно было повторно отправлять заголовки, которые не менялись в каждом сообщении. Характерным примером, использованным в предложении по параметру **state**,

были большие заголовки **Accept**. Однако параметр **state** существенно увеличивает издержки, в то же время нельзя определить выгоды, вытекающие из сохранения состояния. Поэтому все указанные выше параметры были удалены из механизма долговременного соединения.

Спецификация умалчивает о том, как долго соединение должно быть открыто. Каждый сервер может использовать свою собственную эвристику, чтобы решить, когда закрывать долговременное соединение. Фактически при реализации долговременных соединений для Web-серверов часто принимают во внимание совокупность факторов, прежде чем решить, закрывать долговременное соединение или нет.

HTTP/1.1 накладывает ограничения на некоторые методы при работе с преждевременно закрытыми соединениями. Например, HTTP/1.1 требует, чтобы запросы **POST** могли восстанавливать свое состояние после преждевременного закрытия. Клиент должен немедленно прекратить передачу сообщения, как только ему станет известно, что соединение было закрыто. Клиент должен повторить запрос, если он не получил ответа от сервера. Попытка может повторяться после того, как пройдет некоторое время несколько раз, пока не будет получен код состояния (4xx или 5xx).

7.6. Передача сообщений

Основная задача при обмене HTTP-сообщениями состоит в том, чтобы стороны могли распознать полное и без каких-либо потерь получение сообщений. Надежная доставка сообщения по сети — главное для поддержания целостности транзакций. Размер ответа — это полезный признак, позволяющий получателю определить, что получен весь ответ. Единственным средством, с помощью которого исходные серверы, использующие HTTP/1.0, могли сообщить размер тела содержимого, было поле заголовка **Content-Length**. Длина статического ресурса могла быть легко определена с помощью системного вызова операционной системы. При динамически генерируемых ответах, исходному серверу приходится ждать завершения процесса генерации, прежде чем вычислить длину ответа. До того как ответ полностью сформирован, поле заголовка **Content-Length** не может быть заполнено. Следовательно, исходный сервер не может начать передачу ответа до того, как будет заполнено поле заголовка. Это требует буферизации ответа, увеличивая таким образом задержку для конечного пользователя.

В HTTP/1.0 сервер указывает на конец динамически генерируемого ресурса путем закрытия соединения. Если закрытие соединения — единственный способ указать на завершение ответа, то долговременные соединения становятся невозможными. Таким образом, был нужен альтернативный способ задания признака окончания сообщения без закрытия соединения.

HTTP/1.1 решает проблему надежной передачи сообщений с помощью *разбиения тела сообщения на фрагменты* (*chunks*), что позволяет отправителю передавать их независимо. Каждой части предшествует ее размер, позволяющий получателю убедиться, что он получил данный фрагмент полностью. Еще важнее то, что отправитель генерирует фрагмент нулевой длины при завершении передачи сообщения, получение его указывает, что все сообщение было благополучно передано. Разбиение сообщения на фрагменты избавляет от необходимости поддержки буферов произвольного размера, которые могут потребоваться, если прокси-сервер будет сохранять ответ до его передачи устаревшей версии клиента, использующей HTTP/1.0. Этот механизм также устраняет задержку, связанную с ожиданием за-

вершения генерации всего ответа перед добавлением заголовка **Content-Length** (как это обсуждалось выше).

Рассмотрим следующий пример ответа разбитого на фрагменты:

```
HTTP/1.1 200 OK
Server: Apache/1.2.7-dev
Date: Tue, 07 Jul 1998 18:21:41 GMT
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html
691
        <...1681 байтов первого фрагмента данных...>
76
        <...118 байтов второго фрагмента данных...>
0
```

Заголовок **Transfer-Encoding** (имеющий значение **chunked**) указывает получателю, что ответ разбит на фрагменты, и, следовательно, этот ответ должен анализироваться иначе, чем ответы, не разбитые на фрагменты. Ответ, приведенный в примере, разбит на три фрагмента: первому фрагменту длиной 1681 байт (691 в шестнадцатеричной системе счисления), предшествует указатель длины части, за первым фрагментом идет второй фрагмент длиной 118 байтов (шестнадцатеричное 76), которому также предшествует указатель длины фрагмента. Наконец, следует фрагмент нулевой длины — указатель размера фрагмента 0 без следующего за ним тела, что является для получателя признаком завершения ответа. Получатель проверяет длину отдельных фрагментов, а после получения фрагмента нулевой длины он знает, что ответ им был принят полностью. В случае динамически генерируемых ответов сервер может отправлять генерируемые фрагменты в том виде и тогда, когда они были сгенерированы. Это сокращает задержку за счет того, что не нужно ждать завершения процесса генерации всего ответа, прежде чем посылать какой-либо его фрагмент.

Разбиение сообщения на фрагменты не оказывает влияния на то, как транспортный уровень может воспринимать HTTP-сообщение. Один фрагмент может разбиваться на несколько пакетов, или в одном пакете может поместиться несколько фрагментов. Разбиение на фрагменты — это механизм, который полностью реализуется на прикладном уровне. Разбивать на фрагменты можно как сообщения-запросы, так и сообщения-ответы.

Сообщение-ответ, разбитый на фрагменты, завершается фрагментом нулевой длины. Однако за сообщением может следовать необязательный завершающий фрагмент, трейлер (*trailer*). Трейлер является отдельной от тела ответа частью сообщения, и сервер-получатель или агент пользователя определят, что уже получено все тело до обработки следующего за ним трейлера. Трейлер в ответе, разбитом на фрагменты, может состоять только из полей и значений заголовка содержимого. Предположим, что ответ генерируется динамически и отправителю нужно вычислить дайджест всего ответа (можно считать дайджест обычной контрольной суммой). Например, отправителю может потребоваться включить в ответ контрольную сумму для проверки его целостности (например, с использованием алгоритма MD5 [Riv92]), но до того как будут обработаны все байты ответа, невозможно вычислить контрольную сумму. Однако если включить заголовок MD5 в трейлер ответа, то фрагменты тела ответа можно отправлять по мере их генерации, без увеличения задержки. Подобным же образом, трейлер может включать заголовок **Authen-**

tication-Info, как это описано в RFC, посвященном аутентификации с помощью дайджестов [FHBH+99].

Чтобы предупредить получателя о наличии дополнительной информации, заголовок **Trailer** используется для перечисления других заголовков, которые появятся в трейлере сообщения. Трейлеры могут включать обязательную и необязательную информацию. Готовность принять трейлеры на стороне клиента явным образом указывает на то, что клиент будет, если это потребуется, помещать информацию, предшествующую трейлеру, в буфер. Например, исходный сервер может включить в ответ некоторую аутентификационную информацию, которая определяет, должен ли принимающий прокси-сервер передавать ответ дальше или нет. Однако это означает, что принимающему прокси-серверу придется поместить весь ответ в буфер. В соответствии с духом симметричного управления HTTP/1.1 требует, чтобы трейлеры посылались в ответе, разбитом на фрагменты, только в том случае, если прокси-сервер ранее подтвердил свою готовность помещать, если нужно, весь ответ в буфер. Это достигается включением заголовка запроса **TE:trailers**.

Отправитель посылает следующий запрос, чтобы указать на свою готовность принимать поля трейлера в ответе, разбитом на фрагменты:

```
GET /foo.html HTTP/1.1
Host: www.checktrailers.com
TE: trailers
```

Сервер возвращает разбитый на фрагменты ответ с заголовком **Trailer**, указывающим, какие заголовки получатель должен найти в завершающем фрагменте ответа:

```
HTTP/1.1 200 OK
Trailer: Splinfo
Transfer-encoding: chunked
CRLF
691
        <1681 байтов первого фрагмента данных>
76
        <118 байтов второго фрагмента данных >
0
Splinfo: vol=7; pe="u4, 895527629, 5465";
CRLF
```

Разбитый на фрагменты ответ состоит из обычного набора фрагментов с указателями длины, после которых следует фрагмент нулевой длины. Сразу за фрагментом нулевой длины, сервер включает трейлер с заголовком **Splinfo** (указанным в заголовке **Trailer**) и набор значений для этого заголовка. Последняя строка представляет собой символы возврата каретки и перевода строки (CRLF), которые должны завершать разбитое на фрагменты сообщение.

В то время как HTTP/1.0 требует наличия корректного поля **Content-Length** во всех запросах **POST**, HTTP/1.1 этого не требует. В HTTP/1.0 не было другого способа отделить тело содержимого, в HTTP/1.1 это можно сделать с помощью механизма разбиения на фрагменты. Заметим, что разделенные на фрагменты сообщения не обязаны иметь заголовок **Content-Length**, так как длина такого сообщения вычисляется на основе информации о длине фрагментов. Предположим, что сервер, поддерживающий HTTP/1.1, посылает разбитое на фрагменты сообщение прокси-серверу также поддерживающему HTTP/1.1, который должен передать его клиенту, использующему HTTP/1.0. Если в сообщение включен заголовок **Content-Length**, прокси-сервер, использующий HTTP/1.1, проигнорирует этот заголо-

вок. Так как клиенту ничего неизвестно о механизме разбиения на фрагменты, прокси-сервер, использующий HTTP/1.1, из полученных фрагментов создаст обычное сообщение и передаст его клиенту, используя HTTP/1.0, вместе с заголовком **Content-Length**.

7.7. Расширяемость

Разработчики протоколов при создании спецификации часто оставляют возможность внесения изменений. Любому протоколу неизбежно приходится развиваться. Хотя некоторые изменения можно предусмотреть на этапе создания спецификации протокола, все предусмотреть невозможно. В случае с HTTP, существовала дополнительная проблема: имелось несколько популярных реализаций различных Web-компонентов, разработанных на базе промежуточных «стандартов». Версия HTTP/1.1 должна была обеспечить совместимость с существующими реализациями любых сделанных изменений. Это было сделано для того, чтобы браузеры и серверы, совместимые со старыми версиями протокола, продолжали функционировать, пока не будут разработаны и станут популярными новые версии компонентов. Вряд ли прежние версии исчезнут за несколько лет.

Учитывая прошлые уроки, разработчики HTTP/1.1 решили зарезервировать некоторые возможности для *будущих расширений*. Были созданы возможности, не ограниченные теми приложениями, в рамках которых они появлялись. Была проявлена забота о том, чтобы не фиксировать возможные значения конкретных параметров. В условиях большого числа проблем с совместимостью, с которыми столкнулись разработчики протокола HTTP/1.1, они попытались ограничить будущие трудности в следующих версиях этого протокола.

В HTTP/1.1 были предприняты попытки обеспечения расширяемости с помощью следующих трех способов, о чем говорится в последующих разделах:

1. Путем включения методов для получения информации о возможностях сервера до осуществления запросов и для получения информации о том, что действительно получено сервером.
2. Путем добавления новых заголовков для получения информации о промежуточных серверах, включенных в цепочку Web-транзакции.
3. Путем введения поддержки для перехода на другие протоколы.

7.7.1. Получение информации о сервере

Два новых метода получения информации о сервере были введены в HTTP/1.1. Это методы **OPTIONS** и **TRACE**. Мы рассмотрим их здесь подробно.

МЕТОД OPTIONS

В HTTP/1.0 у клиента не было средств узнать о возможностях Web-сервера. Метод **OPTIONS** был введен в HTTP/1.1, чтобы удовлетворить эту потребность.

Рассмотрим следующие ситуации:

- Возможно, что клиенту нужно узнать, реализован ли в Web-сервере метод, не являющийся обязательным для реализации (любой метод, не относящийся к уровню требований MUST). Предположим, что клиентом был послан запрос **PUT** большого объема, который был отклонен сервером из-за невозможности

обработать. Передача отклоняемых сервером запросов большого объема приводит к непроизводительной загрузке сети.

- Возможно, что клиенту нужно узнать, существуют ли какие-то особые требования, связанные с конкретным ресурсом.
- Возможно, агенту пользователя нужно узнать о возможностях прокси-сервера, находящегося на пути к Web-серверу.
- Возможно, что прокси-серверу нужно прозондировать сервер, расположенный выше в цепочке к исходному серверу и претендующий на соответствие с HTTP/1.1, чтобы узнать, может ли он работать с конкретной возможностью HTTP/1.1, например, **Expect**.

По всем перечисленным причинам в HTTP/1.1 был введен новый метод **OPTIONS**. Запрос **OPTIONS** может быть направлен не только серверу-источнику, но и любому серверу на пути от отправителя к получателю. Метод **OPTIONS**, подобно методам **GET** и **HEAD**, является безопасным методом.

Например, чтобы получить список методов, поддерживаемых сервером, клиент может отправить следующий запрос **OPTIONS**:

```
OPTIONS * HTTP/1.1
Host: foo.com
```

Сервер может ответить так

```
HTTP/1.1 200 OK
Allow: HEAD, GET, POST, TRACE, OPTIONS
```

Этот ответ показывает, что сервер поддерживает пять методов, перечисленных в заголовке ответа **Allow**. Заметьте, что URI, заданный в примере, представляет собой "*". Это означает, что клиента интересуют возможности сервера, а не какого-то конкретного URI. Для конкретного URI заголовок ответа **Allow** вернет набор методов, применимых к данному ресурсу.

Чтобы агент пользователя мог получить информацию о конкретном прокси-сервере, задействованном в цепочке передачи запроса, он должен убедиться с помощью метода **OPTIONS**, что данный запрос обработан нужным прокси-сервером. Обычно прокси-серверы пересылают запросы, которые они не могут обработать, исходному серверу. Для того чтобы клиенты могли получить ответ от конкретного прокси-сервера, в HTTP/1.1 был введен новый заголовок запроса. Клиент задает значение для заголовка запроса **Max-Forwards**, вынуждая каждый сервер в цепочке уменьшать это значение на единицу, пока оно не станет равным нулю, после чего соответствующий прокси-сервер должен ответить на запрос **OPTIONS**. Заголовок **Max-Forwards** был включен в протокол и для выбора в качестве цели прокси-сервера и для обнаружения циклов в цепочке прокси-серверов. Это поле похоже на поле времени жизни (TTL) в IP-пакетах (об этом говорилось в главе 5, раздел 5.1.4).

Рассмотрим следующий пример, где задан конкретный сервер в цепочке передачи данных:

```
OPTIONS /bar HTTP/1.1
Host: foo.com
User-Agent: Mozilla/2.0
Max-Forwards: 1
```

Этот запрос будет передан первому получателю, который вычтет единицу из заголовка **Max-Forwards** и передаст запрос следующему получателю. Второй получа-

тель в цепочке должен ответить на запрос, так как значение **Max-Forwards** теперь равно нулю.

Были определены дополнительные возможности расширения для этого нового метода. Например, тело, включенное в заголовок **OPTIONS**, может быть факультативно использовано исходным сервером для передачи дополнительной информации. Реальный синтаксис возможных тел в запросе **OPTIONS** не был определен и был оставлен открытым для будущих расширений. Это еще один пример постепенного развития протокола, без выдвижения непосильных требований к Web-компонентам реализовать все изменения сразу.

В дискуссиях, предшествовавших стандартизации, были сделаны попытки разрешить в ответах на запросы с заголовком **OPTIONS** возвращать информацию о совместимости, например, о совместимости с конкретной версией стандарта или несовместимости по отношению к одной из его возможностей. Однако из-за неопределенности связанной с термином *совместимость* и из-за увеличения сложности, эта возможность не была введена в HTTP. Было добавлено расширение в виде **DAV,** заголовка для расширения HTTP, называемое WebDAV [GWF+99], которое обеспечивает внесение изменений в ресурсы и управление версиями. Эта тема кратко обсуждается в главе 15 (раздел 15.5.2). WebDAV позволяет осуществлять удаленную разработку Web-контента с помощью набора методов, заголовков, запросов, ответов и форматов тела содержимого. Сервер WebDAV при получении запроса с заголовком **OPTIONS** может перечислить свои возможности с помощью заголовка **DAV**. Например, **DAV**:1 означает, что сервер удовлетворяет всем обязательным требованиям (уровень требований MUST), но не поддерживает блокировку. Заголовок **DAV:1, 2** означает, что сервер поддерживает в том числе и блокировку.

МЕТОД TRACE

Метод **TRACE**, введенный в HTTP/1.1, позволяет клиенту узнать содержание сообщения, которое было действительно принято получателем. В отличие от метода **OPTIONS**, который позволяет клиенту узнать о возможностях сервера, метод **TRACE** побуждает сервер вернуть копию полученного сообщения.

В методе **TRACE** все содержимое сообщения-запроса сервер возвращает отправителю в качестве содержимого тела ответа. Например, если клиент сделал следующий запрос:

```
TRACE /bar HTTP/1.1
Host: foo.com
User-Agent: Mozilla/2.0
```

результатом должен быть следующий ответ:

```
HTTP/1.1 200 OK
Content-type: message/http

TRACE /bar HTTP/1.1
Host: foo.com
User-Agent: Mozilla/2.0
```

Телом ответа должно быть полное HTTP-сообщение, полученное целевым сервером. Конечно, как и в любом HTTP-ответе, здесь есть заголовки и тело содержимого. Заголовок **Content-type** используется, чтобы указать, что ответ на запрос с методом **TRACE** имеет тип **message/http**. Тип **message/http** указывает, что тело ответа является HTTP-запросом или ответом и что он подчиняется соответствую-

щим соглашениям MIME для данного типа (например, предельному значению длины строки и допустимым схемам кодирования).

Такого рода обратная связь на прикладном уровне (отправка запроса и получение его обратно) не является чем-то новым, имеющимся только в HTTP. В других протоколах есть похожие механизмы — один из наиболее известных **traceroute** [Jac] (см. главу 5, раздел 5.1.4). Команда **traceroute** отслеживает маршрут, который проходит IP-пакет предназначенный хосту, путем посылки тестовых UDP-пакетов с различными значениями TTL и затем осуществляя прием по протоколу Internet Control Message Protocol (ICMP) ответов **TIME_EXCEEDED**, приходящих от маршрутизаторов на пути следования пакета.

7.7.2. Получение информации о промежуточных серверах

HTTP/1.1 уделяет больше внимания промежуточным средствам в цепи транзакций запрос-ответ. Промежуточное средство в HTTP/1.1 (прокси-сервер или шлюз) добавляет идентификационную информацию о себе и о том сервере, от которого он получил сообщение, в новом заголовке **Via**.

Заголовок **Via** является средством расширения и обеспечения взаимодействия для клиентов и серверов, использующих HTTP/1.1. Когда пользователь создает Web-запрос, то этот запрос может пройти через несколько промежуточных серверов на пути к серверу-источнику. Промежуточные серверы могут быть прокси-серверами, шлюзами или внешними интерфейсами серверов. Аналогично, Web-сервер может быть заинтересован в информации о различных промежуточных серверах, через которые проходит запрос. Фактически, любые из серверов, входящих в цепочку, могут быть заинтересованы в получении информации о промежуточных серверах. Здесь присутствует историческая аналогия с некоторыми командами, которые имеются на нижних уровнях стека протоколов. Способность отслеживать хосты по пути следования IP-пакета была реализована в утилите **traceroute** [Jac] почти за десятилетие до появления Web. Заголовок **Via** является частичным воспроизведением этой возможности на более высоком уровне в стеке протоколов. Сам заголовок **Via** был создан после определения поля **Received** в RFC 822 (стандарт формата текстовых сообщений Internet [Cro82]). Он также похож на механизм в протоколе Border Gateway Protocol (BGP) [RL95, Ste99], где каждая автономная система добавляет свой номер (ASN) к атрибуту AS-PATH.

До того как был предложен заголовок **Via**, существовало предложение по другому заголовку с названием **Forwarded** [FFBL95]. Этот заголовок предназначался для указания переходов между клиентом и сервером в обоих направлениях. Но сюда включались только имена серверов между клиентом и Web-сервером. После добавления возможности получать информацию также и о версии протокола, заголовок **Forwarded** был переименован в заголовок **Via**.

Все HTTP/1.1 прокси-серверы и шлюзы обязаны участвовать (это требование уровня MUST) в механизме **Via**. Промежуточные средства обязаны идентифицировать свое имя и номер версии *предшествующего* сервера, от которого получен запрос или ответ. Из соображений конфиденциальности промежуточное средство может использовать псевдоним вместо настоящего имени хоста. Прокси-серверы или шлюзы, использующие HTTP/1.1 (хотя на практике прокси-серверы, использующие HTTP/1.0, также могут принимать в этом участие) добавляют идентификационную информацию к новому общему заголовку **Via** последовательно по мере продвижения от отправителя к получателю. Таким образом, конечный получатель имеет информацию о серверах по пути следования. Хотя это не так уж необычно

встретить прокси-серверы, использующие HTTP/1.0, которые были избирательно модернизированы, так чтобы они могли добавлять заголовок **Via**, большинству HTTP/1.0 прокси-серверов о механизме **Via** ничего неизвестно.

Например, сервер может получить следующий заголовок **Via**:

```
Via: 1.0 M-PROXY3:8080 (Squid/2.1.PATCH2),
     1.0 tl.us.irc.net:3128 (Squid/2.3.DEVEL1),
     1.1 qd.us.ircache.net:3128 (Squid/2.3.DEVEL1)
```

указывающий, что на пути было три промежуточных сервера, участвовавших в механизме Via. Первый элемент в заголовке **Via**

```
1.0 M-PROXY3:8080 (Squid/2.1.PATCH2)
```

указывает, что сервер, поддерживающий HTTP/1.0, непосредственно предшествовал компьютеру M-PROXY3, и на нем функционировал прокси-сервер Squid 2.1 на порту 8080. Последним из промежуточных средств, участвовавших в механизме Via, является **qd.us.ircache.net**, на котором на порту 3128 функционировал прокси-сервер Squid 2.3, а его предшественником был сервер, поддерживающий HTTP/1.1.

После получения запроса исходному серверу известно сколько прокси-серверов/шлюзов, поддерживающих HTTP/1.1, было на пути сообщения и возможно ему также известно о тех серверах, поддерживающих HTTP/1.0, которые подключены к прокси-серверам, поддерживающим HTTP/1.1. С другой стороны, следует сделать поправку на то, что несколько прокси-серверов одной и той же версии под одним и тем же административным управлением, сгруппированные из соображений безопасности, могут помешать определить точное количество прокси-серверов на пути сообщения. Заголовок **Via** является сквозным заголовком, и поэтому все промежуточные прокси-серверы пересылают его дальше.

Заголовок **Via** используется в сочетании с методом **TRACE** (обсуждался в разделе 7.7.1) для получения информации о пути, пройденном HTTP-сообщением. Заголовок **Via** включает информацию о различных прокси-серверах, поддерживающих HTTP/1.1, или шлюзах, которые могут располагаться на пути между клиентом и Web-сервером. Клиент все же не должен рассылать запрос **TRACE** по всему пути следования до Web-сервера, вместо этого запрос может быть послан одному из прокси-серверов, находящихся на пути сообщения. Возможно, это не одноразовое действие; клиенту может быть придется повторить эту операцию несколько раз, пока он сможет локализовать конкретный прокси-сервер. Несколько прокси-серверов под одним административным управлением, у которых один и тот же номер версии, часто группируются и помечаются одной строкой в заголовке **Via** по соображениям обеспечения безопасности и конфиденциальности. Также как и в методе **OPTIONS**, можно использовать заголовок **Max-Forwards**, чтобы ограничить число переходов, которые пройдет запрос с методом **TRACE**.

Механизм **Via** также можно использовать, чтобы избежать циклов на пути запроса. При обнаружении своего имени в заголовке **Via** прокси-сервер может принимать решение о прекращении пересылки данного сообщения.

7.7.3. Переход к другим протоколам

Весьма вероятна разработка в будущем новых версий HTTP и других протоколов. В HTTP/1.1 имеется возможность перехода на новый протокол, для чего в HTTP/1.1 был введен новый заголовок промежуточных передач **Upgrade**. Когда отправитель включает заголовок **Upgrade** и набор поддерживаемых протоколов,

сервер-получатель может переключиться на один из этих протоколов для данной транзакции или для любых дополнительных транзакций поверх существующего соединения транспортного уровня. Сервер должен указать, на какой протокол он переключился с помощью заголовка ответа **101 Switching Protocols**.

Рассмотрим следующий пример, в котором клиент и сервер, использующие HTTP/1.1, выражают желание перевести существующее между ними соединение на другой протокол.

```
GET http://security.are.us/private-data HTTP/1.1
Host: chase.bronx.com
Upgrade: SafeBank/1.0
Connection: Upgrade
```

Клиент запрашивает ресурс **security.are.us/private-data** и извещает, что он может работать по протоколу **SafeBank/1.0**. Сервер **chase.bronx.com** может ответить по протоколу HTTP/1.1 или в качестве альтернативы переключиться на протокол **SafeBank/1.0** и сообщить об этом с помощью кода ответа **101 Switching Protocols**:

```
HTTP/1.1 101 Switching Protocols
Upgrade: SafeBank/1.0
Connection: Upgrade

<Ответ на запрос клиента>
```

Любое дальнейшее взаимодействие по *текущему* соединению транспортного уровня может теперь выполняться по протоколу **SafeBank/1.0**. Чтобы использовать протокол, отличный от HTTP/1.1, для последующих соединений, сервер должен использовать соответствующие ответы переадресации класса (3xx).

Так как **Upgrade** представляет собой заголовок промежуточных передач, то он удаляется прокси-серверами, прежде чем сообщение пересылается дальше. Сервер-получатель может проигнорировать заголовок **Upgrade**, если он или не воспринимает этот заголовок, или не намерен переходить на другой протокол. Обратите внимание, что переключение распространяется только на текущее соединение транспортного уровня, так как **Upgrade** является заголовком промежуточных передач.

7.8. Сбережение Internet-адресов

В Internet используется четвертая версия протокола IP, который предоставляет максимум 2^{32} адресов хостов. В течение нескольких лет количество адресов, предоставляемых IPv4, рассматривалось как более чем достаточное. Однако популярность Web и быстрое осознание частью бизнесменов, что короткие имена Web-сайтов являются запоминающимися, привело к бурному росту числа доменных имен Web-сайтов типа **www.foo.com**. Эти «чистые» URL хостов состоят только из имени хоста, их легко запомнить. Однако каждое доменное имя хоста требовало отдельного IP-адреса в реализациях HTTP/1.0.

Когда клиент соединяется с сервером, поддерживающим HTTP/1.0, по адресу **www.foo.com** для того, чтобы получить ресурс **http://www.foo.com/bar.html**, в первой строке запроса содержится только

```
GET /bar.hml HTTP/1.0
```

Агент пользователя удаляет имя сервера из URL в HTTP-запросе. Имя сервера было необходимо для установления соединения транспортного уровня, и к тому

времени, когда HTTP-сервер получил сообщение, в нем больше не было надобности.

Популярность Web привела к появлению компаний, занимающихся Web-хостингом. Любая компания, у которой не было желания заниматься сопровождением своего Web-сайта, поручала это другой компании. Такие компании, занимающиеся Web-хостингом, управляют Web-сайтами большого числа компаний. Тем не менее, каждая компания изъявляла желание иметь для сайта собственное доменное имя (например, **http://www.foo.com**), а не доменное имя какой-то другой компании, занимающейся Web-хостингом (например, **www.hostmany.com/foo**). Неспособность сервера извлечь имя сервера из URL означала, что все ресурсы должны были размещаться либо под одним доменным именем, либо нужно было вводить новые доменные имена хостов. Рассмотрим, например, пользователя, запрашивающего ресурс **http://www.serverA.com/bar.html.** Web-сервер с доменным именем **www.serverA.com** получит следующий запрос:

```
GET /bar.html HTTP/1.0
```

В этой ситуации другой Web-сервер не может работать на том же хосте под другим доменным именем **www.serverB.com**, поскольку при этом появляется неоднозначность в интерпретации запроса для **/bar.html**. Оба сервера в равной степени могут иметь право ответить на этот запрос, и не существует способа разрешить эту неоднозначность.

Желание иметь «престижные» URL привела к назначению отдельных IP-адресов для каждого доменного имени. Быстрое увеличение числа Web-сайтов и рост популярности Internet привели к быстрому уменьшению числа свободных IP-адресов. Новая версия IPv6 [DH98] может помочь избежать недостатка IP-адресов. В IPv6 адреса являются 128-разрядными, что разительно увеличивает число доступных адресов по сравнению с 32-разрядным адресным пространством IPv4. На момент публикации данной книги большей части Internet еще только предстоит перейти на IPv6.

Чтобы избежать нехватки IP-адресов, исполнительный комитет IETF, ответственный за координацию процесса стандартизации, уполномочил HTTP Working Group принять меры по сбережению IP-адресов в процессе перехода на HTTP/1.1. У HTTP Working Group был вариант изменить строку запроса, чтобы включить имя хоста следующим образом:

```
GET www.foo.com/bar.hml HTTP/1.0
```

Однако существующие HTTP/1.0 сервера не смогут правильно обрабатывать такую строку запроса. Требование обратной совместимости не может быть нарушено в процессе эволюции протокола, поэтому был предложен компромисс. Сначала было предложено ввести заголовок **Original-URI**, который содержал бы полный URI (включая имя сервера). В процессе дискуссий по развитию протокола, было решено, что такой заголовок был бы длинным и содержал бы избыточную информацию. Так появилась строка заголовка **Host**. Приведенный выше запрос в HTTP/1.1 выглядит так

```
GET /bar.hml HTTP/1.1
Host: www.foo.com
```

Все запросы в HTTP/1.1 *должны* иметь поле заголовка **Host**. Можно также включать номер порта, но если он не указан, по умолчанию предполагается порт с номером 80. Серверы, поддерживающие HTTP/1.1, которые получают запрос от клиента, поддерживающего HTTP/1.1 без заголовка **Host**, обязаны отклонить такой запрос. Надежда здесь на то, что такое строгое требование поможет сберечь

IP-адреса. Поскольку клиентам, поддерживающим HTTP/1.0, ничего неизвестно о заголовке **Host**, HTTP/1.1 серверы будут продолжать принимать запросы HTTP/1.0, у которых нет данного заголовка.

Web-сайтам больше не нужен IP-адрес для того, чтобы иметь собственное доменное имя, например, **www.foo.com**. Множество доменных имен может отображаться на IP-адрес с помощью DNS. Такой метод виртуального хостинга позволяет Web-серверу принимать запросы к нескольким разным Web-сайтам, которые установлены на нем, и перенаправлять запросы. Однако виртуальный хостинг, основанный на доменных именах, не будет работать с SSL, так как при этом шифруется весь заголовок. Заметим, что клиенту для того, чтобы посылать заголовки **Host**, не требуется поддерживать HTTP/1.1. На самом деле, популярные браузеры, которые продолжают использовать HTTP/1.0, включают в сообщение заголовок **Host**.

7.9. Согласование содержания

Если существует единственное представление ресурса, то для запроса такого ресурса можно использовать простое сообщение-запрос HTTP. Но если один и тот же ресурс представлен в нескольких форматах, то клиент и сервер должны будут провести согласование, чтобы данный клиент мог получить ресурс в предпочтительном для него формате. В начале ресурсы в Web были только на английском языке, недавние исследования показали, что даже сейчас большая часть содержания представлена на английском языке [LG99]. Содержимое, представленное на другом языке, было доступно *только* на этом языке (например, на французском, русском, датском). С ростом Web и по мере того, как одно и то же содержание становилось доступным на разных языках, возникал естественный вопрос о возможности выбора различных вариантов одного и того же ресурса. Ресурсы отличались не только языком, но также наборами символов и кодировками. Список возможных наборов символов, например, ASCII, ISO (включая кодировки латиницы или кириллицы), сжатия (gzip и т.д.), зарегистрированы в Internet Assigning Numbers Authority (IANA). Согласование содержания в HTTP было разработано, чтобы помочь клиентам и серверам договариваться о форматах ресурса.

Аспекты, допускающие согласование, могут расширяться. Правила согласования содержания, определенные в HTTP/1.1, будут применимы и к расширениям. Согласование содержания может выполняться динамически. Клиенты могут получать информацию о доступных формах представления содержания и выбирать после согласования с сервером приемлемое представление.

В 1993 г. в одном из первых проектов HTTP [BL93a] довольно детально обсуждалось согласование содержания. В некоторых промежуточных проектах HTTP/1.0 обсуждались различные, связанные с согласованием содержания заголовки, такие как **Accept** и **Accept-Charset**. Браузер Mosaic включал заголовки **Accept** в свои запросы, а первые версии сервера Apache включали поддержку согласования содержания. Расширение возможностей согласования с самого начала было темой, представляющей интерес для HTTP Working Group [Sec95]. Поскольку к моменту завершения документа RFC 1945, описывающего HTTP/1.0, консенсус не был достигнут, дискуссия, относящаяся к пяти заголовкам запросов (**Accept, Accept-Charset, Accept-Encoding, Accept-Language** и **Content-Language**) была вынесена в приложение. В HTTP/1.1 не только оформлены результаты этой дискуссии о согласовании содержания, но и предложены два различных типа согласования:

- **Управляемое агентом.** При управляемом агентом согласовании содержания клиент получает набор альтернативных представлений ответа, выбирает подходящее представление и указывает его в следующем запросе.
- **Управляемое сервером.** При управляемом сервером согласовании содержания сервер выбирает представление, основываясь на том, чем располагает сервер, на заголовках сообщения-запроса или на информации о клиенте, например, на его IP-адресе.

Задержка с реализацией общего для множества клиентов и серверов механизма согласования привела к отделению этих усилий от работы над спецификацией HTTP.

В HTTP/1.0 из-за отсутствия согласования, управляемого агентом, у пользователя было меньше возможностей выбора конкретного варианта ресурса. В HTTP/1.1 отправитель может указать в заголовке **Accept** характеристики содержания, которые он готов принять (например, текст или аудио). Рассмотрим следующий пример:

```
GET /asterix.html HTTP/1.1
Host: www.getobelix.com
Accept-Language: en-us, fr-BE
```

Агент пользователя запрашивает ресурс **asterix.html** и готов принять или вариант на американском английском или бельгийском французском языках. Сервер может выбрать между этими вариантами и вернуть ответ в следующем виде:

```
HTTP/1.1 200 OK
Content-Length: 23819
Content-Language: fr-BE
...
<вариант ответа на бельгийском французском языке>
...
```

Сервер-источник решил выбрать вариант ресурса на бельгийском французском языке, как это указано в заголовке **Content-Language**. Заголовок содержимого **Content-Language** указывает на естественный язык тела содержания и, следовательно, на тот язык, который был выбран получателем. Если содержание представлено на различных естественных языках, то этот заголовок является хорошим способом их различить. Тем не менее, в заголовке может быть перечислено несколько языков, если содержимое может восприниматься более широкой аудиторией. Вот пример использования заголовка **Content-Language** с несколькими языками

```
Content-Language: en-cockney, x-pig-latin
```

в котором говорится, что содержание предназначено для тех, кто говорит на кокни (диалекте английского языка), или для тех, кто говорит на Pig Latin (одном из вариантов латинского языка).

Агенты пользователя могут выражать свои предпочтения в заголовке в терминах, приемлемых типов содержания, но это обычно не имеет смысла, если сервер-источник не имеет возможности предоставить содержание в нужной форме. Агенты пользователя и клиенты могут согласовывать и другие аспекты, например кодировку, но единственным арбитром в согласованиях, связанных с содержанием, является Web-сервер.

HTTP/1.1 разрешает серверу иметь свой собственный алгоритм генерации списка форматов и затем использовать информацию, представленную в заголовках запроса для настройки ответа. Одной из побудительных причин для сервера посылать «наилучший» ответ связан с тем, чтобы избежать получения от клиента запро-

са на другой вариант содержания. Если запрашиваемый формат отсутствует, сервер может вернуть ответ **406 Not Acceptable**. Код **406 Not Acceptable** был введен для реализации управляемого агентом согласования. Сервер также включает информацию о различных свойствах ресурса так, чтобы отправитель запроса мог выбрать из числа доступных вариантов. Оборотной стороной ответа **406 Not Acceptable** является дополнительный запрос, который необходим, чтобы получить приемлемый вариант из списка. Если сервер располагает всего одним вариантом, тогда надо дать ему возможность вернуть содержание, проигнорировав заголовок **Accept**. Спецификация считает, что предпочтительнее вернуть вариант, отличный от тех, которые перечислены как приемлемые, чем ответ **406 Not Acceptable**.

Агенту пользователя сложно передать исходному серверу алгоритм выбора в запросе, поэтому лучшим местом для алгоритма выбора является Web-сервер. Конфигурационный файл сервера является подходящим местом для определения алгоритма выбора. Иначе говоря, сервер поддерживает алгоритм, который определяет лучший вариант ответа. Однако управляемого сервером согласования бывает достаточно не всегда. Уверенность, что агент пользователя располагает выбором пользователя из перечня вариантов, приводит к тому, что сервер не будет способен выполнить эту задачу автоматически.

Код ответа **300 Multiple Choices** в HTTP/1.1 был расширен в основном в результате введения согласования содержания. Этот код используется для указания, что ресурс можно выбирать путем согласования из ряда возможных представлений, расположенных в различных местах. Сервер-источник может указать *предпочтительное* представление ресурса, но переадресация позволяет агенту пользователя сделать правильный выбор.

Одним из способов для агентов пользователя и исходных серверов определить степень предпочтительности конкретного формата являются оценки качества, известные в HTTP как **qvalue**. Как отмечалось в разделе 7.4.3, *qvalue* — это число в диапазоне от 0.0 до 1.0, где меньшее значение оценки качества указывает на меньшее предпочтение, а большее значение оценки качества указывает на большее предпочтение. Оценки качества были впервые предложены в первом проекте документа, описывающего согласование содержания [BL93a], но формализованы были только в HTTP/1.1 после того, как были реализованы компоненты с возможностью согласования. Вес конкретного параметра может теперь задаваться и в запросе, и в ответе. Смысл нулевой оценки качества заключается в том, что содержание с этим конкретным значением параметра неприемлемо для клиента. К сожалению, термин *оценка качества* был выбран до полного определения области его применения; на самом деле оценки качества задают *снижение* качества относительно идеального уровня. Оценка качества 1.0 является идеальной, а 0.9 на 10% хуже, чем искомый идеал.

Согласование содержания является гибким механизмом в том аспекте, что можно задавать не только различные параметры, но и каждый параметр может иметь различные варианты со своими уровнями приемлемости. Например, различные оценки качества могут быть заданы для форматов кодирования:

```
Accept-Encoding: vdelta;q=1.0, gzip;q=0.5, compress;q=0.1
```

Этот пример показывает, что клиент отдает наибольшее предпочтение кодировке ответа **vdelta**. Если сервер не может предложить клиенту этот формат, тогда следует попробовать предложить ресурс в формате **gzip**, а если и это не получается, то в формате **compress**. В этом примере сервер может посылать ответ в одном из указанных форматов или в своем собственном формате. Если клиент вместо этого задал бы

```
Accept-Encoding: vdelta;q=1.0, gzip;q=0.5, compress;q=0.1, identity;q=0
```

тогда сервер имеет ограниченный выбор послать ответ в одном из трех **форматов (vdelta, gzip, compress)**, или послать код ответа **406 Not Acceptable**. Если опустить строку **identity;q=0**, то сервер мог бы послать ответ в кодировке по своему выбору.

Произвольное добавление возможных форматов к заголовку запроса или ответа увеличивает количество байтов, передаваемых по сети. Первые браузеры посылали большой список **Accept** в каждом запросе. Даже если параметры, указанные в заголовке **Accept**, не могли быть использованы получателем, данные уже были переданы. Существует значительное расхождение между объемом передаваемой и реально используемой обеими сторонами информации. Сервер может быть гибким в выборе ответов. Подобным же образом агент пользователя может задавать произвольный набор приемлемых или предпочитаемых форматов. Таким образом, согласование содержания — это мощное средство, которым нужно пользоваться осторожно.

Согласованные ответы, хранящиеся в кэше, требуют более аккуратного отношения. Кэш может ответить на запрос вариантом ресурса с альтернативным местоположением. В HTTP/1.1 это осуществляется с помощью нового заголовка ответа **Vary** и с помощью заголовков содержимого **Content-Location**. Напоминаем о том, что заголовок **Location** используется для переадресации запроса туда, где может находиться ресурс. Заголовок **Content-Location** хранит альтернативное местоположение содержания в запросе; оно отличается от URI ресурса в запросе.

Сочетание согласования управляемого и агентом, и сервером называется *прозрачным согласованием содержания*. Сервер посылает список возможных вариантов, а агент пользователя выбирает наиболее подходящий. Вместо того чтобы заставлять агента пользователя принимать участие в выборе вариантов каждый раз, процесс согласования может извлечь выгоду из наличия в цепочке передачи кэшей. Оптимизация процесса согласования основывается на промежуточных кэшах, на кэшировании списка вариантов и самих вариантах. На самом деле согласование передается от сервера кэшу, который действует от имени сервера. Таким образом, управляемое агентом согласование, обычно выполняемое агентом пользователя, выполняется с кэшем, который значительно ближе к клиенту, чем исходный сервер. Однако политика согласования с кэшем имеет смысл только до тех пор, пока настройки сервера, связанные с процессом согласования, не изменяются. Это естественно порождает вопрос, как кэш мог бы отслеживать изменения на сервере. HTTP Working Group решила убрать прозрачное согласование содержания из спецификации HTTP/1.1, работа над этим продолжается независимо [HM98]. Прозрачное согласование содержания обсуждается более подробно в главе 15 (раздел 15.5.1).

7.10. Безопасность, аутентификация и целостность

Обеспечение безопасности необходимо для гарантии, что только пользователи, прошедшие аутентификацию, получат доступ к ресурсам. Сообщения, которыми обмениваются в Web, не должны быть доступны для перехвата сторонами, не участвующими в обмене. Сообщения могут быть перехвачены, и, следовательно, содержание сообщения должно быть недоступно для всех, кроме тех, кому оно предназначено. Рассмотрим письмо, посылаемое по почте от A к B:

- A и B должны быть уверены, что никто другой не прочтет содержание письма.
- B нужен способ убедиться, что данное письмо действительно пришло от A.

- Письмо должно быть получено В в целости и сохранности (то есть содержание, отправленное А, не должно быть изменено).

Все это относится и к HTTP-сообщениям.

Используется несколько схем аутентификации. Для доступа к некоторым ресурсам достаточно простых схем аутентификации, в которых клиент аутентифицируется с помощью пользовательского имени и пароля, передаваемых открытым текстом. Для других ресурсов могут требоваться более сложные процедуры аутентификации, в которых шифруется пароль.

Одним из распространенных методов аутентификации является схема *запрос-ответ*, в которой сервер посылает запрос и ждет специфического ответа клиента. Запрос может иметь силу только для конкретной *области* (*realm*) — подмножества ресурсов, для которого пользователь считается аутентифицированным. Эта область может относиться к компьютеру или набору каталогов. Каждый сервер определяет для ресурса *защищенное пространство*, комбинируя область ресурса и его URI. Каждое защищенное пространство имеет свою собственную базу данных авторизации. Преимущество защищенных пространств заключается в том, что к различным ресурсам могут применяться различные схемы аутентификации.

Мы начнем с обсуждения проблем модели обеспечения безопасности HTTP/1.0 и усилий, предпринятых для их преодоления в HTTP/1.1. Затем мы рассмотрим проблемы, связанные с обеспечением целостности передаваемого содержания. Решением, принятым в HTTP/1.1, было введение аутентификационной информации, добавляемой к сообщению или в виде заголовка, или в трейлере.

7.10.1. Обеспечение безопасности и аутентификация

Обеспечение безопасности становилось ключевой проблемой по мере того, как росла популярность Web и развивалась электронная коммерция. В HTTP/1.0 обеспечению безопасности уделялось минимальное внимание, была реализована схема запрос-ответ, которая позволяла клиенту получить ресурс после того, как он подтверждал серверу, что он располагает нужными полномочиями. Это механизм *обычной (basic) аутентификации*, описанный в главе 6 (раздел 6.4).

Схема обычной аутентификации в HTTP/1.0 была не слишком защищенной, так как расчет строился на том, что канал между клиентом и сервером был заслуживающим доверия. То, что это не так, было известно уже во время подготовки проекта спецификации HTTP/1.0. HTTP/1.0 не препятствовал использованию сквозного шифрования, но и не предлагал никаких других способов защиты. Имя пользователя и пароль, посылаемые клиентом запрашивающему серверу, передавались закодированными с помощью base64 [FB96a]. Кодирование base64 практически представляет собой открытый текст, имя пользователя и пароль могут стать объектом перехвата. Развитие электронной коммерции привело к передаче через Web номеров кредитных карт, которые и становятся целью перехвата.

Еще одной проблемой в HTTP/1.0 является отсутствие областей действия аутентификации. В отличие от обычной аутентификации, аутентификация с помощью дайджеста позволяет ограничить область действия аутентификации несколькими способами, включая метод запроса, URI и ограничение времени жизни контрольных сумм. Срок действия полномочий в HTTP/1.0 был больше, чем нужно. По сути дела, полномочия были вечными. Это означает, что любой, кто перехватит идентификационные данные, может использовать их безнаказанно в течение долгого времени. Сокращение срока действия полномочий является очевидным усовершенствованием механизма безопасности. Это было реализовано в HTTP/1.1 путем

ограничения ответа одним ресурсом и одним методом. Если кто-то перехватывал сообщения в сети, то он мог бы только повторить тот запрос, ответ на который он уже знал. От клиента требовалось вычислить контрольную сумму, включающую URI, метод запроса, имя пользователя, пароль и уникальное значение временной метки. *Временная метка* является кодом по модулю 16 или по модулю 64, которая гарантированно является уникальной для каждого момента, когда сервер посылает ответ **401 Unauthorized** (ответ **401 Unauthorized** посылается тогда, когда сервер считает, что полномочия клиента являются недостаточными). Значение временной метки создается на основе комбинации времени, генерируемого сервером, атрибута доступа к содержимому ресурса и личного ключа, известного только серверу. Временная метка для клиента непрозрачна.

Аутентификация в HTTP подробно обсуждается в документе RFC 2617, который является документом, сопутствующим спецификации HTTP/1.1 (RFC 2616).

7.10.2. Целостность

Целостность сообщения — это важная составная часть обеспечения безопасности: то, что было отправлено, должно быть доставлено получателю в целости и сохранности. Новый заголовок содержимого **Content-MD5** в HTTP/1.1 используется для проверки целостности тела содержимого. Дайджест MD5 является 128-разрядной контрольной суммой и представляется в HTTP в виде 32-х отображаемых символов ASCII. Биты преобразуются от старших к младшим, группами по четыре бита в шестнадцатеричные цифры от 0 до f (например, битовая строка 1010 представляется шестнадцатеричной цифрой a, а 0100 — шестнадцатеричной цифрой 4). Дайджест позволяет получателю убедиться, что полученное содержание является точно тем же, которое было отправлено. Заголовок **Content-MD5** могут использовать только исходные серверы, поскольку он предназначен для реализации сквозного контроля целостности. Однако все промежуточные серверы (например, прокси-серверы и шлюзы) могут проверять целостность полученной ими информации.

Распространенной в Internet угрозой является атака отказа от обслуживания (DoS — Denial of Service). В Web DoS-атака может принимать множество форм, включая передачу Web-серверу сообщений большого объема. Сообщения большого объема могут переполнить буферы и также привести к нарушению безопасности. DoS -атака — это классическая проблема, с которой сталкиваются серверы; если бы серверу была известна длина сообщения *a priori*, сервер мог бы принять решение о том, следует ли ему расходовать свои ресурсы на обработку запроса. Код ответа **411 Length Required** дает клиенту понять, что серверу необходимо знать длину тела содержимого сообщения до обработки запроса. Это стандартный способ гарантировать, что протокол не заставит сервер принимать сообщения произвольно большого объема. Однако если сообщение разбито на фрагменты (см. раздел 7.6), то можно не указывать длину содержимого в сообщении. Сервер, поддерживающий HTTP/1.1, обязан анализировать сообщения разбитые на фрагменты. Однако сервер в любой момент может принять решение, что сообщение слишком велико и отправить код ответа **411 Length Required** и затем закрыть соединение.

Новый код ответа **414 Request-URI Too Long** может быть возвращен, если сервер не готов обрабатывать слишком длинный URI. Поскольку в протоколе максимальная длина не определена, конкретные реализации пользуются своими собственными ограничениями, чтобы установить соответствующее значение. Хороший способ применения данного кода состояния, это когда сервер полагает, что производится атака типа переполнения буфера. Программы, которые имеют фиксиро-

ванные строковые буферы и не отслеживают переполнение, могут допустить перезапись других разделов программного кода, что может привести к нарушению безопасности.

Обеспечению безопасности в HTTP/1.1 уделено серьезное внимание: предотвращается передача паролей открытым текстом, ограничен срок действия полномочий и введены средства, помогающие сохранить целостность сообщений. Были созданы новые коды ответов, чтобы предоставить реализациям компонентов возможность узнавать о попытках нарушения безопасности.

7.11. Роль прокси-серверов в HTTP/1.1

Одним из решающих улучшений в HTTP/1.1 было осознание той важной роли, которую играют промежуточные компоненты между отправителем и получателем. Хотя туннели и шлюзы также являются промежуточными компонентами, наиболее важным промежуточным компонентом является прокси-сервер. Роль прокси-сервера стала центральной в архитектуре Web. Структура и детали организации прокси-серверов были изложены в главе 3. В частности, вопросы, относящиеся к прокси-серверу как к программе, обрабатывающей запросы и ответы HTTP, были рассмотрены в главе 3 (раздел 3.4.2). Чтобы сосредоточить внимание на той роли, которую играют прокси-серверы, мы разделили обсуждение внесенных изменений на четыре части, следующим образом.

- Типы прокси-серверов.
- Синтаксические требования, предъявляемые к прокси-серверам в HTTP/1.1.
- Семантические требования, предъявляемые к прокси-серверам в HTTP/1.1.
- Влияние других компонентов Web на прокси-серверы в HTTP/1.1.

7.11.1. Типы прокси-серверов

Возможно использование единственного прокси-сервера между клиентом и исходным сервером, также существуют более сложные решения в виде иерархической системы прокси-серверов, которые могут обслуживать целое государство. В том, что касается протокола HTTP/1.1, существует три разных схемы функционирования прокси-серверов.

- Прокси-сервер действует в качестве посредника с другими системами, поддерживая ряд протоколов, таких как WAIS, FTP или Gopher.
- Прокси-сервер работает по протоколу HTTP и является промежуточным компонентом между отправителем и получателем HTTP-сообщений.
- Прокси-сервер играет роль прокси-сервера HTTP, клиента или туннеля в различные моменты времени в зависимости от запроса.

Для прокси-серверов, действующих в качестве посредников с другими системами, в HTTP/1.1 не было введено никаких новых требований. Они продолжают функционировать так же, как они это делали в соответствии с HTTP/1.0.

Во втором случае прокси-сервер может ответить на запрос, если он может сделать это *удовлетворительно,* то есть его ответ гарантированно актуален. Гарантия актуальности может быть ограничена клиентом или сервером. Если так, прокси-сервер может передать запрос дальше по цепочке. Когда спустя какое-то время вернется ответ, прокси-сервер может, но не обязательно, сохранить этот ответ

в кэше, прежде чем пересылать его запрашивающей стороне. Прокси-сервер может изменить запрос или ответ.

В третьем случае прокси-сервер может также функционировать как туннель, когда используется метод **CONNECT**. Хотя в спецификации HTTP/1.1 этот метод просто зарезервирован, он использовался при инициировании Transport Layer Security (TLS) [KL00] поверх существующего TCP-соединения посредством механизма **Upgrade**.

7.11.2. Синтаксические требования к прокси-серверу, поддерживающему HTTP/1.1

В HTTP/1.1 к прокси-серверу предъявляются следующие два класса основных синтаксических требований:

- Требование, связанные с передачей сообщений.
- Требования, связанные с изменением существующих заголовков или добавлением новых.

Далее мы рассмотрим оба эти требования.

ПЕРЕДАЧА СООБЩЕНИЙ

При передаче сообщения прокси-сервер должен учитывать версию протокола отправителя, от которого он получил сообщение, и получателя, которому он передает это сообщение. Предположим, что прокси-сервер, поддерживающий HTTP/1.1, пересылает сообщение-запрос, которое он получил от отправителя, поддерживающего HTTP/1.0. Даже если этот прокси-сервер получил ответ HTTP/1.1 от ближайшего к нему сервера, он не может вернуть этот ответ отправителю. Исходный отправитель, поддерживающий HTTP/1.0, может не суметь правильно воспринять некоторые из возможных аспектов этого сообщения HTTP/1.1. Однако если исходный отправитель правильно воспринимает HTTP/1.1, прокси-сервер, поддерживающий HTTP/1.1, может возвратить ответ HTTP/1.1 нетронутым. На самом деле в спецификации предполагается, что прокси-сервер сохраняет в кэше номер версии своего ближайшего сервера-соседа так, чтобы знать какого типа сообщение ему передавать.

Имеются специальные правила, связанные с передачей ответов класса 1xx. Например, прокси-сервер не может передать ответ **100 Continue**, если прокси-сервером добавлен заголовок **Expect** (т.е. заголовок **Expect** отсутствовал в исходном сообщении, переданном прокси-серверу). Если прокси-сервер получает запрос, который не соответствует его возможностям, он должен вернуть ответ **417 Expectation Failed** вне зависимости от того, обладает ли следующий за ним сервер требуемыми возможностями. Связано это с тем, что механизм **Expect** является механизмом промежуточных передач.

Для методов **TRACE** и **OPTIONS** прокси-серверы должны проверять наличие заголовка **Max-Forwards** (см. раздел 7.7.1) и то, что его значение не равно нулю. Необходимо также уменьшить значение заголовка на единицу, прежде чем передать сообщение дальше. Аналогично, запрос без заголовка **Host** от клиента, поддерживающего HTTP/1.1, не должен передаваться прокси-сервером дальше; вместо этого прокси-сервер должен вернуть **400 Bad Request**.

Заголовки, не воспринимаемые прокси-сервером, должны по-прежнему пересылаться дальше, если только они не перечислены в заголовке **Connection** (описанном в разделе 7.5). Прокси-серверы должны осторожно обрабатывать заголовки промежуточных передач. Ошибочно переданные дальше заголовки, предназначен-

ные для данного соединения, могут создать проблемы. Прокси-сервер должен удалять любые заголовки, совпадающие с лексемами, перечисленными в заголовке **Connection**, прежде чем пересылать сообщение дальше.

ДОБАВЛЕНИЕ ЗАГОЛОВКОВ ИЛИ ИЗМЕНЕНИЕ СУЩЕСТВУЮЩИХ ЗАГОЛОВКОВ

Прокси-серверы, поддерживающие HTTP/1.1, обязаны добавлять информацию о себе в заголовке **Via**, как это обсуждалось в разделе 7.7. Прокси-серверы должны также указывать номер версии протокола вышестоящего сервера в цепочке, от которого получено данное сообщение. Однако прокси-сервер, являющийся внешним шлюзом сети (например, функционирующий как сетевой экран), должен скрывать информацию о компьютерах, находящихся во внутренней сети и не добавлять информацию о них в заголовки **Via**.

Прокси-серверы, поддерживающие HTTP/1.1, не могут изменять порядок значений полей в заголовках сообщений, так как при этом может измениться их семантика. Две кодировки содержания в строке могут дать абсолютно разные результаты, если их применить в другом порядке. Некоторые сквозные заголовки (например, **ETag, Content-MD5**) и поля в заголовках **Expires** не должны изменяться прозрачными прокси-серверами. Кроме того, наличие директивы управления кэшем **no-transform** не допускает изменения некоторых заголовков (например, **Content-Encoding, Content-Type**). Прокси-сервер не должен изменять заголовок запроса **From** (указывающий адрес электронной почты пользователя, пославшего этот запрос) и заголовок ответа **Server** (идентифицирующий программное обеспечение и номер версии Web-сервера).

Прокси-серверам не разрешается изменять полностью определенное имя домена, представленное в URI сообщения. Однако если прокси-сервер получает имя хоста, которое не является полностью определенным доменным именем, то ему разрешается добавить имя домена к имени хоста. Прокси-серверам не разрешается *генерировать* некоторые заголовки (например, сквозной заголовок проверки целостности **Content-MD5**; см. раздел 7.2.2). Прокси-сервер может, тем не менее, использовать значение, представленное в заголовке **Content-MD5**, чтобы проверить корректность тела полученного им сообщения.

7.11.3. Семантические требования к прокси-серверу, поддерживающему HTTP/1.1

Семантические изменения в HTTP/1.1, касающиеся прокси-серверов, являются значительными в четырех областях: кэширование, запросы на диапазоны, долговременные соединения и обеспечение безопасности.

ТРЕБОВАНИЯ К ПРОКСИ-СЕРВЕРУ, СВЯЗАННЫЕ С КЭШИРОВАНИЕМ

Некоторые изменения в HTTP/1.1 относятся к кэшированию. Поскольку прокси-серверы играют главную роль в кэшировании, то изменения в этой области оказывают на них сильное влияние. Например, прокси-серверы должны уделять внимание атрибутам содержимого, которые посылаются в качестве индикаторов кэша и обеспечивают соблюдение семантической прозрачности при возвращении значений из кэша. Прокси-сервер обязан принимать во внимание различные поля в условном заголовке до принятия решения об актуальности кэшированных объектов. При наличии директивы управления кэшем **Cache-Control: s-maxage=0** кэш совместно используемого прокси-сервера должен проверить кэшированный ответ.

Если прокси-сервер действует в качестве кэша и возвращает кэшированные ответы, он должен посылать заголовок **Age** со своими ответами, чтобы указать, что данный ответ не является сгенерированным исходным сервером, т.е. «не из первых рук». Заголовок **Age** включает оценку времени, прошедшего с момента первоначальной генерации данного ответа или с момента подтверждения его актуальности сервером-источником. Получатель может использовать значение **Age** для оценки возраста ответа.

ТРЕБОВАНИЯ К ПРОКСИ-СЕРВЕРУ, СВЯЗАННЫЕ С УПРАВЛЕНИЕМ СОЕДИНЕНИЕМ

Прокси-серверы, поддерживающие HTTP/1.1 и реализующие долговременные соединения, должны обеспечивать долговременность и восходящего (по направлению к исходному серверу), и нисходящего (по направлению к клиенту) соединений. В HTTP/1.0 существовала реализация долговременного соединения (заголовок **Keep-Alive**), которая требовала явного согласования. Однако взаимодействие заголовков **Connection** и **Keep-Alive** может привести к «зависанию» соединения. Это происходит из-за того, что клиент HTTP/1.0 может послать заголовок **Keep-Alive** прокси-серверу, который не воспринимает **Connection**, но может ошибочно передать его дальше. Если нисходящее соединение также поддерживает соединение **Keep-Alive**, находящийся в середине прокси-сервер никогда не получил бы завершающей части этого ответа. Чтобы избежать таких проблем, прокси-серверам, поддерживающим HTTP/1.1, не разрешено устанавливать долговременное соединение с клиентами, поддерживающими HTTP/1.0.

Прокси-серверы могут использовать различные наборы правил (по сравнению с клиентом или исходным сервером) при принятии решения о закрытии долговременного соединения. Исходный сервер может иметь уважительные причины закрывать долговременные соединения раньше, так как он обслуживает множество клиентов. У прокси-сервера обычно меньше клиентов, чем у популярного Web-сервера. И поэтому прокси-серверу «виднее», какие соединения поддерживать открытыми, так как он успевает следить за комбинациями запросов клиентов. В протоколе не указывается как долго (в единицах времени) прокси-серверу следует поддерживать долговременное соединение, но все же устанавливаются общие правила для *количества* долговременных соединений, которые ему следует поддерживать параллельно с любым сервером, расположенным выше в цепочке. В протоколе говорится, что однопользовательский клиент может поддерживать не более двух долговременных соединений с любым сервером или прокси-сервером. Прокси-сервер должен ограничиться 2n соединениями с другими серверами или прокси-серверами, где n — количество одновременно работающих пользователей.

ТРЕБОВАНИЯ К ПРОКСИ-СЕРВЕРУ, СВЯЗАННЫЕ С УПРАВЛЕНИЕМ ПОЛОСОЙ ПРОПУСКАНИЯ

HTTP/1.1 клиенты могут выполнять запросы на диапазоны. Это накладывает на прокси-сервер специфические ограничения. Если прокси-сервер пересылает запрос на диапазон серверу и получает в качестве ответа весь ресурс, он должен вернуть клиенту только указанную им часть ресурса. Тем не менее, сохранить прокси-сервер должен весь ответ. Однако если кэш прокси-сервера не поддерживает запросы на диапазоны, то он не должен кэшировать частичные ответы.

Для прокси-серверов, поддерживающих HTTP/1.1, существуют дополнительные правила работы с механизмом *Expect*. Роль, которую играет прокси-сервер, усложнилась, так как теперь, имея дело с частичным запросом клиента, он должен учитывать возможности нижерасположенного сервера. В соответствии с обсуждением правил передачи заголовков, которое проводилось ранее в этом разделе, про-

кси-серверы, поддерживающие HTTP/1.1, обязаны пересылать заголовок **Expect** вместе с запросом. Однако если прокси-серверу известно, что у ближайшего сервера версия протокола меньше, чем HTTP/1.1, то этот прокси-сервер должен вернуть ответ **417 Expectation Failed**, а не пересылать запрос дальше. Аналогично, если прокси-сервер получил от сервера ответ **100 Continue**, он не должен пересылать этот ответ клиенту за исключением того случая, когда клиентский запрос включает заголовок **Expect**.

ТРЕБОВАНИЯ К ПРОКСИ-СЕРВЕРУ, СВЯЗАННЫЕ С ОБЕСПЕЧЕНИЕМ БЕЗОПАСНОСТИ

Некоторые заголовки и коды ответов были введены для удовлетворения требований к прокси-серверу, связанных с обеспечением безопасности. К ним относятся заголовки запросов, такие как **Proxy-Authenticate**, и коды ответов, такие как **407 Proxy Authorization Required**.

Заголовок запроса **Proxy-Authenticate** требуется тогда, когда ресурс доступен только для клиентов, прошедших аутентификацию. Клиент должен предоставить необходимые полномочия. Заголовок определяет используемую схему идентификации и включает вызов, чтобы дать возможность клиенту вернуть свои идентификационные данные.

Код ответа **407 Proxy Authorization Required** похож на код ответа HTTP/1.0 **401 Unauthorized**. Однако **407 Proxy Authorization Required** служит для аутентификации ближайшего участника соединения в отличие от сквозного заголовка **401 Unauthorized**. Код ответа **407 Proxy Authorization Required** используется прокси-сервером для того, чтобы проинформировать клиента, что требуется санкция на доступ к прокси-серверу, тогда как ответ **401 Unauthorized** используется исходным сервером для отправки вызова клиенту непосредственно. Прокси-сервер посылает заголовок аутентификации **Proxy-Authenticate** клиенту с вызовом, указав подходящую схему аутентификации и параметры запрашиваемого URI. Клиент должен прислать прокси-серверу свои полномочия для заданного URI с помощью нового заголовка запроса **Proxy-Authorization**.

7.12. Различные изменения

В HTTP/1.1 по сравнению с HTTP/1.0 есть и другие незначительные изменения. Мы рассмотрим различные изменения, не вошедших в рассмотренный набор категорий, разделив эти изменения на три части: относящиеся к методам, к заголовкам и к кодам ответов. Эти изменения обсуждаются отдельно, чтобы не отвлекать внимание от более важных изменений рассмотренных в этой главе раньше.

7.12.1. Различные изменения, относящиеся к методам

Семантика отдельных методов HTTP/1.0 была изменена незначительно. В некоторых случаях была уточнена интерпретация этих методов. Изменения коснулись методов **HEAD, PUT, DELETE** и **CONNECT**.

НЕЗНАЧИТЕЛЬНОЕ ИЗМЕНЕНИЯ МЕТОДА HEAD

В HTTP/1.0 нельзя было модифицировать запрос **HEAD** условными заголовками (такими как **If-Modified-Since**), в HTTP/1.1 нет таких ограничений. Условные заголовки в HTTP/1.1 можно использовать с любым методом. Спецификация HTTP/1.1

придерживается принципа вводить ограничения только тогда, когда следует, и приводить конкретные обоснования для любых введенных ограничений.

УТОЧНЕНИЯ МЕТОДА PUT

Метод **PUT** был частично описан в RFC 1945, как часть HTTP/1.0. Последующие дискуссии в HTTP Working Group привели к общему решению о его формальном определении в качестве метода HTTP/1.1. Web-сервер, обрабатывающий запрос **PUT**, проверяет URI запроса, чтобы определить будет ли запрос модифицировать существующий ресурс или создавать новый. Web-сервер должен проверить права доступа пользователя, прежде чем применять данный метод к ресурсу. В формальном определении метода были добавлены отдельные требования к Web-серверам и кэширующим прокси-серверам. Например, Web-сервер должен отвечать **201 Created**, если ресурс был создан в результате запроса с методом **PUT**. Если ресурс, указанный в методе **PUT** (также как **POST** и **DELETE**) был помещен в кэш, кэшированный ответ должен быть помечен как устаревший. Кэширующий прокси-сервер, который заметит запрос с методом, допускающим изменение ресурса на сервере-источнике, должен отметить как неактуальные любые элементы кэша, связанные с этим ресурсом.

В следующем примере запрос с методом **PUT** используется для создания или модификации ресурса **/foobar/foo.html:**

```
PUT /foobar/foo.html HTTP/1.1
User-Agent: Mozilla/2.0
Authorization: Basic ZHeadFuaYow29PinWU=
Content-Type: text/html
Content-Length: 433
Host: bar.com
...
<433 байта возможно нового foo.html>
```

Этот запрос включает информацию о типе содержания и его длине, также как идентификационные данные в заголовке **Authorization**. Идентификационные данные нужны Web-серверу, чтобы убедиться, что данный пользователь имеет необходимые права доступа для модификации или создания ресурса. Если полномочия достаточны, сервер создает или модифицирует **/foobar/foo.html**, а также возвращает ответ **200 OK**.

Сравнение методов PUT и POST. Различия. В процессе формализации определения метода **PUT** возникали вопросы о различии методов **PUT** и **POST**. Чтобы понять разницу между методами **PUT** и **POST**, нужно рассмотреть, как осуществляется интерпретация URI в запросе. **PUT** обязывает сервер применить запрос *только* к данному URI и, если он не может этого сделать, вернуть ответ, относящийся к классу переадресации. Ресурс может быть создан или изменен. Ресурс в методе **POST** ссылается на программу, которая должна будет обработать данные, включенные в содержимое сообщения **POST**. Это тот случай, когда пользователь заполняет форму как составную часть запроса.

МЕТОД DELETE. ФОРМАЛИЗАЦИЯ

Как и **PUT**, метод **DELETE** рассматривался в приложении к RFC 1945, но не был формально определен. Метод **DELETE** приводит к удалению ресурса, ассоциированного с запрашиваемым URI, если запрос не аннулируется по каким-то другим

причинам на Web-сервере. Web-сервер, например, может проверить права доступа инициатора запроса и принять решение не удалять ресурс. В этом случае, сервер должен будет сообщить запрашивающему клиенту, что запрос не принят, возвратив ответ **401 Unauthorized** или **403 Forbidden**.

В HTTP/1.1 требования к компонентам, связанные с реакцией на метод **DELETE**, определены ясно. Например, в протоколе определены коды ответов, которые должен послать Web-сервер, если удаление выполнено успешно. Прокси-сервер, получивший запрос **DELETE** для ресурса помечает его кэшированную копию как устаревшую.

МЕТОД CONNECT

Метод **CONNECT** используется для создания «туннеля» между двумя прокси-серверами и был предложен Ари Луотоненом [Luo97] в 1998 г. в рабочем проекте стандарта (теперь уже устаревшем). Он не был формально стандартизован. Новый метод — **CONNECT** — был зарезервирован в HTTP/1.1 в роли «заполнителя» (то есть для гарантии, что этот метод не будет зарезервирован в будущем для какого-то другого применения). Хотя имя этого метода зарезервировано для будущего применения, одно специфическое применение в RFC упомянуто — прокси-сервер, в целях обеспечения безопасности, способен превратиться в «туннель». Иначе говоря, прокси-сервер начинает механически ретранслировать сообщение, которое он получает. Метод **CONNECT** в HTTP/1.1 [KL00] используется для инициирования Transport Layer Security (TLS) поверх существующего TCP-соединения, для прокладки «туннелей» через прокси-серверы.

7.12.2. Различные изменения, относящиеся к заголовкам

Некоторые заголовки HTTP/1.0 были немного изменены, некоторые новые заголовки были добавлены по узкоспециальным соображениям. Мы рассмотрим эти изменения в трех частях: общие заголовки, заголовки ответов и заголовки содержимого.

ОБЩИЕ ЗАГОЛОВКИ

Если тело сообщения было как-то изменено, то способ изменения указывается в заголовке **Transfer-Encoding**, чтобы получатель знал, как анализировать тело сообщения. В HTTP/1.0 не было гарантии (при отсутствии заголовка **Content-Length**), что получатель действительно получил то, что было передано отправителем. Кодирование при передаче позволило гарантировать, что получатель получил все, что было отправлено. Заголовок **Transfer-Encoding** — заголовок промежуточных передач и, таким образом, имеет смысл только для одного текущего соединения.

В результате введения заголовка **Transfer-Encoding** в HTTP/1.1, по сравнению с HTTP/1.0, изменилось определение HTTP-сообщения (и запроса, и ответа). В HTTP/1.0, чтобы получить тело содержимого, достаточно было удалить только заголовки сообщения, а в HTTP/1.1 исходное сообщение нужно еще и восстановить, применив к нему преобразование, обратное кодированию при передаче.

ЗАГОЛОВКИ ОТВЕТОВ

Один заголовок HTTP/1.0 (**Retry-After**) был изменен в HTTP/1.1 и был добавлен новый заголовок **Warning**.

Заголовок **Retry-After** рассматривался в приложении к спецификации HTTP/1.0 в контексте несовместимых реализаций протокола. Тем не менее, этот заголовок не

стал частью спецификации HTTP/1.0. Он был введен в HTTP/1.1 для уточнения семантики кода ответа **503 Service Unavailable**. Существуют две причины обусловливающие этот код ответа: сервер в течение неопределенного времени работает в автономном режиме, — временно занят и возможно скоро освободится. Если сервер может сообщить клиенту, что данный сервис, возможно, освободится позже, то клиент может повторить запрос после указанного интервала времени. Значение времени указывается или относительно текущего момента или задается конкретное значение в будущем. В HTTP/1.1 заголовок **Retry-After** используется с несколькими ответами, включая **413 Request Entity Too Large** (раздел 7.2.3) и ответами переадресации класса (3xx).

Первые сообщения об ошибках в HTTP/1.0 не были предназначены для чтения их человеком. Классы ошибок 4xx и 5xx включают короткие и простые фразы на естественном языке, но из этих фраз нельзя извлечь информацию о возможных корректных действиях. К сожалению, код состояния и строка сообщения появляются в первой строке ответа и поэтому несколько кодов состояния туда нельзя поместить. Многочисленные коды состояния могли бы запутать получателей. Любая дополнительная информация должна включаться в отдельные заголовки. Эта ситуация была исправлена в HTTP/1.1 путем введения заголовка **Warning**. Теперь вместе с кодом класса предупреждающего сообщения можно посылать информацию, поясняющую предупреждающее сообщение. Обратите внимание, что эти коды предупреждений не относятся к общим кодам ответов HTTP. Как показано в таблице 7.15, в HTTP/1.1 определено семь кодов предупреждений.

Таблица 7.15. Коды предупреждений HTTP/1.1

Код предупреждения (warning)	Краткое описание
110 Response is stale	Кэш предупреждает, что возвращенный ответ устарел
111 Revalidation failed	Неудачная попытка проверить актуальность ответа
112 Disconnected operation	Кэш отключен от сети
113 Heuristic expiration	Срок годности определен на основе эвристики
199 Miscellaneous warning	Предупреждение для вывода пользователю
214 Transformation applied	Кодирование или тип содержания изменены
299 Miscellaneous persistent warning	Предупреждение класса 2xx для вывода пользователю

Классы кодов предупреждений допускают расширение, в будущем возможно появление дополнительных кодов.

Например, предположим, что ресурс был получен с содержанием, закодированным каким-то одним способом (например, **gzip**), но промежуточный прокси-сервер преобразовал ответ в другой формат. Было бы полезно указать на этот факт в предупреждающем сообщении, а не просто пересылать ответ дальше в его трансформированном виде. Прокси-сервер может включить заголовок **Warning** следующим образом:

```
Warning: 214 Transformation applied
```

Аналогично, если прокси-сервер собирается вернуть ответ из кэша из-за невозможности установить связь с исходным сервером, прокси-сервер может указать на то, что актуальность данного ответа не подтверждена, следующим образом:

```
Warning: 111 Revalidation failed
```

ЗАГОЛОВКИ СОДЕРЖИМОГО

В HTTP/1.0 было описано шесть заголовков содержимого, еще один, **Content-Language**, был рассмотрен в приложении к RFC 1945. Некоторые из заголовков содержимого HTTP/1.0 были изменены в HTTP/1.1, чтобы извлечь преимущества из его новых возможностей. Например, в HTTP/1.1 уточнено, что к ресурсу может быть применено несколько кодировок содержания в заголовке содержимого **Content-Encoding**. Web-серверы обязаны проверять, что значение **Last-Modified** совпадает со временем, сгенерированным ими для значения поля **Date** в ответе. Если уже после того как ответ был сгенерирован, содержимое было изменено, то получатель узнает об этом по времени модификации содержимого.

7.12.3. Различные изменения, относящиеся к кодам ответов

При переходе от HTTP/1.0 к HTTP/1.1 почти во всех классах ответов было по несколько кодов, семантика которых была изменена. Мы рассмотрим эти изменения кодов в соответствии с теми классами, к которым они относятся.

КОДЫ ОТВЕТОВ 2XX

Из четырех прежних кодов ответов класса 2xx только два были незначительно изменены. Код ответа **200 OK** дополнен небольшим уточнением, что ответы на запросы **HEAD** не содержат *тела сообщения*, в отличие от *тела содержимого* в HTTP/1.0. Иначе говоря, заголовки содержимого могут быть включены.

В коде ответа HTTP/1.0 **201 Created** было сделано несколько небольших дополнений. Заголовок **Location** можно включать в ответ, указывая конкретный URI для созданного ресурса. Ресурс может иметь несколько адресов, и заголовок **Location** можно использовать, чтобы указать конкретный адрес. В заголовок ответа можно также включать другие свойства, характерные для данного ресурса, такие как тип содержания.

Из трех новых в HTTP/1.1 кодов ответов класса 2xx, два объясняются здесь. Третий (**206 Partial Content**) был рассмотрен в разделе 7.4.1. Здесь мы добавляем пояснение относительно кэширования ответов **206 Partial Content**.

203 Non-Authoritative Information. Этот новый код ответа используется для указания, что метаданные о ресурсе были получены не от самого Web-сервера, а из других источников. Метаданные вероятно не идентичны тем, которые прислал бы исходный сервер. В тех случаях, когда наличие дополнительных метаданных о ресурсе возможно, протокол разрешает включить эту информацию, но с другим кодом ответа.

205 Reset Content. Новый код ответа в HTTP/1.1 **205 Reset Content** используется исходным сервером, чтобы помочь агенту пользователя организовать обратную связь с пользователем. В HTTP/1.0 не было возможности это сделать. Предположим, что пользователь передал данные из формы на сервер. Сервер, возвратив код **205**, может вызвать очистку формы агентом пользователя, показывая пользователю, что данные формы были приняты сервером. Пользователь может продолжить ввод данных в форму для передачи дополнительных запросов. Это пример изменения протокола, которое помогает агенту пользователя использовать ответ сервера, для информирования пользователя об изменении состояния.

206 Partial Content. Кэши, которые не поддерживают заголовки **Range**, не должны кэшировать ответ **206 Partial Content**. Такое предупреждение — пример ограничений накладываемых на реализации, которые не поддерживают новое свойство, но легко могут получать ответы с кодами состояний, связанными с новым свойством. Спецификация протокола включает много таких ограничений, но реализации не всегда проходят проверку на соответствие этим ограничениям, что приводит к потенциальным проблемам. Так изменение протокола оказывает воздействие на реализации серверов и клиентов, которые этим изменением не затронуты.

КОДЫ ОТВЕТОВ 3XX

Четырьмя новыми кодами ответов в HTTP/1.1 являются **303 See Other, 305 Use Proxy, 306 (Unused)** и **307 Temporary Redirect**. Коды ответов **303** и **307** были введены для того, чтобы Web-компоненты могли определять нужное действие более точно. Код ответа **305** был введен в промежуточной (не принятой в качестве стандарта) спецификации RFC 2068. Код ответа **306** был предложен в промежуточной версии проекта спецификации протокола, но был удален в процессе стандартизации протокола.

303 See Other. Ответ HTTP/1.0 **302 Moved Temporarily** включал заголовок **Location**, указывая агенту пользователя, где действительно может находиться данный ресурс. Ответ **303 See Other** похож на код состояния **302 Moved Temporarily**, но был добавлен специально, чтобы исправить известную проблему с агентами пользователя, которые некорректно обрабатывали ответ **302**. Это один из примеров эволюции протокола после обнаружения ошибок в существующих реализациях.

307 Temporary Redirect Код ответа **307 Temporary Redirect** информирует агента пользователя, что данный ресурс может *временно* находиться по новому URI, внесенному в заголовок ответа Location. При наличии ответа переадресации некоторые агенты пользователя ошибочно используют метод запроса GET, чтобы извлечь ресурс по новому URI, независимо от того, какой метод использовался в исходном запросе. И это несмотря на тот факт, что и в RFC 1945, и в одной из промежуточных спецификаций HTTP/1.1 (RFC 2068) говорится ясно, что при переадресации методы запросов не должны изменяться. Переадресация используется в условиях наличия «зеркал» Web-серверов, из-за чего запрос клиента может быть переадресован ближайшей к клиенту «зеркалу». Это снижает нагрузку на отдельный сервер.

305 Use Proxy. Код ответа **305 Use Proxy** сообщает запрашивающему компоненту, что запрос надо повторить через указанный прокси-сервер, заданный в заголовке ответа **Location**. Этот код был первоначально введен, чтобы дать возможность Web-серверу запрещать доступ для всех запросов кроме тех, которые поступают от соответствующего прокси-сервера. Смысл такого решения был в том, чтобы снизить нагрузку на Web-сервер и расположить его так, чтобы он мог указывать каким прокси-сервером должен пользоваться конкретный клиент. Кроме того, промежуточная спецификация RFC 2068 была недостаточно четкой в том, что касается использования кода ответа **305**, и создала возможность появления дыры в системе безопасности. Только один запрос может быть переадресован посредством ответа **305** и такая переадресация может быть сделана только самим Web-сервером. Если это не так, то существует вероятность наличия атаки типа промежуточного компонента (man-in-the-middle). В этом случае некий промежуточный компонент переадресует запросы на «вражеский» прокси-сервер.

306 (Unused). Ответ **306** был введен в одной из первых версий проекта спецификации HTTP и позже исключен. Версия прокси-сервера компании Netscape расширяла требования для кода ответа **306 Switch Proxy** таким образом, что получатель мог использовать прокси-сервер, указанный в новом заголовке промежуточных передач **Set-proxy**, для последующих запросов. Дополнение должно было, главным образом, добавить некоторую недостающую функциональность к семантике ответа **305 Use Proxy** и ограничить область применения этого ответа прокси-серверами, а не Web-серверами. Однако беспокойство о последствиях этого ответа для безопасности (особенно в связи с атакой типа промежуточного компонента) привело к исключению в HTTP/1.1 кода состояния **306**. Однако так как простое исключение может означать, что прежние реализации станут не работоспособными, этот код был зарезервирован.

КОДЫ ОТВЕТОВ 4XX

Несколько новых кодов ответов класса 4xx были реализованы в HTTP/1.1. Коды ответов из таблицы 7.10, не обсуждавшиеся до этого, могут быть разбиты на три категории: уточнение, использование возможности согласования в HTTP/1.1 и новые коды для других свойств.

1. Уточняющие коды состояний. В этой категории мы обсудим три новых кода состояния.

   405 Method Not Allowed. Код ответа **405 Method Not Allowed** был введен для того, чтобы иметь возможность дать несколько более подробный ответ при отклонении запроса на конкретный ресурс. Вместо того чтобы возвращать более общий код ошибки **403 Forbidden** (как это имеет место в HTTP/1.0), сервер перечисляет набор методов, которые разрешены для запрашиваемого URI. Это делается с помощью заголовка содержимого **Allow**. Заметьте, что сервер обязан это делать, если посылается ответ **405 Method Not Allowed**. Отправитель запроса может затем изменить метод запроса. Предположим, что клиент послал запрос **TRACE** серверу, у которого этот метод не реализован. Этот сервер вернет

   ```
   HTTP/1.1 405 Method Not Allowed
   Allow: GET, HEAD, POST, PUT
   ```

   Таким образом клиент узнает набор адекватных методов, которые данный сервер может обрабатывать.

   408 Request Timeout. Другой пример уточнения отразился в коде состояния **408 Request Timeout**. Сервер может оказаться не готов все время ждать, когда клиент пришлет запрос, и может отключиться. Вместо того, чтобы закрывать соединение с неоднозначным кодом ошибки, сервер посылает более детализированный код, указывающий, что он находится в состоянии таймаута.

   410 Gone. Если ресурс отсутствует, HTTP/1.1 сервер вернет ответ **404 Not Found** также как и серверы, поддерживающие HTTP/1.0. Однако если серверу известно, что этот ресурс был удален или если его новый адрес нельзя определить, чтобы указать его, можно использовать новый код ответа **410 Gone**. Информация о том, что ресурс вряд ли появится позже, известна благодаря внутренней конфигурационной информации доступной серверу. В этом случае клиент, в отличие от получения **404 Not Found**, знает, что не следует по-

вторять запрос, который вызвал ответ **410 Gone**. При использовании метода **DELETE**, некоторые ресурсы могут быть удалены навсегда. Различие между **404 Not Found** и **410 Gone** похоже на различие между **301 Moved Permanently** и **302 Found**.

2. Коды состояния согласования. Мы обсудим здесь два новых кода состояния относящихся к категории согласования.

   413 Request Entity Too Large. Код ответа **413 Request Entity Too Large** посылается, если сервер не может обработать сообщение-запрос слишком большого размера. Сервер может разрешить отправителю возобновить тот же запрос позже, если существует вероятность, что у него появится возможность обработать содержимое большого размера. Заголовок **Retry-After** используется для указания времени, после которого клиент может повторить запрос. Этот код ответа особенно полезен в сочетании с механизмом **Expect** — клиент может послать заголовок **Expect** с размером содержимого запроса и если он получит ответ **100 Continue**, то может посылать остальную часть запроса.

   415 Unsupported Media Type. Код ответа **415 Unsupported Media Type** используется тогда, когда содержимое запроса отправителя представлено в формате, который не поддерживается сервером для указанного запроса. Изменение метода запроса может изменить реакцию сервера. Предположим, запрос с методом **PUT** включает содержимое в формате, который не поддерживается сервером. Тот же формат может использоваться в запросе **POST**, возможно сервер сможет принять содержимое и передать его программе, которая это содержимое сможет обработать.

3. Новые коды для других возможностей в HTTP/1.1. Мы обсудим два новых кода ответа в этой категории.

   402 Payment Required. Код состояния **402 Payment Required** был добавлен для электронной коммерции в Web. Тем не менее, за этим кодом не зарезервировано никакой семантики. Это просто «заполнитель» для будущего применения.

   409 Conflict. Код состояния **409 Conflict** был введен в одном из первых проектов HTTP/1.0 для поддержки управления версиями, но формально до HTTP/1.1 в спецификацию не входил. Если два пользователя пытаются изменить ресурс с помощью метода **PUT**, Web-сервер может обнаружить эту ситуацию и послать ответ **409 Conflict**. Тело ответа включает указание на то, что может быть не так, если этот запрос будет выполнен. Ответ **409 Conflict** посылается, если сервер имеет основания рассчитывать на то, что клиент может предпринять корректирующие действия и повторить запрос. Ответ может включать информацию, которая может быть использована агентом пользователя, чтобы принять соответствующие меры.

КОДЫ ОТВЕТОВ 5XX

Из двух новых кодов ответов класса 5xx ответ **504 Gateway Timeout** уже рассматривался в контексте директив управления кэшами. Еще один новый код ответа этого класса — это **505 HTTP Version Not Supported**. Это специфический код ошиб-

ки, указывающий, что номер версии протокола в сообщении-запросе не поддерживается сервером. Такое отсутствие поддержки может быть обусловлено чем-то очень простым, например, сервер может не воспринимать сообщения в формате того номера версии протокола, который использовал отправитель. Или наоборот, возможно сервер не воспринимает семантику и не отвечает требованиям, которые подразумевались отправителем. Данный ответ может указывать на *другой* протокол, поддерживаемый сервером, чтобы отправитель мог отправить соответствующий запрос.

7.13. Резюме

Спустя почти десять лет с момента своего появления протокол HTTP сформировался в наиболее популярный современный протокол прикладного уровня. Потоки данных, передаваемых в доступной измерениям части Internet, показывают, что три из четырех пакетов передаются по протоколу HTTP версий HTTP/1.0 или HTTP/1.1. По любой мерке это ошеломляющая статистика. Хотя с протоколом HTTP/1.0 были связаны некоторые проблемы, значительная часть трафика Internet продолжает передаваться по этому протоколу. Многие браузеры и серверы поддерживают HTTP/1.1, но большинство промежуточных компонентов продолжают поддерживать протокол HTTP/1.0. Это означает, что лишь небольшая часть сквозного трафика полностью осуществляется по протоколу HTTP/1.1. Один из доводов в пользу усовершенствования механизма промежуточных передач в HTTP/1.1 заключается в возможности, что усовершенствования протокола смогут использоваться между любыми двумя компонентами, реализующими HTTP/1.1.

В этой главе приведены логические обоснования перехода к HTTP/1.1 и, где это уместно, прослежена эволюция идей, лежащих в основе этих изменений. Мы надеялись познакомить читателей с процессом разработки протоколов, который часто осуществлялся в непростых условиях. Сравнительное изложение элементов HTTP/1.0 и HTTP/1.1. должно помочь читателю оценить объем работы, проделанной в процессе эволюции протокола. Систематизация основных изменений должна помочь сосредоточиться на важных аспектах протокола с практической точки зрения. Читатели должны помнить, что хотя мы честно пытались охватить большое число проблем, в области корректности изложения последнее слово остается за стандартами IETF. В следующей главе рассматривается взаимодействие протокола транспортного уровня с HTTP.

8

Взаимодействие HTTP и TCP

Хотя протокол HTTP (Hypertext Transfer Protocol) не зависит от протоколов транспортного уровня, почти все его реализации используют TCP (Transmission Control Protocol). TCP был стандартизован в 1980 году, за десять лет до возникновения Web. Ранние протоколы прикладного уровня, использовавшие TCP, заметно отличались от HTTP. Например, Telnet представляет собой интерактивное приложение, использующее одиночное соединение TCP для обмена данными между клиентом и сервером в течение определенного периода времени. В отличие от этого, Web-клиент обычно устанавливает многочисленные соединения для получения различных ресурсов с Web-сервера. Протокол FTP (File Transfer Protocol) поддерживает отдельное соединение между клиентом и сервером для передачи команд, а данные передает с помощью других соединений. В отличие от FTP, HTTP передает и данные, и команды по одному и тому же соединению. По сравнению с файлами, передаваемыми по FTP, размеры большинства HTTP-запросов и ответов небольшие. Эти особенности Web-трафика существенно влияют на эффективность работы TCP.

В этой главе рассматривается взаимодействие между HTTP и TCP, а также его влияние на работу Web. Во-первых, мы обсудим, как TCP использует таймеры для выполнения некоторых ключевых операций, таких как повторная передача утерянных пакетов. Хотя эти таймеры влияют на любой протокол прикладного уровня, функционирующий поверх TCP, особенности трафика HTTP приводят к их более заметному влиянию на производительность Web, чем в более ранних Internet-приложениях. Затем мы изучим, как распределение функций между протоколами транспортного и прикладного уровней сказываются на производительности Web в целом. Некоторые механизмы работы TCP были обусловлены более ранними протоколами прикладного уровня, такими как Telnet и Rlogin. Эти механизмы не всегда хорошо взаимодействуют с HTTP. Затем мы обсудим влияние Web-клиентов, создающих одновременно несколько соединений с одним и тем же Web-сервером, на эффективность и качество работы. Например, браузер может установить несколько соединений для загрузки изображений, встроенных в Web-страницу. Web-серверы должны при этом работать с большим числом одновременных TCP-соединений с различными клиентами. Мы опишем способы снизить нагрузку Web-серверов при обработке большого числа TCP-соединений.

8.1. Таймеры TCP

Реализации TCP используют таймеры для запуска перечисленных ниже ключевых операций:

- **Повторная передача утерянных пакетов.** Конец работы таймера повторной передачи запускает операцию повторной передачи предположительно утерянного пакета.

- **Повтор фазы медленного старта.** Некоторые реализации TCP требуют от TCP-отправителя повторного выполнения фазы медленного старта для управления параметрами скользящего окна после некоторого периода бездействия.
- **Удержание состояния разорванного соединения.** Отправитель TCP-пакета, инициирующий разрыв соединения, окончательно «забывает» его состояние только через некоторый промежуток времени.

В этом разделе мы объясним, почему таймеры, управляющие перечисленными операциями, сильно влияют на производительность Web. Позже, в разделе 8.2.3, мы рассмотрим еще один таймер, который управляет передачей отложенных подтверждений.

8.1.1. Таймер повторной передачи

Иногда процесс загрузки Web-страниц замирает в самом начале. Эти задержки происходят преимущественно из-за того, что у отправителя много времени уходит на то, чтобы обнаружить утерю IP-пакетов. В этом разделе мы расскажем о том, как процесс установления TCP-соединения может растянуться на несколько секунд в результате большого начального значения таймаута повторной передачи (RTO — Retransmission Timeout). Затем мы объясним, почему такие задержки возникают сравнительно часто именно при передаче данных с помощью HTTP, а не при работе других протоколов прикладного уровня.

ЗАДЕРЖКИ В ПРОЦЕССЕ УСТАНОВЛЕНИЯ TCP-СОЕДИНЕНИЙ

С точки зрения пользователя, щелчок мышью на гиперссылке приводит непосредственно к отображению требуемой страницы в окне браузера. Невидимо для пользователя браузер выполняет последовательность действий, необходимых для получения и отображения Web страницы: установление TCP-соединения, передача HTTP-запроса, получение ответа, анализ содержимого и его отображение в окне. Так как браузер представляет собой интерактивное приложение, то задержка при выполнении хотя бы одной из перечисленных операций становится заметной пользователю. Это сильно контрастирует с неинтерактивными приложениями, например, электронной почтой, в которых пользователь не ожидает немедленного ответа. Хотя Telnet и FTP в некоторой мере интерактивны, но в них четко отделяется установление соединения и передача данных. При этом среднестатистический пользователь взаимодействует с удаленным компьютером достаточно долгое время; так что несколько секунд дополнительной задержки при установлении TCP-соединения и передаче имени пользователя и пароля не сильно влияют на степень удовлетворения пользователя работой с приложением.

Установка TCP-соединения представляет собой трехшаговую процедуру: клиент отправляет пакет SYN, в ответ сервер отправляет клиенту пакет SYN-ACK и, наконец, процедура завершается отправкой пакета ACK клиентом. Если пакеты не теряются, то на установление соединения уходит лишь время на отправку и получение перечисленных пакетов. В случае потери пакета SYN или SYN-ACK времени тратится существенно больше. У отправителя TCP-пакета имеется всего два способа обнаружения потери пакета: получение повторного подтверждения или завершение таймаута повторной передачи, как об этом говорилось в главе 5 (раздел 5.2.5). Но получатель не посылает повторных подтверждений, если отправитель не передал ни одного пакета данных. В начале соединения, когда отправитель еще не передал ни одного пакета, скорость передачи пакетов ограничивается не-

большим размером скользящего окна. При отсутствии повторных подтверждений TCP-отправитель должен для обнаружения потери пакета полагаться на таймер повторной передачи. При этом выбор значения RTO достаточно критичен. Слишком большое значение приведет к чрезмерной задержке в случае потери пакета, а слишком маленькое — к излишним повторным передачам. Чтобы найти компромисс, отправитель выбирает величину RTO на основе приблизительной оценки времени прохода пакета до получателя и обратно (RTT — Round-Trip Time).

Но в начале связи отправитель еще не имеет данных об RTT. Это осложняет выбор RTO для начальных пакетов соединения. Чтобы разрешить эту проблему, спецификация TCP предписывает устанавливать начальный RTO равным трем секундам [PA00]. Если SYN-ACK не приходит в течение трех секунд после отправки пакета SYN, то отправитель увеличивает RTO до шести секунд и удваивает его после каждой последующей повторной передачи, как показано на рис 8.1. Сочетание большого начального RTO и увеличение этого времени в геометрической прогрессии позволяют избежать излишних передач пакетов SYN или SYN-ACK в случае взаимодействия компьютеров с большим RTT. Если пакет SYN или SYN-ACK утерян, то задержка дополнительной отправки пакета помогает не перегружать сеть. Эти консервативные методы повторной передачи утерянных пакетов SYN и SYN-ACK не оказывают заметного влияния на неинтерактивные приложения. В то же время потеря одного или нескольких пакетов SYN или SYN-ACK заметно сказывается на работе Web. Потеря двух следующих друг за другом пакетов SYN приведет к суммарной задержке в девять секунд перед тем, как будет отправлен третий пакет SYN. При разработке протокола считалось, что вероятность потери одного или более пакетов SYN или SYN-ACK будет достаточно малой. Но широкое использование Web привело к высокой загрузке сетей и к сравнительно большой вероятности потери пакета (0.05 или более) [Pax99]. Несмотря на применение высокоскоростных линий связи во многих сегментах Internet на критических участках возникают заторы. Участки, подсоединяющие к Internet отдельные организации, университеты или даже целые страны, часто бывают перегружены, особенно в часы пик. Соединение между Internet и провайдерами обычно всегда перегружены. Даже при небольшом трафике сильно перегруженый Web-сервер может терять пакеты SYN. Операционная система, под управлением которой работает Web-сервер, поддерживает очередь TCP-соединений, а если эта очередь переполняется, новые пакеты SYN отбрасываются.

Рис 8.1. Клиент повторно отправляет серверу утерянный пакет SYN

Большие задержки влияют на поведение пользователей. Если пользователю надоело ждать, то он/она может отменить запрос нажатием кнопки Stop в браузере. Затем он нажимает кнопку Reload для повторения запроса. В итоге, когда пользователь нажимает Stop, браузер отдает распоряжение операционной системе о за-

крытии данного соединения (это будет детально описано в разделе 8.2.1). Когда же пользователь нажимает кнопку Reload, браузер немедленно начинает устанавливать новое TCP-соединение. Операционная система при этом посылает серверу пакет SYN и устанавливает начальное значение RTO по умолчанию (скажем, три секунды). Если посланный пакет SYN достигает сервера, соединение устанавливается быстрее, чем при ожидании окончания трех- или шестисекундного таймаута. Действия пользователя по отмене/повторению попытки соединения фактически вызывают немедленную переотправку пакета SYN. Пользователь может также нажать только кнопку Reload, не нажимая Stop. Нажатие Reload одновременно вызывает и закрытие текущего соединения, и открытие нового.

Завершение и повторное установление «зависшего» соединения обычно уменьшает задержку, воспринимаемую пользователем, но происходит это за счет увеличения нагрузки на сервер и на сеть. Например, операционная система компьютера с Web-сервером могут отбросить пакеты SYN от разных пользователей, если все запросы пришли со слишком малым промежутком времени. Если же пользователи при этом повторят свои запросы, то каждый из них пошлет на сервер по одному пакету SYN. Это только увеличит и без того высокую нагрузку на сервер. Такое поведение только увеличивает трафик на загруженном участке сети. Перегрузка приводит к потере пакетов. Так как большинство Web-запросов и ответов коротки, то SYN и SYN-ACK составляют относительно большую долю всех утерянных пакетов. Если в результате потери пакета пользователь останавливает и тут же снова запускает запрос, то клиент вскоре посылает второй пакет SYN. Это увеличивает трафик на перегруженном участке сети.

ЗАДЕРЖКИ В ПРОЦЕССЕ ПЕРЕДАЧИ ДАННЫХ В WEB

По сравнению с началом соединения, длинные таймауты повторной передачи в процесса передачи ответа возникают реже по двум причинам:

- **Меньшее значение RTO.** Отправитель пакетов TCP постепенно уточняет значение RTO, измеряя временя между отправкой запросов и получением ответов. Постепенно RTO приобретает значение, близкое к реальному RTT между отправителем и получателем. Например, RTT между отправителем и получателем равно 200 мс. В начале связи отправитель устанавливает трехсекундное RTO. Затем отправитель выбирает RTO в зависимости от среднего значения RTT. В результате RTO снижается до 250–300 мс.
- **Повторяющиеся подтверждения.** Появление таймаутов повторной передачи менее вероятно, когда отправитель пакетов TCP начинает посылать свои данные. По мере получения данных получатель пакетов TCP подтверждает их получение ACK-пакетами, отражающими число полученных байтов в смежных пакетах данных. В случае утери пакета это число не увеличивается в последующих ACK-пакетах. При получении таких повторяющихся пакетов отправитель понимает, что ранее переданный пакет был утерян. Это служит ему сигналом передать утерянный пакет, не дожидаясь таймаута повторной передачи, как это описывалось ранее в главе 5 (раздел 5.2.5).

Указанные положительные эффекты становятся значимыми, когда по TCP передаются большие объемы данных. Оценка RTT становится более точной при длительном обмене данными между компьютерами, и вероятность повторяющихся подтверждений увеличивается по мере того, как отправитель устанавливает большие размеры скользящего окна.

Большинство Web-ответов, однако, имеют не очень большие размеры, примерно от 8 до 12 килобайтов. Эти короткие пересылки проходят по большей части (если не целиком) на фазе медленного старта. Эта фаза начинается с небольшого начального размера скользящего окна в один или два пакета, как описано в главе 5 (раздел 5.2.6). С небольшим скользящим окном вероятность успешно доставить большое число пакетов после утери пакета очень мала.

Маловероятно, что получатель TCP-пакетов отправит три повторяющихся ACK, необходимых для быстрой повторной передачи утерянного пакета [BPS+98]. Например, утеря первого пакета ответа сервера вызовет передачу единственного повторяющегося ACK в ответ на приход второго пакета, как показано на рис. 8.2. Небольшой размер скользящего окна мешает серверу отправить дополнительные пакеты до тех пор, пока не получено подтверждение о прибытии первого пакета. Ожидание окончания интервала повторной передачи приостанавливает передачу сервером данных. Кроме того, после таймаута повторной передачи отправитель TCP-пакетов вынужден вернуться к своему первоначальному небольшому размеру скользящего окна, как показано на рис. 8.3. Таймауты повторной передачи могут вызвать значительные задержки при передаче данных. Останов и перезапуск сеанса передачи данных может привести к более быстрому отклику, чем при ожидании таймаута повторной передачи.

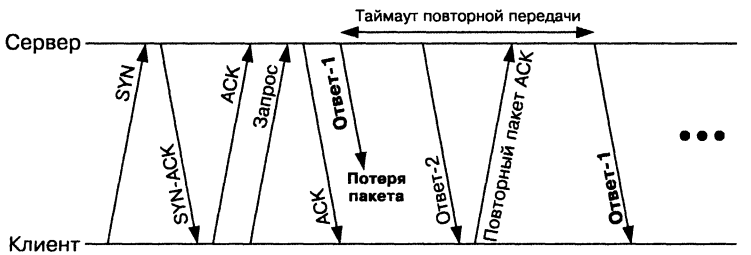


Рис 8.2. Клиент посылает повторный пакет ACK серверу

8.1.2. Повтор фазы медленного старта

Долговременные соединения создают привлекательную альтернативу установлению отдельных TCP-соединений для каждого сеанса передачи данных. Возможность предотвратить фазу медленного старта является одним из преимуществ использования уже имеющегося соединения, как уже говорилось в главе 7 (раздел 7.5). Однако отправка ответного сообщения по пассивному в течение некоторого времени долговременному соединению приводит к появлению большого числа передаваемых пакетов. Чтобы не перегрузить сеть, спецификация TCP требует от отправителя повторить фазу медленного старта для управления скользящим окном после некоторого периода бездействия соединения. Однако имеется несколько рекомендаций, как все-таки избежать этой фазы без перегрузки сети.

НЕАКТИВНЫЕ ДОЛГОВРЕМЕННЫЕ СОЕДИНЕНИЯ

Клиент HTTP часто загружает многочисленные ресурсы с одного Web-сайта за очень короткое время. Например, пользователь может загрузить Web страницу, содержащую несколько изображений. Браузер запрашивает эти изображения, проанализировав содержимое HTML-файла. В дополнение к этому пользователь мо-

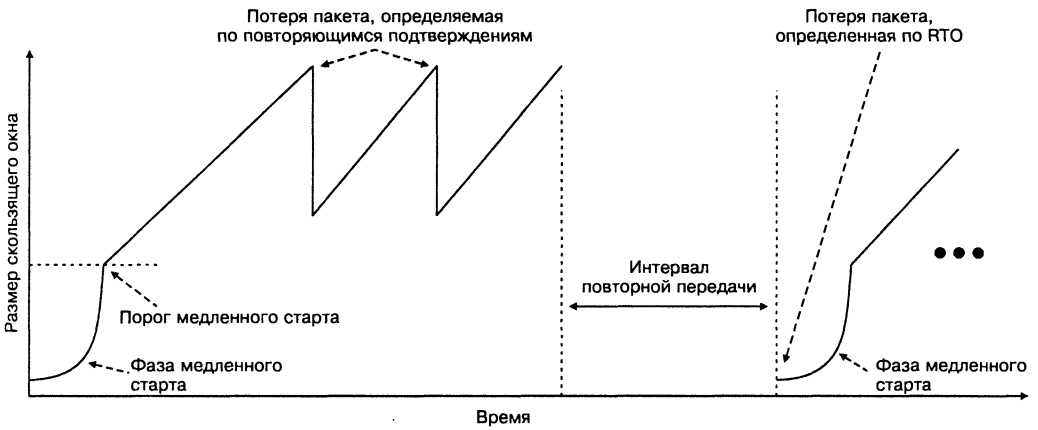


Рис. 8.3. Повтор фазы медленного старта после таймаута повторной передачи

жет щелкнуть на гиперссылке, чтобы просмотреть различные страницы, размещенные на этом же сервере. При наличии долговременного соединения ответы на все эти запросы могут передаваться по одному и тому же TCP-соединению. Пропускная способность может быть большой из-за большого размера скользящего окна. Предположим, что клиент запрашивает HTML-файл, после чего следуют запросы четырех содержащихся в нем изображений. Передача данных начинается при размере скользящего окна в один или два пакета. Размер скользящего окна растет при прохождении фазы медленного старта. Использование того же самого соединения позволяет воспользоваться возросшим размером скользящего окна при загрузке изображений. Далее скользящее окно может продолжать увеличиваться по мере передачи сервером многочисленных ответов.

С другой стороны, использование больших размеров скользящего окна может перегрузить сеть, если долговременное соединение оставалось неактивным в течение некоторого времени. Представим себе пользователя, который загружает Web-страницу с сервера, а после пяти секунд чтения страницы загружает другую страницу с того же сервера. Предположим также, что клиент и сервер оба решили сохранить соединение. Сетевой трафик может существенно измениться за пятисекундный период простоя. Пока соединение было неактивно, другие TCP-соединения с тем же сервером могли расширить свои скользящие окна так, чтобы в полной мере использовать пропускную способность канала. Возьмем простой пример: по участку сети проходит трафик пяти соединений с одинаковыми RTT. В среднем, каждое из них имеет пропускную способность в 20% от общей пропускной способности участка сети. Если одно из этих соединений будет неактивно несколько секунд, то остальные четыре соединения «съедят» его долю и будут занимать по 25% от общей пропускной способности.

Если соединение, которое было некоторое время неактивным, будет работать с тем же размером скользящего окна, который оно имело до периода простоя, то это может вызвать серьезную перегрузку сети. Соединение сразу начнет использовать для передачи 20% пропускной способности сети, хотя остальные соединения продолжат использовать по 25%. В таком случае пропускная способность будет превышена на 20%. Многие из передаваемых пакетов будут теряться. Между прочим, утери пакетов могут заставить некоторые соединения снизить скорость передачи, что приведет к недогрузке сети. Соединениям придется заново медленно уве-

личивать размеры своих скользящих окон, чтобы, в конце концов, полностью использовать пропускную способность сети. В итоге, помимо снижения общей производительности, отправка слишком большого числа пакетов может ухудшить пропускную способность и самого соединения, так как общая перегрузка сети может привести к существенным потерям пакетов данным соединением. Все это приведет к повторной передаче пакетов.

ПОВТОР ФАЗЫ МЕДЛЕННОГО СТАРТА

Чтобы предотвратить «вспышку» активности, отправитель должен после периода простоя передавать данные менее агрессивно, чем до этого. Фаза медленного старта была введена в протокол, чтобы избежать неожиданных и резких «вспышек» трафика. В течение фазы медленного старта TCP-отправитель увеличивает свое скользящее окно и пытается оценить размер своей законной доли пропускной способности соединения с сервером. Позволить соединению, бывшему неактивным, получить сразу большое скользящее окно — это все равно, что позволить новому соединению начать работу с большим скользящим окном. Такая «вспышка» трафика может вызвать перегрузку сети и обусловить большую вероятность утери пакетов. Чтобы избежать этой проблемы, спецификация TCP требует повторения фазы медленного старта после периода бездействия. Этот механизм называется *повтором фазы медленного старта*.

Точное определение механизма повтора фазы медленного старта требует уточнения двух ключевых понятий: начала периода бездействия и установки начального размера скользящего окна. Период бездействия начинается сразу же, как только отправитель перестает передавать данные, а прибытие всех предыдущих пакетов с данными было подтверждено получателем. Получив последнее подтверждение, отправитель запускает таймер. Когда время таймера истекает, отправитель устанавливает начальный размер скользящего окна, равный длине одного-двух пакетов данных. Отправитель TCP-пакетов использует текущее значение RTO для измерения периода бездействия. Иными словами, если приложение не передает данных в течение периода времени, равного RTO, то отправитель устанавливает начальный размер скользящего окна и осуществляет повтор фазы медленного старта.

Повтор фазы медленного старта может начаться, если соединение неактивно дольше, чем несколько RTT, что составляет обычно не более нескольких секунд. Большинство автоматически генерируемых запросов на загрузку встроенных ресурсов приходят именно за такой интервал времени. Помимо этого, долговременные соединения могут обслуживать запросы, генерируемые пользователями, когда они просматривают различные страницы на том же сайте. При этом часто бывают значительные задержки между последовательными запросами. Свойства таких задержек между последовательными запросами более подробно рассматриваются в главе 10 (раздел 10.5.3). Большие промежутки между запросами, обусловленные просмотром страниц пользователями, могут привести к повтору фазы медленного старта. Повтор фазы медленного старта снижает преимущества повышения производительности с помощью долговременных соединений. Сервер не сможет в полной мере использовать высокую пропускную способность канала, если каждый последовательный запрос пользователя будет начинаться с небольшим скользящим окном. С другой стороны, повтор фазы медленного старта также влияет на общую пропускную способность сети.

УМЕНЬШЕНИЕ ВЛИЯНИЯ ПОВТОРОВ ФАЗЫ МЕДЛЕННОГО СТАРТА НА ОБЩУЮ ПРОПУСКНУЮ СПОСОБНОСТЬ СЕТИ

Механизм повтора фазы медленного старта предотвращает генерацию слишком большого числа пакетов для соединений, которые были перед этим неактивны. Предлагается несколько способов для уменьшения или даже устранения влияния повтора фазы медленного старта на снижение производительности соединения [Hei97]:

- **Отключение повтора фазы медленного старта.** Web-сервер может заблокировать механизм повтора фазы медленного старта, оставляя долговременному соединению возможность отправить большое число пакетов за короткое время. Риск уменьшается, если сервер закрывает соединения после небольшого периода простоя. Например, сервер может закрыть соединение, которое было неактивно в течение 15 секунд, чтобы уменьшить общее число соединений, как это описано ниже в разделе 8.4.2.
- **Использование большего таймаута повтора фазы медленного старта.** В операционной системе, под управлением которой функционирует Web-сервер, можно увеличить таймаут, необходимый для повтора фазы медленного старта. Однако не следует забывать, что чем больше таймаут, тем больше вероятность того, что размер скользящего окна, остающийся от предыдущего периода активности, не будет соответствовать текущему состоянию сети.
- **Постепенное уменьшение размера скользящего окна.** Вместо того чтобы использовать решение «все или ничего», отправитель TCP-пакетов может *постепенно* уменьшать размер скользящего окна для неактивного соединения. Отправитель может уменьшать ее пропорционально времени периода простоя. Этот адаптивный подход становится все более консервативным при увеличении неопределенности в предсказании состояния сети.
- **Постепенная отправка пакетов.** Чтобы избежать резкого возрастания числа передаваемых пакетов, отправитель может постепенно увеличивать число отправляемых получателю пакетов. Предположим, что размер скользящего окна равен длине четырех пакетов. Передача всех четырех пакетов сразу друг за другом может резко увеличить трафик и вызвать перегрузку сети. Вместо этого отправитель может делать небольшой перерыв после отправки каждого пакета. Это уменьшит вероятность перегрузить сеть, используя по-прежнему скользящее окна, размер которого равен размеру четырех пакетов.

Реализация TCP на компьютере, где функционирует Web-сервер, может применять комбинацию этих методов. Например, отправитель может использовать больший таймаут, постепенно уменьшать размер скользящего окна и постепенно передавать пакеты.

8.1.3. Сохранение состояния TIME_WAIT

Сильно загруженные Web-серверы обрабатывают большое количество запросов на создание новых TCP-соединений. Поддержка большого числа TCP-соединений требует большого объема оперативной памяти и приводит к большим накладным расходам при обработке получаемых пакетов. В идеале операционная система могла бы освобождать эти ресурсы сразу же, как только соединение закрывается. Но протокол TCP требует, чтобы один или оба компьютера сохраняли информацию о закрытом соединении. В этом подразделе мы объясним, почему хотя бы один из компьютеров должен сохранять состояние TIME_WAIT и почему ответственность за это обычно ложится не на клиента, а на сервер. Потом мы рассмотрим несколько

предложений по снижению накладных расходов, связанных с сохранением состояния TIME_WAIT на Web-сервере.

СОХРАНЕНИЕ ИНФОРМАЦИИ О ЗАКРЫВШИХСЯ СОЕДИНЕНИЯХ

Сохранение информации о закрывшихся TCP-соединениях приводит к значительным накладным расходам на сильно загруженном Web-сервере. Рассмотрим пример сервера, который закрывает соединение после отправки HTTP-ответа клиенту. Закрытие соединения производится с помощью процедуры, состоящей из четырех шагов и начинающейся передачей пакета FIN, как это описывалось в главе 5 (раздел 5.2.3). При получении пакета FIN, клиент посылает пакет с подтверждением. Потом клиент читает HTTP-ответ. За последним байтом сообщения клиент находит символ конца файла (EOF), который означает, что сервер закрыл свою сторону соединения. Клиент затем закрывает свою сторону соединения, посылая при этом пакет FIN серверу. Получив пакет FIN от клиента, сервер посылает ACK, завершающий четырехшаговую процедуру. Получение подтверждающего пакета ACK сообщает клиенту, что соединение закрыто. Но в то же время сервер не может знать, дошло ли его подтверждение до клиента или нет.

Предположим, что последний пакет ACK от сервера был утерян в сети, как показано на рис. 8.4. Если пакет ACK был потерян, то клиент, в конце концов, запустит таймер повторной передачи, по истечению которого он снова пошлет пакет FIN. При потере этого пакета клиент снова запустит таймер повторной передачи, после чего снова пошлет пакет FIN. Так как сервер не знает, дошел ли его пакет ACK по назначению или нет, то сервер не может решить, надо ли передавать ли его снова. Поэтому сервер не может удалить информацию о соединении в течение некоторого времени. Предположим, что сервер все-таки удалил информацию о соединении, некорректно полагая, что клиент получил его пакет ACK. Позже, когда от клиента приходит повторный пакет FIN, сервер не сможет понять, что этот пакет относится к только что закрытому соединению. Думая, что пакет FIN был послан по ошибке, сервер пошлет клиенту пакет RST вместо того, чтобы снова послать подтверждающий пакет ACK.

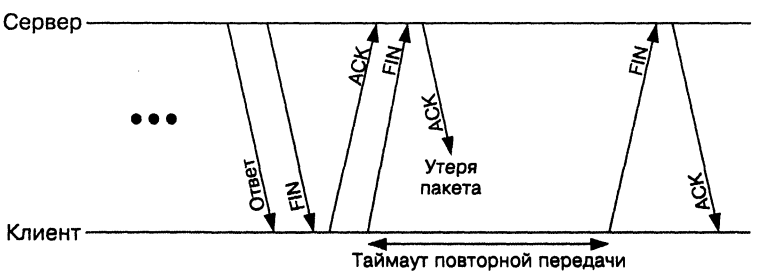


Рис 8.4. Потеря и повторная передача заключительного ACK-пакета сервером

Более серьезная ситуация складывается, если один или более пакетов, относящихся к данному соединению, продолжают путешествовать по сети. Предположим, что сервер получил запрос целиком, а клиент получил ответ полностью. Какой-нибудь повторно отправленный пакет может еще по разным причинам оставаться в сети. Это случается, если пакет был передан более одного раза, причем одна копия дошла до адресата, а другая продолжала блуждать по сети. Такой пакет может, в конце концов, добраться до адресата. Обработка повторного пакета зависит от состояния соединения между компьютерами:

- **Соединение все еще открыто.** Предположим, что повторный пакет приходит, пока соединение еще открыто. Получатель смотрит номер пакета и находит, что эти данные уже были получены. Получатель отбрасывает такой пакет.
- **Соединение было закрыто и нет нового соединения.** Предположим, что старое соединение было закрыто, а новое соединение не было установлено. Когда пакет приходит, операционная система получателя просматривает номера портов TCP в заголовке пакета, чтобы определить, к какому соединению он относится. Не найдя такого соединения, получатель также отбрасывает пакет.
- **Соединение было закрыто, а новое соединение создано.** Предположим, что соединение было закрыто, и компьютеры создали новое соединение с *той же парой* номеров портов. Когда повторный пакет прибывает, номера портов анализируются операционной системой получателя. Номера портов пакета совпадают с номерами портов нового соединения, и данные передаются приложению, связанному с новым соединением. Это ошибочно, так как этот пакет не принадлежит новому соединению.

Чтобы получатель не ассоциировал повторный пакет с другим приложением, оба компьютера должны избегать создания нового соединения с теми же самыми номерами портов, по крайней мере, в течение некоторого промежутка времени после закрытия предыдущего соединения.

Чтобы предотвратить создание нового соединения с теми же номерами портов, хотя бы *один* из компьютеров должен помнить о существовании предыдущего соединения. Спецификация TCP возлагает эту обязанность на компьютер, который первым послал пакет FIN. На этом компьютере TCP-соединение переходит в состояние TIME_WAIT. Операционная система сохраняет информацию, связанную с соединением, чтобы в случае необходимости повторно передать заключительный пакет ACK и предотвратить создание нового соединения с теми же номерами портов. Соединение должно оставаться в состоянии TIME_WAIT столько времени, сколько нужно для того, чтобы в сети не осталось ни одного пакета, принадлежащего данному соединению. Выполнить это очень сложно, так как протокол IP не вводит ограничений на время пребывания пакета в пути. На практике поле в заголовке IP-пакета, содержащее время его жизни (TTL), должно ограничивать время пребывания пакета в сети, как это описано в главе 5 (раздел 5.1.4). Это 8-битное поле позволяет задавать максимальное значение TTL в 255 секунд.

TCP требует, чтобы компьютер оставался в состоянии TIME_WAIT в течение двух периодов *максимального времени жизни сегмента (MSL — maximum segment lifetime)* — оценки времени наибольшей задержки при доставке пакетов. В спецификации TCP указывается, что реализации протокола должны использовать значение MSL, равное двум минутам [J.81], хотя на практике различные реализации используют 30 секунд, одну минуту или две минуты. С одной стороны, малое значение MSL уменьшает длительность состояния TIME_WAIT, что в свою очередь уменьшает объем системных ресурсов, необходимых для хранения информации о закрытых соединениях. Однако небольшое значение MSL увеличивает вероятность того, что повторные пакеты остаются в сети. Кроме того, небольшое значение MSL уменьшает вероятность того, что отправитель сумеет повторно выслать утерянный заключительный пакет ACK. С другой стороны большое значение MSL приводит к тому, что состояние TIME_WAIT длится дольше. В результате, достаточно много соединений могут оказаться в состоянии TIME_WAIT одновременно, что приводит к увеличению требований к системным ресурсам на сильно загруженном сервере. Необходимо также отметить, что большие значения MSL ограничивают скорость взаимодействия между компьютерами.

Предположим, что клиент создает соединение по порту 1025 с сервером, функционирующим на удаленном компьютере на порту 80. После передачи пакета FIN для закрытия TCP-соединения сервер переходит в состояние TIME_WAIT на четыре минуты. Получив подтверждение ACK от сервера, клиент завершает соединение. Клиент может попытаться установить новое соединение с тем же сервером, чтобы запросить другой ресурс. Это соединение должно использовать другой клиентский порт (например, 1026), потому что сервер все еще находится в состоянии TIME_WAIT из-за завершенного соединения. Заголовок TCP пакета имеет 16-битное поле номера порта, что ограничивает число различных портов, которое клиент может использовать для соединения. На практике это не приводит к ограничениям возможностей браузера. Но вот прокси-серверу обычно нужно устанавливать соединения с популярными серверами с очень высокой частотой. Ограничение становится еще более серьезным, если прокси-сервер должен передавать все запросы другому прокси-серверу. Невозможность создавать новые соединения со старыми номерами портов ограничивает скорость, с которой прокси-сервер может посылать запросы следующему прокси-серверу в цепочке.

ВЛИЯНИЕ TIME_WAIT НА WEB-СЕРВЕРЫ

Web-серверы часто инициируют закрытие соединений, что обусловливает дополнительные накладные расходы, связанные с поддержанием состояния TIME_WAIT. Сначала рассмотрим случай, когда клиент и сервер не поддерживают долговременного соединения. В такой ситуации сервер обычно закрывает соединение (посылая пакет FIN) сразу после отсылки HTTP-ответа клиенту. Теперь рассмотрим другую ситуацию, когда клиент и сервер поддерживают соединение открытым. В этом случае все равно наступает момент, когда либо клиент, либо сервер закрывают соединение. Обычно сервер имеет больше причин закрыть соединение. Например, загруженный Web-сервер не может позволить себе поддерживать долговременные соединения с каждым клиентом. Серверы обычно используют таймеры на прикладном уровне для закрытия соединений после некоторого периода бездействия, как будет рассмотрено позже, в разделе 8.4.2. Браузер, напротив, имеет достаточно мало поводов закрыть соединение, потому что использование долговременного соединения позволяет избежать задержек, связанных с созданием нового соединения.

В любом из перечисленных сценариев первым закрывает соединение сервер и, соответственно, он и должен поддерживать состояние TIME_WAIT. Ситуация осложняется, если клиентом является прокси-сервер. Интенсивно функционирующий прокси-сервер связан TCP-соединениями с большим числом клиентов и серверов. У прокси-сервера столько же причин закрыть соединение, сколько и у Web-сервера, если не больше. Если прокси-сервер закрывает соединение, то он и должен поддерживать состояние TIME_WAIT, а не Web-сервер. Поддержка состояний TIME_WAIT сильно влияет на производительность прокси-серверов. В общем случае TIME_WAIT представляет собой неудачный компромисс, так как загруженные компьютеры должны закрывать соединения и в то же время поддерживать состояние TIME_WAIT.

Дополнительная нагрузка, обусловленная TIME_WAIT, увеличивается еще и за счет того, что TCP-соединения — достаточно коротки, так как большинство Web-ответов относительно невелики. Предположим, что клиент запрашивает одиночный ресурс с Web-сервера. Сервер тратит несколько секунд, отвечая на этот запрос. После закрытия соединения сервер находится в состоянии TIME_WAIT в течение четырех минут. Это гораздо больше времени соединения. Более ранние Internet-приложения, такие как Telnet или FTP, обычно использовали TCP-соеди-

нения в течение более долгого времени. По сравнению с FTP-сервером, HTTP-сервер большую часть времени находится в состоянии TIME_WAIT. Использование долговременных соединений частично решает эту проблему. Использование одного соединения для нескольких HTTP-запросов уменьшает общее число как открытых, так и закрытых соединений на сервере. А это, в свою очередь, уменьшает количество соединений, находящихся в состоянии TIME_WAIT. И, тем не менее, интенсивно работающему Web серверу все равно приходится поддерживать достаточно много соединений в состоянии TIME_WAIT.

УМЕНЬШЕНИЕ НАГРУЗКИ, СВЯЗАННОЙ С СОСТОЯНИЕМ TIME_WAIT

Уменьшение нагрузки, обусловленной состоянием TIME_WAIT, чрезвычайно важно для обеспечения высокой работоспособности Web-сервера. Способы уменьшения нагрузки делятся на две основные категории: уменьшение объема ресурсов, идущих на поддержание состояния TIME_WAIT, и передача ответственности за поддержание состояния TIME_WAIT клиенту. Операционная система должна сохранять информацию о состоянии TIME_WAIT для каждого соединения. Сохранение этой информации требует памяти, которая иначе могла бы использоваться для других целей, например, для кэширования часто запрашиваемых Web-ресурсов.

К счастью, операционной системе необходимо хранить меньше информации о соединениях в состоянии TIME_WAIT, чем об активных соединениях. В современных операционных системах требования к памяти, необходимой для повторной передачи последнего пакета ACK и для предотвращения создания соединения с теми же IP-адресами и номерами портов, снижены до минимума. Большое число соединений в состоянии TIME_WAIT увеличивает также ресурсы, необходимые для проверки таймеров, например, таймера повторной передачи. Многие операционные системы проверяют состояния таймеров, периодически сканируя список TCP-соединений. Перенося соединения в состоянии TIME_WAIT в конец списка, операционная система уменьшает время, требуемое для сканирования списка. Уменьшение затрат процессорного времени и памяти на соединения в состоянии TIME_WAIT приводят к заметному увеличению производительности Web-серверов [AD99].

Несмотря на достигнутое уменьшение затрат процессорного времени и памяти, соединения в состоянии TIME_WAIT все равно могут привести к перегрузке Web-серверов. Предлагаются несколько методов переложить ответственность за обработку состояния TIME_WAIT на клиента [FTY99], который, как предполагается, работает с небольшим числом соединений.

- **Внесение изменений в TCP.** Спецификация TCP может быть изменена таким образом, чтобы обработку состояния TIME_WAIT производил *получатель* первого пакета FIN. Например, Web-сервер закрывает соединение и посылает клиенту пакет FIN. Получив пакет FIN, клиент переходит в состояние TIME_WAIT. Чтобы убедиться в том, что сервер не находится в состоянии TIME_WAIT, клиент может послать серверу пакет RST (рис. 8.5). Получение RST заставляет сервер покинуть состояние TIME_WAIT и освободить ресурсы, отведенные для соединения. Альтернативный подход представляет собой модификацию процедуры установления соединения так, чтобы клиент и сервер договаривались, кто будет обрабатывать TIME_WAIT при закрытии соединения.

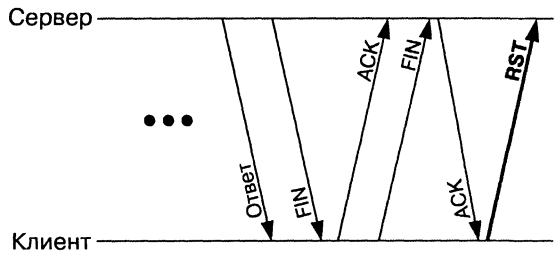


Рис 8.5. Клиент посылает RST, чтобы освободить сервер от необходимости поддержки состояния TIME_WAIT

- **Внесение изменений в HTTP.** Спецификация HTTP может быть изменена таким образом, что закрытие соединения должен инициировать клиент, а не сервер. Например, новый заголовок ответа может содержать инструкцию клиенту закрыть соединение после получения ответа. Новый заголовок запроса может позволить клиенту принять на себя ответственность за закрытие соединения. Например, клиент может закрыть соединение автоматически, скачав встроенные в HTML-страницу изображения. Правда, отдавать управление соединением клиенту — не всегда подходящее решение. Клиент может иметь веские причины не закрывать соединение или не принимать на себя ответственность за обслуживание состояния TIME_WAIT.

И тот, и другой способ требуют внесения изменений в спецификации либо TCP, либо HTTP. Ни один из предложенных подходов не предоставляет привлекательного решения проблемы. Вместо этого администраторы загруженных Web-серверов вносят изменения в настройки операционной системы так, чтобы уменьшить длительность состояния TIME_WAIT (например, до 5 секунд). Это уменьшает нагрузку, связанную с поддержанием состояния TIME_WAIT, увеличивая тем самым вероятность создания соединение с теми же номерами портов.

8.2. Разделение функций между HTTP и TCP

Передача Web-ресурсов базируется на функциях прикладного уровня, предоставляемых HTTP, и на функциях транспортного уровня, предоставляемых TCP. В некоторых случаях становится непонятно, какой уровень должен выполнять ту или иную функцию. В этом разделе содержатся три примера, в которых функции, реализованные на уровне транспортного уровня, влияют на работу Web.

- **Отмененные HTTP-передачи.** Так как в HTTP отсутствует механизм для прекращения передачи данных, то для отмены текущего HTTP-запроса на транспортном уровне необходимо закрыть соединение, по которому производится запрос, как описано в разделе 8.2.1.
- **Алгоритм Нагла.** Алгоритм Нагла (Nagle) ограничивает число небольших пакетов, передаваемых TCP-отправителем, что может задержать отправку последнего пакета HTTP-сообщения, как описано в разделе 8.2.2.
- **Отложенные подтверждения.** Получатель TCP-пакетов может отложить передачу подтверждения в надежде передать его потом вместе с пакетом данных, увеличивая тем самым задержку при передаче HTTP-сообщений, как это подробно описано в разделе 8.2.3.

Обработка прерванных передач HTTP на транспортном уровне позволяет HTTP избежать привязки к какому-либо конкретному транспортному протоколу. Алгоритм Нагла и отложенные подтверждения были добавлены в реализации TCP, чтобы уменьшить накладные расходы по передаче данных в случае таких интерактивных приложений, как Telnet и Rlogin.

8.2.1. Отмененные HTTP-передачи

Пользователь может отменить текущую HTTP-передачу, нажав кнопку Stop или щелкнув мышью на гиперссылке, чтобы перейти на другую страницу. Операции отмены — это обычная составляющая процесса работы в Web. В некоторых случаях браузер может отображать содержимое страницы по мере ее загрузки с сервера. Это позволяет пользователю прочитать часть страницы и, возможно, щелкнуть мышью на гиперссылке до того, как страница будет загружена полностью. Пользователь может щелкнуть мышью на гиперссылке до завершения загрузки изображений. В других случаях пользователь может отменить слишком медленную загрузку и нажать кнопку Reload, чтобы попытаться снова загрузить страницу. Пользователи с медленными подключениями к Internet чаще отменяют запросы. Вероятность прерывания загрузки больших ресурсов достаточно велика. В данном разделе мы объясним, почему для отмены передачи данных требуется закрыть соединение и расскажем о влиянии этого на производительность Web.

ОТСУТСТВИЕ МЕХАНИЗМА ОТМЕНЫ ПЕРЕДАЧИ ДАННЫХ В HTTP

В протоколе HTTP отсутствует механизм прерывания клиентом текущей передачи данных. Возможность отмены текущей передачи данных лежит на границе между прикладным и транспортным уровнями. Включение механизма отмены передачи данных в HTTP затруднительно без привязки к какому-либо конкретному протоколу транспортного уровня. Хотя обычно используется TCP, спецификация HTTP не связана с особенностями какого-либо протокола транспортного уровня. Другие протоколы прикладного уровня, например Telnet, столкнулись с теми же проблемами. В Telnet выбрано другое решение. Во время сеанса Telnet пользователь может нажать Ctrl+C или Delete, чтобы отменить текущее действие. Отправитель может использовать поле указателя срочности в заголовке TCP-пакета, чтобы привлечь внимание получателя, как описано в главе 5 (раздел 5.2.7). Это позволяет получателю избежать необходимости обрабатывать предыдущие байты сообщения. Правда, это решение привязано к протоколу TCP.

В спецификации HTTP/1.1 можно было бы пойти по тому же пути, введя метод запроса для отмены передачи текущего сообщения. Но это осложнит обработку последовательностей запросов. Предположим, что клиент послал последовательность запросов через долговременное соединение, а после этого послал по тому же соединению запрос об отмене передачи данных. Серверу необходимо было бы просмотреть всю очередь входящих запросов, чтобы узнать о том, что клиент послал запрос об отмене. Если клиент послал несколько запросов последовательности, то либо серверу пришлось бы отменять все остающиеся в его очереди запросы, либо клиенту пришлось бы указывать, какие именно запросы нужно отменить. Решение этих проблем сильно усложнило бы протокол. Поэтому спецификация HTTP/1.1 не включает механизма отмены передачи данных.

В результате у Web-клиента нет эффективного способа отмены HTTP-запроса, кроме разрыва соединения, по которому был отправлен запрос. Например, пользователь отменил загрузку 20-ти мегабайтного файла, когда 5 мегабайтов уже полу-

чены. Клиент (то есть браузер) имеет выбор: разорвать соединение или получить 15 Мбайт ненужных данных. Получение данных увеличивает трафик и уменьшает пропускную способность, которая очень пригодилась бы для выполнения следующего запроса пользователя. Все это увеличивает задержку, воспринимаемую пользователем. Соответственно, большинство реализаций Web клиентов предпочитают закрывать соединения.

ВЛИЯНИЕ ОПЕРАЦИЙ ОТМЕНЫ ПЕРЕДАЧИ ДАННЫХ НА ПРОИЗВОДИТЕЛЬНОСТЬ

Разрыв TCP-соединения оказывает заметное влияние на производительность. Например, браузер по протоколу HTTP/1.1 установил долговременное соединение с сервером. Получив HTML-файл, браузер посылает последовательность запросов на изображения, встроенные в страницу. Предположим, что когда сервер передавал эти изображения, пользователь щелкнул мышью на гиперссылке, чтобы перейти на другую страницу того же сайта. Браузер разрывает соединение, чтобы отменить последовательность запросов. Разрыв соединения позволяет избежать получения ненужных изображений. Теперь, пользователь должен ждать, когда браузер установит новое соединение, чтобы получить другую страницу. Для того чтобы открыть соединение, требуется обычная трехшаговая процедура и повторение фазы медленного старта.

Отмена запроса может привести к дополнительным проблемам, если у запроса имеется побочный эффект. Например, запрос пользователя может создать новый ресурс, увеличить значение переменной или запустить сценарий для покупки товара на сайте электронной торговли. Если соединение разрывается до того, как браузер получит ответ сервера, пользователь не узнает, был ли выполнен запрос или нет. HTTP не предоставляет серверу возможности в одностороннем порядке оповестить пользователя о том, что его запрос был обработан. Web-сайт может сохранить дополнительные данные, которые позволят клиенту узнать, был ли обработан запрос. Предположим, что пользователь посетил сайт электронной торговли и послал запрос, который добавляет товар в «магазинную тележку». Отменив запрос, пользователь не знает, был ли товар добавлен в тележку. На Web сайте может быть страница, на которой пользователь может увидеть текущее содержимое своей тележки. Это позволит пользователю узнать, был ли выполнен предыдущий запрос до того, как соединение было разорвано. Другой вариант состоит в том, что HTML-форма может включать уникальный номер транзакции, который позволит сценарию, обрабатывающему запрос, определить, была ли транзакция выполнена ранее.

Операция отмены передачи данных теснее связывает между собой запросы из одной последовательности в одном TCP-соединении. Отмена одного запроса в последовательности приводит к отмене всех еще необработанных запросов в этой последовательности. Например, отмена загрузки страницы прекращает загрузку всех встроенных в страницу изображений. Это совпадает с намерением пользователя отменить загрузку всей страницы, а не отдельных ее ресурсов. А теперь рассмотрим пример прокси-сервера, который посылает последовательность запросов двух клиентов по одному TCP-соединению с Web-сервером. Прокси-сервер сначала посылает запрос от лица клиента A, а затем — от лица клиента B. Если клиент A отменит свой запрос, у прокси-сервера есть две возможности. Он может сохранить соединение открытым и получить весь ответ Web-сервера клиенту A. Это неэкономно, так как ответ уже не нужен клиенту. Прокси-сервер может закрыть соединение. В таком случае серверу потребуется открыть новое соединение и повторить запрос клиента B. Оба варианты плохи. Прокси-сервер не может уйти от этой проблемы, не открывая отдельного соединения для каждого из клиентов.

Операция отмены, инициированная пользователем, не сразу останавливает передачу данных. Рассмотрим пример браузера, контактирующего напрямую с Web-сервером. Когда пользователь нажимает кнопку Stop, браузер инициирует закрытие соединения, посылая серверу пакет FIN или RST. Некоторое время сервер продолжает передавать данные клиенту. В зависимости от времени доставки пакетов и размера ответа сервера, последний может быть целиком передан клиенту до того, как сервер получит RST/FIN. Проблема обостряется, когда клиент передает запрос через прокси-сервер. В таком случае, передача HTTP-данных использует TCP-соединение между клиентом и прокси-сервером, а также соединение между прокси-сервером и Web-сервером. Предположим также, что соединение сервер-прокси имеет большую пропускную способность, чем соединение клиент-прокси. Это обычная ситуация, если у клиента медленное подключение. В связи с несовпадением пропускных способностей пересылка данных между Web-сервером и прокси-сервером осуществляется быстрее, чем между прокси-сервером и клиентом.

Рассмотрим случай, когда клиент запрашивает ресурс объемом 20 мегабайтов и отменяет запрос, получив 5 мегабайтов данных. Прокси-сервер мог получить за это время весь ресурс. Пересылка оставшихся 15 мегабайтов становится бесполезной, если прокси-сервер не получит запрос на этот же ресурс в ближайшем будущем. Если бы клиент связывался с Web-сервером непосредственно, а не через прокси-сервер, сервер не передал бы лишние 15 мегабайтов. Управление трафиком не дало бы Web-серверу посылать пакеты так часто. Предотвращение передачи лишних данных требует некоторой взаимосвязи между соединениями «сервер-прокси» и «прокси-клиент». Прокси-сервер может ограничить объем данных, получаемых им по соединению с сервером. Не читая данных во входном буфере, прокси-сервер фактически уменьшает приемное окно TCP-соединения с сервером. В свою очередь, это ограничивает объем данных, которые сервер может передать. Выбор объема данных, которое прокси-сервер читает из входного буфера, связан с противоречием между получением лишних данных при отмене передачи данных и уменьшением задержек при нормальной передаче данных.

ЗАКРЫТИЕ TCP-СОЕДИНЕНИЯ

Браузер отменяет текущую передачу данных, сделав системный вызов, чтобы закрыть соединение с Web-сервером. В зависимости от браузера и операционной системы, этот вызов приведет к передаче либо пакета FIN, либо RST. Предположим, что система передает пакет FIN. Получив пакет FIN, операционная система передаст приложению Web-сервера EOF. Операционная система продолжит передавать данные из выходного буфера удаленному клиенту. Получив EOF, Web-сервер прекращает передавать данные в выходной буфер. Данные, уже находящиеся в буфере или находящиеся в сети, будут доставлены компьютеру клиента (на котором работает браузер). Получит ли эти дополнительные данные браузер — зависит от того, как было закрыто соединение. Если браузер закрыл концы соединения для передачи и приема данных, то операционная система отбросит данные. Если конец для приема данных остается открытым, то операционная система передаст данные браузеру. Продолжение получения данных может быть полезно, если браузер планирует кэшировать полученное содержимое для обработки последующих запросов пользователя.

Некоторые реализации UNIX не позволяют приложениям инициировать отправку пакета RST. В них отмена запроса порождает пакет FIN, тогда как в других операционных системах возможна передача пакета RST. Получая RST, операционная система сервера отбросит все подготовленные к отправке через данное соединение

данные, включая те данные, которые Web-сервер уже записал в выходной буфер. Это одновременно и хорошо, и плохо. Плюсом является то, что такое закрытие соединения позволяет избежать передачи лишних данных с сервера. В дополнение к этому, пакет RST заставляет операционную систему очистить и входной буфер, вместо того, чтобы передать его содержимое приложению, с которым ассоциировано соединение. Это позволит Web-серверу обойтись без чтения последовательности HTTP-запросов, которые могут находиться во входном буфере. Минусом является то, что RST не очень аккуратно закрывает соединение. Предположим, что часть пакетов, отправленных сервером, не дошла до получателя. Получив RST, операционная система сервера не осуществляет повторную передачу утерянных пакетов.

8.2.2. Алгоритм Нагла

Интерактивные приложения, такие как Rlogin и Telnet, обычно передают много небольших пакетов с клавиатурными командами пользователя и короткими ответами на них. Алгоритм Нагла уменьшает число небольших пакетов, задерживая передачу данных [Nag84]. Описав причины ограничения числа небольших пакетов, мы расскажем, как алгоритм Нагла снижает производительность особенно при использовании долговременных соединений. Потом мы расскажем, как Web-сервер может предотвратить передачу небольших пакетов, даже если алгоритм Нагла отключен.

УМЕНЬШЕНИЕ ЧИСЛА НЕБОЛЬШИХ ПАКЕТОВ

Рассмотрим пример приложения Telnet, позволяющего пользователю взаимодействовать с удаленным компьютером. Это приложение координирует передачу команд пользователя удаленному компьютеру и ответов в обратном направлении. Быстрые ответы нужны, чтобы у пользователя создавалось ощущение непосредственного взаимодействия с удаленным компьютером. Но отправитель TCP-пакетов на компьютере с клиентом Telnet не должен генерировать отдельный IP-пакет для каждой клавиатурной команды. Иначе команда, содержащая единственный символ, приведет передаче 41-байтного пакета: 20-байтный заголовок IP, 20-байтный заголовок TCP и один байт данных. Передача 41 байта на каждый байт данных приводит к значительным накладным расходам, что, в свою очередь, приведет к чрезмерному увеличению трафика в сети. Получив данные, операционная система осуществляет накопление данных перед тем, как их передать. Интерактивные приложения не терпят задержек, поэтому операционная система не должна откладывать передачу данных надолго. Алгоритм Нагла предназначен для поиска компромисса. Алгоритм гарантирует, что отправитель не передает более одного короткого пакета за одно RTT. В этом контексте небольшой пакет — это пакет, содержащий меньше байтов, чем максимальный размер сегмента (MSS — Maximum Segment Size) для данного TCP-соединения (например, 536 или 1460 байтов).

Предположим, что отправитель пакетов TCP передал пакет и ждет подтверждения от получателя. Отправитель не передает небольших пакетов, пока не получит подтверждения о доставке всех уже переданных пакетов. К этому времени отправитель мог накопить дополнительные данные. В глобальной сети для соединений с большим RTT алгоритм Нагла предотвращает наличие нескольких небольших пакетов в сети в одно и то же время. В локальных сетях для соединений с небольшим RTT пакеты с подтверждениями почти всегда приходят до того, как у отправителя появляются новые данные для передачи. Ограничение числа небольших пакетов в пути не вносит дополнительной задержки перед передачей следующего пакета. В дополнение к этому алгоритм Нагла не влияет на работу приложений,

осуществляющих передачу по сети больших объемов данных, потому что пересылка больших файлов обычно производится полноразмерными сегментами. Этот подход оказывает наибольшее влияние там, где это нужно — в интерактивных приложениях, использующих соединения с большими RTT.

АЛГОРИТМ НАГЛА И ДОЛГОВРЕМЕННЫЕ СОЕДИНЕНИЯ

Алгоритм Нагла может отрицательно сказаться на работе Web, если передача данных осуществляется небольшими пакетами. Представим себе Web-сервер, который передает HTTP-ответ, записывая заголовок и тело ответа в буфер по отдельности. Операционная система может передать заголовок отдельным небольшим пакетом до того, как сервер запишет тело ответа в выходной буфер. Предположим, что тело ответа тоже меньше максимального размера сегмента. Теоретически, операционная система отложила бы отправку тела ответа, надеясь накопить еще данных. Но если сервер инициирует закрытие соединения, то операционная система не будет ожидать от него дополнительных данных. Тогда операционная система произведет отправку небольшого пакета независимо от того, прибыло ли подтверждение о предыдущем пакете или нет. Если сервер не закрывает соединение, операционная система не посылает второй небольшой пакет до тех пор, пока не придет подтверждение о получении первого пакета. В дополнение к промежутку времени, за который подтверждение ACK может достичь сервера, операционная система на клиентском компьютере может дополнительно задержать отправку пакета подтверждения, как описано ниже в разделе 8.2.3.

Даже если сервер записывает весь ответ в выходной буфер за один шаг, алгоритм Нагла может уменьшить эффективность долговременных соединений [Hei97]. Представим себе Web-сервер, который записывает HTTP-ответ в выходной буфер. В буфер добавится сравнительно большой объем данных (скажем, 8–12 Кбайт) за короткое время. Операционная система будет передавать такой ответ как серию полноразмерных пакетов. В большинстве случаев, в зависимости от размера сообщения, в конце сообщения может быть один небольшой пакет. Например, сообщение длиной 6000 байтов передается по соединению с MSS, равным 1460 байтам. Сообщение займет четыре 1460-байтных пакета и один 160-байтный. Операционная система пошлет последовательность полноразмерных пакетов и отложит отправку небольшого пакета. Отправка последнего пакета будет произведена, если выполнится одно из следующих условий:

- **Запись дополнительных данных в выходной буфер.** Сервер может добавить данные в выходной буфер, что приведет к отправке еще одного полноразмерного пакета, как показано на рис. 8.6. Пакет содержит конец первого сообщения и начало второго. Однако у сервера не всегда найдутся данные для пересылки клиенту. Например, у сервера отсутствуют запросы, ответы на которые нужно передать по данному соединению.

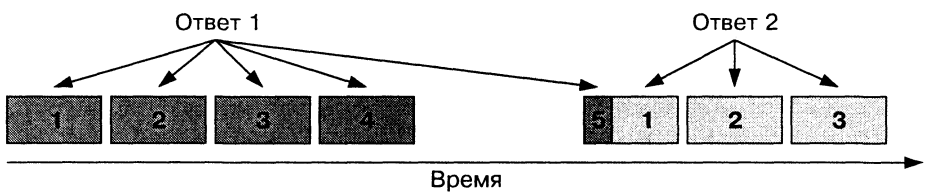


Рис 8.6. Сервер передает полноразмерный пакет, содержащий конец ответа

- **Получение подтверждений на все переданные пакеты.** Могут прийти подтверждения о прибытии всех пакетов, переданных получателю, что позволит операционной системе передать заключительный неполный пакет. На рис. 8.7 показан пример, когда начальный размер скользящего окна на сервере равен длине двух полноразмерных пакетов, сервер получает пакет ACK на каждый второй пакет данных. Ожидание подтверждений обусловливает задержку, равную RTT. Для небольших ответов задержка передачи небольшого заключительного пакета может быть значительной.

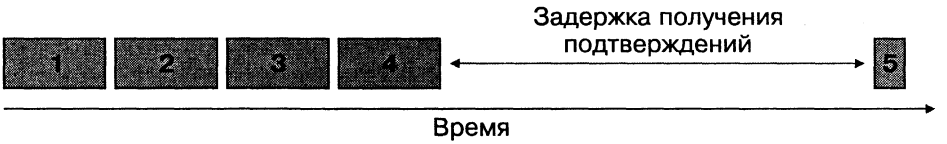


Рис 8.7. Сервер передает небольшой пакет после получения подтверждений

Отключение алгоритма Нагла — простой способ избежать этих потерь производительности. Отключение производится заданием соответствующих параметров во время открытия соединения. Например, в операционной системе UNIX есть функция *setsockopt()* для выполнения различных настроек. Установленный флаг TCP_NODELAY при вызове функции *setsockopt()* отключает алгоритм Нагла. Web-серверы, поддерживающие долговременные соединения, обычно отключают алгоритм Нагла. Web-клиенты также могут отключить алгоритм Нагла, так как в противном случае передача больших запросов, таких как **PUT** и **POST**, может привести к снижению производительности.

НЕДОСТАТКИ ОТКЛЮЧЕНИЯ АЛГОРИТМА НАГЛА

На первый взгляд отключение алгоритма Нагла не будет оказывать никакого отрицательного эффекта на работу Web. На самом деле алгоритм Нагла заметно улучшает работу тогда, когда приложение пишет данные в выходной буфер маленькими порциями. Представим себе Web-сервер, который по отдельности (с помощью вызовов системной функции) записывает каждую строку заголовка HTTP-ответа. Когда алгоритм Нагла отключен, каждый вызов функции *write()* создает отдельный пакет. Это было бы крайне неэффективно. Например, 300-байтовый заголовок ответа состоит из десяти строк. Он может поместиться в один IP-пакет. Помимо этого, в пакет поместится начало тела ответа, если таковое имеется. Запись строк по одной приведет к передаче десяти пакетов вместо одного, если алгоритм Нагла отключен. Эта проблема была характерной для ранних версий Web-сервера NCSA, который вызывал системную функцию для выдачи каждой строки заголовков HTTP. Подобный феномен имеется и в протоколе Network News Transfer Protocol (NNTP) [MSMV99].

Вероятность передачи строк заголовка отдельными пакетами зависит от загрузки сервера. Сильно загруженный сервер менее склонен передавать большое число коротких пакетов. Представим себе загруженный сервер, который генерирует и передает сотни ответов одновременно. Передача IP-пакетов ограничена пропускной способностью подключения сервера к сети. При большой нагрузке операционная система создает очередь данных, передаваемых сервером. Рассмотрим серверный процесс, который делает десять системных вызовов для записи десяти строк заголовка ответа. На загруженном компьютере вторая строка может быть записана в буфер до того, как будет передана первая. Операционная система соединит эти две строки в один

пакет. Таким образом, операционная система может передать все десять строк заголовка в одном пакете. Если бы сервер был менее загружен, то операционная система передала бы все десять строк заголовка в виде отдельных пакетов, каждый из которых содержал бы по 40 байтов заголовков протоколов IP и TCP.

Сервер может избежать передачи небольших пакетов без привлечения алгоритма Нагла, создав сначала весь заголовок и однократно вызвав *write()*, чтобы записать полученный заголовок в выходной буфер. Это гарантирует, что операционная система не будет создавать отдельный пакет для каждой строки заголовка, это также позволит уменьшить число системных вызовов. Сервер может произвести всего один системный вызов, чтобы поместить в буфер заголовок и тело ответа, что будет более подробно описано позже в разделе 8.4.1.

8.2.3. Отложенные подтверждения

Спецификация TCP разработана так, чтобы поддерживать двустороннюю связь между компьютерами. Когда оба компьютера обмениваются данными, подтверждения TCP могут быть присоединены к пакетам данных. Задержка передачи подтверждения увеличивает вероятность его отправки в пакете вместе с данными. Хотя это и эффективно для таких интерактивных приложений, как Telnet или Rlogin, задержка отправки подтверждений замедляет передачу Web-ресурсов. Если включен алгоритм Нагла, отложенные подтверждения могут приостановить загрузку заключительной части Web-страницы. В этом подразделе мы опишем причины применения отложенных подтверждений и изучим их влияние на работу Web.

ПРИЧИНЫ ИСПОЛЬЗОВАНИЯ ОТЛОЖЕННЫХ ПОДТВЕРЖДЕНИЙ

Скорость передачи пакетов TCP отправителем зависит от получения подтверждений от получателя. Скользящее окно управляет передачей пакетов. Когда оно заполнено, отправитель не может передать пакеты, не получив ACK от получателя. Своевременное получение ACK необходимо для поддержания высокой скорости передачи данных. Но посылка ACK требует 40-байтного пакета: 20-байтный заголовок IP и 20-байтный заголовок TCP с уставленным битом ACK. Это очень неэффективно. Рассмотрим отправителя, который посылает получателю 400-байтные пакеты. Посылка 40-байтного пакета с подтверждением на каждый пакет данных увеличит трафик на 10%. Существуют два способа уменьшить трафик подтверждений. Во-первых, получатель не обязан посылать ACK в ответ на каждый полученный пакет данных. Во-вторых, получатель может присоединить подтверждения (флаг ACK и номер подтверждения) к передаваемому им же пакету данных.

Присоединение ACK к передаваемому пакету данных позволит не посылать дополнительных пакетов с подтверждениями. Это очень эффективно, если у отправителя имеются собственные данные, которые нужно отправить. Присоединение невозможно, если у получателя нет собственных данных для передачи. Чтобы увеличить вероятность присоединения, TCP позволяет получателю *отложить* отправку пакета ACK в надежде, что приложение вскоре будет передавать данные. Это очень эффективно для интерактивных приложений. Представим себе, что компьютер A подключился посредством Rlogin к компьютеру B. Предположим, что пользователь ввел один или более символов. Эти символы посылаются по TCP-соединению компьютеру B. Rlogin на компьютере B читает эти символы и генерирует эхо, которое должно отображаться на компьютере A. Эхо помещается в пакет, который будет передан от B к A. В идеальном случае B подтвердит получение начальных символов и пошлет эхо одним пакетом. С другой стороны, B не следует ждать слишком

долго. Подтверждение получения пакета от A очень важно. Ведь компьютер A не сможет отправить еще одну порцию данных, если получение предыдущего пакета не было подтверждено. Следовательно, задержка пакета ACK может замедлить передачу данных от A к B. Чтобы найти компромисс, TCP запускает таймер, который активизирует передачу ACK, даже если нет никаких данных для отправки. Большинство реализаций TCP передают ACK через 200 мс, но некоторые увеличивают задержку до 500 мс. Чтобы избежать задержки пакетов ACK в загруженных соединениях, TCP требует, чтобы подтверждалось прибытие хотя бы каждого второго полноразмерного пакета до завершения таймера задержки подтверждения. Например, если сервер передает длинное сообщение клиенту на высокой скорости, то клиент будет передавать подтверждения после каждого второго пакета данных, даже если клиент не имеет собственных данных для передачи серверу.

ВЗАИМОДЕЙСТВИЕ ОТЛОЖЕННЫХ ACK С ТРАФИКОМ HTTP

Отложенные ACK уменьшают трафик подтверждений двумя путями: присоединяя ACK к передаваемому пакету данных, а также посылая ACK для каждого второго пакета данных. Присоединение ACK к пакетам данных несвойственно для Web-трафика. Обычная передача Web-данных включает в себя короткий HTTP-запрос клиента, после которого следует HTTP-ответ сервера.

Маловероятно, что *оба* компьютера будут передавать данные одновременно, если только сервер не посылает ответ, когда клиент посылает последующие запросы из той же последовательности. Задержка передачи пакета ACK редко позволяет клиенту присоединить его к передаваемому пакету данных. Вместо этого задержка в 200, а то и 500 мс, вызванная таймером задержки ACK, может снизить скорость передачи данных. Эта задержка может быть заметной пользователю. Кроме того, когда скользящее окно сервера заполнено, задержка клиентом передачи подтверждений приостанавливает передачу сервером оставшейся части ответа.

Механизм отложенных ACK может привести к ненужной задержке Web-трафика, обусловленной реализацией Web-сервера. Представим себе сервер, который передает заголовок и тело HTTP-ответа раздельно. Заголовок ответа обычно меньше MSS. Операционная система может передать короткий пакет до того, как сервер запишет хоть какие-нибудь дополнительные данные в выходной буфер. Позже сервер начинает записывать тело ответа в выходной буфер. Если на сервере отключен алгоритм Нагла, операционная система может начать передавать пакеты, содержащие тело ответа. Первый полноразмерный пакет окажется вторым для клиента. Но клиент *не* передаст ACK, так как первый пакет (заголовок HTTP) будет меньше MSS. Клиент должен подождать, пока закончится отсчет таймера задержки подтверждения, прежде чем передаст ACK. В зависимости от размера скользящего окна сервер может оказаться не в состоянии передать дополнительную порцию данных до получения ACK, как показано на рис. 8.8. Чтобы избежать такой ситуации, некоторые операционные системы отключают механизм задержки ACK в начале соединения. Однако это не предотвращает возникновения данной ситуации при передаче последующих ответных сообщений по тому же соединению.

Алгоритм Нагла достаточно сложно взаимодействует с механизмом отложенных подтверждений, когда клиент и сервер поддерживают долговременное соединение [Hei97]; такой же феномен наблюдался на сервере NNTP [MSMV99]. Рассмотрим сервер, который записал HTTP-ответ в выходной буфер. Операционная система разделяет сообщение на сегменты, каждый из которых помещается в пакет IP. Операционная система пытается передавать данные полноразмерными пакетами, хотя последний пакет обычно меньше остальных. Например, сообщение дли-

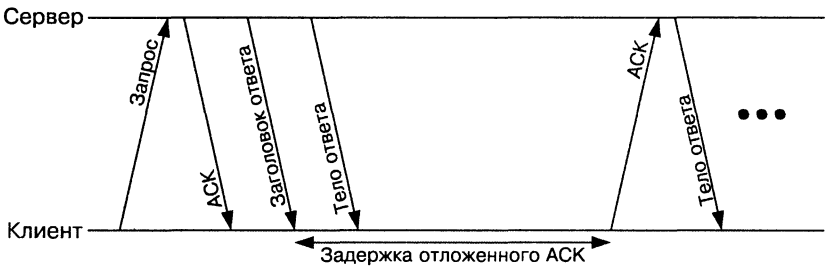


Рис 8.8. Клиент задерживает ACK для первых двух пакетов, переданных сервером

ной 5000 байтов передается в виде трех пакетов длиной 1460 байтов и одного длиной 620 байтов. Согласно механизму отложенных подтверждений, клиент подтверждает каждую пару полноразмерных пакетов. Получив подтверждения на первые два пакета, сервер передает третий полноразмерный пакет. Получив этот пакет, операционная система на клиентском компьютере не передает подтверждения в соответствии с механизмом отложенных подтверждений. Операционная система на сервере, в то же время, не станет посылать последний короткий пакет, пока не получит подтверждения на все предыдущие. Передача данных сервером останавливается, ожидая подтверждения клиента (из-за алгоритма Нагла), а клиент задерживает передачу ACK (из-за механизма отложенных подтверждений), как показано на рис. 8.9. К счастью, отключение алгоритма Нагла на сервере может предотвратить появление такой ситуации на практике.

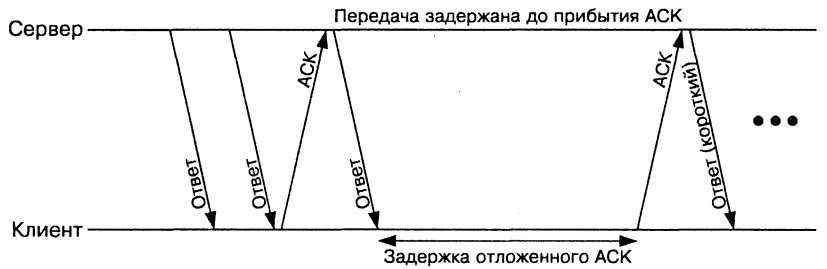


Рис 8.9. Взаимодействие алгоритма Нагла с механизмом отложенных ACK

8.3. Мультиплексирование TCP-соединений

Два предыдущих раздела касались работы отдельно взятого соединения. Но клиент часто устанавливает *несколько* TCP-соединений с сервером. Одновременное наличие нескольких TCP-соединений порождает проблемы с производительностью и равномерным распределением трафика по соединениям. В этом разделе мы изучим причины, по которым клиенты устанавливают множественные соединения, рассмотрим проблемы с производительностью и равномерным распределением трафика, приведем несколько способов решения возможных проблем, связанных с производительностью.

8.3.1. Причины использования параллельных соединений

У Web-клиента есть несколько причин создавать несколько соединений одновременно с одним и тем же сервером:

- **Параллельная загрузка встроенных изображений.** Браузер обычно устанавливает параллельные соединения с сервером, чтобы одновременно загружать несколько встроенных изображений. Первые несколько байтов файла с изображением обычно позволяют получить информацию о размерах изображения. Это позволяет браузеру начать отображать изображение до того, как оно полностью загрузится. Формат GIF поддерживает прогрессивную или чересстрочную развертку, последняя позволяет браузеру отображать все изображение с постепенным улучшением качества в процессе загрузки с сервера. Пользователь может предпочесть увидеть сразу нескольких встроенных изображений с низким качеством, чем одно изображение с высоким качеством, что делает параллельную загрузку встроенных изображений привлекательной альтернативой их последовательной загрузке. Параллельные соединения также используются, когда пользователь открыл несколько окон браузера одновременно.
- **Прокси-сервер, действующий от лица нескольких клиентов.** Прокси-сервер может обрабатывать запросы нескольких клиентов, одновременно запрашивающих ресурсы с одного и того же Web-сервера. Прокси-сервер мог бы послать все запросы по одному TCP-соединению. Это позволило бы сэкономить на создании соединений за счет невозможности распараллелить трафик нескольких клиентов. Предположим, что пользователь A и пользователь B настроили свои браузеры так, чтобы те подсоединялись к одному и тому же прокси-серверу. Теперь предположим, что пользователь B запросил 100-байтную HTML-страницу сразу после того, как пользователь A запросил 100-мегабайтный файл с того же сервера. Если прокси-сервер пошлет оба запроса по одному и тому же (долговременному) соединению, пользователю B придется ждать окончания пересылки 100-мегабайтного файла, чтобы получить свой маленький HTML-файл. Установка двух параллельных соединений с сервером позволит прокси-серверу загружать эти файлы параллельно, что приведет к намного меньшему времени ответа клиенту B.
- **Увеличение общей пропускной способности.** Клиент может добиться большей пропускной способности, устанавливая несколько TCP-соединений с сервером. Предположим, что соединение клиента и сервера характеризуется большим RTT. Даже если сервер и сеть загружены слабо, высокое значение RTT ограничивает пропускную способность TCP-соединения. Пропускная способность ограничена размером входного буфера получателя и размером скользящего окна, и сервер не может передать больше пакетов, чем может поместиться в окно, пока он не получит подтверждения от клиента. Кроме того, начальный размер скользящего окна небольшой из-за фазы медленного старта. Передача ответов по нескольким соединениям может увеличить общую пропускную способность от сервера к клиенту.

Спецификация HTTP рекомендует клиенту открывать не более двух соединений одновременно с одним и тем же сервером, эта рекомендация не открывать более двух соединений для одного клиента распространяется и на прокси-серверы, которые посылают запросы от лица разных клиентов. На практике многие реализации клиентов и прокси-серверов не выполняют эти рекомендации. Кроме того, клиент может открыть несколько параллельных соединений, чтобы получить разные части *одного и того же* ресурса, посылая запросы на диапазоны, как это описано в главе 7 (раздел 7.4.1).

8.3.2. Проблемы с параллельными соединениями

Параллельные соединения увеличивают общую пропускную способность для отдельного клиента, ухудшая работу остальных. Использование параллельных соединений обусловливает несколько проблем:

- **Несправедливое перераспределение трафика по отношению к другим клиентам.** Клиент, который посылает запросы по нескольким соединениям одновременно, достигает большей пропускной способности, отнимая долю пропускной способности у других клиентов. Рассмотрим двух Web клиентов, которые посылают запросы по одному и тому же пути через сеть одному и тому же серверу. Предположим, что клиент А имеет четыре параллельных соединения, когда клиент В имеет всего одно. Клиент А захватит в четыре раза большую пропускную способность, чем клиент В, это приведет к заметно большей скорости работы для клиента А. В то же время клиенту В достанется меньшая пропускная способность и большие задержки. Один из способов для клиента В противодействовать неравномерному распределению ресурсов — также установить больше соединений. Чтобы увеличить пропускную способность, каждый клиент стремится открыть больше соединений, чем другие.
- **Большая загрузка сети и сервера.** В дополнение к неравенству между клиентами, параллельные соединения увеличивают нагрузку на сервер и сеть. Рассмотрим клиента, который устанавливает несколько соединений одновременно, чтобы загрузить набор встроенных в Web-страницу изображений. Это приводит к резкому увеличению трафика, который увеличивает загрузку сети и сервера. Даже если каждое соединение находится в фазе медленного старта, суммарный трафик соединений может быть довольно большим. Как крайний пример возьмем клиента, который открывает 20 соединений с сервером. Клиент пошлет 20 пакетов SYN за короткий период, за которыми вскоре последуют 20 HTTP-запросов. Web-сервер займет большую часть своих ресурсов, пытаясь получить и обработать эти пакеты SYN и HTTP-запросы. В сети суммарная нагрузка, обусловленная большим количеством пакетов SYN, может превысить допустимую пропускную способность, что, в конечном счете, приведет к утере пакетов.
- **Большие задержки для пользователей.** Помимо конкуренции за долю трафика с другими клиентами, параллельные соединения конкурируют между собой. Предположим, что у клиента подключение к Internet имеет пропускную способность в 28,8 Кбит/с (например, телефонный модем). Если клиент установит 20 параллельных соединений, любая передача данных будет проходить со скоростью, не превышающей 1,44 Кбит/с. При загрузке встроенных в Web-страницу изображений, большое число медленных соединений может быть предпочтительнее, чем одно быстрое. В других случаях, пользователь может предпочесть быструю загрузку одного ресурса. Например, пользователь может загрузить различные Web-страницы в разных окнах браузера. Наличие большого числа активных передач данных одновременно может заставить пользователя ждать дольше, чтобы получить хотя бы одну полностью загруженную страницу.

Эти проблемы, связанные с пропускной способностью и неравенством клиентов, можно разрешить следующими способами:

- **Сознательное уменьшение преимуществ параллельных соединений для повышения производительности.** Обеспечение равенства клиентов обычно требует применения алгоритмов, управляющих распределением пропускной спо-

собности сети и системных ресурсов сервера. Например, вместо того, чтобы обрабатывать пакеты в порядке поступления, маршрутизатор может чередовать пакеты от разных отправителей и предназначенные разным получателям. Или же маршрутизатор может подсчитывать, объем трафика получателей и отправителей и «наказывать» слишком агрессивные хосты. Наказание может заключаться в уничтожении части пакетов. Однако на практике такие технологии усложняют устройство маршрутизаторов. Аналогично реализация TCP на Web-сервере может обеспечивать равномерное распределение трафика между клиентами. Web-сервер может обеспечивать равномерное распределение процессорного времени, оперативной памяти и дисковых ресурсов между различными клиентами.

- **Предоставление альтернативы параллельным соединениям.** Долговременные соединения уменьшают выгоду параллельных соединений, как это показано в главе 7 (раздел 7.5). Передача нескольких ответов по одному соединению избавляет от накладных расходов по установлению нескольких соединений. Кроме того, использование долговременного соединения обычно позволяет избежать повторения фазы медленного старта. С другой стороны, долговременные соединения не позволяют загружать несколько встроенных изображений одновременно. Клиент мог бы запрашивать различные изображения по частям по одному долговременному соединению с помощью последовательности запросов на диапазоны. В качестве альтернативы можно было бы внести изменения во взаимодействие между TCP и HTTP, чтобы осуществить чередующуюся загрузку нескольких ресурсов по одному соединению.

Методы равномерного распределения пропускной способности активно изучались по мере эволюции Internet от экспериментальной разработки до сети, предоставляющей широкий набор сервисов [Kes97]. Недавние предложения по изменению схемы распределения трафика конечных пользователей более детально описаны в главе 15 (раздел 15.1).

8.4. Загрузка сервера

Чтение сообщений с запросами, создание ответов и передача данных клиентам требуют существенных ресурсов Web-сервера. Эти операции требуют от сервера выполнить несколько функций, связанных с TCP. В этом разделе мы опишем, как некоторые из этих шагов можно объединить или избежать. Потом мы рассмотрим проблемы, вызванные поддержанием большого числа соединений одновременно. Это относится как к прокси-серверам, так и к Web-серверам.

8.4.1. Совмещение вызовов системных функций

Web-сервер взаимодействует с TCP посредством последовательностей вызовов системных функций. Выполнение нескольких действий с помощью одного системного вызова предоставляет операционной системе дополнительную информацию, которая может увеличить эффективность передачи данных. Рассмотрим Web-сервер под управлением операционной системы UNIX, обрабатывающий запрос GET для статического файла:

1. **Прослушивание запросов на новые соединения.** Сервер прослушивает запросы на новые соединения, а операционная система поддерживает очередь соединений.

2. **Установка нового соединения.** Сервер получает новое соединение из очереди, используя системный вызов *accept()*. Вызвав эту функцию, сервер получает соединение для взаимодействия с клиентом, пославшим запрос. Готовясь передать данные клиенту, сервер может вызвать функцию *setsockopt()*, чтобы отключить алгоритм Нагла.
3. **Создание ответного сообщения.** Сервер делает один или более вызовов функции *read()*, чтобы прочесть запрос клиента. Потом сервер анализирует запрос и идентифицирует запрашиваемый файл. Сервер открывает этот файл с помощью вызова *open()*. Сервер может делать другие системные вызовы для создания заголовка HTTP-ответа (например, для определения текущего времени, размера файла и даты последней его модификации и т.д.).
4. **Передача ответного сообщения.** После создания заголовка HTTP-ответа, сервер использует системный вызов *write()*, чтобы передать его. Затем сервер может начинать передачу файла. Это включает в себя вызов *read()* для чтения файла и *write()* для передачи данных соединению. Если выходной буфер полон, операционная система может не дать серверу сделать еще один вызов *write()*, пока предшествующие данные не будут успешно переданы клиенту.
5. **Закрытие файла и, возможно, соединения.** Записав последний байт ответа, сервер может закрыть файл с помощью *close()* и дополнительно с помощью этой же функцией закрыть соединение. Если сервер использует долговременные соединения, а клиент не передал запроса на закрытие соединения, сервер может оставить соединение открытым для последующей передачи. Наконец, сервер может вызвать *write()*, чтобы внести запись в журнал сервера.

Системный вызов обычно требует переключения контекста между серверным приложением и операционной системой. Выполнение нескольких действий с помощью одного вызова позволяет снизить нагрузку и уменьшить задержки, связанные с переключением контекста. Выполнение нескольких действий с помощью одного системного вызова также оптимизирует работу TCP [NBK99]:

- **Передача содержимого файла напрямую операционной системе.** Чтобы выполнить запрос GET, Web-сервер должен прочитать файл и записать его в выходной буфер. Во многих случаях сервер не анализирует и не изменяет содержимое файла. Чтение файла и затем запись его в выходной буфер приводит к ненужным накладным расходам. Данные копируются из файловой системы в область памяти приложения, а потом из области памяти приложения в выходной буфер операционной системы. Вместо этого данные могут копироваться в буфер напрямую из файловой системы с помощью единственного системного вызова. В большинстве операционных систем имеется системный вызов *sendfile()* или *transmitfile()* для выполнения данной операции.
- **Совместная передача заголовка и тела ответа.** Сервер может передать заголовок и тело ответа с помощью системного вызова *writev()*. Сервер обычно создает заголовок ответа в отдельном буфере. Этот комбинированный системный вызов требует указать буфер (для заголовка ответа) и файл (для тела ответа), а потом передаст содержимое буфера вместе с содержимым файла. Это избавляет от необходимости выполнения отдельных системных вызовов для записи заголовка и тела ответа в один буфер. Это позволяет операционной системе передать начало тела ответа в том же IP-пакете, что и заголовок ответа. В противном случае при выполнении двух системных вызовов операционная система могла бы отправить заголовок, не дождавшись, когда сервер запишет в буфер тело ответа. Объединение этих двух шагов позволяет уменьшить число передаваемых пакетов и избежать потенциально вредных взаимодействий с таймером задержки подтверждений, о чем рассказано в разделе 8.2.3.

- **Отправка ответа и закрытие соединения одним вызовом.** Сервер может закрыть соединение тем же системным вызовом, который использовался для записи HTTP-ответа. Это позволяет серверу записать HTTP-заголовок, передать тело ответа и закрыть соединение одним системным вызовом. Операционная система знает, что сервер хочет закрыть TCP-соединение. Закрытие соединения осуществляется пакетом, в котором установлен бит FIN. Операционная система имеет возможность установить его в последнем пакете данных, как показано на рис. 8.10. Это возможно, так как операционная система уже знает, что сервер хочет закрыть соединение. Иначе операционная система передала бы последний пакет данных до того, как сервер изъявит желание закрыть соединение. В случае длинных ответных сообщений операционная система обычно не успевает отправить весь ответ до того, как сервер вызовет функцию для закрытия соединения. Но короткие ответы чаще требуют отдельных пакетов FIN. Системный вызов для передачи ответа и закрытия соединения гарантирует, что операционная система установит бит FIN у последнего пакета данных. Это уменьшит накладные расходы, сократив число передаваемых пакетов на один.

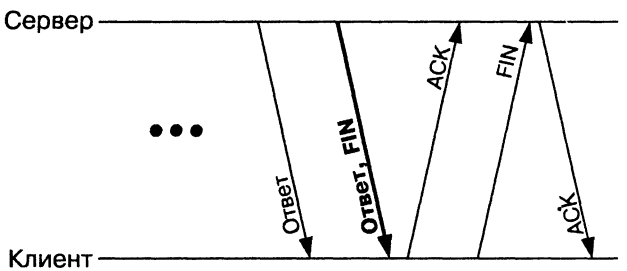


Рис 8.10. Сервер устанавливает бит FIN у последнего пакета данных

Расширение операционной системы новыми системными вызовами может улучшить производительность Web-серверов. Теоретически, Web-сервер мог бы быть встроен в операционную систему. Внедрение приложения в операционную систему является обычной практикой для встроенных операционных систем, предназначенных для выполнения единственной задачи. Напротив, в операционные системы общего назначения дополнительные приложения обычно не включаются. Выполнение требующих значительных ресурсов функций на уровне операционной системы усложняет распределение процессорных и дисковых ресурсов, а также оперативной памяти между различными задачами. В результате операционные системы обычно имеют тщательно отобранный набор системных вызовов. Для нужд Web-серверов можно было бы ввести нескольких новых системных вызовов или изменить реализацию имеющихся. Но нужды отдельных приложений не оправдывают существенных изменений набора функций, предоставляемых операционной системой.

8.4.2. Управление соединениями

Эффективность Web-сервера зависит от числа одновременно открытых соединений. Web-серверы контролируют число открытых соединений двумя основными способами: отвергая запросы на новые соединения и закрывая неактивные соединения.

УПРАВЛЕНИЕ ОДНОВРЕМЕННО ОТКРЫТЫМИ СОЕДИНЕНИЯМИ

Каждое TCP-соединение использует оперативную память на сервере для хранения информации о своем состоянии, входных и выходных буферах. Например, каждое соединение имеет блок памяти, в котором хранятся размер скользящего окна и оценка RTT. Требования к объему памяти могут быть достаточно высокими на сервере, поддерживающем одновременно большое число соединений. Неактивные соединения в состоянии TIME_WAIT также потребляют ресурсы сервера, как это описано в разделе 8.1.3. Кроме того, сервер создает отдельный процесс для каждого открытого соединения, что приводит к дополнительному расходу памяти. Соединения TCP и данные об их состояниях используют память, которую можно было бы использовать для других целей, например для кэширования Web-ресурсов и их метаданных на сервере, как об этом рассказано в главе 4 (раздел 4.3). Кэширование ресурсов на сервере становится менее эффективным, когда сервер работает с очень большим числом соединений одновременно. Кэширование избавляет сервер от накладных расходов, связанных с загрузкой ресурсов с диска и регенерацией метаданных.

Накладные расходы по обработке полученного пакета возрастают с увеличением числа открытых соединений. Когда пакет прибывает, операционная система должна проверить IP-адреса и номера портов, чтобы демультиплексировать пакет и определить, какому соединению он принадлежит. В некоторых старых операционных системах демультиплексирование пакетов требовало линейного сканирования списка соединений; в качестве усовершенствования соединения в состоянии TIME_WAIT могли быть перенесены в конец списка в связи с малой вероятностью получения ими пакетов. Для дополнительного ускорения поиска современные операционные системы используют хэш-таблицу, которая отображает список IP-адресов и номеров портов на список соединений. Но время доступа и требования к памяти по-прежнему увеличиваются при росте числа открытых соединений. Помимо этого в случае управления по событиям, необходим отдельный процесс для прослушивания пакетов, прибывающих по всем открытым соединениям (например, с помощью системного вызова *select()*). Накладные расходы на вызов системной функции увеличиваются с ростом числа одновременно открытых соединений, даже если эти соединения неактивны [BM98].

Оптимальное число соединений зависит от архитектуры сервера и особенностей HTTP-трафика. Создание и передача ответов потребляет дисковые, процессорные, сетевые ресурсы и ресурсы оперативной памяти сервера. Распределение этих ресурсов между большим числом одновременных запросов приводит к большей задержке в создании и передаче ответных сообщений. Как крайний пример, представим себе Web-сервер, который запускает CGI-сценарий, который требует двух секунд для создания каждого ответа. Если позволить процессору переключаться для выполнения сценариев, обслуживающих десять запросов, то это приведет к 20-секундной задержке при создании каждого сообщения. Большая и заметная пользователю задержка может вынудить пользователя покинуть сайт, отменить запрос и передать его заново. Низкая производительность раздражает пользователей и расходует ресурсы сервера впустую. В худшем случае такой сервер будет перегружен, не выполняя никакой полезной работы. Число TCP-соединений должно тщательно контролироваться, чтобы избежать такой ситуации.

Несмотря на недостатки открытия большого числа соединений, серверу не следует ограничивать их число без надобности. Ограничение числа одновременно открытых соединений может заблокировать или задержать запросы на установку соединений для новых клиентов. Когда сервер достигает максимального числа под-

держиваемых соединений, новые пакеты SYN помещаются в очередь. Запросы задерживаются, пока не закроется одно из открытых соединений. Когда очередь заполнена, сервер отбрасывает пакеты SYN, отвергая тем самым запросы на новые соединения. Чрезмерно заниженный лимит допустимого числа соединений также приводит к неоптимальному использованию ресурсов сервера и сети. Серверу требуется определенное число активных TCP-соединений, чтобы использовать доступную пропускную способность сети, потому что скорость передачи сообщений для многих соединений ограничивается величиной RTT и пропускной способностью пути к клиенту. Сервер может посылать ответы большому числу клиентов с низкоскоростными подключениями, не перегружая своего высокоскоростного подключения к Internet.

СТРАТЕГИИ ЗАКРЫТИЯ ДОЛГОВРЕМЕННЫХ СОЕДИНЕНИЙ

В рамках управления числом соединений сервер HTTP/1.1 должен решать, когда закрывать долговременное соединение. Сохранение долговременного соединения позволяет серверу быстрее реагировать на последующие запросы от данного клиента и избежать накладных расходов на установку и закрытие соединения. Поддержание соединения между передачами данных увеличивает общее число одновременно открытых соединений на сервере. Сервер может применять широкий набор стратегий закрытия долговременных соединений. Самая простая стратегия — закрывать неактивные соединения после истечения таймаута. Например, соединение может быть закрыто, если в течение 15 секунд не пришло запроса. Если соединение было неактивно в течение нескольких секунд, то маловероятно получение запроса в ближайшем будущем.

Но даже относительно небольшой таймаут может держать соединение открытым слишком долго, так как многие клиенты запрашивают единственный ресурс с сервера. Чтобы справиться с такой ситуацией, сервер может применять гибридную методику [PM95]. Сервер может использовать небольшой таймаут после первого запроса и увеличивать его, если клиент сделал дополнительные запросы. В дополнение к применению таймаута сервер может ограничить число запросов, обрабатываемых соединением. В противном случае у клиента будет повод периодически отправлять запросы только для того, чтобы поддерживать долговременное соединение открытым. Например, каждые десять секунд браузер может создавать HTTP-запрос, даже если на самом деле ничего не требуется, просто для поддержания соединения в открытом состоянии. Ограничивая число запросов, которое можно сделать через соединение, сервер делает такое поведение невыгодным. С другой стороны, такая методика заставляет корректно ведущего себя клиента нести накладные расходы по созданию нового соединения, когда исчерпывается лимит запросов.

Методики, основанные на таймаутах и числе запросов, относительно легко реализовать, так как они рассматривают каждое TCP-соединение в отдельности. На сервере, создающем отдельный процесс для каждого соединения, такая стратегия имеет преимущество, так как она не требует координации процессов. Сервер может применить более сложные стратегии, совместно управляя всеми открытыми соединениями. Сервер в действительности имеет кэш соединений, и решение, какое соединение закрыть, сведется к задаче о замещении элементов кэша. Например, сервер мог бы закрыть соединение, которое было неактивным дольше всех остальных. Как и в случае стратегий с таймаутом, эта эвристическая стратегия предполагает, что неактивное соединение вряд ли получит запрос в ближайшем будущем. В качестве альтернативы, сервер может принимать решение в зависимости от относительной важности пользователя. Эта стратегия может быть разумной для коммерческого сайта, кото-

рый хочет предоставить наилучшую производительность покупателям, которые тратят на сайте больше денег. Сервер может также постараться избежать закрытия соединения с далеко расположенными клиентами, которые будут испытывать большую задержку при установлении нового соединения с сервером.

8.5. Резюме

Несмотря на преимущества распределения коммуникационных функций по слоям, взаимодействие этих слоев может негативно сказаться на работе приложения. Таймеры, управляющие ключевыми операциями в TCP, оказывают влияние на производительность HTTP. Средства TCP, которые были разработаны, когда Telnet и Rlogin были доминирующими приложениями, могут вредно влиять на передачу данных во Всемирной паутине. Детали реализации компонентов программного обеспечения Web могут смягчить или наоборот усилить эти эффекты. В отличие от более ранних Internet-приложений, Web-клиент обычно использует несколько соединений транспортного уровня одновременно, чтобы параллельно загружать различные Web-ресурсы. Параллельные соединения увеличивают загрузку сервера и сети и приводят к неравномерному перераспределению пропускной способности между пользователями. Сильно загруженные Web-серверы и прокси-серверы поддерживают много TCP-соединений от лица различных клиентов. Эффективность Web-серверов и прокси-серверов зависит от применения эффективных стратегий управления количеством одновременно открытых соединений.

Часть IV
Измерение и описание Web-трафика

9

Измерение Web-трафика

Измерение и анализ Web-трафика играет важную роль в разработке Web-сайтов, в администрировании Web- и прокси-серверов и в эксплуатации IP-сетей. Кроме того, регистрация передач данных является средством изучения ключевых характеристик трафика и испытания новых методов увеличения производительности Web. Измерение Web-трафика состоит из трех основных шагов: наблюдение за передачей данных в каком-либо узле, запись результатов измерений в некотором формате и предварительная обработка записей для последующего анализа. В этой главе мы представляем обзор способов измерения Web-трафика, начиная с мотивации проведения таких измерений. Затем мы рассмотрим пять основных способов наблюдения за Web-трафиком: ведение журнала на клиенте, сервере и прокси-сервере; мониторинг пакетов и активные измерения. Каждая технология имеет преимущества и ограничения, которые влияют на тип регистрируемой информации.

Хотя форматы записи результатов измерений Web-трафика не были стандартизированы, большинство реализаций прокси-серверов и Web-серверов следуют неформальным стандартам по форматам протоколирования. Мы представляем обзор стандартов де-факто ведения журналов Web-серверов и прокси-серверов: Common Log Format (CLF — Общий формат протоколирования) и Extended Common Log Format (ECLF — Расширенный общий формат протоколирования). Так как отсутствуют формальные стандарты регистрации трафика клиентов, мониторинга пакетов и активных измерений, форматы регистрации в различных реализациях отличаются. Далее мы рассмотрим, как размер и разнообразие результатов Web-измерений обусловливают проблемы при хранении и анализе данных. Предварительная обработка результатов измерений предоставляет возможность исключить ошибочные записи, ненужные поля и привести данные к форме, удобной для детального анализа. Выполнение этих предварительных действий упрощает программы анализа и облегчает использование различных инструментов для хранения, анализа и отображения результатов измерений.

Перечисленные способы измерения трафика имеют определенные ограничения, которые затрудняют оценку ряда основных характеристик сети. Часть информации может оказаться недоступной в журнале в зависимости от того, где собираются данные и в какие поля записываются. Мы опишем различные способы косвенного получения отсутствующей информации на основе полей, которые были записаны в журнал. Затем мы рассмотрим четыре исследовательских проекта, которые демонстрируют приложение методов измерения трафика и получения специфических метрик производительности. Эти четыре проекта изучают особенности рабочей нагрузки Web-серверов, изменчивость трафика, создаваемого Web-клиентами, преимущества кэширования Web-ресурсов на прокси-серверах и частоту изменений Web-ресурсов. Мы не будем подробно обсуждать результаты анализа наблюдений, сосредоточив внимание на методах сбора, записи, предварительной обработки и анализа измерений трафика.

9.1. Мотивация измерений Web-трафика

Измерение Web-трафика сыграло существенную роль в описании характера поведения пользователей, производительности программного обеспечения и сетевой инфраструктуры. В этом разделе мы расскажем, какую пользу приносит сбор и анализ данных с точки зрения разработчиков Web-сайтов, администраторов Web-серверов, организаций, предоставляющих сетевые услуги. Некоторых из перечисленных целей трудно достичь из-за ограничений в данных, получаемых при измерении трафика, как это будет описано позже в данной главе.

9.1.1. Мотивация для разработчиков Web-сайтов

Разработчики Web-сайтов могут получить ценную информацию при изучении пользовательских предпочтений. Представим себе сайт электронной торговли, где продаются книги. Статистика посещений сайта влияет на прибыль от рекламы и баннеров, размещенных на Web-сайте. Кроме того, анализ статистики посещений отдельных страниц пользователями может подсказать, как нужно обновлять информацию на сайте. Предположим, что большое число пользователей посещают главную страницу, а потом переходят по определенной последовательности гиперссылок, чтобы найти популярные книги в мягкой обложке. Это дает повод к модификации главной страницы для того, чтобы поместить на ней гиперссылку на страницу со списком наиболее популярных книг. Знание того, как долго пользователи находятся на сайте, и того, сколько страниц они просматривают, также полезно. Если многие пользователи покидают сайт, просмотрев всего одну-две страницы, то может потребоваться реорганизация сайта или публикация более интересного материала. Пользователи, быстро перескакивающие со страницы на страницу, возможно не находят нужную им информацию. Разработчик Web-сайта может решить эту проблему, предоставив пользователям возможность непосредственного выбора некоторого раздела сайта, относящегося к определенной тематике.

Разработчику Web-сайта может оказаться важным знать, откуда пользователи заходят на сайт. Например, если 25% посетителей приходят на сайт электронной торговли с определенного новостного сайта, то имеет смысл разместить на этом сайте рекламу. Измерения могут выявить проблемы, связанные с производительностью сайта. Предположим, что загрузка главной страницы занимает в среднем восемь секунд при соединении со скоростью 56,6 Кбит/с. Эта информация может побудить к созданию более простой главной страницы с меньшим числом встроенных изображений. Измерение задержки при загрузке страниц пользователем может также быть полезной при оценке производительности сервера, на котором размещен сайт. Высокая задержка или низкая пропускная способность могут побудить создателя сайта поменять провайдера, на сервере которого размещается сайт. Большое число запросов, приходящих, скажем, из Италии, могут подсказать разработчику Web-сайта разместить зеркало сайта в Европе и, возможно, наполнить его новым содержимым, предназначенным для европейских пользователей.

9.1.2. Мотивация для компаний, занимающихся хостингом

Измерения также играют важную роль при обслуживании Web-серверов. Рассмотрим компанию, занимающуюся хостингом, т.е. компанию, на Web-серверах которой размещено большое число Web-сайтов. Протоколируя доступ к Web-сайтам, компания может провести статистический анализ количества сообщений-ответов,

количества полученных/отправленных каждым сайтом байтов. Эта информация необходима для подсчета арендных платежей и для распределения системных ресурсов между сайтами. Например, сайт, который получает большое число запросов, может быть реплицирован на несколько компьютеров. Сайт, который получает большую часть запросов днем (например, деловой сайт), может быть размещен на том же компьютере, что и сайт, получающий большую часть своих запросов ночью (например, развлекательный сайт). Кроме того, измерения могут использоваться для обнаружения возможных проблем, связанных с производительностью. Узнав, что пользователи испытывают большие задержки при доступе к Web-сайтам, компания может увеличить пропускную способность канала, приобрести дополнительный канал и установить дополнительные компьютеры для Web-серверов.

Измерения могут помочь в настройке параметров серверов. Web-серверы имеют множество настраиваемых параметров, которые влияют на распределение ресурсов, как это было рассмотрено в главе 4 (раздел 4.6). Например, целесообразность разрешения долговременных соединений на сервере зависит от того, как часто отдельный клиент посылает несколько запросов за короткий промежуток времени. Если долговременные соединения разрешены, таймаут закрытия неактивного соединения можно настроить в зависимости от среднестатистического времени между последующими запросами. Измерения трафика также полезны для сравнения программного обеспечения Web-серверов различных производителей. Они позволяют осуществить сравнительную оценку работы серверов. Подобным образом измерения могут помочь в выравнивании нагрузки исходных Web-серверов. Например, знание пользовательских предпочтений поможет предположить, какая часть запросов может быть удовлетворена сервером-заместителем (прокси-сервером, подключенным непосредственно к Web-серверу) без обращения к исходному серверу.

9.1.3. Мотивация для сетевых операторов

Компании-операторы, поддерживающие работу сетей, также могут получить пользу от измерений Web-трафика. Например, компания, имеющая локальную сеть, может получить выигрыш в производительности, установив кэширующий прокси-сервер. Измерения могут быть использованы для определения доли запросов, которые удовлетворяются из кэша. Подобным образом Internet-провайдеры могут собрать данные о том, как изменится общая пропускная способность, если кэширующий прокси-сервер будет поставлен в том или ином месте сети. Провайдеры могут использовать результаты измерений для сравнительной оценки программного обеспечения прокси-серверов различных производителей. Они могут наблюдать за трафиком от пользователей и к пользователям с разными физическими подключениями к сети, например, с помощью низкоскоростных телефонных модемов и высокоскоростных кабельных модемов. Это поможет провайдерам замерить, насколько высокоскоростное соединение уменьшает задержку и как низкая задержка, в свою очередь, влияет на поведение пользователей. Основываясь на этих наблюдениях, Internet-провайдер может решить, насколько ему нужно увеличить общую пропускную способность, чтобы суметь обслужить увеличивающееся число пользователей с высокоскоростными соединениями.

Измерения Web-трафика позволяют провайдеру установить, какие сайты чаще всего посещают его пользователи, и отследить задержку, испытываемую пользователями при посещении этих сайтов. Низкая скорость работы с популярными сайтами может побудить провайдера увеличить пропускную способность сети или изменить маршрутизацию трафика от Web-серверов и к ним. Если пользователи загружают большие объемы данных с какого-то конкретного Web-сайта, то провайдер

может договориться о проведении прямого канала к компании, на сервере которой установлен этот сайт. Вдобавок, наблюдение за изменениями скорости доступа могут предупредить оператора о потенциальных проблемах в сети, таких как переставший работать канал или возросший трафик от каких-либо клиентов. Результаты измерения трафика можно использовать, чтобы оценить последствия изменений конфигурации сети. Так как именно Web-трафик дает большую часть общего трафика в Internet, то измерение Web-трафика полезно в том числе и для испытания в сложных условиях сетевого оборудования, такого как новые маршрутизаторы, и для оценки нагрузки на DNS-серверы.

9.1.4. Мотивация для исследователей и разработчиков

Измерения Web-трафика бесценны и для научного сообщества, потому что помогают определить характеристики Web-протоколов и компонентов. Web-трафик был активной областью исследований с самых ранних дней существования Web. Исследование трафика сыграло заметную роль в эволюции HTTP. Измерения использовались для оценки реального распространения и эффективности многих новых особенностей HTTP/1.1. Например, изучение результатов измерений Web-трафика повлияло на решение использовать в HHTP/1.1 долговременные соединения [PM95]. Измерения широко использовались для проверки новых методов и механизмов работы Web-компонентов. Измерения Web-трафика сыграли ключевую роль в проверке технологий кэширования, включая алгоритмы замены элементов кэша, проверки актуальности элементов и опережающего чтения.

Помимо исследований Web, анализ Web-трафика заметно повлиял на технологии Internet в целом. Сбор и анализ сведений о Web-трафике привел к намного более глубокому пониманию динамики трафика в Internet. Этот анализ привел к разработке более реалистичных моделей трафика, которые могут использоваться для тестирования новых сетевых протоколов, как это будет обсуждаться в главе 10. Измерения Web-трафика использовались для изучения взаимодействия между HTTP и TCP, а также влияния Web-трафика на работу Internet. Измерения продолжают играть решающую роль для разработчиков, пытающихся понять динамику Web-трафика и его влияние на общую работоспособность сетей. Эти задачи приобретают большую важность в связи с тем, что Internet становится существенной частью мировой экономики.

9.2. Технологии измерений

Web-измерения осуществляются многими способами, от выбора этих способов сильно зависит вид получаемой информации. Программные продукты, работающие в Web, такие как браузеры, серверы и прокси-серверы могут вести журналы во время обработки запросов. Данные о Web-трафике можно собирать, например, осуществляя пассивный мониторинг на маршрутизаторах. Кроме того, специальный Web-клиент может инициировать HTTP-запросы для оценки производительности сети. В этом разделе мы рассмотрим эффективность и ограничения каждого из подходов к измерению Web-трафика.

9.2.1. Протоколирование на Web-серверах

Web-сервер обычно ведет журнал во время обработки запросов клиентов. Каждая запись журнала соответствует HTTP-запросу, обрабатываемому сервером.

В запись включается информация о запрашивающем клиенте, время запроса и информация о запросе и ответе. Разработчики Web-серверов обычно следуют неформальным стандартам форматов журналов, что более подробно описано в разделе 9.3. Сходство в форматах журналов и доминирование небольшого числа типов Web-серверов облегчили создание многочисленных инструментальных средств для анализа журналов. Статистический анализ данных журналов серверов дает ценную информацию для разработчиков и администраторов Web-сайтов. Кроме того, серверные журналы предоставили исследователям уникальную возможность изучить предпочтения групп клиентов по выбору ресурсов. Журналы серверов использовались в большинстве исследовательских работ, посвященных описанию HTTP-трафика и новым Web-технологиям.

Большинство Web-серверов осуществляют по умолчанию протоколирование, но на практике журналы серверов не предоставляют очень подробной информации. В связи с тем, что запись заголовка запроса приведет к значительным накладным расходам, большинство серверов записывают только метод запроса, запрашиваемый URI и код ответа. Кроме того, серверный журнал не предоставляет точной информации о времени. Например, значение времени в записи журнала сервера может быть временем получения запроса, временем начала или конца обработки запроса или временем отправки ответа. Журнал сервера редко включает в себя все эти моменты времени. Вдобавок, значения времени могут быть записаны с низкой разрешающей способностью, например, с точностью до секунды. Низкая точность значений времени затрудняет определение того, как долго сервер обрабатывает одиночный запрос, или сколько времени проходит между последующими запросами. Помимо того, записи в журнале идут не в порядке получения HTTP-запросов, а лишь в порядке регистрации их сервером.

На первый взгляд журнал Web-сервера мог бы использоваться для анализа пользовательских предпочтений и относительной популярности ресурсов на Web-сайте. На самом деле запросы, которые удовлетворяются из кэша браузера или прокси-сервера, не появляются в журнале сервера. Сервер не знает, сколько запросов удовлетворяются из кэша. Между прочим, популярные ресурсы чаще всего удовлетворяются из кэша. Чтобы гарантировать, что все запросы регистрируются в журнале, сервер может быть настроен так, чтобы ограничить кэширование ответов клиентами и прокси-серверами. Например, каждое ответное сообщение с сервера может включать заголовок, запрещающий кэширование или требующий проверки актуальности кэшированных ответов. Правда, такие методики увеличивают трафик в сети, уменьшая эффективность кэширования, как это описано далее в главе 11 (раздел 11.11.1).

Каждая запись в журнале сервера включает информацию о клиенте, инициировавшем запрос. Обычно сервер записывает IP-адрес клиента или его доменное имя. Информация об агенте пользователя важна для изучения вопроса о предпочтениях в выборе пользователями браузеров. В то же время соотнесение запросов с реальными пользователями осложнено по множеству причин:

- **Прокси-серверы.** Запрос может прийти от прокси-сервера вместо агента пользователя. Один и тот же прокси-сервер может генерировать запросы от лица разных пользователей, что делает сложным определение пользователя, пославшего тот или иной запрос.
- **Разные пользователи на одной машине.** Многие организации имеют компьютеры с отдельными профилями пользователей. IP-адрес клиента перестает быть уникальным идентификатором, когда агенты пользователей работают от лица разных пользователей на одном и том же компьютере.

- **Динамическое выделение IP-адресов.** IP-адрес, связанный с определенным компьютером, может меняться. Многие пользователи подключаются к Internet с помощью модема. Провайдеры обычно динамически выделяют клиенту IP-адрес из пула свободных адресов.

Запросы от одного и того же пользователя могут приходить с различных IP-адресов, а различные пользователи могут делать запросы с одного IP-адреса. Чтобы снять эти двусмысленности, некоторые сайты для отслеживания пользователей используют cookies, которые позволяют более точно идентифицировать пользователя.

Хотя серверные журналы сыграли ключевую роль в изучении Web, данные журнала одного сервера не могут быть репрезентативными для других серверов. Web-сайты сильно отличаются по своей популярности и функциональности. Web-сайт высшего учебного заведения существенно отличается от портала или сайта электронной коммерции. На сайтах могут быть размещены различные типы ресурсов, они имеют различную пользовательскую аудиторию. Кроме того, исследователи обычно не имеют доступа к журналам серверов коммерческих Web-сайтов. Компании могут считать, что журналы содержат конфиденциальную информацию о покупателях или другую информацию, которая может оказаться полезной конкурентам. В результате многие проекты по исследованию Web основываются на журналах университетов и некоммерческих учреждений. К сожалению, трудно перенести результаты этих исследований на коммерческие Web-сайты.

9.2.2. Протоколирование на прокси-серверах

Прокси-серверы, также как и Web-серверы, обычно ведут журналы во время своей работы. В отличие от серверного журнала, журнал прокси-сервера производит запись запросов к большому числу Web-сайтов, особенно если прокси-сервер размещен близко к запрашивающим ресурсы клиентам. Например, прокси-сервер может пропускать через себя все HTTP-запросы от всех клиентов одной организации или провайдера. Журнал прокси-сервера предоставляет полезную информацию о пользовательских предпочтениях клиентов, когда они посещают различные сайты. Прокси-сервер зачастую получает более подробную информацию о запрашивающих клиентах, чем исходный Web-сервер. Первый прокси-сервер в цепочке со стороны клиента может различать запросы разных пользователей. Последующие прокси-серверы не смогут различать пользователей, если только HTTP-запросы не включают cookies. Различение пользователей важно для анализа предпочтений пользователей, обращающихся к конкретным сайтам.

Журнал прокси-сервера включает запросы, которые удовлетворяются из кэша прокси-сервера. Исходный сервер, которому предназначаются эти запросы, не получает их. Поэтому журналы прокси-серверов можно использовать для определения относительной популярности различных Web-сайтов и эффективности методик кэширования. Тем не менее, имеется зависимость между контингентами клиентов и серверов в журнале прокси-сервера и расположением прокси-сервера. Прокси-сервер, расположенный близко к клиентам, обрабатывает запросы сравнительно малого числа клиентов к достаточно широкому кругу исходных серверов. В качестве крайнего примера можно привести прокси-сервер, обрабатывающий запросы только одного клиента. Прокси-сервер, расположенный близко к исходным серверам, обрабатывает запросы от большого числа клиентов к небольшому числу серверов. Например, прокси-сервер, являющийся заместителем основного Web-сервера, может обслуживать всего лишь на один Web-сайт.

Некоторые недостатки журналов Web-серверов свойственны и журналам прокси-серверов. Прокси-сервер не видит запросов, удовлетворенных из кэша браузера или другого прокси-сервера, расположенного ближе к клиенту. Поэтому прокси-сервер «не видит» большую часть запросов, направленных какому-то конкретному серверу. Это затрудняет определение некоторых свойств Web, таких как частота запросов на популярные ресурсы. Кроме того, данные о клиентах, полученные на каком-либо одном прокси-сервере, не могут быть распространены всех клиентов Web. Прокси-сервер может иметь достаточно однородный набор клиентов с точки зрения географического положения, пользовательских предпочтений и скорости доступа в Internet. Рассмотрим прокси-сервер, расположенный рядом с провайдером, который обслуживают жилой сектор в Бостоне, штат Массачусетс. Пользовательские предпочтения, наблюдаемые на этом прокси-сервере, скорее всего, будут существенно отличаться от данных, полученных с прокси-сервера, компании в Париже, Франция. Наконец, как и в случае с журналами серверов, многие коммерческие организации не предоставляют исследователям доступа к журналам своих прокси-серверов.

9.2.3. Протоколирование на клиентах

Анализ журналов браузеров потенциально может предоставить детальный обзор пользовательских предпочтений. Браузер может записывать значения времени для разных шагов процесса обмена запросами-ответами. Браузер может регистрировать те действия пользователя, которые не приводят к HTTP-запросам, включая запросы, удовлетворяемые из кэша браузера, равно как и события, связанные с мышью и клавиатурой. Браузер знает, когда пользователь прерывает запрос, нажав кнопку Stop. Отмененный запрос может быть не зарегистрирован на исходном сервере в зависимости от того, какая часть запроса была обработана, когда пользователь отменил его и когда производится запись в журнал. По сравнению с прокси-серверами и Web-серверами браузер работает со сравнительно малым числом запросов одновременно. Запись в журнал полного набора заголовков запросов и ответов не слишком загружает браузер.

В отличие от журналов серверов и прокси-серверов не существует стандарта де-факто формата журналов браузеров. Общераспространенные браузеры не создают журналов по умолчанию; таким образом сбор данных на клиентах требует модификации программного кода браузера и распространения модифицированного браузера в группе пользователей. Следует отметить, что исходный код обычно недоступен для большинства версий общераспространенных браузеров. Кроме того, получение реалистичных результатов при изучении пользовательских предпочтений требует, чтобы большое число пользователей, участвующих в исследовании, использовали модифицированный браузер. Эти пользователи могут и не отражать предпочтений *всех* пользователей Web. Было разработано несколько методов, чтобы собирать подробные записи о пользовательских предпочтениях, не изменяя исходного кода браузера. Например, протоколирование может вестись на прокси-сервере, работающем прямо на компьютере пользователя, ведь браузер может быть настроен так, чтобы направлять все свои запросы прокси-серверу.

Обычный прокси-сервер не узнает о запросах, удовлетворенных из кэша браузера. Чтобы разрешить эту проблему, браузер пользователя может быть настроен таким образом, чтобы не кэшировать ресурсы. Чтобы не вынуждать пользователя менять настройки кэша в браузере, прокси-сервер может запретить браузеру обращаться к кэшу при запросах, модифицируя HTTP-ответы. Прокси-сервер может

вставлять строки в заголовок HTTP-ответа (например, **Expires** или **Cache-Control: no-cache**), которые дадут браузеру указание не помещать ответ в кэш. Это гарантирует, что прокси-сервер «увидит» каждый запрос. С другой стороны, принуждение браузера генерировать HTTP-запрос на каждый ресурс, может негативно сказаться на производительности, что, в свою очередь, может повлиять на поведение пользователя. Такой подход может привести к существенному увеличению нагрузки на прокси-сервер и сеть.

9.2.4. Мониторинг пакетов

Протоколирование на клиенте, сервере и прокси-сервере приводит к дополнительной нагрузке на программное обеспечение Web. Кроме того, журналы, которые ведутся на уровне приложений, не несут или почти не несут в себе информации об активности на уровне протоколов TCP и IP. Такой внутрисетевой мониторинг трафика предоставляет перспективный способ сбора подробной информации, не влияющий на работу высокоуровневых Web-приложений. Но для сбора информации этим методом требуется перехватывать каждый IP-пакет, который идет по участку сети или через маршрутизатор. Некоторые сетевые технологии канального уровня более удобны для мониторинга пакетов, чем другие. Многие локальные сети в настоящее время реализованы на основе Ethernet. В этом случае монитор пакетов может легко перехватить любой пакет в сети. Сбор же информации о пакетах в соединениях «точка-точка», таких как оптоволоконный кабель, требует ответвления от кабеля или наличия поддержки на маршрутизаторе, чтобы обеспечить монитору возможность получения копии любого пакета.

Монитор пакетов может предоставить детальную информацию об активности на уровнях HTTP, TCP и IP. Монитор может записывать временную метку из любого IP-пакета, проходящего по участку сети. Заметим, что Web-серверы и прокси-серверы записывают только одно значение времени для каждого запроса/ответа. Подробная статистика по времени пакетов с запросами и ответами бесценна для анализа задержек, пропускной способности и потерь при пересылке данных. С эффективной аппаратной поддержкой монитор пакетов может генерировать временные метки с разрешающей способностью до миллисекунды или менее, разрешающая способность большинства журналов Web-серверов и прокси-серверов составляет секунды. В связи с тем, что мониторинг не вмешивается в работу Web, пакетный монитор может записывать очень подробную информацию, включая полные HTTP-заголовки запросов и ответов. Более того, монитор может записать сообщение целиком. Кроме того, мониторинг пакетов предоставляет информацию транспортного уровня, такую как время и способ разрыва TCP-соединения. Это помогает исследовать отмененные HTTP-запросы, которые сложно, если вообще возможно анализировать, пользуясь журналами Web-серверов и прокси-серверов. Мониторинг пакетов также предоставляет подробную информацию о пропускной способности участка сети.

В то же время мониторинг пакетов не является панацеей. Как и прокси-сервер, монитор не может перехватить запросы, удовлетворенные из кэша браузера или HTTP-сообщения, которые были зашифрованы с использованием SSL. Помимо этого осуществлять мониторинг пакетов становится все труднее с увеличением пропускной способности сети. Монитор должен успевать перехватывать пакеты, производить необходимую обработку и сохранять результаты со скоростью прибытия пакетов. Ограничения, налагаемые процессором, памятью и скоростью работы с файловой системой, затрудняют мониторинг высокоскоростных сегментов сетей.

За короткое время по участку сети могут пройти пакеты, принадлежащие большому числу различных соединений. Чтобы производить подсчеты на уровне HTTP, монитор должен связывать идущие пакеты с соответствующим HTTP-сообщением. HTTP-сообщения обычно состоят из нескольких IP-пакетов, из-за чего монитору приходится реконструировать информацию уровня HTTP в потоке пакетов.

Разработка надежного программного обеспечения для реконструкции HTTP-сообщений из пакетов — это сложная задача, которая будет рассматриваться далее в главе 14 (раздел 14.1). Кроме того, мониторинг одного сегмента сети предоставляет ограниченное представление о Web в целом. Маршрутизация в Internet не гарантирует, что трафик между клиентом и сервером проходит по одним и тем же сегментам в обоих направлениях. В зависимости от расположения наблюдаемого сегмента в Internet, монитор не обязательно увидит одновременно и запрос, и связанный с ним ответ. Как говорилось выше, трафик на каком-либо отдельном участке не может быть репрезентативным для всей сети. Применение мониторов пакетов на большом числе сегментов стоит очень дорого. По сравнению с протоколированием на клиенте, сервере и прокси-сервере мониторинг пакетов может затронуть личные интересы пользователей. Монитор пакетов имеет доступ к каждому пакету и не является участником пересылки данных. Пользователи обычно не знают о том, что запрос или ответ проходят через монитор пакетов.

9.2.5. Активные измерения

Журналы клиентов, Web-серверов и прокси-серверов, а также результаты мониторинга пакетов обычно не включают достаточной информации, чтобы оценить производительность с точки зрения пользователя. Рассмотрим проблему в определении задержки, возникающей при загрузке Web-страницы **http://www.foo.com**, включая все встроенные изображения. Если одно или более изображений находятся на другом сервере, журнал сервера **http://www.foo.com** не сможет предоставить информацию обо всех запросах. Даже если все запросы были бы записаны в один журнал на сервере или прокси-сервере, то там бы не было бы подробной временной информации о запросах и ответах. Кроме того, в журнале сервера или прокси-сервера не записывается задержка, обусловленная запросом к DNS-серверу, чтобы преобразовать доменное имя **http://www.foo.com** в IP-адрес. Журнал клиента, если таковой имеется, запечатлел бы работу одного конкретного пользователя с конкретным набором Web страниц; эта работа будет отличаться от работы других пользователей. Мониторинг пакетов будет перехватывать трафик только одного сегмента сети, а трафик, вызванный одним запросом, может не полностью проходить по данному сегменту.

Использование журналов клиентов, прокси-серверов и Web-серверов, а также результатов мониторинга пакетов для изучения производительности с точки зрения пользователя имеет два основных ограничения. Во-первых, эти методы измерения регистрируют передачу HTTP-данных в одной точке. Это затрудняет определение производительности «глазами пользователя», а также выявление отдельных составляющих задержки. Во-вторых, эти технологии *пассивны* в том отношении, что они наблюдают за пересылкой HTTP-данных «как есть», без всякого контроля над источниками и временем отправки запросов. Это затрудняет проведение систематических наблюдений за работой Web. Альтернативным подходом является контролируемая отправка запросов и наблюдение за их выполнением. Это называется *активными* измерениями, в отличие от *пассивного* анализа данных журналов или мониторинга пакетов. Активные измерения используют клиента для отправки за-

просов и записи информации о последующих ответах, включая время и HTTP-заголовки. На практике активные измерения могут проводиться с помощью упрощенного клиента, который не поддерживает всех возможностей браузера. Этот клиент обычно читает входной файл, содержащий список URI, которые надо запросить, и моменты времени, когда эти запросы должны быть отправлены.

Проведение эксперимента, основывающегося на активных измерениях, требует ответов на следующие вопросы:

- **Где разместить модифицированные клиенты.** Результаты активных измерений очень чувствительны к расположению клиентов. Производительность у разных пар клиент-сервер разная. Клиенты отличаются с точки зрения скорости доступа, близости к Web-серверу и наличия или отсутствия прокси-серверов. Ощущения у клиента с высокоскоростным доступом через локальную сеть будут существенно отличаться от ощущений клиента, подключенного к Internet с помощью низкоскоростного модема. Ощущения пользователя в Соединенных Штатах при запросе Web-страницы с сервера в Северной Америке существенно отличаются от ощущений пользователя в России при обращении к той же странице. Когда Web-сайты реплицируются на разные компьютеры, два клиента не обязательно получат HTTP-ответы от одного и того же компьютера. Один сервер, содержащий копию сайта, может быть сильно загружен, а другой в то же самое время может простаивать. Поэтому произвести активные измерения, результаты которых могли бы считаться репрезентативными, чрезвычайно сложно.
- **Какие запросы создавать.** Web-сайты существенно различаются по производительности. Web-серверы функционируют на различных аппаратных платформах, используют различное серверное программное обеспечение, подключены к различным каналам, популярность установленных на них Web-сайтов также существенно различается. Кроме того, две Web-страницы на одном и том же сайте могут отличаться по размеру, так и по числу и объему встроенных в них изображений. Один подход состоит в выборе репрезентативного набора запросов на основе популярности Web-страниц или Web-серверов. Например, список популярных URL может быть получен из предыдущих измерений трафика, например, мониторинга пакетов или протоколирования на прокси-сервере. Тогда модифицированный клиент может воспроизводить эти запросы, чтобы измерить качество доступа к этим страницам с точки зрения пользователя. В качестве альтернативы можно определить, какие сайты включить в эксперимент, по списку наиболее популярных Web-сайтов. Понятие популярности сайтов и URL может зависеть от масштаба и места проведения эксперимента; например, пользователи в России обращаются к другим сайтам, нежели пользователи в Соединенных Штатах. Выбор Web-сайтов может определяться и другими критериями, например, желанием сравнить сайты, использующие разное серверное программное обеспечение.
- **Какие данные следует собирать.** Вопрос о том, какие именно данные будут собираться во время эксперимента, влияет на то, какие проблемы производительности можно при этом исследовать. Модифицированные клиенты могут регистрировать самую разную информацию о запросах, такую как задержки, связанные с запросами к DNS-серверам, установкой TCP-соединений, пересылкой HTTP-данных и получением ответов. Но клиенты не могут точно определить источник задержки. Например, журнал не покажет, какие действия выполнил локальный DNS-сервер, был ли таймаут TCP обусловлен утерей пакета, был ли HTTP-запрос удовлетворен исходным сервером или про-

кси-сервером. Дополнительные измерения могут выявить источники задержек. Например, пакеты, перехваченные на клиенте, предоставят более детальную временную информацию, а журналы, собранные в других точках маршрута от клиента к серверу, помогут определить, какие компоненты Web увеличили задержку.

Активные измерения — эффективная технология изучения производительности с точки зрения пользователя. Тем не менее, многообразие ситуаций, возникающих в сети, затрудняет получение общих выводов из результатов измерений. Определение параметров работы Web требует широкомасштабных экспериментов в разное время, с различным расположением клиентов и с разными сайтами. Для этого необходима широкомасштабная измерительная инфраструктура. В течение нескольких последних лет несколько исследовательских групп разместили программное обеспечение во многих географических регионах с целью проведения активных измерений [BC99, KW00]. Мы более подробно рассмотрим одну из таких платформ измерений [KW00] в главе 15 (раздел 15.4). Компании, такие как Keynote Systems [Key], применяют активные измерения, чтобы оценить производительность различных сайтов и провайдеров.

9.3. Журналы Web-серверов и прокси-серверов

Большинство Web-серверов и прокси-серверов ведут журналы в процессе своего нормального функционирования. Каждая запись в журнале соответствует одной паре запрос-ответ и включает некоторое число полей, содержащих информацию о запрашивающем клиенте, времени запроса и HTTP-сообщениях ответа и запроса. Хотя для форматов журналов нет официального стандарта, большинство реализаций прокси-серверов и Web-серверов придерживаются неформальных стандартов по форматам протоколирования. Тем не менее, число полей, равно как и их формат и способ интерпретации варьируются от сервера к серверу.

9.3.1. Common Log Format (CLF)

Каждая запись при использовании Обычного формата протоколирования (CLF — Common Log Format) состоит из семи полей, представленных в таблице 9.1.

Таблица 9.1. Common Log Format

Поле	Значение
Remote host	Имя компьютера или IP-адрес запрашивающего клиента
Remote identity	Учетная запись пользователя на клиентской машине
Authenticated user	Имя, указанное пользователем при аутентификации
Time	Дата/время запроса
Request	Метод запроса, URI запроса и версия протокола
Response code	Код HTTP-ответа
Content length	Число байтов в ответе

- **Remote host.** Данное поле содержит IP-адрес клиента или его доменное имя, например 10.245.131.2 или **users.bar.com**. Сервер может напрямую определить IP-адрес клиента по сокету, связанному с HTTP-запросом. Для получения же доменного имени клиента требуется произвести запрос к DNS серверу, чтобы преобразовать IP-адрес в доменное имя. Сервер записывает IP-адрес клиента, если запросы к DNS-серверу отключены или запрос был безуспешен. Web-серверы, обслуживающие загруженные сайты, обычно отключают обратный поиск в DNS, как это было описано в главе 5 (раздел 5.3.4).
- **Remote identity.** Поле определяет пользователя, который открыл TCP-соединение. TCP-соединение связывает Web-сервер с приложением, запущенным тем или иным пользователем на клиентском компьютере. Имя, связанное с учетной записью пользователя, обычно недоступно на другом конце TCP-соединения. RFC 1413 определяет протокол идентификации [St.93] для получения информации от удаленного компьютера. Но на практике эти запросы требуют много времени, и многие компьютеры на них не отвечают. Поэтому большинство Web-серверов не делают таких запросов. Вместо имени пользователя в таком случае в журнал сервера пишется "–".
- **Authenticated user.** Поле содержит имя пользователя, которое было предоставлено в строке **Authorization** заголовка HTTP-запроса. Это поле соответствует имени, введенному пользователем при запросе информации, защищенной паролем. В это поле записывается "–", если HTTP-запрос не содержал никакой информации для аутентификации или если сервер не записывает такую информацию.
- **Time.** Журнал сервера также содержит время запроса с разрешением до одной секунды. CLF не определяют формат и способ интерпретации этого поля, что приводит к существенным отличиям в журналах различных серверов. Правильная интерпретация этого поля зависит от того, когда сервер вызывает системную функцию (например, *gettimeofday()*), чтобы определить время [Dav99]. В зависимости от реализации сервера, время может соответствовать началу чтения запроса, началу генерации ответа, началу записи ответа в буфер или концу записи ответа в буфер. Эти четыре значения времени могут заметно отличаться, особенно для больших ресурсов или динамически генерируемого содержимого. Сервер не обязательно имеет информацию, когда операционная система передает первый или последний пакеты ответного сообщения.
- **Request.** Поле содержит первую строку заголовка HTTP-запроса, которая состоит из метода запроса, запрошенного URI и версии протокола.
- **Response code.** Поле содержит трехразрядный код, содержащийся в заголовке HTTP-ответа, например 200 для ответа **200 OK**.
- **Content length.** Это поле содержит число байтов в ответе. В зависимости от реализации сервера это поле может содержать размер тела ответа, размер сообщения-ответа или число байтов, переданных сервером. Если сервер записывает размер тела ответа, то поле **Content length** примет значение 0 или "–" для ответов, не включающих тела (например, **304 Not Modified**). Размер сообщения-ответа отличается от числа переданных байтов, если сервер не отправляет все сообщение. Например, при отмене запроса число переданных байтов может быть меньше размера запрошенного ресурса.

На рис. 9.1 представлены четыре записи журнала в формате CLF. В каждом запросе вторые и третьи поля равны "– –", так как сервер не записывает имя удален-

ной учетной записи и имя пользователя. Время содержит день, месяц, год, после чего идет местное время; "−0400" означает, что местное время отстает от Гринвичского на четыре часа.

```
10.245.131.2 - - [15/Oct/2000:00:00:25 -0400] "GET /img/logo.gif
   HTTP/1.0" 304 0
10.245.131.2 - - [15/Oct/2000:00:00:25 -0400] "GET /img/logo.gif
   HTTP/1.0" 304 0
10.245.131.2 - - [15/Oct/2000:00:00:26 -0400] "GET /img/site.gif
   HTTP/1.0" 304 0
10.3.2.16 - - [15/Oct/2000:00:00:38 -0400] "GET /est/bubhi.html
   HTTP/1.0" 304 0
10.16.202.34 - - [15/Oct/2000:00:00:45 -0400] "GET /img/tday.gif
   HTTP/1.0" 200 3699
```

Рис. 9.1. Фрагмент журнала в формате CLF с вымышленными IP-адресами клиентов

9.3.2. Extended Common Log Format (ECLF)

Журналы серверов могут содержать и другие поля, некоторые из них приведены в таблице 9.2. Точный набор полей зависит от настройки сервера. ECLF точно не указывает, какие из полей должны быть включены в журнал.

Таблица 9.2. Поля, которые могут содержаться в журналах в формате ECLF

Поле	Значение
User Agent	Информация об агенте пользователя
Referer	URI Web-страницы, с которой произведен запрос
Request processing time	Время, потраченное на обработку запроса
Request header size	Число байтов в заголовке запроса
Request body size	Число байтов в теле запроса
Remote response code	Код ответа сервера
Remote content length	Размер ответа, отправленного сервером
Remote response header size	Размер заголовка ответа, отправленного сервером
Proxy request header size	Размер заголовка запроса, отправленного серверу
Proxy response header size	Размер заголовка ответа, отправленного клиенту

Обычно используются следующие поля:

- **User agent.** Содержит значение поля **User-Agent** заголовка запроса с названием и версией программы, инициировавшей запрос, как это было описано в главе 6 (раздел 6.2.3).
- **Referer.** Содержит значение поля **Referer** заголовка запроса, если таковое есть. Поле **Referer** заголовка запроса указывает на URI Web-страницы, находясь на которой, пользователь сделал запрос, как это описано в главе 6 (раздел 6.2.3). Например, предположим, что пользователь инициирует запрос, щелкая мышью на гиперссылке на Web-странице поискового сайта; тогда поле **Referer** будет содержать URI страницы поискового сайта.

- **Request processing time.** Это поле содержит время в секундах, потраченное сервером на генерацию ответа. Это поле особенно полезно, если сервер тратит значительное время на обработку отдельных запросов, таких как запросы, требующие исполнения сценария.
- **Request header size.** Это поле содержит число байтов в заголовке HTTP-запроса, посланного клиентом.
- **Request body size.** Это поле содержит число байтов в теле HTTP-запроса, посланного клиентом. Запрос обычно не имеет тела, если только клиент не пошлет запрос **PUT** или **POST**.
- Сумма размеров заголовка и тела запроса составляют полный размер запроса клиента.

Оставшиеся поля, приведенные в таблице 9.2, относятся к прокси-серверам. В отличие от клиентов и серверов, прокси-сервер работает и как клиент, и как сервер. При пересылке данных прокси-сервер получает запрос от клиента, находящегося по одну сторону от прокси-сервера, и посылает его серверу, находящемуся по другую сторону. В результате прокси-сервер может записывать дополнительные поля, отсутствующие в журналах Web-серверов. Например, прокси-сервер может использовать следующие поля:

- **Remote response code.** Это поле содержит трехразрядный код, включенный в поле **Status-Line** заголовка HTTP-ответа, отправленного сервером прокси-серверу. Значение поля **Remote response code** может отличаться от значения поля *response code*, которое содержит код ответа, отправленный прокси-сервером клиенту.
- **Remote content length.** Значение этого поля — число байтов в сообщении, посланном сервером прокси-серверу. Значение в поле **Remote content length** может отличаться от поля *content length*, которое содержит размер сообщения, отправленного прокси-сервером клиенту.
- **Remote response header size.** Это размер в байтах заголовка ответа сервера прокси-серверу.
- **Proxy request header size.** Это поле содержит размер заголовка запроса, посланного прокси-сервером серверу. Этот размер может отличаться от значения поля *request header size*, в котором записано число байтов, отправленных прокси-серверу клиентом.
- **Proxy response header size.** Поле содержит размер заголовка ответа, посланного прокси-сервером клиенту.

Эти дополнительные поля могут быть полезны при анализе работы прокси-сервера в качестве промежуточного звена между клиентами и серверами.

9.4. Предварительная обработка результатов измерений

Результаты измерений Web-трафика обычно имеют большой объем, их можно анализировать различными способами. Предварительная обработка включает в себя следующие три шага:

- Синтаксический анализ для обнаружения ошибочных записей.
- Фильтрация данных для удаления ненужных полей.
- Преобразование данных в формат, удобный для анализа.

Предварительная обработка облегчает работу разнообразных инструментов для хранения, анализа и отображения результатов анализа. С другой стороны, предварительная обработка должна сохранить ключевые данные, чтобы не налагать ненужных ограничений на последующий анализ. Конкретные примеры предварительной обработки данных измерений приведены в главе 14 (раздел 14.2).

9.4.1. Синтаксический анализ результатов измерений

Как первый этап предварительной обработки данных измерений, синтаксический анализ проводится независимо от семантики данных. Результаты измерений обычно состоят из набора записей, состоящих из одного или более полей. Например, журнал в формате CLF, показанный на рис 9.1, содержит четыре записи с семью полями каждая. Результаты мониторинга пакетов могут иметь более сложный формат. Синтаксический анализ включает в себя определение границ между отдельными полями в каждой записи и между записями. Например, записи могут отделяться друг от друга символом перехода на новую строку или пустыми строками, а поля — пробелами или символами табуляции. Записи переменной длины могут включать в себя поле длины, показывающее начало следующей записи. Каждое поле также может иметь собственный формат, в частности, значение времени может заключаться в квадратные скобки (например, **[15/Oct/2000:00:00:25 -0400]**), а строка запроса — в кавычки (например, **"GET /img/logo.gif HTTP/1.0"**).

Программа синтаксического анализа может быть написана на любом языке программирования. Выбор языка зависит от сложности и размеров результатов измерений. Например, анализ журнала в формате CLF не очень сложен по сравнению с результатами мониторинга пакетов. Код программы синтаксического анализа может обнаружить записи и поля, которые нарушают требуемый синтаксис. Например, запись может иметь неправильное число полей или поле может иметь неправильный формат. Программа синтаксического анализа может обнаружить эти записи и удалить их, что позволит остальным программам предварительной обработки иметь дело только с синтаксически корректными данными. Программа синтаксического анализа данных может преобразовывать набор записей в более удобную форму, или она может осуществлять присваивание значения полей переменным для дальнейшей обработки. Кроме того, программа синтаксического анализа может приводить наборы записей в различных форматах к единому формату. Журналы в формате CLF разных Web-серверов часто имеют различия в синтаксисе. Устранение таких различий программой синтаксического анализа позволит остальным программам работать с журналами различных серверов.

9.4.2. Фильтрация результатов измерения

Следующий этап предварительной обработки — фильтрация результатов измерений для удаления ненужных полей и записей. В некоторых случаях отдельные поля могут быть опущены, так как в них нет полезной информации для данного исследования. Например, у каждой записи в журнале формата CLF на рис. 9.1 в полях Remote identity и Authenticated user стоят "-", потому что на сервере отключена идентификация и не применяется аутентификация (не записывается информация о

регистрации пользователей). Из журнала можно исключить эти поля, так как они неинформативны. Кроме того, программа, производящая фильтрацию, может удалить ненужные для анализа поля. Примером может служить результат мониторинга пакетов, который включает полные заголовки ответов и запросов. Использование только первых строк заголовков может быть достаточно для многих задач. Пропуск оставшейся части заголовка сократит накладные расходы, связанные с хранением данных. Но удаление полей требует тщательного определения набора аналитических задач, которые будут проводиться над результатами измерений.

Фильтрация может также включать в себя удаление целых записей в зависимости от содержимого одного или нескольких полей. Записи с некорректными значениями полей могут оказать отрицательное влияние на последующий анализ. Например, запись в формате CLF может иметь хотя и синтаксически правильное значение времени, но не попадающее во временной интервал, который покрывает журнал. Помимо удаления ошибочных записей, фильтрация может использоваться для удаления записей, не связанных с текущим анализом. Например, для анализа количества запросов тех или иных ресурсов с сервера могут быть не нужны запросы, возвращающие некоторые коды ошибок. Подобным образом анализ ресурсов с изображениями может сосредоточиться на запрашиваемых URI с расширениями, соответствующими стандартным форматам изображений (например, **.gif** и **.jpg**).

9.4.3. Преобразование данных измерений

На последнем этапе предварительной обработки осуществляется преобразование данных, чтобы получить новый набор записей для последующего анализа. Преобразованные данные могут иметь более простой синтаксис, который удобнее анализировать. Например, поле запроса в журнале формата CLF может быть преобразовано в три отдельных поля: метод запроса, запрошенный URI и версия протокола. Подобным образом, поля, чувствительные к регистру, могут быть приведены к единому формату. Например, **Content-Length** и **Content-length** следует трактовать как один и тот же заголовок. Кроме того, возможно преобразование содержимого полей в другой тип данных. Значение времени, представленное в журнале как год, месяц, день, час, минута и секунда, может быть преобразовано в одно число, используя, например, представление времени в UNIX (число секунд, прошедших с полуночи 1 января 1970 года). В качестве другого примера IP-адрес клиента в десятичной точечной записи (10.34.197.3) может быть преобразован в 32-разрядное беззнаковое целое число для уменьшения объема данных.

Другие преобразования включают изменение семантики полей с целью добавить или изъять информацию в каждой записи. Например, журнал в формате CLF на рис. 9.1 включает IP-адрес запрашивающего клиента. IP-адрес может быть преобразован в доменное имя с помощью запроса к DNS-серверу. Выполнение этого преобразования уже после того, как журнал был сформирован, позволяет избежать излишней нагрузки на сервер, связанной с преобразованием IP-адресов в доменные имена в ходе обработки HTTP-запросов. Тем не менее, это преобразование следует произвести как можно быстрее, чтобы получить достоверную информацию. Выполнение преобразования в процессе предварительной обработки также избавляет от необходимости переводить IP-адреса в доменные имена по несколько раз в течение последующего анализа. Другие преобразования могут скрыть ту или иную личную информацию пользователей. Например, программа предварительной обработки может включать в себя функцию, которая кодирует IP-адреса пользователей.

Подходящий язык программирования для преобразования данных зависит от сложности требуемых операций. Языки сценариев могут подходить для простых синтаксических преобразований. Более сложные операции, например, запросы к DNS-серверу, могут быть реализованы с помощью системных вызовов. Программа предварительной обработки создает новый набор записей, который может быть проанализирован различными способами. Записи могут храниться в базе данных, которая поддерживает выполнение различных запросов. Записи могут также быть введены в электронную таблицу или в специальную программу для анализа журналов. Сценарии, написанные на awk или Perl, позволяют производить статистические расчеты, а более сложные программы анализа могут быть написаны на C. В любой из этих ситуаций наличие чистого и логично скомпонованного исходного набора записей сильно упрощает анализ результатов измерений трафика.

9.5. Получение информации по результатам измерений

Результаты измерений не всегда содержат всю информацию, необходимую для нахождения точного ответа на основные вопросы о параметрах Web-трафика. Некоторые из этих ограничений следуют из того, что результаты измерений не включают всех нужных полей, например значений времени или HTTP-заголовков. В других случаях необходимая информация просто недоступна там, где проводятся измерения. В данном разделе приводятся методы получения информации об HTTP-заголовках, клиентах/серверах, действиях пользователей и изменениях ресурсов, несмотря на все ограничения в данных измерений.

9.5.1. Ограничения информации об HTTP-заголовках

Заголовки HTTP-запросов и ответов предоставляют богатую информацию о передаче данных в сети. Но некоторые журналы и данные мониторинга пакетов не содержат полных HTTP-заголовков. Большинство журналов серверов записывают строку запроса и код ответа, но не полные заголовки ответов и запросов. Сбор более детальной информации требует перенастройки или модификации сервера. Монитор пакетов, клиент, прокси-сервер или Web-сервер могли бы записывать заголовок каждого сообщения и все тело сообщения. Но на практике это зачастую неосуществимо. Многие исследования опираются на журналы Web-серверов и прокси-серверов, в которых записана только часть всех строк HTTP-заголовков. В некоторых случаях запрос, зарегистрированный в журнале, может быть *уточнен* посредством отправки запроса тому же серверу и наблюдения за ответом. Но такой подход будет неуместен, если запрошенный ресурс был изменен на сервере или если ответ зависит от состава заголовков HTTP-запроса.

Некоторая недостающая информация может быть реконструирована на основе таких полей, как метод запроса, запрошенный URI и код ответа, которые доступны в журнале сервера. По значению поля, содержащего запрошенный URI, можно определить тип запрошенного ресурса. Например, URI, заканчивающиеся на **.html** или **.htm** скорее всего соответствуют HTML-файлам, в то время как URI, заканчивающиеся на **.gif** или **.jpg,** скорее всего соответствуют изображениям; хотя некоторые изображения имеют другие расширения файлов, а некоторые URI, заканчивающиеся на **.gif** или **.jpg,** могут не соответствовать изображениям. Подобным образом по запрошенному URI или методу запроса можно определить, вызывает ли

запрос исполнение сценария на сервере. Запрошенный URI, включающий строку "cgi" или "cgi-bin", обычно соответствует сценарию[1]. Запросы **GET** с параметрами в конце URI, а также запросы **POST** обычно соответствуют HTML-формам. Эти эвристики могут использоваться в простых статистических процедурах, например, при вычислении доли запросов, ответы на которые представляют собой динамически генерируемое содержимое или содержимое определенного типа.

Некоторые коды ответов подразумевают наличие определенных строк в заголовке HTTP-запроса. Например, ответ **206 Partial Content** подразумевает, что клиент запросил часть ресурса, используя заголовок **Range**. Аналогично, ответ **304 Not Modified** подразумевает, что запрос включал проверку актуальности ресурса, например, с помощью строки заголовка **If-Modified-Since**. Однако в ответ на запрос **Range** или **If-Modified-Since** можно получить ответ **200 OK**, как если бы был возвращен весь ресурс. Таким образом, коды ответов могут дать представление только о нижней границе частоты появления определенных типов запросов, но не их точное число. Некоторые аналитические задачи могут использовать коды ответов для определения того, какие запросы игнорировать при проведении тех или иных расчетов. Например, рассмотрим задачу подсчета количества уникальных запросов на ресурсы сервера на основе серверного журнала. Запрошенный URI, который всегда приводит к ответу **404 Not Found** не должен быть включен в обрабатываемые данные, так как запрошенный ресурс отсутствовал на сервере в период формирования журнала.

9.5.2. Неоднозначная идентификация клиента/сервера

На практике бывает трудно выявить, связаны ли пары запрос-ответ с одним и тем же пользователем. Например, IP-адрес клиента, зафиксированный сервером, не обязательно является уникальным идентификатором пользователя. С одним и тем же клиентом могут работать несколько пользователей, а один и тот же пользователь может выходить в Internet с различными клиентскими IP-адресами, как описано в разделе 9.2.1. Подобным же образом, IP-адрес сервера, полученный при мониторинге пакетов, не является уникальным идентификатором. На одном и том же компьютере могут работать несколько Web-серверов, а один и тот же Web-сайт может размещаться на различных компьютерах. Доменное имя сервера можно получить из запрошенного URI или строки **Host** заголовка запроса HTTP/1.1, предполагая, что результаты измерений включают эту строку. В запросах HTTP/1.0 такого поля нет. В некоторых случаях IP-адрес сервера может быть преобразован в доменное имя на более поздней стадии проведения анализа с помощью запроса к DNS-серверу. Но к моменту анализа ответ DNS-сервера может отличаться от того же на момент передачи данных.

Различение уникальных ресурсов также может быть в некоторых случаях затруднено. При обработке запроса сервер преобразует запрашиваемый URI в имя файла. Рассмотрим задачу определения числа уникальных ресурсов, запрошенных с Web-сервера, на основе поля **Request** в журнале сервера. В некоторых случаях различные запрашиваемые URI соответствуют одному и тому же ресурсу. Например, **http://www.foo.com** и **http://www.foo.com/index.html** обычно соответствуют одному и тому же файлу **(/www/index.html)**. Хотя значение URI зависит от регистра символов, Web-сервер может быть настроен таким образом, чтобы трактовать запрос **http://www.foo.com/INDEX.HTML** как другое имя для **http://www.foo.com/index.html**. Некоторые преобразования включают в себя достаточно тривиальные операции, такие

---

[1] Можно добавить, что ресурсы с расширениями .asp, .php, .pl также представляют собой сценарии. — *Прим. ред.*

как удаление повторных символов "/". Запрошенные URI, которые отличаются в чем-то незначительном, могут быть отслежены во время обработки журнала. В других ситуациях сервер может трактовать некоторые URI как псевдонимы других URI. Без доступа к настройкам сервера распознавание таких имен практически невозможно. Большинство исследований обычно предполагает, что различные URI соответствуют различным ресурсам.

9.5.3. Реконструкция действий пользователя

Анализ результатов измерений Web-трафика обычно требует определения того, когда и как часто происходят определенные события, связанные с пользователем. Изучение поведения пользователя включает, например, определение, когда пользователь щелкает мышью на гиперссылке. Даже если программный клиент используется лишь одним пользователем, все равно определение и классификация действий пользователя представляет собой сложную задачу. Информация из HTTP-запросов и ответов, также как длительность передачи и размер отправленных данных, могут быть использованы для реконструкции ключевых событий. Например, поле **Time** в журнале сервера предоставляет удобный способ определить последовательность запросов, отправленных одним и тем же клиентом. Время между последующими запросами может подсказать, какие запросы относятся к одному сеансу посещения Web-сайта и какие из этих запросов соответствуют активизации пользователем гиперссылки, в отличие от запросов на встроенные ресурсы, которые посылаются браузером автоматически.

HTTP не сохраняет своего состояния, что затрудняет определение того, какие запросы связаны друг с другом, так как сеанс взаимодействия клиента и Web-сервера не имеет четкого начала и конца. Вместо этого, начало сеанса взаимодействия между клиентом и Web-сервером можно определить на основе измерений, полученных при протоколировании и мониторинге пакетов. Первый запрос соответствует началу сеанса. Запрос, пришедший через несколько десятков секунд после предыдущего запроса, сделанного тем же самым клиентом, может рассматриваться как принадлежащий тому же сеансу. Более точные выводы возможны при наличии дополнительной информации о структуре сайта. Например, запрошенный URI может соответствовать гиперссылке на одной из предыдущих страниц, что увеличивает вероятность того, что запрос был инициирован в результате щелчка мышью на гиперссылке на одной из этих страниц. Информация о гиперссылках на просмотренных страницах может быть получена из поля **Referer** в HTTP-запросе. Структура гипертекста может быть получена с помощью анализа Web-страниц на этапе обработки данных, хотя их содержимое может быть изменено со времени проведения измерений.

Одиночное действие пользователя, такое как щелчок мышью на гиперссылке, может инициировать несколько HTTP-запросов для загрузки Web-страницы и встроенных в нее изображений. Выявление действий пользователя требует эффективных способов различения запросов, вызванных непосредственно пользователем, и запросов, сгенерированных автоматически. Хотя журнал браузера различает эти запросы, журнал прокси-сервера/Web-сервера или результаты мониторинга пакетов не будут содержать этой информации, если только не доступно содержимое всей страницы. Вместо этого может различаться время между последующими запросами: если запрос приходит меньше чем, скажем, через одну или две секунды после предыдущего запроса, отправленного тем же клиентом, то он автоматически считается выданным браузером. Выводы будут точнее, если запрашиваемый URI соответствует изображению, например, GIF- или JPEG-файлу. Запрошенный ре-

сурс с еще большей вероятностью является встроенным изображением, если запрашиваемый URI содержит тот же путь, что и путь к самой странице. Например, запрос **http://www.foo.com/bar/pict.gif**, который следует в скором времени за запросом **http://www.foo.com/bar/index.html** скорее всего соответствует встроенному изображению.

Большие промежутки времени между запросами соответствуют щелчку мыши, произведенному пользователем, особенно если запрашивается ресурс, не являющийся изображением. Определение того, какие запросы обусловлены действиями пользователя, важно для изучения взаимодействия пользователя с Web-сайтом, например, для определения среднего числа переходов между Web-страницами на сеанс работы пользователя с сайтом. Время между переходами со страницы на страницу может трактоваться как время бездействия или обдумывания. В общем случае эвристика, основанная на времени между переходами со страницы на страницу, слишком проста. Пользователь может потратить несколько минут, «обдумывая» содержимое Web-страницы, перед тем, как щелкнуть на гиперссылке, а может щелкнуть мышью на гиперссылке меньше, чем через секунду. Более того, пользователь может перестать работать на компьютере на какой-то период времени, а затем продолжить просматриватьWeb-сайт. Кроме того, клиент может отправить набор запросов от лица разных пользователей. Но по-прежнему эвристика, основывающаяся на времени, представляет единственный способ получить информацию о поведения пользователя на основе измерений трафика.

9.5.4. Отслеживание модификации ресурсов

Определение того, когда Web-ресурсы создаются, модифицируются и удаляются полезно для анализа изменений на Web-сайте. Кроме того, статистика частоты изменений может влиять на политику Web-кэширования. В идеале Web-сервер мог бы создавать записи о создании, модификации и удалении ресурсов помимо протоколирования запросов пользователей. На практике наблюдение за изменениями затруднено, так как результаты измерений трафика обычно не включают такой информации. Ресурсы, которые не были ни разу запрошены, не появятся в журнале. Кроме того, ресурс мог быть модифицирован несколько раз между последующими запросами, а журналы серверов обычно не включают информацию о возрасте ресурса или времени последней модификации. Статистика модификации ресурсов может быть получена косвенным образом из HTTP-заголовков, размеров ответов и времени каждой записи в журнале. В некоторых случаях этих данных бывает достаточно для оценки возраста запрашиваемого ресурса и времени между модификациями.

Рассмотрим задачу по определению возраста ресурса ко времени запроса, основываясь на информации из HTTP-заголовков. Возраст — это разница между временем запроса и временем модификации. Хотя большинство серверов включают время получения запроса в журнал, время последней модификации обычно не включается. Но монитор пакетов и журнал браузера могут записывать поля **Last-Modified** и **Date** из заголовка HTTP-ответа. Возраст ресурса является разностью этих двух времен. Если результаты измерений не включают время из заголовка **Date**, время запроса может быть отмечено там, где проходит измерение. Например, журнал монитора пакетов или прокси-сервера может записать время наблюдения запроса. Однако регистрация времени наблюдения запроса не отражает времени на сервере, где генерируется заголовок **Last-Modified**. Часы, работающие на разных компьютерах в Internet, могут не быть синхронизированы. Если возможно, следует внести поправку на разницу времени между часами.

Другой подход к изучению времени модификации ресурсов включает в себя сравнение заголовков **Last-Modified,** следующих друг за другом ответов с одним и тем же URI. Если ответные сообщения имеют разные поля **Last-Modified**, то ресурс изменился хотя бы однажды между этими ответами. Разница между значениями **Last-Modified** дает нижнюю границу времени[1], которое прошло между двумя последующими модификациями ресурса. Предположим, например, что один ответ содержал значение **Last-Modified**, равное 13 часам, а второй — 14. Предыдущая версия запроса существовала не более часа. Если ресурс менялся больше, чем один раз в промежутке между двумя ответами, то время между модификациями будет меньше одного часа. Этот подход нечувствителен к разнице между часами сервера и компьютера, так как оба значения были записаны на компьютере, где функционирует Web-сервер.

Заголовки HTTP-ответов не всегда включают строку **Last-Modified**. В некоторых случаях сообщение ответа может включать в себя другой заголовок, например, **ETag**, который используется для различения версий ресурса. Изменение значения этого поля сигнализирует, что ресурс изменился между двумя последующими запросами, не предоставляя информации о том, когда произошло изменение. В других случаях изменение ресурса может быть определено по изменению значения заголовка **Content-Length**. Если два ответа с одним и тем же URI имеют разные значения **Content-Length**, то ресурсы различны. Но обратное предположение неверно. Ресурс может быть изменен без изменения его размера. Тем не менее, сравнение строк **Content-Length** позволяет получить консервативную оценку частоты изменений.

Строки **Content-Length** может не быть среди данных некоторых измерений. Например, журнал сервера может записывать *размер отправленных данных* вместо размера ответа. На некоторых серверах размер отправленных данных может не включать размер заголовка ответа. Размер отправленных данных не совпадает для разных запросов одного и того же URI, если некоторые запросы были отменены до того, как все данные были переданы или ответы имеют разные кодировки данных. В некоторых случаях можно косвенным образом определить, какие запросы были отменены. Рассмотрим в качестве примера журнал сервера, который содержит 20 запросов некоторого URI. Предположим, что один запрос имеет меньший размер отправленных данных, чем остальные, а размеры отправленных данных остальных 19 запросов совпадают. Этот укороченный ответ мог быть обусловлен отменой запроса. Правда, ресурс мог быть изменен дважды, причем второе изменение вернуло ему его первоначальный размер. Еще один эвристический прием основывается на том, что некоторые прокси-серверы и Web-серверы пишут ответы в буфер сокета блоками фиксированного размера (4096 или 8192 байта). Размеры отправленных данных, делящиеся нацело на размер блока, скорее всего, соответствуют отмененным запросам.

9.6. Примеры исследовательских проектов

В этом разделе мы представим в общих чертах четыре исследовательских проекта, которые иллюстрируют способы преодоления ограничений методов измерений для получения информации о характеристиках Web-трафика:

- **Исследование журналов серверов в Университете провинции Саскачеван.** Шесть журналов серверов с записями 1995 года был проанализирован на предмет изучения параметров Web-трафика [AW97].

---

[1] Авторы ошибаются, таким образом можно получить верхнюю границу времени между двумя изменениями, как это следует из дальнейшего. — *Прим. ред.*

- **Исследование журналов прокси-серверов в Британской Колумбии.** Набор из семи журналов прокси-серверов с записями 1996 и 1997 годов был использован для моделирования работы кэширующего прокси-сервера в различных конфигурациях [DMF97].
- **Исследование клиентских журналов Бостонского университета.** Журналы, собранные у студентов университета в 1995 году, использовались для изучения пользовательских предпочтений и объяснения характеристик возникающего при этом сетевого трафика [CBC95, CB97].
- **Мониторинг пакетов в AT&T.** Журнал монитора пакетов 1996 года был использован для анализа частоты изменений Web-ресурсов, а также природы этих изменений [DFKM97].

Имеется сравнительно мало работ по активным Web-измерениям, и мы отложим рассмотрение примеров активных Web-измерений до главы 15 (раздел 15.4). Сейчас же мы обратим основное внимание на методики измерений и анализа данных, а не на конкретные численные результаты этих измерений (последние к тому же существенно зависят от места и времени сбора данных).

9.6.1. Исследование журналов Web-серверов, проведенное Университетом провинции Саскачеван

Изучение журналов Web-серверов, проведенное Университетом провинции Саскачеван, имело целью определить ключевые характеристики Web-трафика [AW97]. Поиск инвариантных характеристик требовал анализа многочисленных журналов для различных типов Web-серверов. Проводился анализ шести журналов из трех университетов, двух научных организаций и одного провайдера. Исследователи не имели возможности влиять на состав полей записей в журналах. Эти журналы были в формате CLF и включали в себя имя запрашивающего компьютера, время запроса, запрошенный URI, код ответа и количество отправленных байтов в ответе в следующем формате:

```
Hostname - - [dd/mmm/yyyy:hh:mm:ss:tz] request status bytes
```

Журналы не включали информации о времени, потраченном на генерацию и передачу ответов, а также о действительных размерах ответов. Кроме того, не было информации о полном наборе файлов на серверах, которые могли быть запрошены пользователями, прокси-серверами или программами индексирования Web-сайтов. Тем не менее, были получены некоторые характеристики на основе данных, взятых из журналов серверов, как показано в таблице 9.3. Некоторые основные статистики были рассчитаны непосредственно, исходя из значений полей журналов. Например, число различающихся запрошенных URI, средний размер ответа, а также частоты появления кодов ответов.

Статистики пользовательских предпочтений были рассчитаны, исходя из взаимоотношений значений полей нескольких записей в журналах. Просмотр последовательностей записей в журнале сделал возможным определение времени между последовательными запросами, приходящими на сервер. Это позволило оценить динамику загрузки сервера. Анализ запрашиваемых URI позволил исследователям определить частоту запросов наиболее популярных ресурсов и время между последующими запросами одного и того же ресурса. Исследование этих свойств было важно из-за их влияния на эффективность Web-кэширования. Другая информация была получена косвенно на основе полей, содержащихся в журнале. Например, запрашиваемые URI были использованы для предположительного определения типа содержимого. Количество переданной информации, связанное с запрошенным URI,

использовалось для предположительного определения размера ресурса, равно как частот изменений ресурсов и частот отмены запросов.

Таблица 9.3. Ключевые метрики исследования журналов Web-серверов, проведенное Университетом провинции Саскачеван

Категория	Метрика
Основные статистики	Число различающихся запрошенных URI
	Средний размер/медиана ответов
	Частоты кодов ответов
Пользовательские предпочтения	Время между запросами
	Популярность запрашиваемых ресурсов
	Время между запросами одного и того же ресурса
Данные, полученные косвенно	Типы содержания запрашиваемых ресурсов
	Среднее значение/медиана размеров ресурсов
	Частоты модификаций ресурсов
	Частоты отмененных запросов

9.6.2. Исследование журналов прокси-серверов в Британской Колумбии

Исследователи Университета Британской Колумбии проводили исследование Web-кэширования на основе семи журналов прокси-серверов, датированных 1996 и 1997 годами [DMF97]. Эти журналы предоставлены различными учреждениями, включая журналы университетов, компаний и одного национального прокси-сервера. Все эти организации использовали прокси-сервер Squid, описанный в главе 11 (раздел 11.10.1). Журналы отличались друг от друга по характеру содержащейся в них информации. Каждая журнальная запись включала IP-адрес клиента, время запроса, запрошенный URI и размер ответа. Чтобы защитить интересы пользователей, IP-адреса клиентов были замаскированы путем перевода их в другое представление. Два сервера меняли функцию преобразования каждый день, что делало невозможным определить, посещал ли данный пользователь тот же самый сервер в течение последующих дней; другие пять серверов применяли одну и ту же функцию на протяжении всего периода протоколирования. Шесть журналов были от кэширующих прокси-серверов. Эти журналы включали дополнительное поле, показывающее, был ли запрос удовлетворен из кэша или нет. Записи в седьмом журнале не включали это поле, но включали значения заголовков **Last-Modified** и **Expires**, когда они присутствовали в HTTP-ответе сервера.

В отличие от предыдущего проекта, данный проект не ставил цели статистически охарактеризовать результаты измерений. Вместо этого журналы были использованы для оценки эффективности кэширования на прокси-серверах при помощи моделирования. Целью проекта были эксперименты с различными размерами кэша, частотой запросов и разными политиками управления кэшированием, используя модель прокси-сервера, основанную на Squid. Моделирование являлось ценной альтернативой экспериментам на работающем прокси-сервере. Во-первых, перенастраивать работающие прокси-серверы во всех семи учреждениях не являлось возможным. Во-вторых,

использование модели позволило экспериментировать с такими размерами кэшей и такими политиками кэширования, которые были бы недоступны на работающем прокси-сервере. В-третьих, использование модели позволило повторять эксперимент на различных конфигурациях, используя один и тот же журнал.

Моделирование использовалось для оценки различных конфигураций кэша, вычислялись четыре основные метрики производительности. Производительность кэша оценивалась по отношению числа сообщений-ответов прокси-сервера и числа байтов в ответах, которые были взяты из кэша, а не из ответов сервера, которому был направлен запрос. Запрос удовлетворялся из кэша прокси-сервера, если моделируемый кэш имел актуальную копию запрошенного ресурса, идентифицируемого запрашиваемым URI. При моделировании также определялась вероятность того, что кэшируемый ответ не мог быть передан прокси-сервером из-за изменения ресурса на сервере. Кроме того, оценивалось преимущество, которое дает кэш прокси-сервера в случае его совместного использования разными клиентами, а не одним клиентом при последовательных запросах. Эти метрики были изучены при разных размерах кэша и разных частотах запросов. Изменение частоты запросов обеспечивалось выборкой запросов из журнала, отправленных случайным набором пользователей. Это имитировало среду, в которой прокси-сервер обслуживал небольшое число пользователей, пользовательские предпочтения которых были такими же, как у клиентов исходного журнала.

Отсутствие информации в данных измерений вызвало несколько проблем. Например, неоднозначная идентификация клиентов затрудняла определение, как часто ресурсы из кэша прокси-сервера использовались совместно несколькими пользователями или последовательно одним и тем же пользователем. Чтобы связать запросы с пользователем требуется наличие уникального идентификатора запрашивающего клиента на протяжении всего журнала. В двух журналах идентификатор, связанный с IP-адресом, менялся каждый день. Эти два журнала не удалось использовать при изучении популярности ресурсов. В случае других пяти учреждений каждый клиент имел один (замаскированный) идентификатор. При изучении этих пяти журналов пришлось допустить, что имелось однозначное соответствие между клиентами и пользователями. Это не так, если несколько пользователей использовали один компьютер, или один пользователь использовал разные компьютеры для выхода в Internet.

Неполная информация о заголовках ответов серверов, связанных с кэшированием, затруднила определение актуальности ресурсов в кэше моделируемого прокси-сервера. В реальности ресурс мог быть со временем изменен. Но в журналах отсутствовала информация о том, что ресурс изменился, а также когда произошли эти изменения. Несмотря на это в некоторых случаях из журналов можно было узнать, что ресурс оставался актуальным между двумя следующими друг за другом запросами. Журнал прокси-сервера сообщал, удовлетворил ли прокси-сервер запрос из кэша или нет. Если запрос был удовлетворен из кэша, то кэшированный ответ был в тот момент актуальным. Тогда полученные сведения применялись к моделируемому прокси-серверу. Когда на моделируемый прокси-сервер приходил запрос этого ресурса, предполагалось, что ресурс актуальный и, следовательно, мог быть передан из кэша.

9.6.3. Исследование клиентских журналов Бостонского университета

Исследователи Бостонского университета собирали журналы браузеров для исследования пользовательских предпочтений и их влияния на сетевой трафик

[CBC95]. На тот момент самым популярным браузером был Mosaic. Программа была свободно распространяемой и могла быть модифицирована для генерации клиентских журналов. Браузер был настроен так, чтобы протоколировать запросы клиентов, и был установлен на рабочих станциях, подключенных к локальной сети Факультета Информатики Бостонского университета. Браузеры протоколировали пользовательскую активность и передавали данные для обработки центральному компьютеру. К концу исследований популярность браузера Mosaic понизилась, код ставших популярными браузеров не мог быть изменен для выполнения исследований. В результате подобное исследование невозможно повторить в наше время. Кроме того, выявленные пользовательские предпочтения студентов университета весьма сильно отличаются от предпочтений других групп пользователей Web.

Каждая запись журнала соответствовала запросу и включала имя компьютера клиента, время запроса (в микросекундах), запрашиваемый URI, размер ответа и время ответа (в секундах). Журналы браузеров использовались для изучения основных свойств Web-трафика [CB97], включая статистики, использованные исследователями Университета провинции Саскачеван, например, размеры ответов и размеры ресурсов. Кроме того, изучалась задержка в получении ресурса и ее связь с размерами ресурса, взаимосвязи типов ресурсов и их размеров (см. таблицу 9.4).

Таблица 9.4. Основные метрики исследования журналов браузеров Бостонского университета

Категория	Метрика
Основные статистики	Размер ресурса
	Размер ответа
	Задержка в получении ресурса
	Тип содержания ресурса
Измерения времени	Время между автоматически генерируемыми запросами
	Время между запросами пользователя
	Совокупная нагрузка на сеть

Исследование касалось также времени между последовательными запросами пользователя, даже если они относились к разным Web-сайтам. Это было возможно, поскольку браузер перехватывал все запросы пользователя, включая запросы к различным Web-сайтам и запросы, удовлетворяемые из кэша самого браузера. Такую информацию невозможно собрать на прокси-сервере и на Web-сервере. Измерения времени позволили дополнительно вычислить время между завершением запроса и началом следующего запроса. Это позволяет оценить время, в течение которого пользователь изучает полученную Web-страницу до того, как послать следующий запрос.

9.6.4. Мониторинг пакетов в AT&T

Для разработки эффективных стратегий кэширования Web-ресурсов необходимо знать, какие ресурсы претерпевают изменения, как часто они меняются и как сильно. Это обусловило детальное изучение частот изменений Web-ресурсов [DFKM97]. При анализе небольшого числа серверных журналов возможно определение частот модификаций их содержимого. Однако это не дает репрезентативной информации о частоте изменения ресурсов в Internet. Вместо этого исследование проводилось в сегментах сети, расположенных вблизи от большого числа выходя-

щих в Internet пользователей, работающих в двух компаниях. В исследовании использовались журналы прокси-сервера компании Digital Equipment Corporation и мониторинг пакетов компании AT&T. Мы обратим внимание на мониторинг пакетов в силу его отличия от других проектов, рассмотренных выше.

Мониторинг пакетов проводился в сегменте сети Ethernet, соединяющем внутреннюю исследовательскую сеть компании AT&T с Internet в 1996 году. Мониторинг проводился для пакетов, передававшихся на порт 80, что отвечает протоколу HTTP. Содержимое таких пакетов сохранялось на диске. После первичного сбора данных другая программа собирала разрозненные пакеты в HTTP-запросы и ответы. Это весьма сложная задача и она будет рассматриваться в главе 14 (раздел 14.1). В совокупности объем HTTP-ответов и запросов составил 20 гигабайтов за 17 рабочих дней мониторинга. Именно запись полной информации позволила обеспечить доступ ко всем HTTP-заголовкам. Сравнивая тела сообщений-ответов, удалось ответить на вопрос о том, где и как изменяются со временем Web-ресурсы. Доступ к полной информации всех сообщений позволил осуществить исследование на уровне, недоступном при использовании журналов Web-серверов и прокси-серверов.

Изучались возраст ресурсов и частоты их модификаций на основе временных меток, рассмотренных в разделе 9.5.4. Заголовки **Date** и **Last-Modified** (когда таковые присутствовали) предоставляли информацию о возрасте ресурса. Рассматривая множество запросов для одного URI на основе заголовка **Last-Modified**, удается выявить, менялся ли ресурс в промежутке между последовательными обращениями к нему. В некоторых случаях заголовок **Last-Modified** отсутствовал, затрудняя изучение вопроса об изменении ресурса. В таких случаях было полезно иметь доступ ко всему сообщению. Сравнение контрольных сумм определенных объектов из HTTP-ответов позволило сделать выводы об изменении ресурса при последовательных запросах с одним и тем же URI. При этом также оказалось возможным изучение еще двух вопросов. Первое, сравнивая контрольные суммы ответов для запросов с разными URI, удалось выявить случаи, когда один и тот же ресурс был доступен по разным URI. Второе, сравнение тел ответов в случаях, когда ресурс изменялся в период между последовательными запросами, позволяло выяснить, *как* он изменялся. Например, было исследовано изменение текста HTML-ресурсов (выяснялось, менялся ли текст полностью или вносились незначительные изменения, например, менялись номера телефонов или гипертекстовые ссылки).

9.7. Резюме

Четыре исследования, рассмотренные в данной главе, иллюстрируют, какая информация может быть извлечена из результатов измерений Web-трафика. Каждая технология измерений позволяет получить ценную информацию о характеристиках Web-трафика. Каждая из рассмотренных технологий имеет ограничения, обусловленные способом и местом проведения измерений и составом доступных данных. Сбор информации обо всех компонентах Web весьма затруднителен. Кроме того, гетерогенная природа Web затрудняет обобщение результатов любых измерений. Несмотря на отмеченные препятствия, измерения играют чрезвычайно важную роль в оценке программного обеспечения компонентов Web, протокола HTTP, в анализе поведения пользователей, основных характеристик Web-ресурсов и Web-сайтов. Измерение трафика в различных местах и в разное время, а также анализ результатов измерений может восполнить ограничения используемых технологий измерений. В последующих главах результаты анализа измерений Web-трафика используются для описания параметров рабочей нагрузки Web.

10

Описание параметров рабочей нагрузки

Определение производительности Web достаточно затруднено на практике. Важные метрики производительности, такие как время ожидания ответа пользователем или число запросов, обрабатываемых сервером за единицу времени, зависят от взаимодействия многочисленных протоколов и компонентов программного обеспечения. Время ожидания, в свою очередь, влияет на реакцию пользователя, например, последний может отменить запрос при задержке ответа. Для определения производительности системы необходимо отделить предложенную нагрузку от свойств самой системы. *Рабочая нагрузка* определяется набором всех входных данных, полученных системой за определенный период времени. Количественные модели, описывающие основные характеристики рабочей нагрузки, могут использовать ряд методов определения производительности, включая выполнение тестов и моделирование. Например, модель рабочей нагрузки Web может использоваться для генерации запросов при сравнении различных прокси- и Web-серверов. Кроме того, модели рабочей нагрузки Web используются при проведении имитационного моделирования новых технологий, используемых для повышения производительности.

Эта глава посвящена характеристикам рабочей нагрузки Web. Разработка модели рабочей нагрузки Web включает три основных шага: выявление основных параметров нагрузки, анализ измеренных данных для получения количественных значений этих параметров и проверка модели в реальных ситуациях. Начнем обсуждение с мотивации описания и обзора параметров рабочей нагрузки Web. Построение модели нагрузки требует понимания техники статистического анализа измеренных данных и представления ключевых свойств Web-трафика. Хотя такие характеристики, как размеры Web-ресурсов или число встроенных в Web-страницу изображений, меняются при различных измерениях, но определенные статистические характеристики достаточно устойчивы для всех проводимых опытов. Распределения вероятностей обеспечивают эффективный способ описания результатов измерений в сжатом виде. Поэтому перед обсуждением деталей описания рабочей нагрузки Web мы кратко рассмотрим распределения вероятностей.

Мы опишем основные свойства рабочей нагрузки Web на основе результатов измерений трафика:

- **Характеристики HTTP-сообщений.** В HTTP определено несколько методов запросов и множество ситуаций, связанных с ответами. Относительная популярность методов запросов и ситуаций, связанных с ответами, иллюстрирует, как протокол используется на практике. Реальная модель Web-нагрузки должна основываться на репрезентативной смеси запросов и ситуаций, связанных с ответами.

- **Характеристики ресурсов.** Web-клиенты запрашивают ресурсы с различными свойствами, включая тип содержания, размер, популярность и частота модификации. Кроме того, число встроенных ресурсов существенно различается для различных Web-страниц. Эти характеристики оказывают существенное влияние на нагрузку, связанную с хранением и передачей Web-ресурсов.
- **Поведение пользователей.** При посещении Web-сайта пользователи запрашивают Web-страницы с некоторой задержкой между последовательно загружаемыми Web-страницами. Поведение пользователя оказывает значительное влияние на количество запросов, получаемых Web- и прокси-серверами, а также на трафик в сети.

Далее мы обсудим, как эволюция Web сказывается на свойствах рабочей нагрузки. Затем остановимся на том, как использовать параметры рабочей нагрузки для генерации потока HTTP-запросов, который отражает основные свойства реального Web-трафика. Эти синтезированные модели нагрузки должны использоваться в различных тестах для определения характеристик производительности прокси- и Web-серверов. Точность тестирования зависит от детального анализа Web-ресурсов и поведения пользователей. Следует отметить, что детальный анализ Web-трафика и поведения пользователей может привести к нарушению прав пользователей на неприкосновенность личной информации. В этой главе будут подробно обсуждаться взаимоотношение неприкосновенности личной информации пользователей и определение характеристик рабочей нагрузки Web.

10.1. Характеристики рабочей нагрузки

Модель рабочей нагрузки состоит из набора параметров, определяющих ключевые особенности нагрузки, на которые влияют расположение ресурсов и производительность системы. В этом разделе будут обсуждены побудительные мотивы для описания и определения параметров модели рабочей нагрузки Web.

10.1.1. Применение модели нагрузки

Модель нагрузки может использоваться в различных задачах оценки производительности, например:

- **Выявление проблем с производительностью.** У Web-сервера могут быть проблемы с производительностью, включая недопустимо большое время ответа, низкая пропускная способность в условиях, возникающих в определенное время суток. Определение причин требует тестирования сервера при реальной нагрузке. Нахождение решения проблемы может потребовать многократного тестирования в условиях такой нагрузки.
- **Компоненты для тестирования производительности.** Решение о приобретении прокси-сервера или Web-сервера зависит от цены и производительности. Сравнение производительности различных конфигураций требует приложения одинаковых воспроизводимых нагрузок к каждой из них. Нагрузка должна отражать основные свойства Web-трафика для того, чтобы результаты моделирования давали представление о том, как данная конфигурация будет работать на практике. Тесты производительности можно также использовать для оценки новых конфигураций еще до их развертывания.

- **Планирование производительности.** Производительность зависит от пропускной способности сети, мощности процессора, объема дисковой подсистемы, имеющейся оперативной памяти прокси- и Web-серверов. Например, перегруженный Web-сайт можно реплицировать на несколько компьютеров. Решение о том, сколько компьютеров нужно для сайта, определяется компромиссом между ценой и производительностью. Оценка производительности конфигурации зависит от наличия точной модели ожидаемой нагрузки.

Модели нагрузки имеют ряд преимуществ и недостатков по сравнению с другими популярными методами определения производительности Web. Один из естественных подходов заключается в построении запросов, непосредственно исходя из имеющихся журналов Web-компонентов. Преимущество повторного использования журналов состоит в воспроизведении известной рабочей нагрузки, избегая промежуточных шагов анализа трафика. Этот подход особенно полезен для понимания специфичных ситуаций, вызвавших появление проблем с производительностью. Однако повторное использование журналов не обеспечивает достаточной гибкости, если необходимо изменять нагрузку. Кроме того, использование журналов не позволяет четко разделить факторы, обусловленные рабочей нагрузкой и производительностью системы. Записи в журналах зависят от текущей производительности системы. Использование этих данных при других конфигурациях системы может привести к неправильным выводам. Например, анализ данных, полученных на перегруженном сервере, может выявить длительные задержки между последовательными запросами и малое число обращений пользователей на сеанс работы с сервером. Однако эти свойства могу оказаться результатом исключительно низкой производительности сервера, а не факторов, определяющих рабочую нагрузку сервера.

Другой подход заключается в передаче запросов с максимальной частотой для испытания сервера в экстремальных условиях. Такое стрессовое тестирование является важной частью оценки новой системы. Однако производительность системы под такой нагрузкой может не дать представление о том, как система будет работать при реальной нагрузке. Генерация потоков запросов с помощью модели нагрузки исключает некоторые недостатки представленных выше подходов. В противовес нагрузке, сгенерированной по журналу, синтезированная нагрузка создается на основе математической модели, которая может быть протестирована, проанализирована и критически рассмотрена. В противовес стрессовой нагрузке, синтезированная нагрузка имеет возможности для представления основных свойств реального Web-трафика. Что особенно важно, модели рабочей нагрузки представляют возможности для оценки производительности системы в контролируемом режиме с помощью изменения значений параметров распределений вероятностей. Эти эксперименты позволяют исследовать влияние отдельных параметров на производительность системы в целом, а также узнать, как меняется поведение системы при изменении рабочей нагрузки.

10.1.2. Выбор параметров рабочей нагрузки

Для того чтобы убедиться, что модель нагрузки отражает реальную нагрузку, параметры модели должны иметь определенные свойства:

- **Независимость от системы.** Параметры модели не должны зависеть от используемой системы, например, от реализация прокси-сервера или Web-сервера. Модель нагрузки не должна включать такую информацию, как время реакции пользователя, производительность сервера, процент утерянных паке-

тов, т.к. эти параметры зависят от серверной платформы и нагрузки на сеть. Однако трафик, синтезированный с помощью модели нагрузки, может использоваться для оценки этих параметров для отдельно взятой платформы.

- **Надлежащий уровень детализации.** Для оценки производительности системы параметры должны представлять рабочую нагрузку на соответствующем уровне детализации. Например, модель рабочей нагрузки на сетевом уровне может включать параметры, связанные с размерами IP-пакетов и номерами пакетов. Модель рабочей нагрузки на прикладном уровне может включать время между успешными HTTP-запросами клиента и размер ответных HTTP-сообщений, но не внутреннее их содержимое.
- **Независимость от других параметров.** Зависимость между параметрами нагрузки усложняет работу по получению простой модели нагрузки, которая достаточно точно представляет реальный Web-трафик. Однако выбрать достаточно малое число независимых параметров на практике очень трудно. Например, размер ресурса обычно связан с типом содержания. В модель можно ввести дополнительный параметр для отражения этой зависимости. Например, модель может включать такие параметры, как размеры HTML-документов и размеры изображений.

Модели нагрузки меняются во времени вместе развитием систем. В таблице 10.1 приведен список параметров рабочей нагрузки, выбранных за несколько лет измерений Web-трафика. Первая категория охватывает основные характеристики HTTP-сообщений: методы запросов и коды ответов, что иллюстрирует, как протокол используется на практике. Вторая категория охватывает основные свойства Web-ресурсов, включая тип содержания, размер и популярность. Третья категория включает поведение пользователя, посещающего Web-сайт, включая время между обращениями различных пользователей и число гипертекстовых переходов на сеанс работы с Web-сайтом.

Таблица 10.1. Примеры параметров рабочей нагрузки Web

Категория	Параметр
Протокол	Метод запроса
	Код ответа
Ресурс	Тип содержания
	Размер ресурса
	Размер ответа
	Популярность
	Частота обновления
	Временное местоположение
	Количество встроенных ресурсов
Пользователи	Время между двумя сеансами
	Число гипертекстовых переходов на сеанс
	Время между запросами

Эти параметры определяют временные параметры запросов, поступающих на серверы. На практике различные параметры таблицы 10.1 не являются независи-

мыми друг от друга. Например, время между двумя гипертекстовыми переходами пользователя зависит от размера ответа на запрос. Дальнейшее изучение может обеспечить более детальное понимание зависимостей между параметрами. После краткого введения в статистику и в распределения вероятностей будет изложено текущее понимание характеристик нагрузки Web.

10.2. Статистики и распределения вероятностей

Описание рабочей нагрузки включает связывание каждого параметра модели рабочей нагрузки с количественными значениями, полученными на основе анализа результатов измерений. Хотя использование простых статистик, например, среднего, медианы и дисперсии достаточны для некоторых параметров нагрузки, однако распределения вероятностей обеспечивают более общий способ выяснить, как изменяются параметры в некотором диапазоне значений.

10.2.1. Среднее, медиана и дисперсия

Статистики, такие как среднее, медиана и дисперсия, описывают основные свойства многих параметров нагрузки Web. Рассмотрим файл регистрации сервера, в котором записывались метод запроса и код ответа для каждой Web-транзакции. Определение доли запросов, использующих метод **GET**, или доли ответов с кодом **200 OK** включает в себя число передач каждого типа. Другие параметры, такие как размер ответного Web-сообщения, имеют существенно более широкий диапазон значений. Вычисление доли ответов с каждым возможным размером сообщения было бы очень утомительным. Вместо этого, характеристики размеров ответов можно представить некоторой статистикой, например, средним размером ответа. Однако на практике среднее значение не охватывает всех характеристик изменчивости большинства параметров Web-нагрузки.

Фактически, когда параметр изменяется в широких пределах, среднее значение является очень обманчивой статистикой, так как может быть искажено достаточно малым количеством больших значений. Например, предположим, что сервер генерирует пять ответных сообщений с размерами 4100, 4700, 4200, 20000 и 4000 байтов, соответственно. Основанное на этих пяти ответах среднее значение равно 7400 байтам. Однако это значение не дает правильного представления о типичном размере ответа. Альтернативной статистикой является *медиана* — размер «серединного» ресурса — первая половина ресурсов имеет значения большие, чем медиана, а вторая половина — меньшие. В этом примере медиана равна 4200 байтам, что лучше описывает типичный размер ответа. Однако медиана не указывает на возможность того, что размер ответа может принимать очень большое значение. Последовательность 4100, 4700, 4200, 4800 и 4000 будут иметь ту же самую медиану.

Вычисление среднего и медианы улучшает картину. Факт того, что медиана много меньше среднего, говорит о наличии относительно малого числа больших значений. Например, первая последовательность имеет среднее значение 7400, а медиану 4200, что как раз и свидетельствует о наличии больших значений ответов сервера. В противоположность этому, вторая последовательность имеет среднее значение 4360, которое очень близко к медиане, равной 4200. Поэтому можно предположить, что размеры ответов в этом случае меняются не очень сильно. Кроме среднего и медианы имеются и другие статистики, такие как дисперсия и среднее квадратическое отклонение, определяющие, насколько сильно параметр отклоняет-

ся от своего среднего значения. Небольшие значения этих статистик говорят о том, что параметр остается близким к среднему значению, в то время как большие значения свидетельствуют о том, что параметр может принимать значения, существенно отличающиеся от среднего. Однако подобно среднему и медиане, дисперсия и среднее квадратическое отклонение дают только обобщенную характеристику параметра. Эти обобщенные характеристики не предоставляют достаточно информации для генерации Web-нагрузок, которые бы давали представление о том, как параметр меняется на практике.

10.2.2. Распределения вероятностей

Распределение вероятности определяет, как параметр меняется в широком диапазоне значений. Рассмотрим распределение $F(x)$ размеров ответов сервера. $F(x)$ представляет собой долю ответов, значения которых больше x байтов. Такая функция называется *дополнительной функцией распределения*. Значение $F(x)$, как показано на рис. 10.1, может быть определено непосредственно из результатов измерений. Для последовательности 4100, 4700, 4200, 20000 и 4000 $F(x)$ равна в 1 для интервала x от 0 до 3999, падает до 0,8 при x = 4000, уменьшается до 0,2 при x равном 4100, 4200, 4700, достигая нуля при x = 20000. В этом примере $F(x)$ изменяет значения на небольшом числе точек. На практике больший набор результатов измерений может привести к функции, которая меняется многократно.

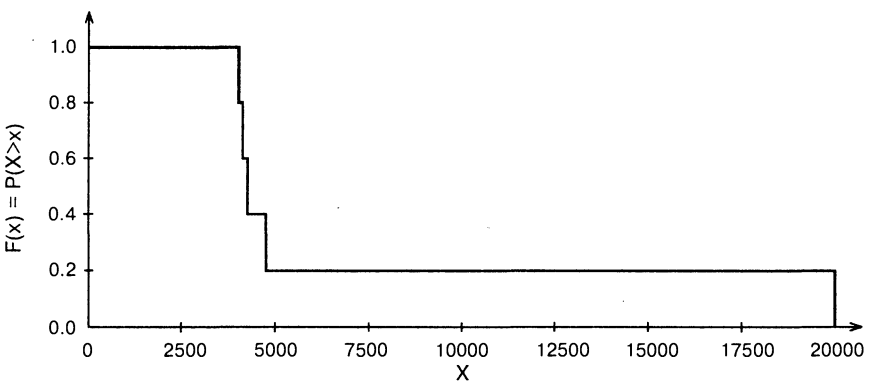


Рис. 10.1. Пример дополнительной функции распределения

Работа с распределением, построенным непосредственно по результатам измерений, может быть затруднена. Представление распределения в виде математического выражения упрощает описание параметров рабочей нагрузки в виде аналитической модели.

Несколько распределений вероятностей хорошо изучены и широко используются для описания рабочей нагрузки. Каждое из распределений может быть представлено относительно простой функцией, зависящей от одной или нескольких переменных. Одним из наиболее популярных распределений является экспоненциальное распределение, которое имеет вид:

$$F(x)=e^{-\lambda x}$$

со средним значением равным $1/\lambda$, где x может принимать значения большие или равные нулю. Другие распределения описываются функциями одной или более переменных, влияющих на форму распределения.

При связывании опытного распределения вероятности с теоретическим распределением требуется проверить гипотезу, что теоретическое распределение не противоречит опытным данным. Проверка гипотезы состоит из двух основных шагов. Сначала определяются параметры теоретического закона распределения. Например, значение λ для экспоненциального распределения может быть определено из среднего размера ответа, полученного на основе измеренных данных. Далее вычисляется значение критерия согласия для сравнения теоретического и экспериментального распределений. Значение критерия согласия определяет уровень совпадения теоретического и экспериментального распределений. Если совпадение не удовлетворительно, то можно рассмотреть другое теоретическое распределение. Например, экспоненциальное распределение не может точно описывать большинство параметров рабочей нагрузки в таблице 10.1. В некоторых случаях вообще затруднительно подобрать теоретическое распределение, соответствующее измеренным данным. Тогда можно попробовать представить различные части экспериментального распределения с помощью различных теоретических распределений. Например, небольшие значения ответа могут описываться одним распределением, а большие — другим. В течение многих лет сопоставление измеренных данных с теоретическими распределениями и проверка гипотез об уровне их соответствия является активной областью статистических исследований. Это подробно обсуждается в различных книгах [DS86, LK99].

Описывать разнообразие и эволюцию в таких сложных системах как Web чрезвычайно трудно. Измерения, проведенные в различных сегментах Web, могут привести к совершенно различным выводам о характеристиках нагрузки. Например, распределение размеров ответов среди клиентов, пользующихся беспроводной связью, может заметно отличаться от распределения размеров ответов клиентов, использующих широкополосные соединения. Кроме того, появление новых приложений и сервисов может изменить характеристики рабочей нагрузки Web. В некоторых случаях изменение параметров нагрузки может быть описано с помощью изменения параметров распределения вероятности без изменения вида распределения. Например, хотя средний размер ответа может меняться в зависимости от пользователя, но изменчивость размера ответа может быть одинакова. Далее при обсуждении параметров нагрузки будет выяснено, как характеристики нагрузки меняются во времени и будут найдены ее основные инвариантные свойства. Кроме того, будет обсуждено, какие изменения необходимо сделать в Web-приложениях, чтобы оказать влияние на некоторые параметры рабочей нагрузки в дальнейшем.

10.3. Характеристики HTTP-сообщений

Основные свойства HTTP-запросов и ответов позволяют выяснить, как протокол используется на практике. Статистический анализ методов запросов и кодов ответов может привести к созданию реальной картины Web-нагрузки и выявить проблемы отдельного сайта. Популярность определенных методов запросов и определенных кодов ответов меняется во времени по мере появления новых Web-приложений и перехода на протокол HTTP/1.1.

10.3.1. Методы HTTP-запросов

Знание того, какие методы запросов используются на практике, полезно для оптимизации серверных приложений для наиболее используемых методов, а также разработки реалистичных тестов для оценки производительности прокси- и Web-

серверов. В протоколе HTTP/1.1 определены методы **GET, HEAD, POST, PUT, DELETE, OPTIONS, TRACE** и **CONNECT,** как об этом говорилось ранее в главе 7 (раздел 7.2.1). Для подавляющего большинства HTTP-запросов используется небольшое число методов.

Характеристики трафика. Проведение большого числа экспериментальных исследований показало, что подавляющее большинство Web-запросов используют метод **GET** для загрузки ресурсов и запуска сценариев [AJ00, PQ00]. Небольшая доля HTTP-запросов использует метод **POST**, обычно для передачи данных форм. Другие методы запросов используются редко, так как они не инициируются типичными действиями пользователя при работе с браузером. Измерения выявили небольшое число запросов, использующих метод **HEAD**, которые могут быть сгенерированы вручную как часть операций по тестированию Web-сервера. Несмотря на доминирование запросов **GET**, точное распределение методов запросов меняется от сайта к сайту. Сайт с необычной смесью методов запросов может иметь производительность, отличную от других сайтов. Например, рассмотрим Web-сайт, который позволяет пользователям посылать и принимать сообщения электронной почты. Для отсылки сообщения электронной почты пользователь вводит текст в окне браузера и нажимает кнопку Submit. Такой Web-сайт может получать относительно большое число запросов **POST**, и эти сообщения могут включать большой объем данных. Выполняя тестирование сайта в предположении, что большинство запросов для получения данных с сайта используют метод **GET**, не даст репрезентативного описания нагрузки этого сайта. Знание того, что сайт может использовать необычные методы запросов, поможет выделить такую ситуацию, в то время как серверные тесты, использующие обычную смесь методов запросов, не обеспечивают определение того, как реально будет работать такой сайт.

Влияние распространения новых технологий. Распределение методов запросов может изменяться во времени, следуя за эволюцией Web-приложений. Многие Web-сайты позволяют пользователям отправлять формы разнообразного назначения, включая передачу запросов поисковым машинам или передачу сообщений электронной почты. Когда сайт, позволяющий пользователям отправлять данные, становится популярным, то количество запросов **POST** увеличивается. В процессе работы ряд приложений позволяет клиентам создавать новые ресурсы. Например, приложения, использующие Web Distributed Authoring and Versioning (WebDAV), позволяют группе пользователей объединяться для написания и редактирования документов, как это будет обсуждено ниже в главе 15 (раздел 15.5.2). Если это приложение становится популярным, то число запросов **PUT** и **DELETE** будет увеличиваться. Приложения WebDAV также предоставляют новые методы, которые используются все чаще по мере того, как растет их популярность. Кроме того, существующее приложения могут начать использовать методы запросов по-другому. Например, браузер или прокси-сервер перед выполнением запроса на ресурс могут использовать метод **HEAD** для проверки актуальности этого ресурса в кэше. Такое приложение будет подробно рассмотрено в главе 13 (раздел 13.1.2). Кроме того, появление инструментальных средств тестирования и отладки Web-компонентов может увеличить частоту использования метода **TRACE**.

10.3.2. Коды HTTP-ответов

Знание того, как сервер отвечает на клиентские запросы, является важной частью построения реалистичной модели Web-нагрузки. В протоколе HTTP определено большое число кодов ответов, разделенных на пять классов, как это обсуждалось в главе 7 (раздел 7.2.3).

Характеристики трафика. Ответом на успешный запрос в большинстве случаев (от 75% до 90%) является **200 OK** [AJ00, PQ00]. Следующим, наиболее часто возвращаемым кодом (от 10% до 30%), является **304 Not Modified**. Наиболее частыми кодами ответов из оставшихся являются 3xx (переадресация) и 4xx (ошибки клиента). Например, код **404 Not Found** возвращается, когда сервер не может сгенерировать запрашиваемый ресурс. Некоторые сайты используют переадресацию для выравнивания нагрузки нескольких серверов. Например, частота кода ответа **302 Found Redirection** меняется от сайта к сайту. Подобная статистика указывает на то, что тесты не должны рассчитывать, что в ответ на все успешные запросы будет получен код **200 OK**.

Необходимо учитывать разнообразные факторы, которые обусловливают данный код ответа. Ответ **304 Not Modified** возвращается, когда клиент проверяет актуальность копии ресурса в кэше. Web-сайт, где доминируют динамические ресурсы, не будет получать много запросов для проверки актуальности элементов кэша и следовательно не будет посылать много ответов с кодом **304 Not Modified**. С другой стороны, частота появления кода ответа **304 Not Modified** зависит как от частоты изменений ресурсов, так и от вероятности запроса на кэшированные ресурсы. Web-сайт со статическим содержанием, таким как HTML файлы и встроенные изображения, будет возвращать относительно большое число ответов с кодом **304 Not Modified**. Это характерно для сайтов, не использующих заголовки управления кэшированием ответов (т.е. **Expires**) для указания того, как долго клиенту можно благополучно возвращать кэшированный ответ. Без этой информации клиент должен регулярно проверять актуальность кэшированных ответов сервера. В результате число ответов с кодом **304 Not Modified** увеличивается, особенно если ресурс запрашивается большим числом пользователей.

Большое число ответов с кодами класса 4xx (ошибки клиента) может указывать на проблемы с организацией Web-сайта или с другими Web-страницами, на которых имеются гипертекстовые ссылки на данный сайт. Рассмотрим, например, что произойдет, когда администратор реорганизует Web-сайт. Некоторые URL, которые присутствуют на Web-страницах или в закладках пользователей, могут теперь указывать на отсутствующие ресурсы. Большое число ответов **404 Not Found** указывает на то, что многие запросы используют старые URL. Основываясь на этой информации, администратор сайта может настроить сервер так, чтобы последний трактовал старые гиперссылки как синонимы новых или переадресовывал на новый URL. В некоторых случаях администратор может идентифицировать Web-страницы с устаревшими гиперссылками. Сообщение с кодом **404 Not Found** может также говорить о том, что в HTML-файле имеется гипертекстовая ссылка с опечаткой, например, пропущенный символ в URL или ошибочное написание (например, **http://www.bar.com/necesary.fig** вместо **http://www.bar.com/necessary.fig**)

Влияние распространения новых технологий. Некоторые коды ответов не появляются, пока клиент не использует определенные заголовки запросов. Следовательно, частота появления кодов ответов может зависеть от построения Web-клиентов. Например, рассмотрим код ответа **206 Partial Content**, используемый в случае, когда сервер возвращает диапазон байтов запрашиваемого ресурса, как этот было описано ранее в главе 7 (раздел 7.4.1). Запросы на диапазоны будут становиться все более распространенными по мере того, как реализации прокси-серверов станут поддерживать HTTP/1.1. Кроме того, браузеры могут использовать дополнительные модули, генерирующие запросы на диапазоны. Например, модуль, отображающий PDF-файл, может выдавать запросы на диапазоны для доступа

к отдельным страницам документа. Новые технологии, повышающие производительность браузеров и прокси-серверов, могут также увеличить число запросов на диапазоны. Например, клиент может кэшировать часть ресурса большого объема и выдавать запрос на диапазон для получения оставшейся части [STR99]. Клиент, выдающий запрос на диапазон в паре с сервером, возвращающим диапазон байтов, увеличивают частоту появления кода ответа **206 Partial Content**.

Частота кода ответа **302 Found Redirection** варьируется от сайта к сайту. Некоторые Web-сайты используют переадресацию как способ выравнивания нагрузки для группы дублирующих друг друга серверов. В этом случае в начале обработки запроса определяется сервер, которому необходимо перенаправить клиентский запрос, основываясь на загрузке серверов или близости к клиенту. Ответ переадресации дает указание клиенту повторить запрос к альтернативному серверу. Растущие масштабы Web-приложений позволяют предположить, что дублирование серверов станет повсеместным, особенно для популярных Web-сайтов. Однако переадресация на уровне HTTP — это только один из путей координации обработки клиентских запросов. Если другие подходы станут более популярными, то частота запросов переадресации может со временем не увеличиться, а уменьшиться. Различные подходы к переадресации клиентских запросов на дублирующие серверы будут детально обсуждаться в главе 11 (раздел 11.12). Код ответа **302 Found Redirection** может также возвращаться Web-сайтом, который при попытке доступа к несуществующему ресурсу возвращает страницу, используемую по умолчанию, вместо кода ответа **404 Not Found**.

10.4. Характеристики Web-ресурсов

Для моделирования рабочей нагрузки Web необходима детальная информация о характеристиках Web-ресурсов. Ресурсы характеризуются размером, популярностью, частотой изменений. Кроме того, HTML-страницы характеризуются числом встроенных изображений. Подытоживая характеристики Web-ресурсов, мы обсудим типы и размеры содержания, размеры ответов, популярность, частоту модификаций и число встроенных ресурсов.

10.4.1. Типы содержания

Web-сайты обеспечивают доступ к разнообразным ресурсам с различными типами содержания. Статистические данные о типах содержания дают информацию о видах данных, имеющихся на Web-сайте. Тип содержания имеет также непосредственную связь с другими ключевыми параметрами рабочей нагрузки, такими как размер ресурса и частота модификации. Кроме того, некоторые типы содержания, такие как текст, поддаются сжатию для уменьшения размера ответного сообщения, в то время как другие типы содержания, например, изображения, уже находятся в сжатом виде.

Характеристики трафика. На большинстве Web-сайтов преобладают текстовые ресурсы (типы **text/plain** и **text/html**) и изображения (типы **image/jpeg** и **image/gif**) [AW97, DFKM97, AJ00]. Оставшиеся типы содержания представляют сравнительно малую долю ресурсов. Эти типы содержания представляют собой документы Postscript и PDF, сценарии JavaScript, апплеты Java, аудио и видео. Однако соотношение типов содержания может существенно меняться от сайта к сайту. Например, один Web-сайт, содержащий архивные документы, может иметь большое число Postscript-

и PDF-документов, другой Web-сайт может интенсивно использовать апплеты Java. Кроме того, популярность некоторых типов содержания может варьироваться в различных пользовательских группах. Пользователи с низкой скоростью подключения к Internet предпочитают реже загружать изображения или большие документы, в то время как технические специалисты, работающие на платформе UNIX, могут отдавать предпочтение большим файлам в формате Postscript. В некоторых случаях смесь типов содержания, наблюдаемая клиентом, может зависеть от настроек прокси-сервера. Например, прокси сервер может отфильтровать некоторые типы содержания для исключения загрузки апплетов, запуск которых может привести к нарушению безопасности.

Влияние распространения новых технологий. Появление новых приложений может оказать неожиданное и глубокое влияние на распределение типов содержания. Например, растущий интерес к мультимедиа в Web привел к увеличению числа сайтов с аудио и видео [AS98].

10.4.2. Размеры ресурсов

Размеры Web-ресурсов влияют на требования к объему памяти на Web-серверах и приводят к перегрузке кэширующих прокси-серверов и браузеров. Кроме того, размеры ресурсов влияют на загрузку сети и задержки при доставке ответных сообщений.

Характеристики трафика. Хотя точное распределение размеров ресурсов меняется в зависимости от сервера и от времени, по анализ Web-измерений позволяет выявить некоторые общие характеристики. В общем случае средний размер ресурса относительно мал, хотя небольшая доля ресурсов имеет достаточно большой размер. Текстовые документы, HTML-документы и файлы с изображениями составляют основную часть ресурсов большинства Web-сайтов. Такие ресурсы обычно имеют размер меньший, чем ресурсы других типов, особенно аудио и видео. Средний размер HTML-файлов составляет 4–8 Кбайтов, а средний размер изображения около 14 Кбайтов, хотя точное значение изменяется от сайта к сайту [Pit99, AJ00]. Хотя большинство таких файлов имеют малый размер, некоторые из них бывают очень большими. HTML-файлы имеют значение медианы около 2 Кбайтов, что существенно меньше, чем средний размер (4–8 Кбайта). Это говорит о большом разбросе размеров HTML-файлов. Размеры изображений также существенно меняются в зависимости от источника изображения и его назначения. Логотипы, навигационные изображения, рекламные объявления обычно имеют небольшие размеры, в то время как фотографии бывают сравнительно большого размера. Большой разброс размеров ресурсов также характерен и для других типов содержания. Кроме того, Web-сайт содержит смесь различных типов содержания, что приводит к еще большему разбросу размеров ресурсов. Например, сайт, содержащий смесь текста, изображений и данных мультимедиа, может иметь размеры ресурсов в диапазоне от нескольких сот байтов до нескольких гигабайтов.

Хотя значение медианы и описывает типичный ресурс, по распределение вероятности дает лучшее описание размеров ресурсов. Знание распределения размеров ресурсов полезно для принятия решения о выделении дискового пространства на сервере или на прокси-сервере. Кроме того, тесты, оценивающие производительность Web-серверов и прокси-серверов, должны учитывать присущий ресурсу диапазон изменений размеров. Высокая изменчивость размеров ресурсов обычно описывается распределением Парето:

$$F(x)=(k/x)^a,\ x \geqslant k$$

где значение параметра a определяет форму распределения, а значение параметра k – масштаб. Распределение имеет среднее значение равное $ka/(a-1)$ для $a>1$. На

рис. 10.2 приведен график распределения Парето для двух значений параметра a (1.1 и 1.5) со средним значением, равным единице. При значении a = 1.5 около 80% ресурсов имеют размер меньший единицы. Оставшееся 20% ресурсов имеют достаточно большой размер, в результате среднее значение размера равно 1. При значении a = 1.1 разница еще более существенна: менее 10% ресурсов имеют размер, больший единицы.

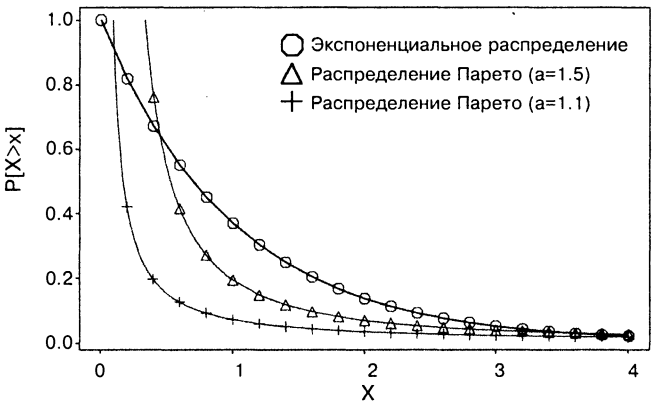


Рис. 10.2. Экспоненциальное распределение и распределение Парето (со средним значением, равным 1)

Для сравнения на рис. 10.2 приведен также график для экспоненциального распределения $F(x)=e^{-\lambda x}$ со средним значением равным единице (λ = 1). Экспоненциальное распределение широко используется в математическом анализе и имеет существенно меньшую изменчивость значений, чем распределение Парето. При экспоненциальном распределении только 60% ресурсов имеют значения меньше единицы. Хотя оставшиеся значения и больше единицы, но лишь малое число ресурсов принимают значения большие 4 или 5. Это легко видеть на рис. 10.3, где изображены те же самые распределения, но в логарифмическом масштабе по обеим осям. Кривая экспоненциального распределения спадает ниже значения 10^{-10} для

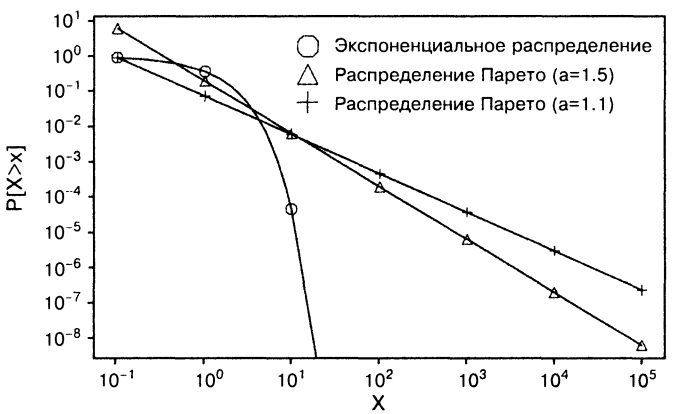


Рис. 10.3. Экспоненциальное распределение и распределение Парето в логарифмическом масштабе

$x > 40$. Это означает, что только один из десяти миллиардов ресурсов может иметь значение больше 40. Значения обоих распределений Парето в противоположность экспоненциальному распределению уменьшаются только до 10^{-3}, демонстрируя, что один из тысячи ресурсов принимает значение большее 40. Очень большие значения размеров ресурсов имеют место гораздо чаще для распределения Парето, чем для экспоненциального распределения, при одинаковом среднем значении.

Распределение Парето при значениях a от 0 до 2 является распределением с *медленно убывающим «хвостом»*. «Хвост» показывает насколько медленно распределение $F(x)$ уменьшается при больших значениях x. Распределение Парето в двойном логарифмическом масштабе линейно (рис. 10.3). Наклон графика в таком масштабе определяется значением коэффициента a. Меньшие значения a определяют меньший наклон, что соответствует большей изменчивости размеров ресурсов. Например, на рис. 10.3 график распределения при $a = 1.1$ лежит выше графика при $a = 1.5$ для больших значений x. Большое число измерений показало, что хвост распределения Парето достаточно хорошо описывает изменчивость размеров у Web-ресурсов. Хотя значения a и отличаются от одной серии измерений к другой, но большинство исследований показало, что значение a находится в диапазоне от 1.0 до 1.5 [AW97, CB97, BBBC99]. Распределение имеет очень большое среднее значение при значении a близком к единице, что делает его практически неприменимым для описании размеров ресурсов.

Несмотря на хорошую точность описания больших ресурсов, распределение Парето не слишком хорошо описывает оставшиеся ресурсы. Гибридная модель, объединяющая несколько распределений, обеспечивает лучшее соответствие результатам измерений. Данные гораздо лучше описываются *логарифмически нормальным распределением* [BC98]. *Нормальное* распределение имеет колоколообразный вид с центром, соответствующим среднему значению; половина значений распределения будут меньше среднего, а вторая половина — больше среднего значения. У логарифмически нормального распределения нормальное распределение имеет логарифм x. Использование логарифма увеличивает долю значений, которые меньше среднего, как показано на рис. 10.4. Для сравнения на рисунке также приведен график экспоненциального распределения с тем же средним. Логарифмически нормальное распределение достаточно точно описывает характеристики Web-ресурсов за исключение очень больших значений на хвосте распределения. Распределение

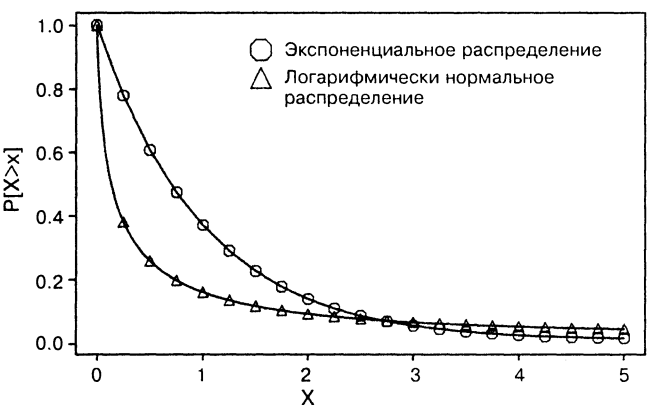


Рис. 10.4. Логарифмически нормальное распределение

Парето вместе с логарифмически нормальным распределением образуют модель, описывающую значения размеров Web-ресурсов во всем диапазоне значений.

Влияние распространения новых технологий. Размеры Web-ресурсов могут меняться со временем. Пользователи с высокоскоростным подключением к Internet гораздо чаще запрашивают ресурсы большого размера [AFJ99]. Это, в свою очередь, может позволить разработчикам создавать Web-страницы бо**л**ьшего размера. Однако тенденция увеличение размера Web-страниц необязательно влечет увеличение изменчивости размеров ресурсов. Изменение набора используемых типов содержания может оказать существенное воздействие на размеры Web-ресурсов. Например, аудио- и видеоресурсы обычно имеют весьма значительные размеры. Увеличение доли мультимедийного содержания Web может изменить распределение размеров ресурсов. Однако несмотря на изменения видов ресурсов, имеющихся в Web, чрезвычайно высокая изменчивость размеров ресурсов остается всегда. Распределения с медленно убывающими хвостами наблюдаются не только для размеров Web-ресурсов, но и, например, для размеров файлов на компьютерах под управлением операционной системы Unix или для времени выполнения программ на компьютерах [HB97].

10.4.3. Размеры ответов

Распределение размеров ресурсов отражает изменчивость размеров файлов, хранящихся на Web-серверах. Однако нагрузка на сервер и сеть зависит от того, какие из этих ресурсов в действительности передаются по запросам клиентов. При анализе производительности сервера и сети размер ответных сообщений является более важным фактором. Число байтов в ответном сообщении определяет часть пропускной способности, необходимую для обработки клиентского запроса. Кроме того, размер ответа определяет пропускную способность используемого TCP-соединения. Небольшой размер скользящего окна на фазе медленного старта TCP ограничивает скорость передачи для многих ответных сообщений. Размер ответа также определяет относительный вклад различных источников задержки при обработке клиентского запроса. Затраты на преобразование доменного имени в IP-адрес сервера и установление TCP-соединения составляют основную часть задержки при передаче коротких сообщений. В противовес этому при передаче длинных сообщений более важна загруженность сервера и сетевого соединения.

Влияние распространения новых технологий. Размеры ответов могут отличаться от размеров ресурсов по ряду причин. Во-первых, некоторые ответные сообщения не пересылают ресурсы. Например, ответы **204 No Content** и **304 Not Modified** не содержат тела сообщения. Эти сообщения содержат не более нескольких сот байтов в заголовках. Как было отмечено ранее в разделе 10.3.2, ответ **304 Not Modified** составляет существенную часть ответных сообщений. Во-вторых, некоторые Web-ресурсы никогда не запрашиваются и, следовательно, не влияют на распределение размеров ответных сообщений. Проведенные исследования выявили, что в среднем ресурсы большего размера являются менее популярными, чем ресурсы меньшего размера. Например, при просмотре Web-страниц пользователь не будет часто скачивать Postscript- и PDF-файлы. В третьих, некоторые ответы могут быть прерваны до того, как завершится их передача, что укорачивает размер ответа. Пользователи чаще прерывают длинные ответы, чем короткие. Эти факторы приводят к тому, что обычно размер ответа типичного сообщения меньше, чем размер типичного ресурса.

Исследования показали, что медиана распределения размеров ответов на несколько сот байтов меньше, чем медиана распределения размеров ресурсов [BBBC99, AFJ99]. Однако размер ресурса является первичным фактором, воздействующим на размер ответных сообщений. Оба распределения вероятности очень похожи. Подобно размерам ресурсов, размеры ответов могут быть представлены комбинацией логарифмически нормального распределения и распределения Парето [BC98, BBBC99]. Кроме того, подобно распределению размеров ресурсов распределение размеров ответов также имеет медленно убывающий хвост. Переменная a в распределении Парето определяет убывание хвоста. Оба распределения вероятностей имеют значение a в диапазоне от 1.0 до 1.5, указывая на наличие очень медленно убывающих хвостов [Mah97, CB97, AFJ99]. Это говорит о том, что размеры ответа, аналогично размерам ресурса, покрывают весьма широкий диапазон значений.

Некоторые клиентские запросы обрабатываются кэширующими прокси-серверами без привлечения исходного сервера. Любая зависимость между размером ресурса и вероятностью его кэширования прокси-сервером может изменить распределение размеров ответов исходного сервера. Более вероятно, что будут кэшироваться наиболее популярные ресурсы. Более высокая популярность небольших ресурсов позволяет предположить, что именно они и будут кэшироваться прокси-сервером. Кроме того, прокси-серверы с ограниченным объемом дисковой памяти могут не кэшировать ресурсы большого размера, еще более увеличивая вероятность того, что ресурс большого размера будет передаваться исходным сервером. Однако при проведении исследований часто предполагается, что ресурсы малого размера запрашиваются не намного чаще, чем ресурсы большого размера, а прокси-серверы имеют достаточно дисковой памяти для кэширования сообщений больших размеров. В результате не выявляется значительной разницы между размерами ответов, переданных кэширующими серверами, и размерами ответов, переданных исходными серверами.

Влияние распространения новых технологий. Точное соотношение между размерами ресурсов и их популярностью может зависеть от ситуации. Клиент с низкоскоростным подключением к Internet будет намного реже запрашивать ресурсы большого размера, а прокси-сервер, обслуживающий такого клиента, может принять решение не кэшировать ответы большого размера. С другой стороны, все больше и больше клиентов получают высокоскоростной доступ через кабельные модемы и DSL. Такие пользователи будут более склонны запрашивать большие ресурсы. Это может ослабить существующую корреляцию между размером ресурса и его популярностью. Перечисленные конкурирующие факторы говорят о том, что неразумно предполагать, что распределение размеров ресурсов такое же самое, как и распределение размеров ответов. Реальная модель рабочей нагрузки должна учитывать обе характеристики и отслеживать, как соотношение между ними меняется во времени.

10.4.4. Популярность ресурсов

Популярность различных ресурсов оказывает важное влияние на производительность. Кэширование в браузере и на прокси-сервере наиболее эффективно, если большинство запросов делается на небольшое число ресурсов. Клиент может заранее загрузить ряд ресурсов, участвующих в дальнейших запросах. Популярность также воздействует на нагрузку сервера из-за необходимости ответов на большее число запросов. Наиболее популярные ресурсы скорее всего будут находиться в оперативной памяти сервера, что исключает необходимость их чтения с жесткого диска. Сервер может сохранять наиболее популярные ответы, сгенерированные динамически. Например, поисковый сервер может многократно получать

одинаковые запросы. Результаты поиска по этим наиболее популярным запросам могут быть сохранены на сервере. На практике сервер может не иметь достаточно памяти для хранения большого числа ответов. Компромисс между объемом памяти и повышением производительности зависит от распределения популярности ресурсов.

Характеристики трафика. Популярность ресурса определяется как отношение числа запросов на этот ресурс к общему числу запросов. Функция $P(r)$ описывает долю запросов, относящихся к каждому из ресурсов. Рассмотрим в качестве примера Web-сайт со 100 ресурсами. Если все 100 ресурсов имеют одинаковую популярность, то $P(r) = 0.01$ для $r = 1, 2, ..., 100$. Кривая, изображенная на рис. 10.5, указывает на то, что некоторые ресурсы значительно более популярны, чем другие. В этом примере на долю наиболее популярного ресурса приходится 20% запросов из набора из 100 ресурсов. Доля второго по полярности ресурса составляет около 10%. Сайт получает очень мало запросов на малопопулярные ресурсы. График, приведенный на рис. 10.5, представляет типичное распределение популярности Web-ресурсов сайта.

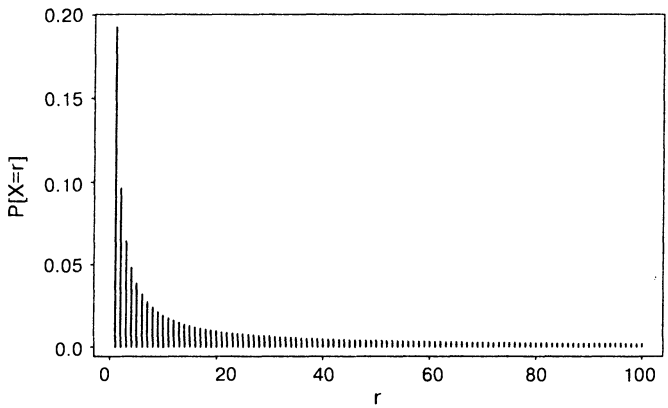


Рис. 10.5. Закон Зипфа

Доля запросов на ресурс обратно пропорциональна его рангу. Распределение соответствует *закону Зипфа* [Zip49]:

$$P(r)=kr^{-1}$$

где k — коэффициент пропорциональности, нормирующий сумму $P(r)$ к единице. Закон Зипфа используется для описания, например, частоты появления различных слов в документе; слова "the" или "and" встречаются в документе на английском языке гораздо чаще, чем "jejune". В более общем случае распределение Зипфа имеет вид:

$$P(r)=kr^{-c}$$

где c — некоторая константа. Меньшее значение c соответствуют меньшему разбросу популярности в наборе ресурсов. Предельный случай, когда c=0, соответствует случаю равной популярности всех ресурсов.

Значение c варьируется для разных прокси- и Web-серверов [CBC95, ABCdO96, BCF+99, PQ00]. В ранних исследованиях запросов к Web-серверам были получены значения с близкие к единице. Последние исследования оценивают эту величину

в диапазоне от 0.75 до 0.90. Кэширование в браузерах и прокси-серверах приводит к уменьшению числа запросов, достигающих Web-сервера для наиболее популярных ресурсов. В результате доля запросов к некэшируемым ресурсам существенно возрастает. Для Web-сервера запросы, перехваченные прокси-сервером, приводят к меньшим значениям константы *c*. Прокси-серверы обрабатывают запросы для различных Web-сайтов, что обычно связано с большим числом ресурсов со сравнимой популярностью. В дополнение к этому, эффективность кэширования браузером направлена на уменьшение числа повторяющихся запросов клиента к одному и тому же ресурсу. Тем не менее ресурсы, получаемые от прокси- и Web-серверов имеют широкий разброс по популярности.

Аналогичные результаты получены и для Web-сайтов — небольшое число популярных Web-сайтов получают наибольшее число запросов. Разброс в популярности оказывает важное влияние на производительность. Во-первых, локальный DNS-сервер скорее всего имеет кэшированную копию соответствий доменных имен и IP-адресов для большинства популярных сайтов. Это позволяет устранить задержки, связанные с повторными обращениями к другим DNS-серверам за исключением случаев, когда кэшированная DNS-запись устаревает. Во-вторых, на практике самые загруженные Web-сайты реплицируются на нескольких Web-серверах для того, чтобы приблизить содержимое к клиентам. Поскольку эти сайты определяют значительную часть трафика, то эффективная схема репликации этого небольшого числа сайтов может привести к существенному снижению сетевой нагрузки и уменьшению времени ожидания ответов пользователями. В-третьих, новые технологии повышения производительности не нужно развертывать на большом числе сайтов, чтобы иметь значительный выигрыш. Достаточно избирательного развертывания на небольшом числе наиболее популярных сайтов.

Влияние распространения новых технологий. Распределение Зипфа получило широкое распространение при описании результатов измерений в Web, с помощью него описывается поведение ряда других сложных систем вне Web. Однако значения параметров распределения Зипфа меняются со временем и от сайта к сайту. Изменения в распределении популярности ресурсов на Web-сайте или в исследуемом наборе сайтов оказывают влияние на эффективность кэширования. Менее асимметричные распределения с меньшей долей запросов на наиболее популярные ресурсы делают кэширование менее эффективным. Разнородность пользовательских сообществ приводит к тому, что большая часть ресурсов, имеющихся в Web, станет относительно популярной. Например, чем больше людей в мире подключатся к Web, тем больше новых сайтов будут обеспечивать эти пользовательские сообщества. Эти Web-сайты могут оказаться чрезвычайно популярными в некоторых странах и регионах. Однако другие социальные факторы с течением времени могут привести к еще большей неоднородности в популярности, когда пользователи будут иметь легкий доступ к Web по всему миру. В любом случае эти изменения популярности должны воздействовать на кэширование ресурсов для снижения нагрузки на серверы и сети.

10.4.5. Изменения ресурсов

Web-ресурсы изменяются со временем в результате модификации их содержания. Частота изменений и разница между последовательными версиями ресурсов оказывают существенное влияние на множество Web-приложений. Например, поисковые серверы зависят от спайдеров, индексирующих Web-страницы. Если Web-страницы изменятся на исходном сервере, то поисковый сервер вернет некор-

ректный результат в ответ на запрос пользователя. Зная частоту изменений ресурсов на сайте и значимость этих изменений можно определить частоту индексирования Web-страниц спайдером. Точно так же изменение содержимого ресурсов влияет на производительность Web-кэширования. Зная, что некоторые виды ресурсов изменяются чаще, чем другие, можно сформулировать политику кэширования для браузеров и прокси-серверов. Например, ресурсу, который меняется менее часто, может быть отдано предпочтение при кэшировании, обновляя его с исходного сервера менее часто.

Характеристики трафика. Любой ресурс может изменяться между двумя последовательными запросами, а сценарии могут создавать динамически формируемые ответы любых типов. На практике предполагается, что частота изменений ресурса зависит от типа содержания. На большинстве Web-сайтов изображения меняются не слишком часто. Некоторые Web-сайты хранят встроенные изображения на отдельном компьютере, предназначенном для статического содержания, как это будет детально рассмотрено в главе 11 (раздел 11.13). Например, HTML файл **http://www.foo.com/index.html** может иметь встроенное изображение **http://www.foo.com/pic.gif**. Точно так же текстовые и HTML-ресурсы часто хранятся как статические файлы на исходном сервере. Эти ресурсы меняются только в случае модификации файлов. Однако текстовые и HTML-файлы меняются гораздо чаще, чем изображения. Иногда текстовые и HTML-файлы создаются сценариями в ответ на запросы пользователей. Например, пользователь может обратиться к поисковому серверу, который ищет информацию по ключевым словам. В ответ поисковый сервер возвращает HTML-файл с гиперссылками на найденные Web-страницы. Повторение запроса на следующий день может привести к созданию HTML-файла с другим списком гиперссылок.

Некоторые ресурсы изменяются регулярно, например, ежеминутно, каждые четверть часа, ежечасно или ежедневно [DFKM97, PQ00]. Приведем примеры регулярно изменяемых Web-ресурсов с прогнозами погоды или сведениями о котировках акций, обновляемыми через заданное время. Администратор сайта может использовать периодическую природу этих обновлений для сокращения нагрузки на сервер. Эффективное использование заголовков кэшированных ответов в HTTP/1.1 может снизить частоту запросов для проверки актуальности кэшируемых копий ресурсов. Например, заголовок **Expires** может указывать время, когда ресурс должен быть обновлен. Это позволяет уверенно вернуть в ответ на запрос кэшированный документ, будучи уверенным, что он не будет изменен на исходном сервере. После истечения времени, заданного в заголовке **Expires,** любой запрос будет направляться исходному серверу для получения обновленной копии. Точное задание времени в ответных HTTP-сообщениях может уменьшить загрузку исходного сервера и сократить задержку, испытываемую пользователем.

В модели рабочей нагрузки необходимо учитывать частоту изменений ресурсов. Для некоторых Web-приложений существенен и вид изменений. Например, HTML-файл может включать счетчик, показывающий число обращений к Web-странице. Этот счетчик изменяется при каждом запросе. В некоторых случаях изменение не являются семантически важным. Пользователю не обязательно знать точное значение счетчика. Как было сказано в главе 7 (раздел 7.3.3), в протокол HTTP/1.1 включены слабые атрибуты содержимого для случаев, когда два варианта ресурса не являются семантически различными. Администратор сайта, зная, что изменение ресурса несущественно, может использовать слабые атрибуты содержимого для увеличения эффективности кэширования браузерами и прокси-серверами. В других случаях изменения ресурса могут быть семантически значимыми, но ка-

саться только нескольких байтов. Например, в некоторых текстовых и HTML-файлах изменения могут касаться небольшого фрагмента текста, например, номеров телефонов, или гиперссылок, например, на встроенные изображения. Это требует искать пути, позволяющие передавать в HTTP-ответе только *изменения*, а не весь ресурс. Эта проблема будет рассмотрена в главе 15 (раздел 15.2).

Влияние распространения новых технологий. Частота и виды модификаций ресурсов могут меняться на различных этапах жизненного цикла Web-сайта. Более широкое использование динамического содержания, например, запросов к базам данных влияет на частоту модификации ресурсов. Если информация в базе данных изменяется со временем, то одинаковые запросы могут приводить к различным результатам. Подобно регулярному изменению цен акций на бирже, приложения могут обусловить периодические изменения ресурсов. Эволюция инструментальных средств разработки Web-содержания также влияет на частоту изменений ресурсов. Например, инструментальное средство разработки может назначать новый URL для каждой новой версии файла и изымать старый URL, а не связывать один и тот же URL со всеми версиями. Это приведет к уменьшению вероятности того, что Web-ресурсы будут меняться со временем. Клиенты могут посылать серверу обновления ресурсов в теле HTTP-запроса. В конечном счете, в дальнейшем могут возникнуть новые типы содержания, которые будут меняться чаще или реже типов содержания, которые существуют в настоящее время. На изменяемость ресурсов будут воздействовать приложения, включая средства индексирования Web-содержания и кэширования.

10.4.6. Временная локализация

Время между последовательными запросами одного и того же ресурса оказывает значительное влияние на Web-трафик. Популярность ресурса определяет частоту запросов без указания интервалов между ними, временная локализация характеризует вероятность того, что запрошенный ресурс будет запрашиваться снова в ближайшем будущем. Когда последовательность запросов выявляет высокую временную локализацию, то имеется высокая вероятность того, что запрашиваемый ресурс уже находится в оперативной памяти исходного сервера или в кэше прокси-сервера. Точное определение временной локализации в потоке запросов является важной частью моделирования рабочей нагрузки сервера. Тестирование сервера тестами с низкой временной локализацией приведет к недооценке потенциальной производительности, которая будет достигнута, если фактический поток запросов будет иметь высокую временную локализацию. Высокая временная локализация также увеличивает вероятность того, что запрос будет удовлетворен браузером или прокси-сервером, и уменьшает вероятность того, что ресурс будет изменен со времени предыдущего доступа.

Характеристики трафика. Временная локализация охватывает иные, чем популярность ресурса, свойства рабочей нагрузки сервера. Например, рассмотрим запросы для ресурсов *a* и *b*. В последовательностях запросов (*a, b, a, b, a, b, a*) и (*a, a, a, a, b, b, b*) имеется одинаковое число обращений к обоим ресурсам. Временная локализация сильнее у второго потока, т.к. запросы для каждого из ресурсов сгруппированы. Временная локализация может быть измерена с помощью помещения каждого запроса в последовательности в вершину *стека*, в качестве меры временной локализации используется *расстояние в стеке* — число запросов между двумя запросами к одному и тому же ресурсу. Последовательность (*a, b, a, b, a, b, a*) начнется с состояния стека (*a*). Затем второй запрос будет помещен на вершину стека, со-

стояние стека в данном случае (*b, a*). Третий запрос требует доступа к ресурсу *a*, который находится во второй позиции стека. Следовательно, этот запрос имеет расстояние в стеке равное 2, а новое состояние стека имеет вид (*a, b, a*).

Малое расстояние в стеке подразумевает высокую временную локализацию. Например, последовательность (*a, a, a, a, b, b, b*) начнет стек с (*a*). Второй, третий и четвертый запросы для *a* будут иметь единичное расстояние в стеке. Аналогично второй и третий запросы для *b* будут тоже иметь единичное расстояние в стеке. Временная локализация ресурса может быть охарактеризована путем рассмотрения распределения расстояний в стеке для последовательности запросов. Изучение Web-трафика показало, что расстояния в стеке для запросов ресурсов соответствуют логарифмически нормальному распределению [AW97, BC98], приведенному на рис. 10.4. Большинство запросов к ресурсам, которые были сделаны в последнее время, дают небольшое расстояние в стеке, что предполагает сильную степень временной локализации. Небольшая, но существенная доля ресурсов не запрашивается длительное время или вообще не запрашивается.

10.4.7. Число встроенных ресурсов

Действие пользователя, например, щелчок мышью на гиперссылке, часто инициирует серию HTTP-запросов для загрузки как самого HTML-документа, так и встроенных в него ресурсов. Эти встроенные ресурсы могут представлять собой изображения, сценарии JavaScript, другие HTML-файлы, отображаемые в фреймах. Число встроенных ресурсов оказывает значительное влияние на нагрузку на сервер и сеть. Клиент выдает отдельный HTTP-запрос для каждого из встроенных ресурсов, если нет его кэшированной копии. Автоматически сгенерированные запросы загружают сервер и сеть. Фактически клиент одновременно открывает ряд параллельных TCP-соединений для получения необходимых встроенных ресурсов. Кроме того, множественные запросы могут быть переданы последовательно по установленному TCP-соединению. Модель рабочей нагрузки, которая не учитывает автоматическую загрузку встроенных ресурсов, не будет описывать ситуации внезапного увеличения нагрузки на сервер и сеть. Кроме того, число встроенных ресурсов оказывает влияние на эффективность установленного долговременного соединения. Наличие большого числа встроенных ресурсов увеличивает вероятность обработки последовательности HTTP-запросов, с помощью одного долговременного TCP-соединения.

Характеристики трафика. Выполненные в Web измерения показали, что медиана числа встроенных в Web-страницу ресурсов лежит в диапазоне от 8 до 20 [CP95, Mah97]. Распределение имеет сильный разброс значений, соответствуя распределению Парето [BC98]. Небольшое, но существенное число страниц имеет весьма большое количество встроенных ресурсов. Кроме того, число встроенных ресурсов имеет тенденцию к возрастанию по мере увеличения числа высокоскоростных подключений клиентов на работе и дома. Большое число встроенных ресурсов не обязательно преобразуется в большое число запросов к Web-серверу. Во-первых, на встроенные ресурсы могут иметься гиперссылки на кэшированных Web-страницах. Во-вторых, некоторые встроенные изображения не обязательно находятся на том же самом сервере, что и Web-страница. Например, Web-страница может содержать рекламные баннеры или изображения, которые расположены на другом сервере, как это будет описано в главе 11 (раздел 11.13). Все это уменьшит число запросов на встроенные ресурсы к серверу, с которого была осуществлена загрузка содержащей их Web-страницы.

10.5. Характеристики поведения пользователя

Характеристики рабочей нагрузки Web зависят от того, как пользователи загружают Web-страницы с различных сайтов. Заход пользователя на сайт, число страниц, которое он загружает, и время ожидания между последовательными загрузками вносят свой вклад в трафик.

10.5.1. Сеанс и прибытие запроса

Рабочая нагрузка прокси-, Web-серверов и сети зависит от временных соотношений в последовательности HTTP-запросов, исходящих от клиентов. Нагрузка, обусловленная одним клиентом, может быть смоделирована на трех уровнях: сеансов, переходов между страницами и запросов. Хотя Web-серверы не имеют явных сеансов с пользователями, но серия запросов одного пользователя к одному серверу может рассматриваться как *сеанс*. Пользовательский сеанс начинается с первого запроса и заканчивается после последнего запроса, после которого следует период неактивности. В процессе одного сеанса пользователь выполняет несколько *гипертекстовых переходов* по Web-страницам сайта. Переход соответствует пользовательскому действию, например, щелчку мышью на гиперссылке, отправке формы или вводу URL в адресной строке браузера. Каждый переход инициирует отправку браузером *HTTP-запроса* на ресурс, за которым, возможно, следуют автоматически генерируемые запросы на встроенные ресурсы, гиперссылки на которые имеются на данной странице.

С точки зрения сервера, каждый сеанс осуществляется с новым пользователем. Некоторые действия, например, создание cookies, могут осуществляться на уровне сеанса. Каждый пользовательский переход инициирует поток HTTP-запросов к серверу. Клиент может установить новое TCP-соединение для обработки запроса или передать запрос по существующему соединению. Модель рабочей нагрузки должна учитывать характеристики прибытия запросов на уровнях сеанса, TCP и HTTP. Временные характеристики начал сеансов могут быть изучены путем измерения времени между началом одного сеанса пользователя и началом сеанса следующего пользователя. Исследования показали, что интервалы между началами сеансов подчиняются экспоненциальному распределению [LNJV99, Fel00b]. Это одна из редких ситуаций моделирования рабочей нагрузки, которая описывается экспоненциальным распределением.

Однако экспоненциальное распределение не является достаточно точной моделью для описания TCP-соединений и HTTP-запросов, которые часто испытывают всплески из-за изменений пользовательского поведения и автоматической загрузки встроенных ресурсов. Для описания этих эффектов модель нагрузки должна учитывать последовательность переходов во время сеанса работы пользователя, время между последовательными переходами и число встроенных ресурсов на каждой из страниц. Каждый из этих параметров обусловливает всплески HTTP-запросов. Высокая изменчивость означает, что среднее число запросов не может описывать нагрузку сервера. Сервер может получать запросы со значительно большей частотой во время некоторых периодов времени. Эффективная обработка всплесков запросов требует дополнительных ресурсов от сервера и сети. Модель рабочей нагрузки, которая не учитывает всплески нагрузки, будет переоценивать потенциальные возможности Web-сервера.

10.5.2. Число переходов на сеанс

Число переходов, связанное с сеансом пользователя, оказывает существенное влияние на рабочую нагрузку сервера. Если типичный пользователь загружает большое число страниц, то нагрузка, обусловленная каждым сеансом, будет складываться из разнообразных передач данных. С другой стороны, если большинство пользователей обращаются только к одной странице, то каждый сеанс будет включать передачу всего одной страницы и ее встроенных элементов. В этом случае серверу нет смысла поддерживать установленное долговременное TCP-соединение после пересылки страницы и встроенных изображений, поскольку мала вероятность получения дополнительных запросов от клиента. В дополнение к этому, число переходов на сеанс влияет на эффективность кэширования браузером. Рассмотрим Web-сайт, имеющий ряд изображений, которые одновременно встроены в несколько страниц. Если пользователь посещает эти страницы, то изображения будут доступны из кэша браузера. Наоборот, если пользовательский сеанс включает в себя единственное обращение к Web-странице, то серверу придется передавать полный комплект встроенных изображений для этой страницы.

Пользователи существенно различаются по числу страниц, которые они просматривают на Web-сайте. Некоторые пользователи посещают только основную страницу Web-сайта и не обращаются к другим страницам. Другие могут провести час или более за просмотром различных страниц сайта. Большинство сеансов включает небольшое число переходов (от 4 до 10) [Mah97, LNJV99]. Однако результаты варьируются от сайта к сайту. Например, сайты электронной коммерции пытаются удержать пользователя, в то время как поисковые машины направляют пользователя к содержимому других сайтов. Кроме того, кэширование в браузерах и прокси-серверах приводит к уменьшению числа страниц, запрашиваемых с сайта. Число переходов описывается распределением Парето [LNJV99], это подразумевает, что число переходов некоторых сеансов гораздо больше, чем остальных. В результате небольшая часть посетителей сайта ответственна за большую часть запросов. Когда эти пользователи посещают сайт, то большое число запросов, генерируемое ими, может существенно снизить производительность сервера, наблюдаемую другими пользователями. Следовательно, реалистичная модель рабочей нагрузки должна включать в себя всплески нагрузки на сервер и сеть от таких пользователей.

10.5.3. Временные интервалы между запросами

Время между двумя последовательными запросами пользователя тоже оказывает влияние на рабочую нагрузку сервера и сети. Обычно браузер выдает запросы на встроенные ресурсы в процессе получения и синтаксического анализа содержимого HTML-файла. Однако время между двумя переходами зависит от поведения пользователя. В некоторых случаях пользователь просматривает Web-страницы очень быстро, затрачивая совсем мало времени на изучение содержимого каждой из них. В других случаях пользователь может потратить по несколько минут на чтение страницы до того, как запросит следующую страницу. Время между моментом полной загрузки страницы и моментом перехода пользователя на другую страницу называется *временем обдумывания или бездействия*. Характеристики времени бездействия пользователя влияют на эффективность политики закрытия долговременных соединений. Простая политика сервера заключается в разрыве TCP-соединения после некоторого времени простоя, в течение которого не приходит новых HTTP-запросов. Если время бездействия превышает этот период, то следующий HTTP-запрос потребует установления нового TCP-соединения.

Типичное время между переходами варьируется от сайта к сайту, считается, что в большинстве случаев это время не превышает 60 секунд. В небольшой части случаев время бездействия является очень большим по сравнению со средним значением. Исследования показали, что время бездействия подчиняется распределению Парето с медленно спадающим хвостом и со значением параметра распределения a около 1.5 [BC98]. Считается, что время бездействия не полностью определяется размером ответа, распределение которого имеет значение параметра a в диапазоне от 1.0 до 1.5.

В общем и целом Web-трафик проявляет изменчивость в нескольких направлениях. Медленно спадающие хвосты распределений вероятностей описывают ряд свойств Web-трафика: размеры ресурсов, размеры ответов, количество встроенных ресурсов в Web-страницу, количество переходов на сеанс, время между последовательными переходами. В результате сеансы могут быть смоделированы, как последовательности активных и пассивных периодов, в каждом активном периоде происходит загрузка страницы и встроенных элементов, а пассивный период соответствует времени бездействия пользователя. Продолжительность как активного, так и пассивного периодов имеют распределения с медленно спадающими хвостами.

Web-трафик состоит их суперпозиции множества последовательностей активных и неактивных периодов, каждый из которых соответствует разным пользователям. В результате нагрузка Web-сервера и сети *самоподобна,* имеется несколько характерных времен существенного изменения трафика от нескольких микросекунд до нескольких минут [LTWW94, WTSW97, CB97]. Это свидетельствует о том, что значение средней нагрузки не является хорошей оценкой для требований к сети или к серверу. Типичный Web-сервер и сетевые компоненты должны располагать дополнительными возможностями для работы в периоды повышенной нагрузки. Изменчивость Web-трафика также существенно влияет на выбор сервера и сетевых компонентов. Тесты, используемые для прокси-серверов и Web-серверов, должны предусматривать присущую трафику изменчивость нагрузки. Аналогично оценка сетевых протоколов, таких как TCP, должна учитывать изменчивость размеров передаваемых данных и нагрузки во времени.

10.6. Применение моделей рабочей нагрузки

Глубокое понимание характеристик рабочей нагрузки необходимо при создании модели рабочей нагрузки для оценки Web-протоколов и компонентов программного обеспечения. Создание Web-трафика с помощью модели включает создание потока HTTP-запросов, который учитывают различные параметры нагрузки и использование этого потока в работе прокси-серверов и Web-серверов.

10.6.1. Объединение параметров нагрузки

Создание синтезированного трафика включает в себя генерацию последовательностей псевдослучайных чисел с распределениями вероятностей каждого из параметров. Рассмотрим модель нагрузки, которая объединяет параметры и распределения, обсужденные в двух предыдущих разделах и сведенные в таблицу 10.2. Начала пользовательских сеансов представляют собой Пуассоновский случайный процесс с некоторым средним значением интервала между началами сеансов. Каждый новый сеанс начинается через некоторое время после начала предыдущего. Временной интервал между началами сеансов может быть получен с помощью случайного числа

на отрезке от 0 до 1, соответствующего экспоненциальному распределению $F(x)$. Это значение, в свою очередь, может использоваться для вычисления значения временного интервала x. Например, для экспоненциального распределения со средним $\lambda=1$ и случайного числа 0.3679 получим значение $x=1$, т.к. $e^{-1}=0.3679$. Повторяя эту процедуру, создаем последовательность чисел, определяющих времена начал сеансов. Каждый сеанс состоит из некоторого числа переходов, которое может быть вычислено из распределения Парето. Каждый переход генерирует запрос Web-страницы, состоящей HTML-файла и некоторого числа встроенных ресурсов. После загрузки Web-страницы, задается некоторое время бездействия до следующего перехода; это время бездействия также вычисляется с помощью распределения Парето.

Таблица 10.2. Распределения вероятностей в моделях рабочей нагрузки Web

Распределение	**Параметры нагрузки**
Экспоненциальное	Интервалы между сеансами
Парето	Размеры ответов (хвост распределения) Размеры ресурсов (хвост распределения) Число встроенных страниц Время между двумя запросами
Логарифмически нормальное	Размеры ответов (тело распределения) Размеры ресурсов (тело распределения) Временная локализация
Зипфа	Популярность ресурса

Подобным образом могут быть сгенерированы и характеристики ресурса. Например, набор синтезированных Web-страниц может быть сгенерирован заранее и сохранен на сервере. Рассмотрим задачу создания 1000 Web-страниц для синтеза рабочей нагрузки Web-сервера. Распределение вероятности размеров HTML-ресурсов может быть использовано для определения размеров каждого из 1000 HTML-файлов. После этого можно использовать функцию распределения числа встроенных ресурсов для определения числа встроенных ресурсов для каждой из страниц. Размер каждого из встроенных ресурсов также определяется с помощью распределения вероятности. Каждый из этих ресурсов будет иметь URL, который появится в HTTP-запросах, исходящих от Web-клиента. Далее, для определения доли клиентских запросов для каждого из ресурсов используется распределение Зипфа, описывающее популярность ресурса. Наконец, каждая Web-страница на сервере состоит из HTML-файла определенного размера и популярности, с которыми связано некоторое число встроенных ресурсов, каждый из которых имеет собственный размер.

Генерация синтезированной рабочей нагрузки требует объединения характеристик ресурсов со временем клиентских запросов. Например, запрос может быть связан с определенным URL с помощью распределения вероятности. Это дает уверенность в том, что клиент посещает каждую страницу в соответствии с ее популярностью. Однако это приближение не учитывает временной локализации последовательности запросов для одного и того же ресурса. Для того чтобы убедиться, что клиентские запросы соответствуют как распределению популярности, так и распределению временной локализации могут быть использованы дополнительные

приемы [BC98]. Другой важной проблемой является моделирование изменяющихся на сервере ресурсов, чтобы учесть воздействие изменений в размере ресурсов в реальном трафике, прошедшем кэширование. Например, времена модификации ресурсов могут быть получены из распределения вероятностей. В каждый из моментов модификации ресурс может быть изменен на сервере. Дальнейшие запросы для этого URL будут требовать от сервера передачи ресурса, а не ответа **304 Not Modified**.

10.6.2. Проверка достоверности модели рабочей нагрузки

Создать синтезированный трафик, который точно воспроизводил бы реальную нагрузку, чрезвычайно трудно особенно с таким большим количеством взаимовлияющих факторов. Синтезированная рабочая нагрузка не обязательно должна учитывать все основные особенности Web-трафика. Использование модели нагрузки для оценки производительности основано на предположении, что реально система будет давать такую же или похожую производительность. В результате проверка достоверности синтезированной нагрузки является важным шагом в создании и использовании модели рабочей нагрузки. Проверку достоверности часто путают с верификацией. Верификация включает проверку того, что синтезированный трафик имеет те же статистические свойства, которые заложены в модель, включая правильное распределение размеров ресурсов и времен бездействия. Проверка достоверности требует демонстрации того, что производительность системы подвергнутой синтезированной нагрузке, соответствует производительности той же самой системы при реальной нагрузке в соответствии с заранее определенными метриками производительности.

Необходимость проверки достоверности модели возникает в большом числе случаев. Когда производительность системы оценивается с помощью синтезированной модели нагрузки или абстрактной модели системы, то достоверность результатов находится под сомнением. Однако модели редко подвергаются строгой проверке на достоверность. Такая проверка может быть трудной, длительной или просто невозможной. В других случаях модель, которая прошла проверку достоверности для некоторых приложений, ошибочно используется в другой ситуации. Обычно модель рабочей нагрузки конструируется для определенных целей, таких как оценка производительности сервера и времени ожидания ответа пользователем. Может оказаться неподходящим использование одной и той же модели нагрузки для получения других метрик производительности, например, загрузки процессора сервера. Идеальным было бы исследование каждой из основных метрик производительности с помощью реальной нагрузки. Тот же самый сервер может быть оценен и с помощью синтезированной нагрузки, что облегчает прямое сравнение результатов измерений.

Модель синтезированной нагрузки используется также для тестирования серверов в различных ситуациях, которые могут случиться на практике. В этом случае сравнение с реальной нагрузкой может оказаться невозможным. Некоторые методики могут увеличить вероятность того, что синтетическая модель отражает реальные ситуации. Люди с опытом наблюдения Web-трафика могут испытать модель нагрузки, чтобы увидеть, учитывает ли она особенности реальной рабочей нагрузки. Анализ реального Web-трафика может помочь проверить заложенные в модель предположения. Например, предположим, что в модель рабочей нагрузки учитывает тот факт, что размер HTML-файла не зависит от числа и размеров встроенных в Web-страницу изображений. Анализ Web-измерений позволяет определить, соответствуют ли предположения действительности. Кроме того, очень важно исследо-

вание чувствительности производительности системы к малым изменениям модели рабочей нагрузки. Если слабые изменения входных параметров оказывают существенное влияние на производительность системы, то это означает, что небольшие неточности в моделировании реальной рабочей нагрузки скомпрометируют используемую синтезированную модель.

10.6.3. Генерация синтезированного трафика

Приложение синтезированной нагрузки к прокси- или Web-серверу требует решения нескольких практических задач. Тестирование высокопроизводительного сервера требует создания высокоскоростного потока запросов от большого числа синтезированных клиентов, каждый из которых выдает запросы на основе модели нагрузки. Однако использование отдельного компьютера для каждого клиента является дорогостоящей и труднореализуемой задачей. В идеале множество синтезированных клиентов можно запустить на одном компьютере [BC98, BD99]. Однако совместное использование ресурсов компьютера может привести к возникновению корреляции между клиентами. Клиенты, запущенные на одном компьютере, будут конкурировать за доступ к процессору, оперативной памяти, дисковой подсистеме, а также за доступ к сети. Для предотвращения взаимодействия между синтезированными клиентами, число клиентов, приходящихся на один компьютер, должно быть ограничено и основываться на тестировании, определяющем момент, когда система не сможет обслуживать ни одного дополнительного клиента без искажения вида трафика. Поддержание большого числа клиентов на одном компьютере требует наличия системы с большим объемом оперативной памяти и эффективной реализацией стека протоколов TCP/IP [BD99].

Создание синтезированного трафика обеспечивает благоприятную возможность оценить возможности прокси- или Web-сервера контролируемым способом. Накопление статистики производительности является решающей частью этого процесса. Однако измерение и запись данных о производительности может повлиять на работу клиента и сервера. Например, синтезированный клиент может зарегистрировать задержку в ответе сервера на HTTP-запрос. Время, затраченное на измерение и запись статистики, может повлиять на посылку последующих запросов. Искажения можно сократить путем записи статистики в оперативную память, а не в файл на диске. Однако для некоторых систем оперативная память может оказаться критическим ресурсом. Искажения можно также сократить путем уменьшения числа клиентов, исполняемых на каждом компьютере. Сервер может регистрировать данные о производительности, такие как загрузка процессора или число TCP-соединений. Ограничение частоты получения и записи этих данных может уменьшить нагрузку на сервер.

Свойства сети также оказывают существенное влияние на производительность прокси- и Web-серверов. Internet проявляет высокую изменчивость в задержке пакетов и вероятностях их потерь, что влияет на управление скользящим окном TCP. Реальная оценка производительности Web-сервера должна учитывать эти взаимодействия. Клиенты с высокоскоростным подключением непосредственно к серверу, будут создавать нагрузочный трафик, существенно отличающийся от трафика клиентов, разбросанных по всему миру и имеющих низкоскоростные подключения. Низкоскоростные подключения с большим временем ожидания ответов приводят к необходимости поддержания более длительных TCP-соединений на сервере, что увеличивает число соединений, которые должны обслуживаться одновременно. Кроме того, повторные передачи утерянных пакетов, увеличивают объем данных,

передаваемых сервером. Идеальное решение этой проблемы заключается в распределении синтезированных клиентов в разных точках Internet. Однако на практике это может оказаться достаточно дорогостоящим решением. Альтернативой является направление клиентских запросов и ответов сервера через компьютер, который *эмулирует* свойства сети, задерживая или теряя пакеты [BD99].

Производительность Web зависит от соотношений между пользовательским поведением, характеристиками ресурсов, загрузкой сервера и динамикой сети. Охватить все эти факторы в модели рабочей нагрузки на практике чрезвычайно трудно. Однако синтезированная модель нагрузки помогает оценить и сравнить компоненты программного обеспечения Web контролируемым образом. Было проведено большое число экспериментов для сравнения различных реализаций прокси- и Web-серверов [Wbe, TS95, Spea, Pol, MCS98, AC98]. Эти эксперименты используют набор клиентских и серверных программ для синтеза рабочей нагрузки, а также программ сбора и анализа данных. Обычно синтезированная нагрузка в этих экспериментах имеет стандартные параметры, задаваемые для создания одинаковой нагрузки при каждом их применении. С течением времени коммерческое программное обеспечение становится все более сложным, учитывающим все больше параметров нагрузки и новые особенности HTTP. Кроме того, тесты изменяются в направлении анализа других важных аспектов Web-трафика, например, динамической генерации ресурсов и ответов об ошибках.

10.7. Конфиденциальность пользователей

Сбор и анализ Web-трафика позволяет получить важную информацию о функционировании Web. Однако Web-запросы и ответы включают информацию о конечных пользователях и их привычках. В этом разделе будут обсуждены различные виды личных данных и те, кто имеет доступ к этим данным. Далее будет обсужден вопрос о том, как измеренные данные могут привести к нарушению конфиденциальности пользователей.

10.7.1. Доступ к данным пользователей

Действия пользователя, такие как переходы по гиперссылкам и отправка форм, преобразуются в последовательность запросов. Браузер обрабатывает пользовательский запрос путем извлечения данных из кэша, либо связываясь с сервером. Браузер также инициирует автоматические запросы для загрузки встроенных изображений и повторяет запросы для ответов переадресации. Далее браузер может запустить специальное приложение для просмотра полученного документа, что может обусловить дополнительные HTTP-запросы. Пользователь может не знать о полном наборе запросов, инициированных его действиями. В запросах содержится информация о том, какие Web-страницы просматривал пользователь, что искал в Web, какие делал оплайновые покупки, какие текстовые данные вводил в формы. Содержимое HTTP-запросов и ответов может быть записано сторонами, вовлеченными в обмен, также как и другими компонентами, которые перехватывают сетевой трафик. Например, HTTP-ответ может быть записан в кэш браузера или перехвачен монитором пакетов.

Web создает иллюзию анонимности. Однако HTTP-запросы могут содержать различную информацию, идентифицирующую пользователя. В посланный запрос браузер может вставить различные HTTP-заголовки, которые содержат личную

информацию и позволяют отслеживать пользовательские запросы. Заголовок **From** содержит адрес электронной почты пользователя для регистрации или для целей идентификации. Заголовок **User-Agent** содержит информацию об используемом браузере и, возможно, операционной системе, установленной на компьютере пользователя. Другие заголовки, такие как **Accept-Language**, могут дать дополнительную информацию о пользователе или позволить серверу отслеживать последовательные запросы одного и того же пользователя. Браузер посылает HTTP-запрос и принимает HTTP-ответ через TCP-соединение. IP-адрес пользовательского компьютера ассоциируется с браузером — конечной точкой TCP-соединения, и отображается в каждом IP-пакете при передаче. Простой DNS-запрос может преобразовать IP-адрес в доменное имя пользовательского компьютера, которое может помочь выяснить личность пользователя или название организации пользователя.

Потеря анонимности зависит от того, насколько существенные выводы о пользователе может сделать другая сторона. Информация об организации пользователя или его провайдере весьма полезна. Например, рассмотрим служащего компании, который заходит на Web-сайт, позволяющий проводить поиск патентной информации по ключевым словам. Такая информация может быть полезна для конкурентов. Имя или адрес электронной почты пользователя могут дать еще более существенную информацию. Например, знание о том, что пользователь с адресом электронной почты **viv@foo.com** посетил Web-сайт с информацией по определенным заболеваниям, может оказаться полезно компании, осуществляющей медицинское страхование пользователя. Знание специфики запросов, посланных пользователем, дает еще больше информации. Пусть пользователь загрузил Web-страницу, содержащую список вакантных должностей в некоторой компании. Эта информация будет интересна руководству пользователя. Кроме того, любые детали поведения пользователя могут быть использованы для выяснения, какую рекламу пользователь увидел при просмотре сайта. Администратор Web-сайта может также продать адрес электронной почты и информацию о просмотренных страницах сторонней компании.

Запросы могут также включать заголовки **Authorization**, которые содержат имя пользователя и пароль на удаленном компьютере, как это обсуждалось в главе 7 (раздел 7.10). При использовании обычной (Basic) схемы аутентификации в HTTP/1.0 пользовательский пароль не шифруется. Любой, кто запишет пароли, закодированные с помощью Base64, сможет выступать в роли данного пользователя. Протокол HTTP/1.1 решает эту проблему путем поддержки аутентификации с помощью дайджестов. В дополнение к заголовку **Authorization**, HTTP-запрос может включать заголовок **Cookie**, который позволяет серверу и другим сторонам отслеживать последовательные запросы одного и того же пользователя. Сервер может связать cookie с именем пользователя, адресом электронной почты, почтовым адресом, номером кредитной карточки и историей покупок. Информация заголовков **Authorization** и **Cookie** используется многими сайтами. Администраторы Web-сайтов могут сотрудничать между собой с целью объединения информации и отслеживания пользователей при доступе к различным сайтам. Пользователи могут не замечать получения ресурсов с других сайтов в результате автоматических запросов на загрузку встроенных изображений или ответов переадресации.

Поле заголовка запроса **Referer** содержит URL ресурса, с которого был осуществлен запрос данного ресурса. Это позволяет проследить последовательность переходов пользователя от одного сайта к другому. Кроме того, поле заголовка **Referer** может информировать сторонних лиц о существовании ресурсов, не предназначенных для широкого доступа. Например, предположим, что некто создал Web-страницу и послал ее URL определенной группе лиц. Никаких других гипер-

ссылок на эту страницу нет. Но если пользователь щелкнет на гиперссылке на этой странице, то ее URL будет записан в поле заголовка **Referer**. Любые данные измерений, которые включают содержимое поля **Referer,** будут содержать этот URL. Запросы к таким URL могут фиксироваться прокси-серверами. Кроме того, клиент может обнаружить такую страницу путем последовательного перебора набора URL до тех пор, пока не будет возвращена нужная страница. На практике ограничить доступ к частным Web-страницам можно только с помощью аутентификации по имени пользователя и паролю.

Пользователь может контролировать, какая личная информация предоставляется другим лицам. Например, пользователь указывает адрес электронной почты, который появляется в заголовке **From** путем настройки браузера, хотя этот заголовок обычно и не включается в HTTP-запрос. Кроме того, пользователь может настроить прием cookies браузером. Большинство браузеров могут быть настроены так, что они будут отвергать cookies всех Web-сайтов за исключением указанного ограниченного числа Web-сайтов или получать пользовательское разрешение перед приемом cookies. Объем персональных данных, связанных с cookies, зависит от того, какие данные пользователь ввел в форму, отправляемую на сайт. Например, при отправке формы пользователь может отказаться вводить адрес электронной почты. Пользователь может ограничить объем личной информации, передаваемой в HTTP-заголовке, направляя запрос прокси-серверу, осуществляющему анонимизацию. Если запрос должен содержать личную информацию, например, номер кредитной карточки, то использование SSL и HTTPS эффективно предотвращает перехват информации.

10.7.2. Информация, доступная компонентам программного обеспечения

Конфиденциальность пользователя зависит от того, кто имеет доступ к информации, присутствующей в HTTP-запросе и ответном сообщении. Браузер имеет наиболее полную информацию о пользовательских предпочтениях, пользовательских действиях, содержимом HTTP-запросов и ответных сообщений. По сравнению с другими Web-компонентами, браузер пользователя является наиболее «доверенным» приложением. Однако он может передать личную информацию другим программным компонентам через заголовок HTTP-запроса. Это происходит по ряду причин. Во-первых, сервер может использовать эту информацию для настройки ответного HTTP-сообщения. Во-вторых, браузер и серверное программное обеспечение часто разрабатываются одной и той же компанией, что обусловливает побудительный мотив для принудительного включения браузером определенных заголовков в HTTP-запросы. В-третьих, провайдер может также распространять предварительно настроенные браузеры для того, чтобы направлять запросы через прокси-серверы и включать дополнительные заголовки в HTTP-запросы.

Прокси-сервер может также быть «доверенным» приложением, как обсуждалось в главе 3 (раздел 3.7). Например, прокси-сервер может быть использован работодателем пользователя или провайдером в качестве способа повышения производительности, сокращения нагрузки на сеть и фильтрации несоответствующего профилю компании содержания. Однако прокси-сервер имеет доступ ко всем HTTP-запросам, передаваемым от лица пользователя Web-серверам. Кроме того, сетевой провайдер обычно имеет доступ к дополнительной информации, такой как IP-адрес или доменное имя пользовательского компьютера, что позволяет отождествить каждый запрос с определенным клиентом. В некоторых случаях учреждение

может иметь побудительные мотивы отслеживать передаваемые для отдельных пользователей данные. Например, компания может идентифицировать пользователей, которые ищут информацию о вакансиях или об определенных заболеваниях. Провайдер может отслеживать пользовательские сообщения с целью поиска потенциальных клиентов для новых сервисов. Кроме того, провайдер может собирать информацию о пользовательских предпочтениях в интересах других организаций.

В большинстве случаев Web-серверы управляются организациями, которые не связаны с пользователями, делающими запросы к серверу. Сервер обычно не является компонентом, которому пользователь может доверять. Основываясь на заголовках HTTP-запросов, сервер может распознать, какие из запросов приходят от определенного клиента. По сравнению с прокси-сервером, Web-сервер имеет большие трудности в распознавании того, какие из запросов пришли от *данного* клиента. Однако если пользователь посылает личную информацию на Web-сайт, то сценарий, обрабатывающий пользовательский запрос, может связать эти данные с cookie. Принимая эти cookie в последующих HTTP-запросах, сценарий может идентифицировать определенного пользователя. Аналогичным образом сценарий может извлекать личную информацию из заголовка **Authorization**. В некоторых случаях, пользователь может не возражать против предоставления личной информации Web-сайту. Например, многие компании имеют Web-сайты, которые предоставляют своим сотрудникам доступ к личным данным о зарплате и других выплатах. Кроме того, некоторые Web-сайты проводят политику конфиденциальности, которая описывает, как сайт использует данные и могут или нет они быть доступны третьей стороне.

Как участники передачи данных, браузеры, прокси- и Web-серверы непосредственно связаны с запросами и ответными сообщениями. Однако другие компоненты тоже могут иметь доступ к сообщениям. Поскольку Web-трафик проходит по сети, то монитор пакетов или промежуточный прокси-сервер могут перехватывать IP-пакеты, подлежащие пересылке. Собирая последовательности пакетов, эти компоненты могут восстановить HTTP-сообщение, как и другой IP-трафик, включая сообщения электронной почты и файлы, загружаемые по протоколу FTP. В зависимости от того, где перехватывается трафик, IP-адрес клиента может быть связан с пользователем, посылающим запросы. Эти компоненты имеют доступ к тем же данным, что и Web-браузер. К тому же такие компоненты могут быть установлены компанией пользователя или провайдером. Однако в отличие от прокси-сервера, указываемого обычно пользователем при настройке браузера, другие компоненты «невидимы» для пользователя, который не знает, где и как осуществляется наблюдение за его деятельностью.

10.7.3. Использование данных пользовательского уровня

Несмотря на наличие доступа к тексту HTTP-сообщений большинство компонентов программного обеспечения не регистрирует эту информацию. На практике журналы прокси-серверов и Web-серверов содержат ограниченное число полей. Сетевой администратор может перенастроить или изменить программное обеспечение для записи дополнительных полей, что, кстати, увеличит нагрузку на сервер. Политика конфиденциальности также может определить, какие поля подлежат записи, а также в какой форме. Например, отслеживание или регистрация может быть отключена для шифрованных полей, содержащих идентификационную информацию пользователя. Отождествление последовательности запросов с пользователем является важной частью анализа рабочей нагрузки Web. Однако этот ана-

лиз не требует опознавания каждого пользователя. Аналогичным образом, основываясь на последовательностях просматриваемых пользователем страниц, Web-измерения можно использовать для изменения информационного наполнения Web-сайта или рекламных материалов. Для этого нет необходимости в идентификации пользователя, запрашивающего страницы.

В дополнение к ограничению количества полей, записываемых в регистрационные журналы, политика конфиденциальности может ограничивать число пользователей, имеющих доступ к этим данным и способ использования этих данных. Например, провайдер может принять решение о мониторинге пакетов или ведении журнала прокси-сервера для повышения производительности и эффективности работы сети. Для этого администратор Web-сайта может использовать журналы Web-сервера для его настройки или для реорганизации информационного содержания. Важная информация о клиентах, такая как имена клиентов, почтовые адреса, истории покупок и номера кредитных карточек, может быть записана в отдельную закрытую базу данных. Журналы серверов могут быть использованы для внутренних целей с ограниченным доступом к ним. Файлы, передаваемые другим организациям, могут содержать уменьшенное число полей или зашифрованные значения полей. Продажа третьим лицам информации о пользователях может быть ограничена явным образом политикой обеспечения безопасности сайта.

Новые технологии могут быть направлены на повышение защищенности пользователей. Например, Консорциум Всемирной паутины (W3C) разрабатывает Платформу для описания параметров безопасности (P3P — Platform for Privacy Preferences) [P3P], которая позволяет пользователю определять состав личной информации, сообщаемой им Web-сайту. Основываясь на пользовательских предпочтениях и политике сайта, пользователь или браузер могут решить, осуществлять ли доступ к сайту и какую личную информацию раскрывать. В конечном счете, основа политики безопасности состоит в том, что пользователь сознательно предоставляет личную информацию. Требование явного раскрытия личной информации не очень понятно и удобно для пользователей, большая часть из них недостаточно глубоко знакома с «внутренней кухней» Web. Новые разработки могут помочь найти компромисс между необходимостью защиты пользовательской информации и желанием собрать данные для анализа пользовательского поведения и рабочей нагрузки Web.

10.8. Резюме

Оценка производительности Web-протоколов и программных компонентов требует понимания характеристик рабочей нагрузки Web. Учет изменчивости, присущей нагрузке Web, необходимо учитывать при разработке моделей рабочей нагрузки. Хотя рабочая нагрузка меняется во времени и в пространстве, основные характеристики относительно стабильны. Большая часть изменений характеристик нагрузки проявляет себя в виде изменений значений параметров распределений вероятностей, а не самого распределения. Кроме того, соотношение между параметрами и их влиянием на производительность Web остается относительно стабильным. Разброс характеристик нагрузки Web, тенденции изменения во времени и пространстве подчеркивают важность оценки Web-протоколов и программных компонентов в различных ситуациях. Синтезированная рабочая нагрузка, которая зависит от различные параметров нагрузки, обеспечивает метод сравнения производительности различных систем. Эти модели нагрузки играют важную роль в тестировании прокси- и Web-серверов различного назначения. Наконец, несмотря на

необходимость проведения детальных измерений Web-трафика, сбор информации о HTTP-запросах и ответах может нарушать конфиденциальность пользователей. HTTP-заголовки включают идентифицирующую пользователя информацию, а прокси- и Web-серверы могут использовать различные методы отслеживания последовательностей запросов данного пользователя. Мониторы пакетов и перехватывающие прокси-серверы, которые непосредственно не участвуют во взаимодействиях запрос-ответ, могут также получать информацию о пользователях. Предоставление пользователям возможности контроля над тем, какие личные данные раскрываются, является важной частью компромисса между пользовательской конфиденциальностью и необходимостью получения информации о характеристиках рабочей нагрузки.

Часть V
Web-приложения

11

Кэширование

Кэширование было первой технологией, использованной для снижения времени ожидания ответов пользователями и снижения трафика. Измерения трафика в Web выявили также примечательную тенденцию, состоящую в том, что большинство пользователей предпочитают посещать небольшое число *одних и тех же* сайтов. За несколько лет кэширование стало основной технологией сокращения трафика. В этой главе будет проведен обзор двух основных аспектов кэширования:

- **Основы и механизм кэширования**. Обзорная часть главы начинается с основных целей и эволюции кэширования. В последующих четырех разделах будет обсуждено, зачем, что, где и как кэшировать. Раздел *зачем* объясняет мотивы кэширования на каждой стадии запроса от пользователя через провайдера и Internet, через различные уровни стека протоколов (DNS, TCP, HTTP) к исходному серверу и обратно. Раздел *что* объясняет, что же влияет на принятие решения о кэшировании сообщений. Семантически прозрачный кэш, который возвращает ответ на запрос в том виде, какой он имел бы, если бы был получен от исходного сервера, должен удовлетворять ограничениям протокола HTTP. В разделе *где* говорится о местах, где выполняется кэширование на пути между пользователем и сервером. В разделе *как* перечислены основные шаги, выполняемые кэширующим программным обеспечением при кэшировании сообщений. Обзор продолжает обсуждение актуальности кэша и замены его элементов. В заключение говорится о влиянии частоты изменений ресурсов на принятие решения о кэшировании.
- **Практика кэширования**. Практическая часть главы начинается с взгляда на то, как кэши совместно используют информацию, и на различные протоколы, используемые при их взаимодействии. Далее рассматриваются особенности оборудования и программного обеспечения, используемых при кэшировании. В качестве примера программного обеспечения рассматривается популярная система кэширования Squid, а аппаратная часть представлена редиректорами, а также другими устройствами. Обсуждаются проблемы, связанные с кэшированием, на коммерческом и на социальном уровнях. Далее будет рассмотрена репликация — концепция, близкая к кэшированию, но основанная на других принципах и преследующая другие цели. Будут кратко упомянуты два новых направления: распределение и адаптация Web-содержания.

Кэширование обсуждается также в других главах книги:

- В главе 2 кратко обсуждается кэширование браузерами.
- В главе 3 кратко показана роль прокси-серверов в кэшировании.

- В главе 7 (в разделе 7.3) в связи с кэшированием обсуждается протокол HTTP/1.1.
- В главе 13 рассмотрены новые технологии кэширования.

В этой главе имеется исчерпывающий обзор темы. Отметим, что в главе термины *объект* и *ответ* считаются взаимозаменяемыми.

11.1. Истоки и цели Web-кэширования

Проект стандарта HTTP/1.1 (RFC 2616) определяет кэширование как локальное хранение ответных сообщений. Более свободная трактовка определения кэширования — перемещение содержания ближе к пользователю. Кэширование является, по-видимому, наиболее изученным Web-приложением. В настоящее время на рынке имеется ряд коммерческих программных и аппаратных решений различных компаний. Начнем с краткой истории кэширования, далее определим цели кэширования и перечислим проблемы, возникающие при решении задач кэширования.

Первый Web-сервер *httpd* (созданный в Швейцарии, в Женевской лаборатории ЦЕРН) имел прокси-сервер, который включал кэш [LA94]. Один из самых ранних проектов, связанных с Web-кэшированием, Harvest [BDH+94], индексировал информацию в Internet. Harvest кэшировал и реплицировал информацию, собранную другими средствами. Иерархический объект кэша был фундаментальным компонентом его архитектуры. Реестр сервера позволял собирать информацию о других кэшах. Реестр отвечал за запросы о расположении других кэшей. Разнесенные в пространстве кэши могли объединять свои ресурсы. После разработки Harvest начали развиваться и ряд других проектов, связанных с кэшированием.

Ко времени появления HTTP/1.0 кэширование уже стало важным и необходимым компонентом Web. Ранние эксперименты, связанные с кэшированием и измерениями, проведенные до 1994 г. [BC94], продемонстрировали возможности значительного сокращения объема данных, передаваемых через сеть. В кэши начали записывать разнообразную информацию, связанную с документами, включая тип содержания, его размер и среднее время между модификациями. Проверка актуальности содержимого кэша и доставка кэшированных ответов аутентифицированным пользователям становилась весьма важной. Группы кэшей организовывались в иерархические системы, которые выходили за границы регионов и национальные границы. Кэширование было первым из основных Web-приложений, используемых в повседневной практике обычными пользователями.

Целями кэширования является уменьшить:

- время между началом Web-запроса и моментом получения агентом пользователя ответа (время ожидания ответа пользователем);
- нагрузку на сеть путем устранения повторных передач одинаковых ответов;
- нагрузку исходного сервера за счет наличия прокси-сервера на пути между клиентом и сервером, обрабатывающим запрос.

Первая цель является наиболее известной целью кэширования. Сокращение времени ожидания пользователя имеет важное значение не только для пользователя Web, но и для разработчика Web-содержания. Исследования показали, что когда пользователь быстро получает ответ от Web-сайта, то он проводит на этом сайте больше времени. Многие пользователи еще имеют относительно медленное подключение к Internet. Быстрое получение Web-страниц увеличивает пользовательские ожидания на получение большего объема информации с сайта.

Вторая цель главным образом влияет на сеть, но имеет интересный побочный эффект. Передача только *необходимой* информации (т.е. устранение повторных передач содержания, которое пользователь может получить из кэша, расположенного к нему ближе) уменьшает нагрузку на сеть. Сокращение нагрузки приводит, в свою очередь, к увеличению производительности для всех, кто использует сеть, т.к. теряется меньшее число пакетов, а затраты на повторную их передачу уменьшаются.

Третья цель подразумевает, что исходный сервер может обработать большее число запросов от различных клиентов. Увеличение числа обрабатываемых запросов отражается не только на прикладном, но и на транспортном уровне. Это приводит к уменьшению числа отказов в установлении TCP-соединений из-за переполнения очередей и задержек обработки запросов.

11.2. Зачем нужно кэширование?

Начнем с мотивации кэширования. Хостинговые компании платят за полосу пропускания используемых каналов, и, естественно, кэширование может помочь им сократить свои расходы. От кэширования могут получить выгоду все участники обмена Web-сообщениями. Конечные пользователи получают значительный выигрыш от кэширования, так как уменьшаются задержки при получении ответов. Значительная доля разрывов соединений, происходящих во время сеансов взаимодействия с Web-сайтами, обычно связано недостаточно быстрым получением ответов пользователями. Пропускная способность сети снижается из-за повторных передач утерянных данных. Если предположить, что перегрузка возникает в различных точках Internet, то может оказаться полезным сокращение трафика или перенос его от магистральных сетей к периферии. Это дает двойную выгоду на уровне сети: передаются только полезные данные, а высвободившаяся пропускная способность может быть использована для передачи дополнительных данных. Расстояние кэша от пользователя также является существенным фактором при определении выигрыша пользователя от использования кэша. Близкий к пользователю кэш может существенно уменьшить время ожидания ответа по сравнению с кэшем, расположенным ближе к серверу.

Рассмотрим различные составляющие задержки, возникающей при загрузке ресурсов, и их влияние на пользователя, сеть и исходный сервер. Кэширование ресурсов браузером пользователя и прокси-сервером, расположенным близко к пользователю, уменьшает задержки в получении ответов пользователем. Перенос передаваемого содержания ближе к пользователю позволяет сократить некоторые составляющие задержек, возникающих при получении ответов. Как было сказано в предыдущих главах, с точки зрения пользователя составляющие задержек в процессе загрузки ресурсов определяются следующими факторами:

- пропускная способность соединения пользователя с провайдером и провайдера с Internet;
- если ответ на запрос к DNS-серверу отсутствует в кэше, то затрачивается время на обращение к DNS-серверу для преобразования доменного имени в IP-адрес;
- возникающие на пути между пользователем и исходным сервером заторы в сети;
- перегрузка исходного сервера;
- время создания ответа;
- время отображения ответа браузером.

Эти факторы задержки прослеживаются от передачи запроса браузером серверу до воспроизведения им полученного ответа. Рассмотрим влияние кэширования на каждый из перечисленных факторов задержки.

Сетевое соединение. Задержка передачи данных между клиентом и провайдером меняется существенно меньше, чем задержка в Internet. Если содержание расположено достаточно близко к провайдеру, то тогда можно достаточно точно предсказать время ожидания ответа. Если скорость соединения провайдера с Internet высокая, то ответ будет доставляться до провайдера достаточно быстро. Однако если скорость соединения небольшая, то время передачи ответа в сеть провайдера может составлять основную часть общего времени ожидания ответа. Кроме того, если соединение между провайдером и исходным сервером невозможно в момент запроса, то кэш провайдера может вернуть ответ. Ответ может оказаться устаревшим, однако в некоторых случаях это может быть не так важно. В ряде случаев ответы исходных серверов персонализированы и поэтому отсутствуют в кэше.

Задержка, связанная с DNS. Увеличение времени жизни (TTL) таблиц соответствий между доменными именами и IP-адресами в DNS также может сократить общее время ожидания ответа, поскольку соответствие между доменным именем и IP-адресом меняется не слишком часто. Как это будет обсуждаться в разделе 11.13, некоторые схемы кэширования могут не только располагать Web-содержание ближе к пользователю, но и уменьшать время поиска соответствия между доменным именем и IP-адресом. Расположение Web-содержания ближе к пользователю означает уменьшение сетевого расстояния (т.е. числа промежуточных передач) ответа. Кэширующий прокси-сервер может хранить список часто используемых доменов и тем самым сокращать задержки, связанные с DNS-операциями.

Сетевая нагрузка и производительность сети. Сокращение числа промежуточных передач пакетов может увеличить пропускную способность между клиентом и сервером, на котором размещено Web-содержание. Поскольку TCP является основным протоколом транспортного уровня для HTTP, то сокращение времени прохождения пакетов от клиента к серверу и назад (RTT) повышает производительность. Производительность TCP обратно пропорциональна RTT. Уменьшение числа промежуточных передач приводит также к уменьшению вероятности задержек в промежуточных точках из-за перегрузок. Как было отмечено в главе 5 (раздел 5.2.6), большое число пакетов в сети может вызвать перегрузку сети, приводящую к утере пакетов. Утерянные пакеты должны быть переданы повторно. Если в результате кэширования уменьшается нагрузка на сеть, то оставшиеся данные могут быть переданы быстрее. Отправитель TCP-пакетов для более полного использования сети может увеличить размер скользящего окна.

Нагрузка на исходный сервер. Исходный сервер может также выиграть от кэширования. Если будут кэшироваться наиболее популярные ответы, то нагрузка сервера сократится, поскольку существенно уменьшится количество запросов, достигающих этого сервера. Сервер сможет обработать большее количество запросов пользователей без постановки в очередь соединений транспортного уровня, без задержек в обработке HTTP-запросов. Исходный сервер переносит задачу обработки запросов на вспомогательные кэширующие прокси-серверы. Исходный сервер может также поддерживать долговременные соединения с меньшим числом клиентов, и эти соединения дольше могут оставаться открытыми. Время на генерацию излишних ответов сокращается, оставляя больше времени на выполнение сервером других задач.

Время генерации ответа. Время генерации ответа на запрос может быть сокращено путем изменения архитектуры Web-сервера. Если обработка входящих запросов требует немного ресурсов сервера (т.е. циклов процессора), то на генерацию ответа их тратится гораздо больше. Если генерация ответа будет занимать значительную часть времени ожидания ответа пользователем, а кэширование полностью отсутствует, то такая задержка будет наиболее значимой. Кроме того, если архитектура Web-сервера требует последовательной обработки запросов, то генерация длинного ответа будет негативно сказываться на обработке остальных запросов данного соединения.

Воспроизведение ответа браузером. Браузер должен синтаксически проанализировать и отобразить ответ. Время на анализ ресурса браузером не может быть сокращено кэшированием, если только не хранить версию воспроизводимой страницы, непосредственно в оперативной памяти браузера. Чтение и воспроизведение требует времени, даже если ответ записан в кэш браузера. Однако поскольку весь документ доступен локально, то для отображения ответа могут быть использованы различные технологии. Например, браузер может изменить процесс отображения содержания документа, если все встроенные изображения уже имеются в наличии. Большинство Web-страниц отображаются по мере загрузки ресурсов браузером, тем самым отображение загружаемых ресурсов может быть медленнее, чем отображение из кэша.

Кэширование связано и с финансовыми аспектами, так как влияет на задержки. Кэширование сокращает нагрузку на сеть провайдера. *Неожиданная популярность* у пользователей одного или двух ресурсов могут привести к перегрузке провайдера. Например, некоторые события, например, ожидание приговора суда, популярные спортивные мероприятия вызывают резкий всплеск обращений пользователей к определенным сайтам. Может оказаться дешевле приобрести кэширующий сервер, чем наращивать пропускную способность магистральной сети. Часть полосы пропускания, высвобожденная в результате кэширования, позволит провайдеру обслуживать большее число пользователей в пределах существующей инфраструктуры.

Даже при отсутствии неожиданных всплесков нагрузки провайдер обычно обязан платить своему вышестоящему провайдеру за полосу пропускания сети независимо от того, какую часть этой полосы он фактически использует. Сокращая объем загружаемых данных, провайдер может существенно сократить свои расходы. Провайдеры заключают соглашения по обмену трафиком друг с другом, если обмениваются примерно равными объемами данных. Такие соглашения требуют симметричного трафика, так что имеется сильный финансовый стимул сократить объем данных, передаваемых наружу из сети провайдера. В свою очередь, крупные провайдеры также используют соглашения об обмене трафиком. Кэширование может помочь провайдеру в конкурентной борьбе. Новый сервис может быть добавлен без боязни значительного ухудшения качества обслуживания клиентов.

11.3. Что кэшировать?

Кэш может принимать решение о необходимости кэширования на основе двух факторов: требований протокола HTTP и кэшируемого содержания. Требования, связанные с протоколом, определяют необходимость выполнения директив, касающихся кэширования сообщений. Требования, связанные с содержанием, отделены от факторов, определяемых протоколом. Эти соображения обусловлены коммерческими требования и политикой обновления кэша, проверяющей, содержатся ли в кэше актуальные данные. В свою очередь политика кэширования может определяться свойствами сообщений, например, размером.

11.3.1. Требования, определяемые протоколом

Хорошо функционирующий кэш должен подчиняться требованиям HTTP. Проект стандарта HTTP/1.1 определяет простые правила того, какие ответы необходимо кэшировать; методы запросов, поля заголовков запросов, коды и заголовки ответов должны указывать на то, что ответ можно кэшировать. Несоответствие одному или нескольким из перечисленных требований не позволяет кэшировать ответ. Например, ответы на запросы с методами **OPTIONS**, **PUT** и **DELETE** (рассмотренные в главах 6 и 7) не кэшируются. Ответы на запросы с методом **POST** не кэшируется, если не содержат заголовков **Cache-Control** и **Expires**. Если кэш не поддерживает заголовков **Range**, то ответ с кодом **206 Partial Content** не может кэшироваться.

Некоторые ответы включают зависящую от ресурса информацию от исходного сервера, которая может препятствовать кэшированию сообщений. Такая информация бывает двух видов: информация о возможности кэширования и директивы управления кэшированием. Если ответ включает информацию о возможности кэширования, то решение о кэшировании должно определяться ею. Например, сервер может дать явное указание, когда следует обновить ресурс в кэше с помощью заголовка **Expires**. Если время, указанное в **Expire**, близко ко времени прибытия сообщения, то ресурс кэшировать нельзя. Директивы управления кэшированием могут предотвращать кэширование некоторых ответов. Например, директива **Cache-Control: Private** указывает, что прокси-сервер, совместно используемый несколькими клиентами, не может кэшировать данный ответ. Ответное сообщение, включающее директиву управления кэшированием

```
Cache-Control: no-store
```

не должно сохраняться. Директива управления кэшированием

```
Cache-Control: no-cache
```

уменьшает вероятность того, что кэш будет сохранять ответ, поскольку перед тем, как отправить клиенту кэшированную копию, будет осуществлена проверка ее актуальности. Директивы не обязательно должны быть директивами в ответе, переданном исходным сервером. Они могут быть директивами запроса, определенными запрашивающим клиентом. Например, **Cache-Control: no-store** может появляться как в запросе, так и в ответе. Протокол использует эти директивы для обеспечения конфиденциальности ответа, а также указывает, что данные ресурсы могут быть изменены после их отправки. Кэш должен внимательно отслеживать такие директивы и, если они приходят, обеспечивать семантически прозрачное кэширование.

Присутствие в запросе заголовка **Authorization** или заголовка **Vary** в ответе снижает шанс того, что ресурс будет кэширован. Заголовок запроса **Authorization** указывает на то, что запрашиваемый ресурс доступен не для всех. Это уменьшает вероятность, что ответ используется многими пользователями. Аналогично, присутствие заголовка **Vary** указывает на то, что сохраняемый кэшем ответ будет ограничен версией, указанной в заголовке **Vary**.

Конфликтующие цели сторон, вовлеченных в кэширование, были известны разработчикам HTTP. Стандарт протокола не может дать оптимальное решение для всех сторон, связанных с транзакцией. Однако разработчики протокола в основном уделяли внимание семантической прозрачности кэширования, предоставляя управление последней всем сторонам, вовлеченным транзакцию.

11.3.2. Соображения, определяемые Web-содержанием

У кэша может быть набор своих собственных правил, не связанных с требованиями протокола, для решения вопроса о необходимости кэширования ответа. Другими словами, возможность кэширования ответа не означает, что он будет обязательно кэширован. Сообщение может оказаться слишком большим, динамически генерируемым или включать в себя cookies, что может оказать влияние на кэширование. Политика кэширования может руководствоваться другими соображениями, отличными от требований протокола, например, атрибутами сообщений. Например, частота, с которой кэш проверяет актуальность ресурсов на исходном сервере, может диктоваться политикой продолжительности кэширования ресурсов. С другой стороны, если обладателю кэша платят за объем данных, доставляемых пользователям, то он может принять решение игнорировать дату последней модификации и посылать полный ответ вместо передачи **304 Not Modified**. Совместно используемый кэш может не кэшировать ответы на запросы, содержащие внедренную в них личную информацию (например, cookies). Страницы ASP и запросы на документы, требующие аутентификации, являются, вероятно, не лучшими кандидатами на кэширование.

Можно сократить коммерческую стоимость передач, игнорируя некоторые ограничения, связанные с кэшированием. Кэши могут сохранять ресурсы, которые не предполагалось кэшировать, например, в случае использования **Cache-Control: private.** Цели пользователя — сократить время ожидания ответа, а провайдера — сократить объем пересылаемых данных, могут не совпадать с побудительными мотивами компании, осуществляющей кэширование, приводя к игнорированию ограничений протокола.

Кэши могут также учитывать расходы по хранению данных и не кэшировать большие ресурсы, даже если нет никаких других препятствий для их кэширования. Если сообщение слишком большое, то, возможно, что многие другие объекты будут вытеснены из кэша. В терминах времени задержки ответа, стоимость загрузки большого числа небольших ресурсов с соответствующих исходных серверов может быть выше, чем стоимость получения их из кэша. Таким образом, кэш может отказаться от кэширования слишком больших ответов. С другой стороны, большие кэшируемые ответы, которые неоднократно требуются клиентам, находящимся за кэшем, могут давать значительную экономию трафика. Например, предположим, что новая версия популярной программы появилась на Web-сайте. Такой ресурс имеет и большой объем, и достаточно высокую частоту запросов. Его кэширование дает значительную выгоду. Частично причина заключается еще и в том, что некоторые провайдеры должны платить за объем загружаемых ими данных, и они предпочитают использовать кэши, чтобы не загружать ресурсы многократно.

Многие кэши воздерживаются от сохранения ответов, сформированных сценариями, предполагая малую вероятность того, что те же значения параметров запросов будут повторно использованы. Решение о том, что ответ был сгенерирован как результат вызова сценария, основано на эвристике. Основное предположение, влияющее на решение о кэшировании ответа, заключается в том, что в дальнейшем будет генерироваться один и тот же ответ, а запрос на такой ответ может поступить в ближайшее время. По сравнению с запросами на статические ресурсы, которые меняются не слишком часто, каждое выполнение сценария с высокой вероятностью приводит к отличному от предыдущего ответа результату. Однако присутствие в динамическом ответе такой информации о возможности кэширования, как заголовки **Expires** и **Etag,** может указать на целесообразность кэширования ресур-

са. Например, рассмотрим CGI-сценарий, который возвращает n-ую цифру числа p. Ответ не будет изменяться до тех пор, пока передаваемый сценарию параметр равен n. Многие запросы часто приводят к одному и тому же ответу и некоторые кэши уже учитывают это. Данное соображение актуально для поисковых серверов, где очень большое число запросов выполняется для малого числа ключевых слов, например, "mp3", "sex" и т.д. Если поисковый сервер обновляет результаты таких запросов, например, не чаще, чем раз в день, то запрос будет приводить к одному и тому же ответу. В общем случае не всегда возможно сказать, был или не был возвращенный ответ сгенерирован динамически.

Миф о том, что ответы на CGI-запросы и другие динамически генерируемые страницы не должны кэшироваться, должен развеяться. Другой категорией ответов, которые могут выглядеть как некэшируемые, являются ответы, включающие данные, подготовленные для определенного пользователя. Например, ответ, содержащий cookie, рассматривается прокси-сервером как некэшируемый, т.к. ожидается, что он будет разным для разных пользователей. Но некоторые сообщения с cookies могут все же кэшироваться, если они возвращают один и тот же ответ для различных значений cookies [WM99]. Заметим, что некоторые типы содержания, которые обычно рассматриваются как кэшируемые (например, изображения в форматах GIF или JPEG), могут на самом деле включать нестандартную информацию, или вообще информация о необходимости их обновления может отсутствовать, что затрудняет принятие решения о кэшировании.

Решение о кэшировании определяется частотой изменения ресурсов. Некоторые ресурсы изменяются не очень часто. Примерами таких ресурсов является электронные книги. Некоторые статические ресурсы могут изменяться периодически. Основная страница пользователя, например, периодически изменяется, но не особенно часто. Таким образом, определение скорости изменения ресурса является корректной метрикой для принятия решения о кэшировании ресурса. Первоначально решение о возможности кэширования ресурса основывалось на времени его последней модификации. Предположение заключалось в том, что если ресурс давно не изменялся, то маловероятно, что он изменится в ближайшем будущем. Это делает ресурс кандидатом на кэширование. Если ресурс был сохранен в кэше, то время последней модификации используется также для решения о том, когда проверить актуальность кэшированного ответа. И наоборот, этот подход предполагает, что если ресурс был модифицирован недавно, то имеется высокая вероятность того, что он вскоре будет снова изменен. Таким образом, кэшированная копия такого ресурса очень быстро устареет. В то же время высокая частота изменения ресурса может указывать на высокий интерес и на большую частоту запросов, что является аргументом в пользу его кэширование. Поскольку выгода от кэширования обусловлена в основном большим числом обращений к кэшируемому ресурсу, то частота обращений к ресурсу должна учитываться при принятии решения о кэшировании.

Нагрузка на кэш тоже должна учитываться при принятии решения о кэшировании ресурса. Например, заполненный кэш может обрабатывать запросы к кэшу, просто посылая отказы в кэшировании, и, соответственно, не кэшируя ответные сообщения в этих случаях.

11.4. Где выполняется кэширование?

Кэши имеются в браузерах, они также располагаются на пути между агентом пользователя и исходным сервером. В дополнение к кэшу браузера может исполь-

зоваться кэш прокси-сервера. Кэширование имеет смысл выполнять в нескольких местах, а не только в одном месте. Кэширование браузером и прокси-сервером имеет следующие особенности:

- Кэш браузера помогает избежать повторной загрузки страниц, просматриваемых пользователем во время сеанса работы с браузером. Однако кэш браузера никак не учитывает частые запросы других пользователей на тот же ресурс.
- Кэширующий прокси-сервер может иметь десятки, если не сотни пользователей. Однако браузер может хранить набор принятых ответов за большее время, чем кэширующий прокси-сервер. Работая с большим числом пользователей, кэширующий прокси-сервер может удалять некоторые ответы из кэша быстрее, чем браузер.
- Региональный кэш может помочь нескольким другим кэшам, географически расположенным недалеко от него. Национальный кэш может группировать региональные кэши и помочь им сократить объем трафика через национальные границы. С каждым шагом, удаляющим от клиента, добавляются потери в производительности даже при наличии кэшированных ответов.

В главе 3 (раздел 3.8) были кратко описаны обратные и перехватывающие прокси-серверы. Обратный прокси-сервер работает как представитель одного или более исходных серверов и может включать кэш. Кэширование на нем выполняется от лица исходных серверов, а не пользователей. Это сокращает нагрузку на исходные серверы, задачи, с которыми сталкиваются обратные прокси-серверы, сходны с задачами для прокси-серверов, находящихся близко от пользователей. Наиболее часто запрашиваемые ресурсы исходного сервера с большой вероятностью перемещаются в кэш обратного прокси-сервера. Если ресурс не найден на обратном прокси-сервере, то последний направляет запрос исходному серверу. При этом прокси-сервер действует как ретранслятор.

Перехватывающий прокси-сервер может быть расположен в произвольном месте сети и анализировать трафик на сетевом и транспортном уровнях. Перехватывающий прокси-сервер перехватывает HTTP-запрос и принимает ответ. Обычно такой прокси-сервер располагается ближе к клиентам. Отметим, что перехватывающий прокси-сервер не обязательно должен находиться на пути прохождения пакетов. Устройство, которое может проанализировать пакеты на транспортном уровне, может переадресовывать трафик перехватывающему прокси-серверу. После того как пакеты извлечены, запрос может быть переадресован кэшам, находящимся под тем же административным управлением, что и перехватывающий прокси-сервер. Такой перехват обычно невидим пользователям, которые могут почувствовать только сокращение времени ожидания ответа и предположить, что ответ получен из кэша. К сожалению, некоторые перехватывающие прокси-серверы не поддерживают непосредственно протокола HTTP, а принятый ответ может быть не тот же самый, который был бы получен от обычного прокси- или исходного сервера.

11.5. Как выполняется кэширование?

Обсудим теперь подробно вопрос о том, *как* выполняется кэширование, т.е. операции, которые происходят при кэшировании. Во-первых, кэш должен определить, нужно ли сообщение кэшировать. Затем решить, имеется ли для этого достаточно свободного места и, если места нет, решить, какие объекты в кэше следует замес-

тить. При получении запроса кэш должен определить, может ли он его обработать, если может, то вернуть кэшированный ответ и обновить некоторую информацию. Кэш должен иметь согласованную политику по проверке актуальности кэшированных ресурсов.

11.5.1. Решение о необходимости кэширования

Кэш должен сначала решить, нужно ли сообщение кэшировать. Различные кэши используют различные стратегии принятия решений о кэшировании. Как было обсуждено в разделе 11.3, общими критериями, используемыми при принятии решения о кэшировании, являются:

- Имеются ли требования протокола, которые предотвращают кэширование ответа?
- Является ли содержание ответа некэшируемым (т.е. динамические данные или особый тип содержания)?
- Насколько вероятно, что кэшированный ответ будет использован повторно?
- Приведет ли кэширование данного ресурса к замещению в кэше одного или более ресурсов?

Кэш использует некоторые или все перечисленные критерии для принятия решения о необходимости кэширования.

11.5.2. Замещение содержимого кэша и запись ответа в кэш

После принятия решения о записи сообщения осуществляется проверка, можно ли записать сообщение без замещения других объектов в кэше. Если это не так, то запускается специальный алгоритм замещения. Замещение содержимого кэша является достаточно трудоемким процессом, особенно если замещается большое количество небольших объектов (замещение содержимого кэша будет рассмотрено более детально в разделе 11.6). Дополнительная нагрузка возникает, когда выполняются новые запросы на замещенные объекты, так как должны быть установлены новые соединения для их загрузки. Часто, когда становится известно, что ресурс устарел, то он может быть удален из кэша, даже если кэш не заполнен до конца. Это понижает затраты на выполнение алгоритма замещения содержимого кэша в момент обработки запроса, в свою очередь это уменьшает время ожидания ответа.

После того как освобождается место для записи ответа, кэш извлекает информацию о сообщении, включая дату последнего обновления и информацию об устаревании. Обрабатываются следующие заголовки сообщений: **Expire** и **Cache-Control: max-stale,** несущие информацию об актуальности ресурса. Эти поля заголовков помогают кэшу выполнять ограничения HTTP по периоду времени, в течение которого может быть возвращен семантически корректный ответ. Кэш, поддерживающий протокол, обязан гарантировать, что любой возвращенный им ответ, рассматривался бы исходным сервером как актуальный. В главе 7 (раздел 7.7.3) обсуждались различные заголовки, семантически связанные с кэшированием. При отсутствии специфической информации об актуальности ресурса кэш использует эвристически полученное время истечения срока актуальности. Значение этого времени может быть получено, основываясь на времени последнего обновления ресурса (значения поля заголовка **Last-Modified**). Например, сервер может добавить фиксированный промежуток времени, скажем, 10 минут к значению **Last-Modified**

и использовать его как время обновления ресурса. И, наконец, создается ключ ресурса, используемый при последующих поисках этого ресурса. Ключ создается с помощью хэш-функции на основе на URL запроса. Когда кэш получает новый запрос, то он использует URL для поиска ресурса в кэше.

11.5.3. Возврат кэшированного ответа

Когда в кэше находится ответ, соответствующий ключу, то встает вопрос о том, что возвращать пользователю. В зависимости от политики кэширования и ограничений, относящихся к кэшированию, которые содержатся в заголовках ответа, может быть выполнена проверка актуальности для того, чтобы убедиться, что ответ еще не устарел (проверка актуальности и общая согласованность кэшированных результатов будут обсуждены ниже в разделе 11.7). Если проверка подтверждает, что ответ не устарел, то запрос удовлетворяется из кэша. В противном случае кэш получает новую копию ресурса и использует политику кэширования для решения о том, нужно ли его снова кэшировать, одновременно направляя ресурс клиенту. Если запрос не найден в кэше, то запрос перенаправляется исходному серверу.

11.5.4. Обслуживание кэша

Периодически кэш может проверяться на наличие устаревших объектов, чтобы удалить их. Кэш может также контролировать частоту запросов кэшированного объекта, чтобы определить наиболее популярные ресурсы и выполнить специальные действия. Например, кэш может проверить актуальность популярного ресурса. Такая проверка может быть сделана с помощью запроса **HEAD**, который приводит к получению только метаданных ресурса. Кэш может также обратиться к исходному серверу, определить наличие изменений ресурса и обновить его содержимое в кэше. Такой подход направлен на уменьшения времени ожидания ответа пользователями. Однако он увеличивает трафик между исходным сервером и кэшем.

11.6. Замещение объектов в кэше

После того как кэш заполняется, необходимо выполнить удаление некоторых объектов для кэширования новых ответов. В течение последних лет было исследовано несколько подходов к замещению объектов в кэше. Некоторые из них заимствованы из традиционных подходов к кэшированию в файловых системах, а некоторые специально предложены для Web-кэширования. Одним из хорошо известных подходов является алгоритм LRU (Least Recent Use) — замена объектов, запрошенных наиболее давно. Книги на полках, которые давно не использовались, больше покрыты пылью, чем те, которыми пользовались недавно. Чем больше времени прошло с момента последнего обращения, тем больше пыли. Задачи кэширования, такие, как уменьшение объема данных, пересылаемых по сети, и сокращение времени ожидания ответа, приводят к необходимости комплексного решения задачи замещения объектов в кэше. Комплексный подход состоит в использовании комбинации метрик, включая размер кэшированного ответа, тип его содержания и информация о расстоянии до исходного сервера.

Полезность хранимого в кэше ответа может быть оценена по ряду факторов, включая:

- **Стоимость извлечения ресурса**. Стоимость извлечения ресурса с исходного сервера определяется возможностями кэша в организации взаимодействия с сервером и расстоянием, которое должен пройти ресурс до того, как достигнет кэша. Цена замещения ресурса, требующего больших затрат по извлечению, должна быть оправдана тем, что ресурс потребуется в будущем.
- **Стоимость хранения ресурса.** Кэш использует фиксированный объем дисковой памяти, сохранение одного объекта требует уменьшения объема памяти для других. Большой ресурс занимает больше пространства, однако повторное его извлечение с исходного сервера будет также дорогостоящим.
- **Число обращений к ресурсу.** Объект, к которому многократно обращались в прошлом, будет, вероятно, запрошен снова и является достойным кандидатом на кэширование в течение длительного срока.
- **Вероятность того, что ресурс будет запрошен в ближайшее время.** Если такая вероятность высокая, то нет смысла удалять ресурс из кэша. Вероятность доступа к ресурсу может быть рассчитана *заранее* или оценена на основе подходящих образцов.
- **Время последней модификации ресурса.** Ресурс, который не менялся в течение длительного времени, скорее всего, не изменится и в ближайшем будущем. Ресурс, созданный совсем недавно, может оказаться или динамически создаваемым ресурсом, или он снова изменится в ближайшем будущем. Ресурсы, которые чаще изменяются, как правило, оказываются более популярными, поскольку они меняются именно в результате своей популярности и, следовательно, являются хорошими кандидатами на кэширование. Однако кэшируемый ответ должен в этом случае заменяться достаточно часто (с частотой его изменения на исходном сервере). Время последней модификации, таким образом, может использоваться для принятия решения о том, какие ресурсы можно замещать.
- **Эвристическая оценка времени истечения срока годности ресурса.** Если время истечения срока годности ресурса не указано сервером, то кэш должен найти эвристическую оценку этого времени. Если отсутствуют ресурсы с истекшим сроком годности, то наиболее вероятным ресурсом на замещение является тот, у которого осталось наименьшее время до истечения его срока годности.

Каждый из этих факторов принимается во внимание при решении о том, какой механизм замещения будет наиболее подходящим для текущего состава объектов в кэше.

Различные факторы, принимаемые во внимание при замещении объектов в кэше, приводят к использованию различных алгоритмов. *Одноуровневый* алгоритм использует одну метрику, в то время как *двухуровневый* алгоритм использует комбинацию алгоритмов с первичной и вторичной метриками. Комбинированные алгоритмы используют взвешенные значения оценок. Если сложность замещающего алгоритма увеличивается, то возможно снижение его эффективности. Это происходит по причине больших накладных расходов в процессе добавления ресурса, который, скорее всего, скоро будет замещен. Накладные расходы на выполнение алгоритма рассматриваются как стоимость и являются фактором в оценке общей полезности алгоритма. Ниже приведен краткий обзор предложенных и используемых алгоритмов замещения объектов кэша.

Самый старый из используемых (LRU—Least Recently Used). Является одним из наиболее протестированных алгоритмов. Этот алгоритм просто удаляет из кэша самый старый объект (с точки зрения времени его последнего использования). Идея подхода весьма прямолинейна — объект, который запрашивался недавно, скорее всего, будет запрошен снова в ближайшее время, и необходимо заместить более старые объекты до удаления любого из более новых ресурсов. Ряд исследований [AW97, ASA+95, WAS+96, CI97] показали, что этот алгоритм не является лучшим для максимизации доли наиболее часто используемых объектов. Среди причин можно указать отсутствие временной локализации в ссылках на документы и то, что многие объекты запрашиваются только однократно.

Самый редко используемый (LFU — Least Frequently Used). Этот алгоритм ранжирует документы по частоте доступа к ним и удаляет документы, которые имеют самую маленькую частоту. Стратегии, основанные на частотах использования кэшированных ресурсов, были рассмотрены в работах [WAS+96, SSV97].

Размер объекта (SIZE). Другим критерием для выбора объекта, подлежащего замещению, является его размер. Удаляя из кэша объект самого большого размера, можно освободить место для нескольких объектов меньшего размера.

Hyper-G (LFU/LRU/SIZE). Алгоритм Hyper-G объединяет три предыдущие стратегии. Первыми кандидатами на замещение в этом алгоритме являются наименее часто используемые объекты (LFU). Если имеется несколько ресурсов, удовлетворяющих этому критерию, то удаляется самый старый из них (LRU). Если предыдущие операции все еще не определяют единственного ресурса, то удаляется тот из них, который имеет самый большой размер (SIZE).

Алгоритмы, основанные на полезности (GreedyDual-Size). Этот алгоритм был предложен в контексте замещения страниц в памяти компьютера [You94]. Все рассматриваемые страницы должны быть одного размера, а стоимость извлечения страницы из вторичного источника хранения должна быть составной частью ключа. Позднее алгоритм был модифицирован для анализа Web-ресурсов различных размеров [CI97]. Модифицированный алгоритм определяет значение параметра, называемого *полезностью*, и замещает ресурс, имеющий наименьшее значение полезности. Отталкиваясь от стоимости загрузки ресурса в кэш и его размера, полезность принимает во внимание также временной фактор, значение которого обновляется каждый раз, когда ресурс удаляется из кэша.

Замещение объектов в кэше было давней проблемой кэширования. Устойчивое снижением цен на устройства хранение данных привело к тому, что объем кэшей существенно увеличился. Проблема замещения объектов кэша сошла со сцены по следующим четырем основным причинам:

- Устойчивое снижение цен на устройства хранения данных, привело к появлению кэшей, имеющих объем памяти достаточный для хранения большинства запрашиваемых ресурсов.
- Общее сокращение доли кэшируемого трафика.
- Разработка алгоритмов, которые могут использоваться в большинстве ситуаций замещения кэшируемых объектов. Алгоритмы типа Greedy Dual-Size и Hyper-G относятся к этой категории.
- Изменение ресурсов во времени сокращает время хранения ресурсов в кэшах.

11.7. Согласованность кэша

Исходный сервер определяет время актуальности кэшированного ответа. Кэш же должен перед тем, как отправить запрашивающему клиенту кэшированный ресурс, убедиться, что он еще актуален. Согласованность кэша — хорошо изученная проблема для всех форм кэширования на компьютерах. Первоначально согласованность изучалась в связи с использованием многоуровневой памяти компьютера — кэшированная копия файла могла измениться на диске. Были предприняты попытки минимизировать объем затрат, требуемых для проверки актуальности, однако цена поддержания согласованности менялась в зависимости от контекста. Для Web-кэшей в последнее время были предложены различные алгоритмы поддержания согласованности. Согласованность кэша может зависеть от ресурсов и политики сохранения элементов в кэше. Кэши могут просто возвращать более старое кэшированное значение с добавлением признака устаревания ресурса. Одна из причин такого поведения — отключение исходного сервера или перегрузка кэша. Заголовок **Warning** протокола HTTP/1.1 (обсужденный в главе 7, в разделе 7.12.2) может использоваться для указания того, что возвращаемое кэшированное значение возможно уже устарело. Агент пользователя может сделать эту информацию доступной пользователю.

В распределенных файловых системах проблема кэширования была также глубоко изучена. Web отличается от распределенных файловых систем по ряду признаков. Обновление ресурса происходит только в одном месте — на исходном сервере, а в распределенных файловых системах это обновление может происходить в различных местах. Однако в Web присутствие большого числа прокси-серверов и возможность кэшировать *части* ответа меняют картину. Кроме того, в Web цена проверки актуальности выше из-за необходимости установления соединения с исходным сервером.

Протокол HTTP/1.1 обеспечивает несколько путей для поддержания согласованности кэша. Если исходный сервер задает определенное время истечения срока годности ресурса, то прокси-сервер, обеспечивающий семантическую прозрачность кэширования, обязан строго придерживаться этих сроков. Единственным исключением может быть наличие в заголовке запроса клиента директивы **Cache-Control: only-if-cached**, которая заставляет прокси-сервер возвратить кэшированный ответ без проверки его актуальности на исходном сервере. Если же исходный сервер не задает срок годности ресурса, то прокси-сервер может использовать эвристически вычисленное время годности. Это время часто связывается с проверкой актуальности. Наиболее общим подходом к проверке актуальности в Web является запрос с методом **Get** или **Head** и заголовком **If-Modified-Since** (обсужденном в главе 6, раздел 6.2.3). Этот заголовок включает время последней модификации ресурса, который был указан исходным сервером. В некоторых случаях время создания ответа может совпадать со временем последней модификации. Атрибуты содержимого протокола HTTP/1.1 в совокупности с заголовком **if-Match** могут быть использованы для выполнения проверки актуальности для версий ресурсов (см. главу 7, раздел 7.3.3). Исходный сервер может вернуть полную копию ресурса (вместе с ответом **200 OK**) или ответ **304 Not Modified** без тела ответа. Однако проверка актуальности требует полного цикла HTTP-обмена запросом и ответом.

Если кэширующий прокси-сервер посылает запрос для проверки актуальности при каждом запросе ресурса, то такая политика кэша называется *строго согласованной*. Если кэш использует эвристические процедуры для оценки сроков годно-

сти кэшированного ответа без взаимодействия с исходным сервером, то такая политика называется *слабо согласованной*. В зависимости от вида кэша и типичного набора кэшированных ответов может быть подходящей либо первая, либо вторая политика. Эвристика, основанная на фиксированном времени хранения, а также эвристики, основанные на времени жизни, относятся к слабо согласованным политикам. Эти слабо согласованные подходы отличаются тем, какой компонент берет ответственность за определение периода актуальности ресурса:

1. **Эвристика, основанная на фиксированном времени хранения.** Кэш может хранить ответ в течение фиксированного интервала времени без проверки актуальности. Однако сервер обещает уведомить кэш в случае изменения ресурса в *течение* периода хранения. После истечения этого периода, кэш может проверить ресурс на актуальность. Данный подход, описанный в [LC97], переносит накладные расходы по проверке актуальности ресурсов на сервер, который теперь должен взаимодействовать со всеми кэширующими прокси-серверами, которым он обещал сообщить об обновлениях ресурсов. Если исходный сервер должен уведомить большое число прокси-серверов, то этот способ не слишком подходит. Отметим, что при этом требуется взаимодействие для того, чтобы сервер мог уведомить кэши об истечении времени хранения. Алгоритмы, основанные на фиксированном времени хранения, широко обсуждаются в литературе, но они еще не реализованы в популярных кэширующих прокси-серверах. Одна из причин заключается в том, что они требуют совместных действий сервера и кэша.
2. **Подход, основанный на времени жизни** (TTL — Time To Live). С каждым ответом связано время истечения срока его годности. Когда этот интервал времени заканчивается, ответ рассматривается как устаревший. В течение срока годности кэш не перепроверяет его актуальность и, тем самым, остается возможность того, что будет использован устаревший ответ. Значение TTL может изменяться в зависимости от ресурса и основано на следующих факторах:

   Время истечения срока годности определено в поле заголовка ответа. Кэш может его использовать непосредственно или модифицировать на основе своей политики. Если время годности ресурса не указано, то кэш может назначить определенное значение TTL для ресурсов данного вида.

   Частота запросов кэшированного ресурса. Ресурсам, запрашиваемым чаще, можно назначить большее TTL.

   Мобильная среда. Непостоянное подключение и низкая скорость соединения пользователей мобильных устройств [HL96] может привести к заданию специальных значений TTL для ответов, запрашиваемых такими пользователями.

   Время последней модификации ресурса. Адаптивный алгоритм может предположить, что недавно модифицированный ресурс может снова измениться [Cat92]. Следовательно, кэш может назначить меньшее значение TTL для недавно измененного ресурса. Кэшированному ответу может быть назначено большее TTL, если на исходном сервере ресурс не изменялся в течение длительного времени. Информация о времени изменения берется из заголовка ответа **Last Modified**.

Подходы, основанные на TTL, являются достаточно популярными.

Поддержание согласованности может оказать существенное влияние на время ответа кэша, т.к. каждый запрос для проверки актуальности требует установления со-

единения с исходным сервером [Joh99]. Высокая цена установления соединения с исходным сервером приводит к необходимости сокращать число запросов на обновление. Как было отмечено в главе 10 (раздел 10.3.2) и в ряде исследований [AFJ99, KW97], ответ **304 Not Modified** составляет значительную долю (от 10 до 30 процентов) всех ответов. В главе 13 (раздел 13.1) будут обсуждены некоторые методы сокращения числа ответов **304 Not Modified**. Такие методы снижают число отдельных соединений, требующихся для проверки актуальности ресурсов, путем использования «подсказок» исходного сервера, указывающих, когда ресурс изменился.

11.8. Скорость изменения ресурсов

Скорость, с которой меняются различные ресурсы в Web, варьируется очень сильно. Некоторые ресурсы, будучи однажды созданы, уже никогда не меняются. Другие меняются при каждом запросе. Скорость изменения ресурсов определяет значимость следующих вопросов, определяющих необходимость кэширования:

- Какие ответы необходимо кэшировать?
- Имеется ли корреляция между скоростью изменения ресурса и частотой обращения к нему?
- Как долго хранить ответ в кэше?

Обнаружение изменений в ресурсах на уровне файлов достаточно легко выполнимо. Однако кэширующий прокси-сервер не может быть осведомлен, как часто динамически изменяемый ресурс меняется на исходном сервере. Сервер может получить некоторую информацию об этом из анализа URL ресурса. Например, если в тексте URL содержится подстрока "cgi" или знак вопроса "?", то можно предположить, что ответ создается динамически. Однако это может оказаться не всегда верным. Точно также, некоторые ресурсы, не содержащие ни подстроки "cgi", ни знака вопроса "?", могут, тем не менее, изменяться динамически. Динамически созданные ресурсы могут возвращать одинаковое содержимое для каждого запроса, а некоторые ресурсы, которые могут рассматриваться как статические, могут на самом деле изменяться довольно часто. Например, сайт **www.cnn.com** регулярно обновляет свои страницы путем добавления новостного Web-содержания. Некоторые страницы изменяются при каждом доступе к ним, например, на странице может оказаться счетчик доступа к ней. Таким образом, знание *скорости изменение* набора ресурсов может помочь в принятия решения о необходимости и длительности кэширования.

Влияние скорости изменения ресурсов на кэширование может определяться категорией ресурса и следующими факторами:

- **Тип.** Изображения меняются гораздо реже, чем текстовые ресурсы, и, таким образом, могут дольше не обновляться.
- **Частота и периодичность изменений.** Если известно, что ресурс меняется только раз в день (например, комикс на сайте **www.dilbert.com**), то и кэшировать его надо тоже только в течение суток. Для такого ресурса нет необходимости проверять актуальность, так как известно, что в течение определенного времени он меняться не будет.
- **Размер ресурса.** Если небольшие ресурсы имеют тенденцию часто меняться, то для решения вопроса о времени кэширования этот фактор может быть учтен вместе с частотой обращения к таким ресурсам.

Если ресурс имеет время истечения срока годности, тогда нет необходимости определять его эвристически. Однако исходный сервер может не указывать каких-либо определенных сроков годности ресурса. Одной из причин такого поведения может оказаться, что сервер сам не знает точного времени изменения ресурса, другой причиной является то, что указание сроков годности сокращает трафик от кэширующих прокси-серверов. Это в свою очередь уменьшает число обращений к исходному серверу, от которого зависит плата за размещаемую рекламу.

11.9. Протоколы кэширования

В зависимости от своей организации в группы кэши могут посылать и принимать информацию о ресурсах, в которых они заинтересованы. Такое взаимодействие является внешним и отделяется от сообщений, содержащих запросы и ответы, передаваемые между клиентом и сервером. Межкэшевый обмен может использовать протокол HTTP. Однако обычно используется специальный облегченный протокол. Если кэши имеют иерархическую организацию, то на одном уровне иерархии кэш может контактировать с другим для запроса хранящегося в нем о объекта. На запрос об отсутствующем ресурсе может ответить один или несколько серверов, имеющих такой ресурс. Получить копию ресурса из локального кэша выгоднее, чем от исходного сервера. Однако ожидание ответа от всех прокси-серверов данного уровня иерархии может увеличить время ожидания ответа пользователем. Далее будет рассмотрено несколько известных решений для сокращения стоимости межкэшевых взаимодействий.

Ниже также будут обсуждены несколько протоколов, которые различаются своими возможностями и применением. Основной целью создания этих протоколов является исследование различных подходов к взаимодействию кэшей друг с другом. Будут обсуждены четыре протокола: Internet Cache Protocol (ICP), Cache Array Resolution Protocol (CARP), Cache Digest Protocol (CADP) и Web Cache Coordination Protocol (WCCP).

11.9.1. Протокол Internet Cache Protocol (ICP)

Кэш, в котором отсутствует запрашиваемый объект, может проверить его наличие в другом кэше. Такой метод взаимодействия отличается от традиционного запроса ресурсов у исходного сервера. В этом случае кэши являются как источниками, так и получателями сообщений. Для этого необходим специальный протокол межкэшевого взаимодействия. Одним из самых ранних протоколов для взаимодействия кэшей и был разработан Internet Cache Protocol (ICP) [WC97b, WC97a]. Этот протокол является примером протокола запросов — сообщение, посланное клиентским кэшем, является запросом о наличии кэшированной копии определенного ответа, необходимого клиентскому кэшу. Популярность ICP основана на том, что он реализован в свободно распространяемом и широко используемом кэширующем прокси-сервере Squid [WC98].

Протокол ICP оказался подходящим и для иерархических наборов кэшей, взаимодействующих на каждом уровне иерархии и имеющих общего родителя. При перемещении вверх по иерархии эта схема повторяется, соответствуя перемещению к центральному кэшу, который может быть подключен к другому центральному кэшу в другой локальной сети по такой же схеме. Центральный кэш может иметь в качестве родителя региональный кэш, а набор региональных кэшей может в каче-

стве родителя иметь национальный кэш. Предположим, что некоторый кэш (назовем его исходным) не имеет запрашиваемого ресурса. Он посылает ICP-запрос (обычно это UDP-сообщение) ко всем своим соседям на данном уровне иерархии одновременно. Успешный ответ будет указывать о наличии требуемого ресурса в одном из кэшей данного уровня, и исходный кэш может запросить этот ресурс, используя протокол HTTP. Если в кэшах данного уровня иерархии отсутствует данный ресурс, то исходный кэш пошлет ICP-запрос родительскому кэшу. Возможен также вариант, что ответа от кэшей данного уровня иерархии не поступит за определенный интервал времени, что инициирует в исходном кэше событие, связанное с таймаутом. Родительский кэш повторяет указанную процедуру. Если от кэшей не поступит ответа, то исходный кэш запросит данный ресурс у исходного сервера. Предположение, лежащее в основе ICP-протокола, заключается в том, что передача кэшам набора ICP-запросов столько раз, сколько имеется уровней в иерархии, выполняется быстрее, чем взаимодействие с исходным сервером. Кроме того, некоторая оптимизация помогает сократить общие издержки. Например, когда ответ возвращается от исходного сервера или кэша в иерархии, то промежуточные кэши могут сохранять ответ для будущего использования. Не ответившие соседние кэши сообщают о возможности своего взаимодействия с исходным кэшем.

Хотя региональные и национальные прокси могут сократить расстояние, которое проходят извлекаемые ресурсы для большого числа пользователей, но переходы между уровнями иерархии могут снижать производительность. Если на текущем уровне иерархии ресурс не найден, то это дает дополнительный вклад в общую задержку. Даже если ресурс будет обнаружен в национальном прокси-сервере, то это все равно обусловит задержку, связанную с передачей ресурса клиенту, сделавшему запрос.

11.9.2. Cache Array Resolution Protocol (CARP)

Протокол Cache Array Resolution Protocol (CARP) [CAR] определяет механизм, с помощью которого группа кэширующих прокси-серверов может функционировать как единый логический кэш. Набор ответов, который коллективно кэшируется группой прокси-серверов трактуется как один большой кэш. Специальные хэш-функции используются для кодирования URL ресурсов, хранящихся в различных кэшах. Клиент, пытающийся найти кэшированный ресурс, может направить запрос соответствующему кэшу, кодированный с помощью этой функции. Хэш-функция преобразует запрошенный URL и идентификатор требуемого прокси-сервера создавая специальный *путь разрешения запроса*. По сравнению с ICP протокол CARP имеет строго *определенный* путь разрешения запроса, что исключает необходимость посылок дополнительных запросов. По сравнению с ICP в CARP используется гораздо меньше повторных запросов. CARP использует как HTTP, так и вызов удаленных процедур для взаимодействия прокси-серверов друг с другом. CARP связывает прокси-сервер с коэффициентом нагрузки, который может быть принят во внимание до того, как запрос будет направлен данному прокси-серверу. CARP [CAR] реализован основными производителями программных продуктов.

Когда конфигурация кэширующей системы изменяется путем удаления или добавления прокси-сервера, то кэшированные URL должны быть реорганизованы. При замене содержимого кэша хэшированные значения должны быть вычислены снова. URL не могут быть представлены в нескольких кэширующих прокси-серверах, и, следовательно, выравнивание нагрузки прокси-серверов затруднено. Высокая популярность некоторых ресурсов приводит к тому, что лишь небольшое под-

множество кэшей получает основную часть запросов. Если поиск кэшированного ответа оказывается неудачным, то на некоторое время прокси-сервер должен считаться неактивным.

11.9.3. Cache Digest Protocol (CADP)

Протокол Cache Digest Protocol [HRW98] является расширением протокола ICP. Основная его цель заключается в обмене *дайджестами* кэшированных ресурсов. Дайджест представляет собой сжатое описания кэшированных ресурсов и определяет наличие набора ресурсов в кэше. Когда кэш имеет дайджесты всех других кэшей на данном уровне иерархии, то можно быстро проверить наличие искомого объекта. Если поиск в дайджесте оказывается успешным, то кэш становится кандидатом на получение запроса на искомый объект. Запрашивающий кэш может даже выбрать из *нескольких* кэшей, для которых поиск в оказался успешным. Если проверка по дайджесту оказалась неудачной, то взаимодействие с кэшем не осуществляется. В результате сокращается число сообщений, рассылаемых кэшам одного иерархического уровня.

Одной из очевидных проблем данного протокола является старение дайджестов и пересылка ошибочных данных. Объект может быть удален из кэша уже после создания дайджеста. Другой проблемой является размер набора дайджестов и обмен дайджестами на данном уровне иерархии кэшей. При наличии большого числа кэшей размер набора дайджестов становится очень большим. Проведенные исследования позволили уменьшить размеры дайджестов [FCAB98].

Для обмена дайджестами между кэшами может использоваться схема, принятая в ICP и основанная на протоколе UDP. Однако для надежности обмен дайджестами осуществляется с помощью HTTP-сообщений поверх TCP. Дайджесты могут рассматриваться как обычные ресурсы, к которым для проверки актуальности применима технология обновления ресурсов HTTP (с помощью заголовков **Expire** и **If-Modified-Since**).

11.9.4. Web Cache Coordination Protocol (WCCP)

В отличие от высокоуровневых протоколов типа ICP и CARP протокол Web Cache Coordination Protocol (WCCP) [WCC99] является координационным механизмом, который связан с сетевым уровнем. Назначением WCCP является перехват HTTP-запросов и переадресации их кэшу. Поскольку запрос может оказаться отвергнутым (если кэш по какой-либо причине будет недоступен), то необходим координационный механизм. Задача координатора заключается в выравнивании нагрузки между множеством кэшей. Периодически проверяя работоспособность кэша, данная технология гарантирует, что пакеты не будут посланы неработоспособному кэшу.

Такой координационный механизм заложен в ядро протокола WCCP, который реализован как часть технологии Cisco Cache Engine. Этот механизм настроен на получение Web-запросов, перенаправленных ему маршрутизатором. Маршрутизатор, поддерживающий протокол WCCP, способен проанализировать все IP-заголовки и TCP-пакеты, поступающие на 80-й порт (стандартный порт HTTP), и перенаправить их кэширующему серверу. Кроме того, WCCP-маршрутизатор периодически связывается с кэширующим сервером, чтобы убедиться в его доступности.

11.10. Программное и аппаратное обеспечение кэширования

Кэширование, как это было рассмотрено в главе 3, может быть обеспечено с помощью прокси-серверов. Это решение требует добавления прокси-серверу некоторой функциональности. Кроме того, кэширование может выполняться с помощью специально предназначенного для этого оборудования. В первой части этого раздела в качестве примера будет рассмотрено программное решение — Squid — прокси-сервер, широко используемый для кэширования. Во второй части будут рассмотрены различные аппаратные решения, требующие установки и настройки новых устройств.

11.10.1. Программное обеспечение кэширования. Squid

Squid является широко используемой кэширующей системой с открытым исходным кодом, которая работает на большом числе платформ, включая Unix, Solaris, Linux, FreeBSD, NetBSD и OS/2. Другими популярными кэширующими системами являются NetApp [Neta], Inktomi [Ink] и Novell Bordermanager [Nov]. Кроме HTTP, Squid способен работать и с другими протоколами (включая FTP, SSL, Gopher и WAIS). Он является настраиваемым кэшем: можно определить, по каким правилам должен обрабатываться кэшированный ответ (например, тип содержания, продолжительность кэширования и т.д.). Для получения более полного представления о Squid можно ознакомиться с руководством пользователя Squid [Pea97].

Как и для любого другого прокси-сервера, для Squid необходима настройка на стороне клиента. Браузеры должны быть настроены на использование прокси-сервера для любого протокола, поддерживаемого Squid (как это обсуждалось в главе 2, раздел 2.4.2). Например, номер порта для HTTP устанавливается равным 3128, поскольку именно его Squid использует для взаимодействия с Web-клиентами по умолчанию.

Squid может общаться с другими кэшами на данном уровне иерархии по протоколу ICP. Взаимодействие с другими кэшами осуществляется по протоколу UDP через порт 3130 по умолчанию, хотя для взаимодействия может использоваться и HTTP.

Иерархические отношения между кэшами определяются с помощью конфигурационных файлов, иерархия может быть локальной, региональной, национальной и даже интернациональной. Условное кэширование может быть настроено через границы доменов. Например, один Squid может использоваться для кэширования ряда национальных доменов, таких как .fr и .au. Для кэширования определенного сайта, прокси-серверы, расположенные вблизи от него, должны быть зарегистрированы в центральной базе данных.

Squid контактирует с набором кэшей на данном уровне иерархии, используя UDP и предполагая, что кэш, который первым сообщает о наличии ресурса, столь же быстро сможет выдать и ответ. Другими словами, Squid собирает информацию, касающуюся загрузки сети, основанную на времени ответа. Ответ, полученный от кэша на том же уровне иерархии не должен кэшироваться на запрашивающем кэше. Вместо этого запрашивающий кэш может просто позволить кэшу на том же уровне иерархии служить кэшем для этого объекта, если кэш находится в той же высокоскоростной сети с малым временем ожидания. Если кэшированной копии запрашиваемого ресурса нет ни на текущем уровне иерархии, ни в родительском

кэше, то Squid направляет запрос исходному серверу. Ответ сервера может быть записан в кэш перед отправкой его клиенту.

Squid способен работать в кластере: множество кэшей в одном сетевом сегменте могут действовать как кэши одного уровня. Преимущество кластерного кэширования заключается в высокой надежности и низкой цене, так как используются несколько дешевых компьютеров, а не мощный и дорогой сервер. DNS-сервер также участвует в схеме кластерного кэширования. Определение адреса кэширующего сервера выполняется таким образом, что браузер может обращаться к любому кэшу кластера.

Squid использует списки управления доступом (ACL — Access Control List) для определения клиентов, которые могут к нему обращаться как к прокси-серверу. Список IP-адресов используется для фильтрации входящих запросов. Различные адреса могут быть определены для различных протоколов. Например, группа клиентов может использовать Squid как кэш для HTTP-запросов, но не для FTP-запросов. Squid может быть настроен так, чтобы предотвратить доступ к некоторым Web-серверам; ACL в этом случае состоит из набора доменов. Например, рассмотрим список управления доступом с меткой **dumb,** содержащий два домена **notbridght.com** и **nitwit.com**:

```
acl dumb dstdomain notbright.com nitwit.com
http_access deny dumb
```

Это будет предотвращать запросы к любому из этих двух доменов от клиентов, находящихся позади Squid.

Squid также может функционировать как *акселератор* для Web-сервера: он может быть расположен перед одним или несколькими исходными серверами и обрабатывать входящие запросы. Для этого Squid должен работать на стандартном для протокола HTTP восьмидесятом порту. После этого Squid можно настроить для обработки запросов к одному или более исходным серверам и в случае необходимости переадресовывать входящие запросы на исходные серверы, находящиеся за Squid. Squid может обрабатывать заголовок **Host** протокола HTTP/1.1 (см. главу 7, раздел 7.8) для определения, какой исходный сервер должен получить запрос. Squid может обеспечивать более высокую производительность, чем исходный сервер. Когда он настроен как акселератор, то может взять на себя заботу об обновлении ресурсов при изменении их на исходном сервере. Акселератор на основе Squid работает как зеркало и способен отслеживать изменения на исходном сервере. Squid не обязан обрабатывать все запросы. Имеется интересная возможность разделить ресурсы сервера на кэшируемые и некэшируемые. Squid обрабатывает запросы на кэшируемые ресурсы, а Web-сервер, запущенный на порту, отличном от стандартного, обрабатывает запросы на некэшируемые ресурсы. Например, Squid может направлять все CGI-запросы исходному серверу. Если для внешней сети сделать видимым только Squid, то можно предотвратить атаку на исходный сервер, расположенный позади Squid. Например, следующая настройка, сделанная на Squid

```
acl safe dstdomain safe1.com safe2.com
http_access deny !safe
```

позволяет сделать доступными только серверы **safe1.com** и **safe2.com**. Кэширующий прокси-сервер будет выдавать отказ при запросе любого другого домена.

Подобно многим другим кэширующим прокси-серверам, Squid при записи ресурса в кэш использует модель истечения срока годности. Последние версии Squid используют различные модели, основанные на частотах обновления ресурсов. Вместо ожидания истечения срока годности ресурса актуальность объекта проверяется

только при необходимости. Истечение срока годности ресурса проверяется во время запроса, если объект устаревает, то он обновляется с исходного сервера. При необходимости в результате обновления записывается новая версия ресурса. Squid поддерживает заголовки **Cache-Control** и **Expires**, которые могут присутствовать в заголовках запросов и ответов. Предпочтение отдается заголовку клиентского запроса (например, **Cache-Control: max-age**), а не заголовку **Expires** ответа сервера. Можно настроить Squid так, что он будет изменять срок годности, указанный в заголовке **Expires** ответа.

Последние версии Squid также поддерживают дайджесты кэшей. Новая версия способна использовать особенности HTTP/1.1, включая долговременные соединения и аутентификацию прокси-сервера. Squid поддерживает многопоточное исполнение, если его поддерживает операционная система. В случае перехватывающих прокси-серверов пакеты, полученные на один порт, перенаправляются на другой, а Squid действует как получатель перенаправленных пакетов. Squid также поддерживает протокол WCCP (обсужденный с разделе 11.9.4).

11.10.2. Аппаратное обеспечение кэширования

Кэширование может быть выполнено также с помощью специального аппаратного обеспечения. При использовании кэширующих прокси-серверов имеет место несколько проблем:

- браузеры пользователей должны быть настроены для взаимодействия с кэширующим прокси-сервером;
- при наличии нескольких кэшей, необходима организация взаимодействия между кэшами для предотвращения дублирования данных;
- если прокси-сервер в какой-то момент недоступен, то придется перенастроить браузеры пользователей.

Альтернативным способом решения проблем кэширования является размещение специфической аппаратуры в сети. Такое решение связано с перехватом трафика на различных уровнях (сетевом, транспортном и прикладном). Перехваченный трафик направляется к одному или более устройствам, выполняющим функции кэширующего прокси-сервера. Аппаратное решение для кэширования может быть отнесено к одной из двух категорий: редиректору и комплексному решению. Редиректор может быть либо маршрутизатором, либо коммутатором, перехватывающим запросы. Комплексное решение кроме функций перенаправления осуществляет и кэширование.

ПЕРЕХВАТЫВАЮЩИЕ ПРОКСИ-СЕРВЕРЫ И РЕДИРЕКТОРЫ

Функционирование редиректора основано на перехвате трафика между запрашивающим клиентом и исходным сервером. Аппаратный кэш может использовать инфраструктуру сети для перехвата трафика. Такие устройства обычно размещают на границе сети провайдера, они способны просматривать все пакеты, проходящие между клиентами и Internet. Перехват исключает необходимость для пользователя настраивать браузер на взаимодействие с прокси-сервером. Во время перехвата проверяется как Web-, так и локальный трафик, обусловливая тем самым задержки при передаче локального трафика.

Конечный пользователь не имеет контроля над перехватывающим прокси-сервером. Как было отмечено в разделе 11.4, пользователь не ощущает присутствия перехватывающего прокси-сервера. Пользователь, принимая ответ от перехваты-

вающего прокси-сервера, полагает, что он пришел от исходного сервера. Предположим, что кэш перехватывающего прокси-сервера функционирует некорректно и отсылаемый им ответ будет либо устаревшим, либо не соответствовать протоколу HTTP. У пользователя нет способа узнать причину и место возникновения проблемы. Перехватывающий прокси-сервер обусловливает проблемы в области безопасности, так как пользователи не знают о возможной угрозе своей безопасности со стороны перехватывающего прокси-сервера.

Перехватывающие прокси-серверы используются достаточно широко. Перехват или перенаправление могут быть осуществлены с помощью маршрутизатора или коммутатора. *Маршрутизатор* может иметь специальную политику для проверки одного или более портов (порт 80 для протокола HTTP) и перенаправления этого трафика. Это можно рассматривать как коммутатор уровней L3/L4, т.е. на третьем — сетевом и четвертом — транспортном уровнях. Коммутатор L4 может проверить TCP-пакет SYN и решить, необходимо ли перенаправлять запрос.

Реализованная Cisco политика маршрутизации позволяет перехватывать весь сетевой трафик на 80-ом порту. Однако слепое перенаправление трафика на другой порт может привести к снижению производительности, так как многие сообщения не должны кэшироваться. Например, нельзя кэшировать динамически созданные сообщения. *Web-коммутатор* — это сетевое устройство, имеющее программное обеспечение для перехвата и перенаправления трафика на один или более прокси-сервер. Коммутатор обычно работает на транспортном уровне (L4) и только перехватывает пакеты TCP/IP на 80-м порту. В результате трафик, не относящийся к Web, не изменяется. Коммутатор L4 может быть также расположен на границе сети. Коммутатор является более простым, более дешевым и имеющим меньше функциональных возможностей устройством, чем маршрутизатор.

Коммутаторы могут анализировать *содержание* трафика, а не только номер порта. Интеллектуальные коммутаторы могут анализировать часть заголовков на прикладном уровне. Анализ содержания требует затрат, т.к. перехватывающий прокси-сервер должен работать с заголовками HTTP. Далее коммутатор может принимать эвристические решения о необходимости кэширования Web-содержания. Если Web-содержание не кэшируется, то коммутатор передает запрос непосредственно исходному серверу без переадресации его кэшу. Кроме проверки типа содержания, коммутатор может выполнять избирательную переадресацию для определенных доменов и фильтрацию возвращаемого содержания. Коммутатор, проверяющий содержание, является примером коммутатора уровня 4+ (уровень 5 или 7).

Проверка заголовков на прикладном уровне (в данном случае заголовков HTTP) может помочь в принятии решений о необходимости кэширования. Для принятия соответствующего решения может быть проанализирован ряд HTTP-заголовков (номер версии протокола, **Cache-Control, Cookie** и т.д.). Например, если присутствует заголовок **Cache-Control: no-cache**, то нет смысла переадресовывать запрос кэшу. Однако для слежения за заголовками прикладного уровня, коммутатор L5 неизбежно должен прекращать соединение на транспортном уровне (TCP). Для выравнивания нагрузки можно выполнить распределение запросов по различным кэшам с помощью хэш-функций, основываясь на URL.

Проверка заголовков прикладного уровня до обработки запроса в общем случае весьма проблематична. Откладывание операции до момента проверки заголовков прикладного уровня может увеличить время ожидания. Проверка может выявить отсутствие заголовков, помогающих принять решение. Далее, как было сказано ранее, URL не является достаточно хорошим индикатором необходимости кэширования. Решение, основанное на том, что URL содержит подстроку **cgi**, может оказать-

ся необоснованным. Независимо от того, полезным или нет оказался анализ заголовков и URL, соединение на транспортном уровне уже разорвано и необходимо устанавливать новое соединение с сервером, находящимся на пути к клиенту.

Обычно аппаратуру, непосредственно предназначенную для кэширования содержания, отделяют от перехватывающих устройств. Может использоваться один или несколько кэшей, а коммутатор переадресовывает запрос соответствующему кэшу. Отделение перехвата от кэширования с помощью нескольких кэшей позволяет осуществлять выравнивание нагрузки и устранить отказы из-за выхода из строя одного кэша.

Поскольку перехватывающий прокси-сервер располагается на пути всего трафика между клиентом и сетью, то при выходе из строя прокси-сервера клиент будет отключен от Internet. Способом предотвращения таких отказов является отслеживание работы перехватывающего прокси-сервера и отключение механизма перехвата в случае его отказа, что позволяет передать клиентский запрос непосредственно исходному серверу. Более устойчивый механизм [Fou] повторяют запрос исходному серверу, если кэш не обнаруживает соответствующих пакетов TCP SYN/ACK для SYN.

КОМПЛЕКСНЫЕ АППАРАТНЫЕ РЕШЕНИЯ

Некоторые провайдеры предпочитают приобретать устройства, позволяющие решить все проблемы кэширования. Преимущество таких устройств заключается в сокращении затрат на администрирование по сравнению с редиректорами. Используемая в соответствующей операционной системе модель кэширования объединяется со специфической аппаратной платформой. Обычно такое оборудование монтируется в стойку.

Имеются две категории таких комплексных решений: устройства первой категории требуют отдельного редиректора, в устройствах второй категории имеется внутренний редиректор. Современные типы устройств не требуют отдельных коммутаторов L4/L5. Гибридная версия устройств включает модули, которые осуществляют коммутацию на четвертом уровне. При обнаружении отказа кэша устройство может посылать запрос непосредственно исходному серверу.

Подход, заложенный в такие устройства, не всегда обеспечивает необходимую гибкость из-за специфических особенностей используемой операционной системы. С комплексными аппаратными решениями обычно поставляются специализированные ядра ОС и модифицированные файловые системы. Изменение трафика может потребовать перенастройки кэша. Возможности перенастройки (добавление или удаление кэшей) зависят от интерфейса, реализованного в комплексном решении. Потребитель может оказаться неспособным собрать подробную информацию о трафике своих кэшей. В общем случае ему приходится использовать те виды измерений, которые обеспечиваются комплексным решением. Универсальная техника измерений в этом случае не может быть использована по причине использования специализированных ядра ОС и файловой системы.

Однако комплексные решения весьма популярны и занимают существенную долю рынка. Они обеспечивают необходимую функциональность кэширования с минимальными затратами на поддержание функционирования со стороны потребителя. Пока будет присутствовать необходимость в комплексных решениях, потребителям придется удовлетворяться информацией, обеспечиваемой ими.

11.11. Препятствия для кэширования

Хотя кэширование и помогает сократить время ожидания ответа пользователем, нагрузку на сеть и исходный сервер, имеется несколько препятствий для кэширования. Обсудим некоторые из проблем, имеющих отношение к кэшированию, и попытки их преодолеть. Начнем с отключения кэширования (cash busting) — преднамеренных действий некоторых исходных серверов, чтобы предотвратить кэширование ответов. Затем обсудим потенциальные возможности нарушения конфиденциальности, которые могут произойти в результате кэширования запросов и ответов.

11.11.1. Отключение кэширования

Не все исходные серверы заинтересованы в кэшировании ответов прокси-серверами. Не все ответы можно кэшировать, у сервера должна быть возможность решать какой ресурс можно кэшировать, а какой нет. Когда прокси-сервер доставляет кэшированное Web-содержание, есть некоторая вероятность того, что это Web-содержание уже устарело. Протокол HTTP/1.1 предоставляет серверу возможность выразить свои предпочтения о кэшировании отдельных ресурсов. Однако ранее протокол не был достаточно гибким, и сервер был вынужден прибегать к технике, не предусмотренной протоколом для того, чтобы убедиться, что неправильное кэширование не нарушает функциональности Web-сайта.

Более важно, что у исходного сервера нет способа узнать число раз, когда клиенту можно доставлять одно и то же содержание. Даже те сервера, которые заинтересованы в снижении собственной нагрузки, могут захотеть узнать количество запросов к ним. В случае Web-страниц, на которых имеются реклама, исходному серверу нужно узнать точное число клиентов, которые загружали его страницы и подсчитать, сколько пользователей видело рекламу. Многие Web-сайты получают доходы, зависящие от числа запросов на ресурсы, т.е. от числа пользователей, видевших рекламу.

По этой и по некоторым другим причинам серверы используют отключение кэширования. Отключение кэширования представляет собой методику, позволяющую предотвратить кэширование ответа. Первоначально такая методика рассматривалась как простой механизм управления сервером. В более широком смысле отключением кэширования является любая методика, препятствующая кэшированию ресурса, который в обычном случае может кэшироваться прокси-сервером или браузером. Обычно отключение кэширования включает установку таких значений параметров кэширования, которые эффективно отключают кэширование. Например, установка значения заголовка **Expires** на уже прошедшее время гарантирует, что ресурс считается устаревшим сразу после поступления в кэш. Кэш не хранит такие ответы. Для предотвращения кэширования исходный сервер может включить в заголовок ответа директивы **Cache-Control: no cache** или **Cache-Control: no store**. Другая методика, которая может использоваться для отключения кэширования — умышленное изменение страницы без фактического изменения ее содержимого. Например, объявление на странице может только слегка перемещено, изменяя тем самым компоновку страницы.

Рассмотрим два способа снижения мотивации для отключения кэширования. Первый способ использует HTTP-заголовки, другой предполагает перемещение изображений с исходного сервера на прокси-сервер.

Одной из попыток решения проблемы отключения кэширования было предложение методики названной *подсчет числа обращений к ресурсу (hit-metering)*

[ML97]. Методика вводит новый HTTP-заголовок, **Meter**, который будет использоваться кэшем для передачи исходному серверу числа обращений к ресурсу. Основная цель введения данного заголовка — информировать исходный сервер о приблизительном числе пользователей, получивших доступ к кэшированному документу. Была надежда на то, что указание числа обращений уменьшит количество некэшируемых документов. Фактически исходный сервер мог явно указать прокси-серверу вернуть кэшированный ответ клиентам ограниченное число раз перед тем, как снова обратиться к исходному серверу. Прокси мог сообщить количество обращений через заголовок **Meter** при любом контакте с исходным сервером, например во время проверки актуальности ресурса. Если прокси-сервер удалял ответ из кэша, то обязан был сообщить исходному серверу количество обращений к данному ресурсу. Для этой цели использовался запрос с методом **HEAD**, который требовал отдельного сеанса связи с исходным сервером. Такой метод измерения количества обращений не оказался успешным. К сожалению, не удалось качественно реализовать измерения числа обращений к ресурсам в Web-компонентах. Администратор сайта не мог быть уверен в том, что кэш корректно указывает число обращений. Отключение кэширования, в некотором смысле, оказалось самым простым и надежным путем для администраторов исходных серверов.

Вторая методика, называемая *ad-insertion* [Ami99], предполагает, что прокси-сервер добавляет рекламные объявления на страницу и снимает эту задачу с исходного сервера. Такой подход требует, чтобы прокси-сервер взаимодействовал с исходным сервером таким образом, чтобы последний был уверен в том, что рекламные объявления действительно помещаются на Web-страницы. Выигрыш агентов пользователя и прокси-серверов двойной. Во-первых, кэширование не отключается, что дает возможность использовать кэшированный ответ в течение большего времени. Во-вторых, пользователь не должен ждать пересылок рекламных объявлений с исходного сервера, поскольку эти объявления вставляются прокси-сервером.

Другая методика, с помощью которой можно подсчитывать число обращений к ресурсам без отключения кэширования, связана с использованием «Web-жучков». «Web-жучки» являются маленькими изображениями, которые невидимы пользователю и размещаются на странице в качестве счетчиков. Большинство пользователей позволяют браузеру загружать изображения, встроенные в документ. Путем проверки содержимого HTML-страницы можно обнаружить присутствие таких «изображений». Загрузка «Web-жучков» может привести к вызову сценария на исходном сервере и обмену cookies. Таким образом, могут вестись пользовательские профили и определяться, какие рекламные объявления должны быть показаны.

11.11.2. Проблемы конфиденциальности кэширования

Нарушение конфиденциальности пользователей является одной из широко обсуждаемых тем. Пользовательский запрос ресурса в Web может использоваться прокси-сервером для построения профиля пользователя. Ответ исходного сервера может быть кэширован серверами-посредниками на пути к клиенту против желания исходного сервера, который может хотеть, чтобы ответ был направлен непосредственно запрашивающему пользователю. Таким образом, конфиденциальность пользователя и информационного содержания могут быть нарушена. Протокол HTTP/1.1 предлагает новые заголовки и механизмы для защиты конфиденциальности. Например, директива запроса и ответа **no-store** предотвращает кэширование сообщения на всех этапах передачи запроса и ответа. Кэш может быть персональ-

ным (для одного пользователя) или коллективным (многие пользователи получают данные из кэша), и это различие является достаточно важным. Имеется ряд директив, позволяющих управлять кэшированием и влияющих на конфиденциальность. Однако для большинства HTTP-сообщений нет необходимости включать такие директивы, даже если они присутствуют, то могут и не выполняться. У клиента нет никаких средств, с помощью которых он мог бы удостовериться, что ресурс не кэшировался на пути своего прохождения от сервера к клиенту. Аналогичным образом исходный сервер тоже не может узнать, был ли ресурс где-нибудь кэширован.

Имеется интересный альтернативный взгляд на то, как кэширование может помочь сохранению конфиденциальности, заключающийся в том, что когда клиенты получают кэшированный ответ от прокси-сервера, то они на самом деле защищены от исходных серверов. Исходный сервер, заинтересованный в отслеживании пользователей, будет получать только IP-адрес прокси-сервера. Клиент должен доверять прокси-серверу, так как прокси-сервер имеет подробную и точную информацию о запросах, выданных всеми клиентами, находящимися позади него.

Использование перехватывающих прокси-серверов, которые практически невидимы для пользователя, является еще одним источником нарушения конфиденциальности. В то время как некоторые исходные серверы можно рассматривать как системы занимающиеся сбором информации о клиентах со всеми вытекающими последствиями, то пользователи не могут обратиться к перехватывающим прокси-серверам, так как об их присутствии неизвестно клиенту. Таким образом, пользователь не может знать о том, какая именно информация о нем имеется у перехватывающего прокси-сервера.

Популярность Web и легкость, с которой нарушается конфиденциальность, привлекли к этой проблеме повышенное внимание. Проблема конфиденциальности становится все более и более серьезной. Имеются законодательные попытки Европейского Союза защитить конфиденциальность пользователей и провайдеров Web-содержания. Некоторые корпорации в США имеют теперь в штате сотрудника отвечающего за безопасность пользователей (Chief Privacy Officer), обязанностью которого является проверка нарушений конфиденциальности потребителей.

11.12. Кэширование и репликация

Репликацию можно рассматривать как альтернативу кэшированию. При кэшировании запрашиваемый ресурс перемещается *ближе к пользователю*. *При репликации* Web-содержание исходного сервера копируется на зеркальный сайт. Клиенты могут иметь доступ к зеркальному сайту непосредственно, или запрос может быть переадресован. Преимущество репликации заключается в том, что на зеркальном сайте *все* ресурсы могут быть доступны в любое время, а не только ресурсы, кэшированные близкими к клиенту прокси-серверами. Зеркала могут использовать для обновления протокол, отличный от HTTP, и поддерживать тесную связь между собой для устранения устаревания ресурсов. Однако часто на зеркалах публикуются только некоторые наиболее часто используемые ресурсы. Другие переносятся на зеркала по необходимости.

Одним из преимуществ репликации над кэшированием заключается в модели обновления. Обновление ресурсов при репликации выполняется вне протокола HTTP. Список реплик заранее известен администраторам зеркал, что позволяет осуществлять групповую доставку ресурсов. При этом всего одна копия ресурса

передается по сети репликам. После того, как ресурс устаревает, для обновления также используется групповое вещание.

Для пересылки изменений зеркалам можно также использовать современные методы. В главе 15 (раздел 15.2) будут изложены основы механизма пересылки зеркалам только измененных ресурсов. Разработка данного механизма требует значительных усилий для обеспечения обратной совместимости протоколов. Доставка данных репликам может происходить и без таких сложностей. Более того, для зеркальных сайтов кодирование ресурсов может обеспечить некоторые преимущества. Например, зеркальный сайт может уметь обрабатывать новый формат данных, стандартно не поддерживаемый в Web. Кроме того, поскольку исходный сервер управляет ресурсом и знает, когда ресурс меняется, то он может принудительно доставить изменения без ожидания запроса от него. Устаревание ресурса может быть вовремя ликвидировано, хотя и ценой некоторых затрат, т.к. некоторые ресурсы могут меняться довольно часто.

Доступ к зеркальным сайтам может осуществляться несколькими способами:

- На сайте публикуется список зеркал, и пользователь может выбрать ближайшее к нему зеркало. Предполагается, что клиент из Новой Зеландии скорее выберет зеркало, находящееся в Австралии, чем в США. Запросы к сайту существенно перераспределяются на клиентском уровне. Общее время ожидания пользователей может быть существенно сокращено путем доступа клиентов к ближайшему зеркальному сайту. Это обычно делается для сайтов, обеспечивающих доступ к программному обеспечению большого объема.
- Когда запрос принимается исходным сервером, то он может выполнить его переадресацию на уровне HTTP с помощью кодов запросов **302 Moved Temporarily, 303 See Other** или **307 Temporary Redirect**. Клиент использует информацию в заголовке **Location** для контакта с репликой. Переадресация на уровне HTTP не всегда приятна для пользователя, поскольку приводит к задержке ответа, вызванной созданием двух соединений вместо одного.
- Пользователи могут быть переадресованы на зеркальный сайт довольно простым способом. DNS-сервер при преобразовании доменного имени, может возвращать различные IP-адреса в зависимости от адреса исходного запроса. Все клиентские запросы из Новой Зеландии могут переадресовываться на новозеландский зеркальный сайт в Окленде, а не на основной сайт в Индии. Пользователям не требуется явно указывать адреса зеркальных сайтов в своем браузере. Дополнительное преимущество динамического подключения заключается в том, что DNS может выполнять выравнивание нагрузки путем переадресации запросов к различным зеркальным сайтам (как это было описано в главе 5, раздел 5.3.5).

11.13. Распределение Web-содержания

Распределение Web-содержания представляет собой избирательную репликацию. Основная идея распределения Web-содержания заключается в распределении нагрузки исходного сервера. Это может быть сделано путем перераспределения некоторой части или всего Web-содержания, которое обычно доставляется с одного сервера, на несколько серверов. Для этого могут использоваться различные технологии, в том числе и DNS. Один из способов заключается в распределении основных и встроенных ресурсов. Основные документы — это документы-контейнеры, а

встроенные ресурсы, — изображения или скрипты, — являются частью Web-страницы. Серверы, используемые для доставки встроенных ресурсов, называются *серверами распределенного Web-содержания*. Они могут быть расположены близко к исходному серверу либо в любых других местах и содержать реплицированное Web-содержание. В момент запроса сервис распределенного Web-содержания пытается обнаружить ближайший к пользователю сервер, содержащий встроенные изображения. Термин «ближайший» может трактоваться либо в географическом смысле, либо в сетевом, либо в смысле минимизации задержки ответа. Такой подход сокращает нагрузку на основной сервер и уменьшает время ответа на стороне конечного пользователя.

Цель распределения Web-содержания не отличается от целей кэширования. Оба подхода перемещают Web-содержание ближе к пользователю для сокращения времени ожидания на стороне пользователя и уменьшения нагрузки на исходный сервер. Прокси-серверу в процессе кэширования необходимо беспокоиться о согласованности Web-содержания и проверять актуальность кэшированных ресурсов. При использовании распределенного Web-содержания исходный сервер управляет Web-содержанием и может выполнять произвольное перемещения Web-содержания на другие серверы. Это, в свою очередь, позволяет перераспределить Web-содержание в соответствие с текущими потребностями.

При использовании зеркал большая часть сайта реплицируется на несколько серверов в Internet. При подходе, использующем распределение Web-содержания, исходный сервер решает, какой из ресурсов должен быть реплицирован, и, что более важно, перепоручить задачу поддержания зеркал другим организациям. Исходный сервер обязан уведомлять такие организации, занимающиеся распределением Web-содержания, об изменении ресурсов.

Рассмотрим в качестве примера распределение Web-содержания, предложенное Akamai [Aka]. Сайт, который распределяет часть своих ресурсов с помощью Akamai, должен изменить их URL, добавив специальный префикс. Префикс включает строку с именем компьютера, например, **a1025o.akamaitech.net**. Посредством DNS это имя преобразуется в IP-адрес зеркального сервера в предположении, что на нем имеется копия ресурса. Решение, какой IP-адрес возвратить клиенту, принимается DNS-сервером. Указанный таким DNS-сервером ресурс должен находится близко к DNS-серверу клиента, пославшего запрос. Клиент, скорее всего, находится достаточно близко к своему DNS-серверу в терминах сетевого расстояния, и, таким образом, ресурс будет передаваться на значительно меньшее расстояние. Поскольку строка запроса к серверу должна быть преобразована, то можно использовать DNS-сервер, чтобы определить подходящий сервер Akamai, имеющий запрашиваемый ресурс. Например, рассмотрим запрос на встроенный ресурс **secdef.gif** для документа-контейнера **http://www.cnn.com**. Встроенное изображение **secdef.gif** ресурса **http://www.cnn.com/secdef.gif** будет переименовано в **http://a138g.akamai.technet/cnn.com/secdef.gif**. Префикс **http://a138g.akamai.technet** относится к серверу Akamai, который обслуживает этот ресурс. Когда DNS преобразует доменное имя **a138g.akamai.technet**, то определяется некий IP-адрес (скажем **1.2.3.4**), который адресует к ближайшему к клиентскому DNS-серверу серверу Akamai. Этот сервер должен иметь изображение **secdef.gif**. Если его там не окажется, то сервер Akamai, используя внутренний протокол, запросит ресурс или с другого сервера Akamai, или с исходного сервера **http://www.cnn.com**. Далее этот ресурс будет кэширован сервером Akamai для использования при последующих запросах. Другой клиент в другой части Internet, затребовав тот же самый документ–контейнер, получит аналогично сформированный URL **http://a138g.akamai.technet/cnn.com/foo.gif**. Различие будет заключаться в том,

что будет возвращен IP-адрес **5.6.7.8** другого сервера Akamai, который находится ближе к DNS-серверу клиента.

Распределение Web-содержания по серверам Akamai должно выполняться таким образом, чтобы DNS-преобразования определяли ближайший зеркальный сайт. Алгоритм Akamai запатентован, а механизм — нет. Значение TTL DNS-сервера должно быть установлено таким образом, чтобы ответы DNS не кэшировались чересчур долго. В противном случае, Web-клиент, который обращается к **a138g.akamai.technet** может получить IP-адрес зеркального сервера Akamai, который не является наилучшим для запрошенного ресурса. Имеется компромисс между поиском лучшего выбора для каждого запроса и издержками DNS-сервера на выполнение преобразований. Хотя изменение значений TTL DNS-сервера Akamai не влияет на формирование ссылок на другие сайты, однако оно увеличивает DNS-трафик в сети и недостаточно изучено.

Существует несколько проблем при распределении Web-содержания. Исходные серверы выигрывают от сокращения нагрузки, а пользователи — от получения ресурсов с ближайших серверов. Однако местоположение серверов распределения Web-содержания может представлять проблему для ряда клиентов. Для некоторых клиентов сетевая связь с исходным сервером может оказаться лучше в терминах RTT, чем с выбранным сервером распределения Web-содержания. В главе 15 (раздел 15.4) приведены некоторые результаты последних исследований влияния сайтов распределения Web-содержания на производительность.

Технически сайты распределения Web-содержания функционируют от лица основного сервера. Клиенты устанавливают прямой контакт с сервером распределения Web-содержания, предполагая, что сервер распределения Web-содержания совместим с протоколом HTTP. Сайты распределения Web-содержания используют различные протоколы для взаимодействия с основным сервером и могут использовать другие средства для проверки актуальности доставляемого ими Web-содержания. Способ, которым они это делают, непрозрачен для внешнего мира.

Среди компаний, использующих решения для распределения Web-содержания можно назвать Adero [Ade], Akamai, Cisco [Cis], Digital Island [Dig], Exodus [Exo], Mirror Image [Mir], Netcaching [Netb], SolidSpeed [Sol], Speedera [Speb] и Unitech [Uni]. Adero является примером сети распределения Web-содержания, где доставляется все Web-содержание сайта, причем обращаться к исходному серверу за контейнерным документом нет необходимости. Существуют технические различия между подходами, используемыми различными компаниями, однако они не слишком существенны. Некоторые продукты распределения Web-содержания также могут работать с потоковым Web-содержанием.

11.14. Адаптация Web-содержания

В последнее время появилось несколько новых идей по снижению нагрузки на исходные серверы путем перемещения части его функций ближе к клиенту. *Адаптация Web-содержания* включает преобразование ресурсов в различные форматы, перевод с одного естественного языка на другой или выполнение дорогостоящих вычислений. На момент написания этой книги разработки по адаптации Web-содержания находились на концептуальной стадии, а идеи не полностью созрели. Цель распределения Web-содержания — переместить Web-содержание ближе к клиенту и подальше от исходного сервера. Адаптация Web-содержания имеет целью дальнейшее уменьшение нагрузки на исходный сервер путем перенесения самых трудоемких задач ближе к пользователю.

Проект протокола ICAP (Internet Connection Adaptation Protocol) [ICA01] предлагает отделения функций кэширующего прокси-сервера от клиента адаптации Web-содержания. Клиент адаптации Web-содержания посылает HTTP-сообщение серверу адаптации Web-содержания. Для взаимодействия между клиентом и сервером адаптации Web-содержания используется протокол HTTP/1.1. Клиент ICAP может располагаться на прокси-сервере и играть роль как прокси-сервера, так и ICAP-клиента. В некотором смысле размещение ICAP-клиента может освободить прокси-сервер от излишней нагрузки.

В качестве примера адаптации Web-содержания рассмотрим ресурс, имеющийся в специфическом виде: видеоклип с звуковым сопровождением на тамильском языке. Предположим, что несколько пользователей, расположенных за прокси-сервером, заинтересованы в получении последовательности статических изображений с звуковым сопровождением на голландском языке. Выбор языка определяется их персональными настройками; преобразование видеофрагмента в последовательность неподвижных изображений может быть связано с отсутствием мультимедийного плейера. Возможно, что исходный сервер поддерживает различные версии как звукового сопровождения, так и форматов Web-содержания. Однако преобразование видеофрагмента в последовательность статических изображений с звуковым сопровождением на голландском языке требует существенных изменений ответа. Исходный сервер может быть не заинтересован ни в поддержании избыточных форматов, ни в выполнении преобразований. Он может позволить некоторой другой доверенной стороне выполнять такие преобразования. Традиционно кандидатом на такую работу является прокси-сервер, если только он не слишком перегружен и может выделить значительную часть своих ресурсов для выполнения адаптации Web-содержания. Роль серверов адаптации Web-содержания заключается в снижении нагрузки как на исходные серверы, так и прокси-серверы при выполнении задач, требующих больших вычислительных ресурсов. Однако компонентам, выполняющим адаптацию, необходимо определить, какое именно адаптированное Web-содержание нужно клиенту, связаться с исходным сервером, чтобы убедиться в актуальности ресурса, подлежащего адаптации, доставить его прокси-серверу, через который ресурс поступает к клиенту. Адаптация Web-содержания пока что переживает период становления и еще рано обсуждать ее потенциал.

11.15. Резюме

В этой главе был представлен краткий обзор обширной темы, связанной с кэшированием. Кэширование является зрелой технологией с решениями, предлагаемыми десятками компаний. Кэш, который не обеспечивает семантическую прозрачность, рискует либо доставить пользователю устаревшие данные, либо, что еще хуже, доставить данные, про которые неизвестно, устарели они или нет. Был рассмотрен ряд вопросов, связанных с кэшированием, включая вопросы о том, что кэшировать, как кэшировать и где кэшировать. Кратко обсуждены новые методы распределения Web-содержания и его адаптации. Кэширование является развивающейся областью бизнеса, связанного с Web, и хотя кажется, что необходимо разрабатывать другие идеи, основная идея остается относительно стабильной — перемещение Web-содержания ближе к пользователю.

12

Доставка мультимедийных потоков

Web предоставляет пользователям доступ к множеству разнообразных ресурсов, вне зависимости от формата или места размещения данных. На первых порах существования Web доминирующее положение занимали Web-сайты с ресурсами в виде текста и изображений. Кэширование в Web стало средством, позволившим сократить затраты на доставку популярных ресурсов, запрашиваемых клиентами. Последнее время очень популярным стало аудио и видео, особенно для пользователей, имеющих высокоскоростные подключения к Internet. Подобно обычному тексту и изображениям, мультимедийные данные могут храниться в файлах на сервере и доставляться обратившимся с запросом клиентам. Такой подход типичен для передачи копий видеоклипов или звукового сопровождения для последующего воспроизведения. Однако многие мультимедийные приложения, такие как видео по запросу или телеконференции, дают возможность пользователю просматривать или прослушивать потоки данных по мере их поступления. Воспроизведение данных получателем во время передачи их отправителем называется *потоковой* передачей мультимедиа. Приложения для передачи и воспроизведения мультимедийных потоков предъявляют жесткие требования к производительности компьютеров и сети.

В этой главе мы сфокусируем внимание на протоколах доставки мультимедийных потоков. Сначала мы вкратце познакомимся с характеристиками аудио- и видеоданных и сделаем обзор популярных мультимедийных потоковых приложений. В продолжение темы, мы остановимся на проблемах доставки мультимедийного содержания через Internet. Хотя для передачи мультимедийного содержания может использоваться протокол HTTP, обычно передача аудио и видео лишь инициируется с помощью HTTP. Браузер запускает вспомогательное приложение (медиаплейер), которое взаимодействует с мультимедийным сервером с использованием различных протоколов, специально предназначенных для доставки потоков аудио и видео. Далее мы рассмотрим ряд Internet-протоколов, которые поддерживают большое число приложений, таких как мультимедиа по запросу, телеконференции, игры с большим числом участников и телевизионные трансляции.

В оставшейся части главы более подробно рассматриваются протоколы, поддерживающие мультимедиа по запросу. Потоковая передача по запросу для предварительно записанной аудио- и видеоинформации следует той же модели клиент–сервер, что и Web. Эти приложения обычно используют протокол Real Time Streaming Protocol (RTSP) как альтернативу HTTP. Определенный в документе RFC 2326 [SRL98], протокол RTSP координирует доставку мультимедийных потоков от мультимедийного сервера обратившемуся с запросом клиенту. Синтаксис и семантика RTSP во многом заимствованы из HTTP/1.1. Мы поговорим о сходстве между этими двумя протоколами, чтобы проиллюстрировать общий характер концеп-

ций, заложенных в HTTP/1.1. Кроме того, мы остановимся на различиях между двумя протоколами, чтобы особо выделить коммуникационные требования потоковых приложений. В ходе обсуждения RTSP также будет проиллюстрировано, как различные мультимедийные протоколы совместно работают при доставке мультимедийных потоков.

12.1. Мультимедийные потоки

Мультимедийные потоки отличаются от традиционных Web-ресурсов форматом данных и требованиями к производительности при доставке данных. В этом разделе мы вкратце познакомимся с представлением аудио- и видеосодержания. Затем мы рассмотрим различные приложения для работы с мультимедийными потоками, популярными в Internet. Затем мы определим общие требования, которые эти приложения предъявляют к доставке мультимедийных потоков. Более детальное рассмотрение мультимедийных потоков можно найти в книгах [CHW99, SR00].

12.1.1. Данные аудио и видео

В сравнении с традиционным Web-содержанием, приложения для доставки мультимедийных потоков характеризуются сложными взаимодействиями между потоками и внутри потоков. Например, *поток* видео состоит из последовательности изображений, или *кадров*, каждый из которых, в свою очередь, состоит из набора *пикселов*, как описано в таблице 12.1. Каждый пиксел, или элемент изображения, соответствует небольшой прямоугольной области изображения. Размер изображения выражается в числе пикселов по каждому измерению. Изображение 640×480 имеет ширину в 640 пикселов и высоту в 480 пикселов. Цвет или яркость каждого пиксела представляется числом. Например, реалистичное цветовоспроизведение использует минимум 24 бита для каждого пиксела. Интенсивность каждого из основных цветов: красного, синего и зеленого, которая воспринимаются человеческим глазом, представляется восемью битами. Представление каждого пиксела в изображении требует большого объема данных — для изображения 640×480 с 24 битами на пиксел потребуется более 7 мегабайтов (т.е. 640×480×24). К счастью, возможно сжатие изображений путем устранения избыточности, например, больших областей, закрашенных одним цветом. Форматы изображений, использующие сжатие, такие как GIF или JPEG, весьма распространены в Web.

Таблица 12.1. Иерархия мультимедийного содержания

Термин	Описание
Пиксел	Элемент изображения
Кадр	Двухмерный набор пикселов
Поток	Последовательность кадров, передаваемых за определенное время
Сеанс	Синхронизированный набор потоков
Презентация	Набор мультимедийных сеансов

Каждый кадр в видеопотоке соответствует неподвижному изображению, «выхваченному» в определенный момент времени. Аудиопоток состоит из последовательности звуков (сэмплов). Мультимедийный поток перед отображением у получателя проходит через следующие стадии:

- **Ввод и преобразование в цифровую форму.** Поток аудио или видео должен быть введен с аналогового устройства, такого как микрофон или видеокамера, и преобразован в цифровую форму.
- **Кодирование.** Кодировщик преобразует исходные цифровые данные в определенный аудио- или видеоформат. Кадры могут кодироваться по мере ввода, не ожидая, пока будет введен весь поток.
- **Хранение.** Сервер может сохранять закодированный поток для дальнейшей передачи.
- **Доставка.** Поток передается одному или нескольким получателям. «Живой» поток может передаваться по мере ввода и кодирования, тогда как сервером передается записанный ранее поток.
- **Декодирование.** Получатель декодирует и отображает данные по мере их получения. Возможен и альтернативный вариант — получатель сохраняет весь поток до начала его воспроизведения.

Получатель воспроизводит сэмплы или кадры способом, не допускающим образования паузы между ними. Чтобы отобразить поток видео с частотой 30 кадров в секунду, медиаплейер должен показывать новый кадр каждую 1/30 секунды. Видеопотоки обычно состоят из большого объема данных. Для отображения тридцати изображений 640×480 в секунду потребуется скорость передачи 210 Мбит/с. Подобно изображениям и текстовым данным, мультимедийные потоки допускают сжатие. Фактически аудио- и видеоданные дают дополнительную возможность для сжатия — использование избыточной повторяемости в последовательности сэмплов или кадров. Некоторые схемы видеокомпрессии генерируют кадры небольшого объема, которые представляют собой различия между последовательными изображениями. Схемы видеокомпрессии также могут использовать регулярность видеообъектов от одного кадра к другому. Эффективные технологии сжатия могут уменьшить размер видеопотока в 25 или даже в 100 раз. К используемым в Web видеоформатам относятся RealVideo, AVI (Audio Video Interleave), QuickTime и MPEG (Moving Pictures Expert Group) [AS98]. Широко используются аудиоформаты RealAudio, AU (Audio), WAV (Waveform Audio) и MP3 (MPEG Audio Layer 3).

Кодирование потоков аудио и видео характеризуется принципиальным компромиссом между объемом данных и качеством их воспроизведения у пользователя. Подобного компромисса нет при передаче традиционного текстового содержания. Видеопотоки, доступные в Web, обычно характеризуются потерей качества во многих отношениях, а именно:

- **Низкая частота кадров.** Видеопоток с низкой частотой кадров воспринимается как прерывистая последовательность статических изображений. Частота кадров меньше 24 или 30 кадров в секунду плохо воспринимается глазами. Тем не менее, многие мультимедийные потоковые приложения используют более низкую частоту кадров в пределах от 10 до 15 кадров в секунду.
- **Малые размеры кадра.** Большинство видеопотоков, доступных в Web, имеют небольшие размеры кадра, обычно несколько дюймов в высоту и в ширину.

- **Низкое разрешение.** Отдельные кадры могут иметь низкое качество из-за излишней зернистости изображения или вследствие избыточной компрессии данных.

Потеря качества ведет к уменьшению объема данных в видеопотоке. Схожие методы сжатия могут быть применены и к аудиоданным. На практике, однако, люди более восприимчивы к потере качества аудиоинформации. Аудиопотоки требуют гораздо меньшей скорости передачи данных, чем видеопотоки, что делает не столь важной проблему снижения качества при уменьшении объема передаваемых данных.

Многие мультимедийные приложения работают не с одним потоком, а с несколькими. Например, один мультимедийный *сеанс* может состоять из потока аудио и потока видео. Хотя два потока могут использовать различные методы кодирования и сжатия, они связаны между собой во времени. Воспроизведение потоков аудио и видео должно быть скоординированным, чтобы сохранить временные свойства, присущие источнику. Медиаплейер должен воспроизводить каждый поток с соответствующей скоростью, а звук и изображения должны быть синхронизированы друг с другом. Наконец, мультимедийная *презентация* может состоять из множества сеансов во времени, использующих различные участки экрана. Допустим, университетский профессор читает лекцию удаленной аудитории через Internet. Презентация может включать видеоизображение и речь лектора, изображение текущего поясняемого материала и видеоизображение аудитории, находящейся в другом месте.

12.1.2. Приложения для мультимедийных потоков

Имеется множество мультимедийных приложений для воспроизведения аудио и видео по мере их поступления. Хотя многие из этих приложений существуют отдельно от Web, Web-браузеры предоставляют интерфейс для вызова этих приложений. Желание поддерживать мультимедийные приложения оказало прямое влияние на разработку протоколов для доставки мультимедийных потоков. К популярным классам мультимедийных потоков относятся следующие:

- **Мультимедиа по запросу.** В приложениях мультимедиа по запросу клиент выдает запрос серверу на поток мультимедийного содержания для немедленного воспроизведения. Например, клиент может запросить видеоклип, новости или музыкальную композицию, воспроизвести данные по мере их поступления. Приложения мультимедиа по запросу следуют той же модели клиент–сервер, которая обычно используется в Web. Пользователь может инициировать запрос на мультимедийный сеанс, щелкнув на гипертекстовой ссылке на Web-странице.
- **Internet-телефония.** В технологиях IP-телефонии (IP telephony) и Голос через IP (Voice over IP) пользователи общаются друг с другом в реальном времени. Главным побудительным мотивом применения IP-телефонии является низкая стоимость передачи аудиоданных через Internet и возможность интегрировать телефонию с другими IP-приложениями. Как и в традиционной телефонной сети, одна сторона инициирует вызов другой стороны. После установления соединения аудиоданные передаются в обоих направлениях. Каждый участник использует приложение, которое кодирует и декодирует аудиоданные и взаимодействует с сетью. Пользователи могут говорить и слушать друг друга с помощью своих компьютеров или с помощью обычных телефонных аппаратов.

- **Телеконференции.** Близко связанные с IP-телефонией приложения для телеконференций предназначены для организации общения группы участников между собой. В простейшем случае группа пользователей обменивается аудио- и, возможно, видеоинформацией в реальном времени. Приложение может также поддерживать общее рабочее пространство, например, сетевую доску объявлений, а также функции управления аудиторией, например, предоставление слова участнику конференции. Приложения для IP-телеконференций, такие как CU-SeeMe [Dor95] и средства vic/vat, используемые в MBone [Eri94], предшествовали появлению Web.
- **Радио/телевидение.** Приложения для радио и телевидения дают возможность пользователю настраиваться на определенную станцию или программу. Множество пользователей по всему миру могут одновременно смотреть программу. Подобно тому, как IP-телефония конкурирует с традиционной междугородней телефонной связью, приложения для радио и телевидения в Internet предоставляют альтернативу традиционным сетям радио и ТВ. Internet дает возможность поставщикам содержания расширить свою аудиторию при относительно низких затратах и без ограничений, например, на частотные диапазоны.
- **Многопользовательские игры.** Игровые приложения дают возможность группе пользователей, находящихся в различных местах, участвовать в игре. Игровые приложения обычно дают возможность пользователям включаться в игру и выходить из нее, а также настраивать свое положение в игре. Например, пользователь может перемещаться по игровому полю и взаимодействовать с другими игроками на определенном расстоянии от текущего местонахождения.
- **Виртуальная реальность.** Подобно игровым приложениям, виртуальная реальность дает возможность пользователю перемещаться в виртуальном окружении. Например, пользователь может совершить виртуальную прогулку по городу, встречая на своем пути трехмерные изображения различных зданий. Действия пользователя, такие как поворот налево или движение вперед, приводят к изменению того, что он видит. Некоторые приложения виртуальной реальности позволяют нескольким пользователям участвовать в виртуальной прогулке.

Протоколы доставки мультимедийных потоков пытаются учесть общие требования этих приложений, а также поддерживают сервисы, которые объединяют свойственные приложениям различных классов возможности. Например, пользователь может инициировать воспроизведение видеопотока в видеоконференции.

12.1.3. Свойства мультимедийных приложений

Мультимедийные приложения обладают определенными характеристиками, которые имеют важное значение для передачи мультимедийных данных через Internet.

Начальная задержка. Как только получатель начинает воспроизведение мультимедийного потока, последовательность кадров должна поступать к нему своевременно. Однако передача потока через сеть связана с задержкой, которая может варьироваться от одного пакета или кадра к другому. Получатель обычно вводит задержку перед воспроизведением первого кадра, чтобы избежать прерывистого воспроизведения потока, если последующие кадры прибудут с большой задержкой. Плейер может выделить буфер для хранения поступивших данных. Мультимедийные приложения сильно различаются по величине начальной задержки. Для некоторых приложений медиаплейер может ожидать поступления нескольких кадров,

прежде чем начать воспроизведение. Пользователь может не возражать относительно задержки в несколько секунд перед просмотром ранее записанного видеопотока или прямого спортивного репортажа. В некоторых случаях пользователь может согласиться подождать несколько часов перед просмотром видеоклипа, чтобы дать возможность серверу передать данные с меньшей скоростью. Другие приложения, такие как телефония и распределенные игры, не могут себе позволить длительных задержек. Для таких интерактивных приложений задержка, превышающая несколько сотен миллисекунд, уже становится довольно ощутимой. Терпимость к величине задержки зависит и от пользователей. Пользователи, активно участвующие в телеконференции, не могут примириться с длительными задержками, тогда как пользователи, являющиеся лишь слушателями телеконференции, могут пойти на увеличение задержки до нескольких секунд в обмен на более высокое качество или более низкую цену.

Создание содержания. Потоки в реальном времени кодируются и передаются по мере их создания. В противоположность этому предварительно записанный поток вводится и кодируется до начала передачи. Для предварительно записанных потоков возможно более сильное сжатие, поскольку данные являются доступными перед их передачей. Предварительно записанные потоки также предоставляют серверу большую гибкость при передаче данных клиенту. Сервер может поддерживать функции видеомагнитофона, включая паузу, перемотку и ускоренное воспроизведение, которые дают возможность пользователю просматривать различные фрагменты предварительно записанных данных. Некоторые приложения комбинируют содержание в реальном времени и предварительно записанное содержание. Телевизионная станция, ведущая спортивный репортаж, может также осуществить запись потока для повторного его воспроизведения в дальнейшем. В процессе просмотра репортажа в реальном времени пользователь может решить записать оставшуюся часть потока. Кроме того, предварительно записанное содержание, например телевизионная реклама, может быть вставлено в «живой» поток.

Число потребителей. Мультимедийные приложения различаются по числу потребителей. В простейшем случае пользователь посещает Web-сайт и запрашивает предварительно записанный мультимедийный сеанс. Мультимедийный сервер передает запрошенные аудио- и видеопотоки пользователю. Другие приложения предусматривают двунаправленное взаимодействие. Например, большинство приложений телефонии предполагает наличие двух участников, хотя на каждой стороне имеется только один потребитель. В то же время трансляция прямого спортивного репортажа может предназначаться сотням или даже тысячам потребителей. Передача мультимедийного потока множеству потребителей может вызвать существенное повышение нагрузки на отправителя и на сеть. *Групповое вещание* (multicasting) позволяет отправителю передавать одну копию потока множеству потребителей. Потребители принимают данные, подписываясь на услугу группового вещания, идентифицируемую по IP-адресу; сеть обеспечивает получение копии данных потребителем. Для услуг группового вещания выделяется часть пространства IP-адресов, о чем говорилось в главе 5 (раздел 5.1.3).

Инициирование сеанса. Мультимедийные приложения различаются и по способу инициирования получения потока пользователем. В приложениях мультимедиа по запросу пользователь может запросить сеанс, щелкнув на гипертекстовой ссылке; сервер удовлетворяет запрос, инициируя передачу запрошенного потока (потоков). В приложениях IP-телефонии пользователь может получить сообщение, которое сигнализирует об установке соединения между двумя компьютерами.

В определенный момент перед получением сигнального сообщения пользователь должен выразить желание получать такие сообщения. Сторона, инициировавшая вызов, должна иметь возможность обнаружения предполагаемого получателя и обмена сигнальными сообщениями. В противоположность этому приложение для телевизионного вещания может объявить о наличии группы вещания, ассоциированной с сеансом. Тем самым пользователям предоставляется необходимая информация, позволяющая им присоединиться к группе вещания для приема передачи в соответствующее время.

12.2. Доставка мультимедийного содержания

В противоположность традиционным данным в виде текста и изображений, мультимедийные потоки предъявляют строгие временные требования по доставке информации получателю. В этом разделе рассматриваются требования к производительности приложений для доставки мультимедийных потоков. Затем мы обсудим, почему эти требования трудно удовлетворить в традиционных IP-сетях. Наконец, мы рассмотрим ограничения в применении HTTP в качестве протокола прикладного уровня для потоковых приложений.

12.2.1. Требования к производительности

Доставка мультимедийных потоков предъявляет жесткие требования к производительности, отличные от тех, которые имеют место для текста и изображений.

Задержки. Пользователи Web испытывают задержку при загрузке текста и изображений. В некоторых случаях Web-браузер воспроизводит текст и изображения по мере их поступления. Несмотря на некоторые неудобства для пользователей, задержка не оказывает влияния на качество содержания после его полного получения. Задержки порядка нескольких десятков или сотен миллисекунд не имеют существенного значения для пользователя. Для мультимедийных потоков ситуация иная. После начала воспроизведения аудио- или видеопотока последовательные звуковые сэмплы или кадры должны поступать своевременно. В противном случае медиаплейер должен компенсировать отсутствующие данные. Например, плейер может сделать паузу в воспроизведении (ожидая поступления следующего кадра) или повторить предыдущий кадр. В обоих случаях возникает прерывистое изображение или звук. Задержка в поступлении аудиопотоков приводит к еще более заметному снижению производительности. Медиаплейер обычно буферизует группы кадров перед началом воспроизведения потока, чтобы компенсировать длительные задержки при получении последующих кадров.

Потери. Сообщения-запросы и сообщения-ответы HTTP передаются целиком, если только передача не прерывается. HTTP предполагает, что сообщения доставляются надежным транспортным протоколом, например, TCP. Надежная доставка не является главной целью для многих мультимедийных приложений, поскольку повторная передача утерянного пакета может вызвать слишком большую задержку в получении данных. Повторно переданный пакет вряд ли окажется полезным, если он поступит после того, как медиаплейер отобразит соответствующий кадр. Кроме того, медиаплейер не сможет получить последующие данные из буфера сокета до тех пор, пока потерянный пакет не будет успешно передан. Протокол User Datagram Protocol (UDP) часто используется в качестве альтернативы TCP для

транспортировки мультимедийных потоков. Многие мультимедийные приложения могут примириться с небольшими потерями данных. Например, видеоплейер может восстановить отсутствующую часть кадра на основе ближайших пикселов в этом кадре, либо наборов пикселов в предыдущем и последующем кадрах. Однако поскольку потери или задержки пакетов снижают качество потока, некоторые приложения все же имеют ограниченную поддержку для восстановления утерянных данных. Это обычно достигается посредством выборочной повторной передачи потерянных пакетов отправителем или передачи некоторых избыточных данных, чтобы помочь получателю восстановить потерянную информацию.

Скорость передачи. Большинство Web-ответов являются небольшими и имеют средний размер от 8 до 12 килобайтов. В то же время потоки аудио и видео обычно имеют достаточно большой размер. Передача потоков аудио и видео требует высокой пропускной способности. Например, несжатое речевое сообщение телефонного качества требует скорости передачи данных 64 Кбит/с; эффективная техника сжатия может уменьшить требования к пропускной способности до 10 Кбит/с. Требования к пропускной способности для видеопотоков сильно варьируются в зависимости от качества, частоты кадров, размера изображения. Высококачественное сжатое видео требует скорости передачи 4–8 Мбит/с, что соответствует передаче около 200 мегабайтов данных в течение пяти минут. Хотя для традиционного Web-содержания допустимы короткие периоды снижения скорости передачи, строгие требования к задержке для мультимедийных потоков не позволят допускать вариаций в скорости передачи. Для предварительно записанных потоков сервер может отправлять дополнительные данные в периоды с высокой скоростью передачи данных в сети, чтобы приспособиться к периодам с более низкой скоростью передачи в дальнейшем. Для потоков в реальном времени отправитель имеет ограниченные возможности в использовании периодов с высокой пропускной способностью.

12.2.2. Ограничения, свойственные IP-сетям

Разработчики Internet ничего не знали о мультимедийных приложениях, так как их в то время не было. Совместное использование IP и TCP обеспечивает оптимальный и надежный транспорт, который хорошо подходит для большинства Internet-приложений. Ниже перечислены характеристики, которые наиболее критичны для мультимедийных потоков.

Качество сервиса. IP предоставляет оптимальный сервис без установления логических соединений, который допускает потерю, задержку пакетов или доставку их не в том порядке, как это рассказывалось в главе 5 (раздел 5.1.2). Мультимедийные приложения могут мириться с вариациями в задержке, потерями данных и с изменениями пропускной способности сети. Однако значительные колебания производительности сети приводят к заметному снижению качества воспроизведения на стороне получателя. Чтобы обеспечить более высокую и более предсказуемую производительность, необходимо изменить инфраструктуру сети или развернуть дополнительные мощности. Поставщики маршрутизаторов и сетевые провайдеры начали поддерживать дифференциацию трафика в Internet. Это способствует уменьшению задержек, повышению скорости передачи данных и снижению потерь в сетях.

Управление пропускной способностью. Рост популярности мультимедийных приложений поднял проблему пересмотра требований, которым должна удовлетворять инфраструктура Internet. Мультимедийные потоки требуют постоянной скорости передачи на определенный период времени. Кроме того, один поток может быть

передан нескольким получателям с помощью группового вещания или путем нескольких индивидуальных передач. Это еще больше увеличивает нагрузку на сеть. Мультимедийные приложения обычно используют в качестве базового транспортного протокола UDP. UDP-отправитель не обязательно реагирует на затор в сети снижением скорости передачи. В результате UDP-сообщения могут перегрузить сеть и вызвать снижение пропускной способности TCP-соединений. Это уменьшает производительность приложений, использующих TCP, таких как передача Web-сообщений с помощью HTTP. В последние годы были предприняты значительные усилия по разработке *дружественных для TCP* методов управления пропускной способностью [FF99] для UDP-трафика, в том числе и для группового вещания. Цель дружественного для TCP управления пропускной способностью состоит в том, чтобы обеспечить «справедливое» ее распределение между мультимедийными потоками и другим Internet-трафиком, не заставляя мультимедийные приложения использовать TCP в качестве транспортного протокола.

Групповое вещание. В противоположность традиционным Internet-приложениям, многие мультимедийные приложения передают одни и те же данные множеству получателей. Поддержка группового вещания IP-маршрутизаторами предоставляет возможность значительного сокращения нагрузки на отправителя и сеть. Сетевая поддержка группового вещания в течение многих лет является областью активных исследований. Хотя многие маршрутизаторы имеют встроенную поддержку группового вещания, оно слабо поддерживается операционными системами. Групповое вещание на уровне приложений является популярной альтернативой, обеспечивающей эффективное распространение информации среди множества получателей, не предъявляя при этом к маршрутизаторам требований поддержки группового вещания. При реализации группового вещания на уровне приложений отправитель осуществляет индивидуальную передачу на небольшую группу серверов-посредников. Каждый сервер-посредник, в свою очередь, может отправлять данные другим серверам-посредникам. Серверы-посредники эффективно формируют *наложенную* сеть, построенную поверх IP-сети. Одним из первых воплощений наложенной сети стала сеть Multicast Backbone (MBone), используемая для видеоконференций. В последующие несколько лет групповое вещание на уровне приложений стало стандартным решением для распространения мультимедийного содержания группе получателей. Помимо ретрансляции данных, серверы-посредники могут выполнять множество других функций, таких как кэширование или адаптация содержания.

12.2.3. Мультимедиа по запросу по HTTP

Первоначально мультимедийные потоки в Web передавались точно так же, как традиционные текст и изображения. Клиент отправляет HTTP-запрос на Web-сервер, а сервер передает запрошенный ресурс клиенту. Заголовок **Content-Type** в сообщении-ответе HTTP указывает на формат кодирования (например, **Content-Type: video-mpeg**). На основе значения заголовка **Content-Type** браузер вызывает соответствующее вспомогательное приложение, чтобы интерпретировать ответ, как об этом говорилось в главе 2 (раздел 2.4.3). Для данных аудио и видео вспомогательным приложением обычно является медиаплейер, который декодирует и отображает ответ. Плейер может также предоставить графический интерфейс для регулирования громкости и выполнения функций видеомагнитофона, таких как пауза, прямая и обратная перемотка. С точки зрения браузера медиаплейер не

отличается от любого другого вспомогательного приложения — браузер просто направляет данные приложению.

Обработка мультимедийного содержания точно таким же образом, как текстовых данных и изображений, имеет несколько преимуществ. Все данные могут храниться на Web-сервере и быть доступными с использованием одного протокола — HTTP. Кроме того, мультимедийное содержание может быть кэшировано на прокси-серверах, расположенных на пути между сервером и клиентом. Однако обслуживание мультимедийных потоков как традиционных Web-данных имеет следующие недостатки:

- **Начальная задержка.** В зависимости от реализации браузер может не активизировать медиаплейер, пока весь HTTP-ответ не будет получен сервером. Это вносит значительную начальную задержку для больших аудио/видео клипов. Чтобы избежать этого, браузер должен направить данные на медиаплейер сразу при поступлении первых пакетов данных.
- **Копирование данных между браузером и плейером.** Мультимедийные данные должны быть переданы от браузера плейеру по мере их поступления от сервера. В зависимости от конкретной реализации это может потребовать копирования большого объема данных между двумя процессами на клиентском компьютере. Плейер должен получать данные своевременно, чтобы обеспечить плавность воспроизведения.
- **Чередование потоков аудио и видео.** Вспомогательное приложение работает с содержимым одного сообщения-ответа. Рассмотрим HTTP-запрос на мультимедийный сеанс, который содержит как аудио-, так и видеоданные. Web-сервер должен чередовать аудио- и видеосодержание в одном сообщении-ответе, чтобы дать возможность плейеру воспроизводить данные по мере их поступления. В некоторых случаях аудио- и видеоданные уже объединены вместе в одном файле.
- **Издержки надежного транспорта.** HTTP подразумевает наличие надежного протокола транспортного уровня, такого как TCP. Передача данных потокового мультимедиа с помощью TCP сама по себе неэффективна. Обнаружение и повторная передача утерянных пакетов приводит к дополнительной задержке при отображении мультимедийного содержания. Это может вызвать прерывистое воспроизведение потока медиаплейером.
- **Сложности с функциями видеомагнитофона.** Трактовка медиаплейера как вспомогательного приложения усложняет реализацию функций видеомагнитофона. Плейер может обеспечить ограниченную паузу, перемотку назад и ускоренную перемотку вперед путем обращения к данным, полученным с сервера. Воспроизведение с произвольного момента времени в аудио- или видеопотоке — еще более сложная задача. Это потребует от плейере выдачи нового HTTP-запроса для соответствующего диапазона данных.

HTTP не слишком хорошо подходит для передачи потоков мультимедийного содержания; для преодоления указанных выше ограничений были разработаны другие подходы.

Вместо того чтобы получать данные от браузера, медиаплейер может взаимодействовать непосредственно с сервером. Это взаимодействие не обязательно должно осуществляться по HTTP. Плейер может взаимодействовать с мультимедийным сервером с использованием протокола, который лучше подходит для работы с мультимедийными потоками. Параллельно с разработкой стандартных протоколов, поставщики программных продуктов реализовывали свои собственные подходы к коорди-

нации действий между плейером и сервером. Как правило, для инициирования мультимедийной передачи используется HTTP. В ответ на HTTP-запрос сервер возвращает ответ, содержащий метаданные о мультимедийном потоке. Например, тело HTTP-ответа может включать URL мультимедийного сервера с указанием желаемого содержания (например, **pnm://media.foo.com/clip**). Браузер активизирует вспомогательное приложение, которое обрабатывает ответ, устанавливая свое собственное соединение с мультимедийным сервером.

Медиаплейер должен понимать формат метаданных в теле HTTP-ответа. Кроме того, плейер должен знать, как связаться с мультимедийным сервером. Первые компании, занявшиеся доставкой мультимедийных потоков, например, RealNetworks, разрабатывали и предлагали мультимедийные серверы, а также бесплатно предоставляли заинтересованным пользователям медиаплейеры. Управление как сервером, так и плейером предполагало использование специфичных для производителя протоколов для коммуникационного взаимодействия плейера и сервера, а также для представления метаданных в теле HTTP-ответа. Разработка и распространение стандартных протоколов являются необходимыми этапами в разделении функций, относящихся к метаданным, а также в разграничении обязанностей между плейером и сервером. Различные варианты стандартных протоколов реализованы в некоторых медиаплейерах и серверах.

12.3. Протоколы для передачи мультимедийных потоков

К передаче мультимедийных потоков предъявляются особые требования, которые не могут быть удовлетворены HTTP. В этом разделе мы обсудим протоколы, которые способны удовлетворить требованиям, относящимся к следующим категориям.

- **Передача данных.** Потоковое мультимедиа требует доставки данных с теми же временными характеристиками одному или нескольким получателям. Транспортировка потока обслуживается двумя тесно взаимосвязанными протоколами. Протокол Real-time Transport Protocol (RTP) делит поток на пакеты, каждый из которых содержит отметку времени, порядковый номер и информацию об отправителе [SCFJ96]. Спецификация RTP также определяет протокол RTP Control Protocol (RTCP), по которому осуществляется обратная связь с информацией о качестве передачи и идентификационных данных участников потока.
- **Создание сеанса.** Потоковое мультимедиа требует, чтобы отправители и получатели могли выражать свою заинтересованность в участии в мультимедийном сеансе. Соответствующий протокол зависит от конкретного применения. Протокол Session Initiation Protocol (SIP) используется для приглашения пользователя присоединиться к сеансу [SSR99]. Протокол Session Announcement Protocol (SAP) используется для объявления группы вещания, к которой клиенты могут присоединиться [HPW00]. Протокол Real Time Streaming Protocol (RTSP) позволяет клиенту отправлять сообщение-запрос мультимедийному серверу [SRL98].
- **Описание сеанса.** Участникам сеанса нужно получить метаданные, такие как параметры передачи и кодирования составляющих потоков, продолжитель-

ность, название и назначение сеанса. Эта информация передается с помощью протокола Session Description Protocol (SDP) [HJ98].

- **Описание представления.** Создание мультимедийных документов требует эффективного способа для организации воспроизведения сеансов во времени. Являясь альтернативой HTML, язык Synchronized Multimedia Integration Language (SMIL) предоставляет возможность управлять, когда, где и как будут воспроизводиться мультимедийные сеансы [Syn98, Hos00, Smi].

RTP, RTCP, SIP, SAP, RTSP и SDP специфицированы в документах запросов на комментарии (Request for Comments), разработанных Internet Engineering Task Force (IETF). За спецификацию SMIL отвечает World Wide Web Consortium (W3C).

12.3.1. Передача данных

RTP и RTCP координируют доставку мультимедийного потока одному или нескольким получателям. RTP [SCFJ96, Tho96] удовлетворяет основным требованиям по транспортировке мультимедийных потоков для разнообразных приложений, таких как мультимедиа по запросу, IP-телефония и телеконференции. Данные в RTP-пакете ассоциируются с определенным звуковым сэмплом или видеокадром в потоке; один сэмпл или кадр может размещаться в нескольких RTP-пакетах. Заголовок RTP помогает получателю интерпретировать данные.

Таймирование. Получатель должен иметь информацию для координации воспроизведения потока. Промежутки между кадрами могут быть определены из RTP-заголовков без необходимости синхронизации часов отправителя и получателя. RTP-заголовок содержит 32-битное поле временной метки относительно момента времени, когда был воспроизведен первый сэмпл. Все пакеты звукового сэмпла или видеокадра имеют одно и то же значение временной метки. Первый RTP-пакет в потоке имеет случайную отметку времени, подобно случайному начальному порядковому номеру первого пакета в TCP. Временные метки способствуют корректному воспроизведению последовательности сэмплов или кадров в потоке. Это дает возможность получателю воспроизводить данные с соответствующими интервалами времени между последовательными сэмплами или кадрами.

Упорядочение. В RTP отсутствует механизм надежной доставки и управления информационным потоком. Доставка RTP-пакетов зависит от транспортного протокола. RTP не предполагает, что транспортный протокол обеспечивает упорядоченный, надежный байтовый поток, поскольку повторная передача потерянных пакетов для многих мультимедийных потоков нежелательна. В действительности RTP обычно работает поверх UDP, а каждый RTP-пакет передается в одном UDP-пакете. Альтернативой является использование TCP для обеспечения надежного транспорта либо передачи потока через межсетевой экран, который отвергает UDP-пакеты. Однако IP-пакеты могут быть потеряны или поступать не в том порядке, а UDP не осуществляет какой-либо повторной передачи или упорядочения пакетов. RTP-заголовок содержит порядковый номер, дающий возможность получателю обрабатывать поступающие пакеты в нужном порядке и обнаруживать потери пакетов. Получатель может вести статистику потерь пакетов для обратной связи с отправителем. Данные такой статистики могут заставить отправителя скорректировать скорость передачи данных.

Идентификация источника. Помимо информации для таймирования и упорядочения, RTP-заголовок идентифицирует источник (источники), ответственные за

данные в пакете. Это особенно полезно для приложений, которые могут иметь несколько источников данных (например, телеконференции). Каждый источник идентифицируется 32-битным числом. Каждый RTP-пакет имеет *синхронизирующий источник*, ответственный за временную метку и порядковый номер данных. В некоторых ситуациях синхронизирующий источник генерирует данные с помощью одного или нескольких *дополнительных источников*. Рассмотрим приложение для организации телеконференций с несколькими участниками. Каждый участник генерирует аудиопоток и направляет RTP-пакеты промежуточной системе — *микшеру*. Микшер объединяет пакеты участников и генерирует один аудиопоток. Микшер является синхронизирующим источником для нового потока. Изначальные участники являются *участвующими* источниками. При объединении отдельных аудиопотоков микшер формирует список участвующих источников для синхронизации составляющих потоков. Включая идентификаторы участвующих источников в RTP-заголовок, микшер информирует получателей обо всех сторонах, ответственных за объединенный поток.

Формат представления. Интерпретация данных RTP на стороне получателя зависит от выбранного формата кодирования. RTP-заголовок содержит 7-битное поле типа полезной нагрузки, идентифицирующее формат мультимедийных данных. Тип полезной нагрузки ассоциируется с именем и описанием кодировки, включая частоту временных меток. Формат полезной нагрузки определяет синтаксис и семантику данных RTP, в том числе специфичных для содержимого заголовков. Первый набор типов полезной нагрузки был определен в документе RFC 1890 [Sch96]. Новые типы кодировок регистрируются организацией Internet Assigned Numbers Authority (IANA). Подробные описания типов полезной нагрузки, содержащиеся в соответствующих документах, таких как RFC, позволяют разработчикам осуществлять поддержку схем кодирования. Типы полезной нагрузки были также определены для общеупотребительных мультимедийных сервисов, включая RTP-микшеры и избыточное кодирование аудиоинформации, устойчивое к потерям пакетов. Для достижения гибкости и расширяемости RTP также поддерживает использование динамических типов полезной нагрузки, которые не были зарегистрированы IANA.

Хотя RTP предоставляет большую часть информации, необходимой для передачи мультимедийных данных от отправителя одному или нескольким получателям, иногда требуется некоторая дополнительная управляющая информация. Определенный в том же RFC, что и RTP [SCFJ96], RTCP выполняет следующие три функции:

- **Обратная связь.** RTCP предоставляет обратную связь о качестве приема, например, статистику потерь RTP-пакетов получателем. Это помогает RTP-отправителю выявлять проблемы с производительностью и соответствующим образом адаптировать скорость передачи данных.
- **Идентификация.** RTCP связывает каждый идентификатор источника из RTP-пакетов с уникальным каноническим именем. Это дает возможность получателю ассоциировать набор потоков, возможно, имеющих различные 32-битные идентификаторы источников, с одним отправителем.
- **Синхронизация.** RTCP связывает реальное время с временными метками в RTP-пакетах. Это позволяет получателю синхронизировать воспроизведение нескольких потоков, включая аудио и видео, принадлежащих одному сеансу.

12.3.2. Создание сеанса

Отправители и получатели должны иметь способ выразить свою заинтересованность в участии в определенном мультимедийном сеансе. Детали установления соединения между двумя или более сторонами варьируются в зависимости от типа приложения, как говорилось ранее в разделе 12.1.3. Это получило воплощение в нескольких различных протоколах:

- **Session Initiation Protocol (SIP).** SIP [SSR99, SR99] поддерживает приложения, такие как IP-телефония, в которых принимающая сторона получает персональное приглашение для участия в сеансе. Сторона, инициирующая сеанс, может использовать SIP для нахождения одного или более пользователей и пригласить их принять участие в сеансе. SIP выполняет пять основных функций: (1) решает, с каким компьютером установить контакт, (2) определяет, какие параметры использовать, (3) определяет, желает ли пользователь принять вызов, (4) устанавливает соединение с пользователем, (5) обслуживает передачу и завершает вызов.
- **Session Announcement Protocol (SAP).** SAP [HPW00] поддерживает такие приложения, как Internet-радио, в которых предполагаемые участники заранее не известны. Вместо того чтобы организовывать обращение к получателям на стороне отправителя, заинтересованные пользователи посылают запрос на вступление в группу вещания, связанную с сеансом. Однако пользователь может не знать, какая группа вещания соответствует данному мультимедийному сеансу. SAP периодически отправляет анонсирующий пакет, содержащий описание сеанса, на известный адрес группового вещания и номер порта (9875). Это можно сравнить с телевизионной станцией, которая сообщает о программе передач средствам массовой информации. SAP-слушатель получает анонс, присоединяясь к известной группе вещания. SAP-слушатели могут сами распространять информацию о сеансе среди других пользователей.
- **Real Time Streaming Protocol (RTSP).** RTSP дает возможность пользователям извлекать мультимедийное содержание по запросу, выдавая запрос мультимедийному серверу, подобно отправке HTTP-запроса на Web-сервер. Мультимедийные сеансы и составляющие их потоки идентифицируются URL. Клиент выдает RTSP-запросы для получения информации о сервере и мультимедийном сеансе, а также для воспроизведения, паузы, записи или просмотра потоков. RTSP предоставляет абстракцию «дистанционного сетевого управления» мультимедийным сервером. RTSP обычно не доставляет данные. Он используется для выбора транспортного механизма (например, RTP и RTCP), транспортного протокола (например, индивидуальное вещание по UDP, групповое вещание по UDP или TCP) и определенных номеров портов (например, портов 6970 и 6971). Спецификация RTSP во многом основана на синтаксисе и семантике HTTP/1.1, о чем подробнее рассказывается в разделе 12.4.

12.3.3. Описание сеанса

Вне зависимости от того, как был создан мультимедийный сеанс, участникам нужно получить описание сеанса. Например, участникам нужно знать имя и назначение сеанса, параметры представления составляющих его потоков аудио и видео, IP-адреса и номера портов. Эта информация передается в стандартном формате, определяемом SDP (Session Description Protocol) [HJ98]. В действительности SDP — это не протокол, а язык или формат доставки информации о сеансе. Описа-

ния сеансов передаются другими протоколами, например, SIP, SAP, RTSP и HTTP. Например, HTML-файл может иметь гипертекстовую ссылку, которая указывает на SDP-файл. С точки зрения пользователя щелчок мышью на гипертекстовой ссылке инициирует передачу данных аудио или видео. В реальности же Web-браузер посылает HTTP-запрос Web-серверу для извлечения описания сеанса, а затем активизирует медиаплейер для обработки сообщения HTTP-ответа (например, SDP-файла). После этого медиаплейер осуществляет синтаксический разбор SDP-файла и связывается с одним или с несколькими мультимедийными серверами для извлечения аудио- и видеопотоков.

Описание SDP состоит из одной или нескольких строк текста в формате ASCII. Каждая строка включает состоящий из одного символа *тип* и *значение* в виде текстовой строки, разделенные знаком равенства. Формат строки значения зависит от типа. Описание содержит одну или несколько строк с информацией сеансового уровня, за которой может следовать дополнительная информация об отдельных потоках в сеансе. Информация сеансового уровня применяется ко всему сеансу и к каждому из мультимедийных потоков. Информация сеансового уровня включает имя и назначение сеанса в виде текстовых строк, например

```
s=Web Seminar
i=What everyone needs to know about the World Wide Web
```

Раздел сеансового уровня может также содержать имя, адрес электронной почты и номер телефона лица, ответственного за сеанс, а также URL с дополнительной информацией. Кроме того, раздел сеансового уровня может содержать информацию о ключах шифрования, необходимых для участия в мультимедийном сеансе или для получения индивидуальных потоков. Описание также включает время начала и время окончания сеанса в формате Network Time Protocol (NTP) [Mil92] и необязательные указания по требованиям к пропускной способности.

Описание SDP также содержит информацию о каждом мультимедийном потоке, такую как тип содержания и формат кодирования (например, видео в формате MPEG). Кроме этого, SDP сообщает информацию по доставке потока: транспортный протокол (например, UDP) и разбивка на кадры (например, RTP). Для потока группового вещания описание сеанса включает IP-адрес и номер порта группового вещания. Получатель использует эту информацию для вступления в соответствующую группу вещания. Для потока индивидуального вещания описание сеанса содержит доменное имя или IP-адрес источника данных. Получатель может выдать DNS-запрос для преобразования доменного имени в IP-адрес, а затем связаться с удаленным компьютером для установления коммуникационного взаимодействия. Описание может также указывать на требования потока к пропускной способности сети. Получатель может использовать параметр пропускной способности для того, чтобы определить, принимать ли поток.

12.3.4. Описание презентаций

Создание мультимедийного содержимого требует наличия стандартного способа для подготовки презентаций. Язык гипертекстовой разметки Hypertext Markup Language (HTML) служит этой цели применительно к традиционному Web-содержанию. Однако HTML не имеет составляющих для управления временными свойствами мультимедийных презентаций. Разработанный W3C стандарт SMIL представляет собой язык разметки для построения потоковых мультимедийных презентаций с привязкой по времени, где объедены аудио-, видеоданные, текст и

изображения [Syn98, Hos00, Smi]. SMIL позволяет разработчикам задавать, *какое* содержание отображать (и где оно находится), *где* содержание следует отобразить на экране и *когда* содержание должно быть отображено. Подобно HTML, SMIL дает возможность авторам встраивать гипертекстовые ссылки, по которым осуществляются переходы, когда пользователь щелкает мышью на соответствующей чувствительной области на экране.

Автор SMIL-файла может определять разметку в виде набора *областей*. Каждая область представляет собой прямоугольник с определенными размерами в определенном месте на экрана. Каждая область ассоциируется с мультимедийными данными. Рассмотрим пример, когда университетский профессор читает лекцию удаленной аудитории. Презентация может связывать различные области с видеоизображением преподавателя, картинкой текущего слайда, демонстрируемого аудитории, и видеоизображением аудитории. Каждое из трех представлений занимает определенную часть экрана и может быть извлечено из различных источников с помощью различных протоколов и форматов кодирования. SMIL поддерживает встроенные гипертекстовые ссылки, например, щелчок на области с изображением в презентации может инициировать HTTP-запрос на Web-страницу, которая отображает подробную информацию по теме, освещаемой текущим слайдом. В более общем случае гипертекстовая ссылка может ассоциироваться с определенной областью и интервалом времени.

Помимо связывания отображения содержания с областями экрана, SMIL дает возможность автору координировать отображение данных во времени. В SMIL имеются возможности для указания, когда начинать воспроизведение потока и какую часть потока воспроизводить. SMIL содержит средства для синхронизации различных потоков или сеансов. Отдельные сеансы могут воспроизводиться последовательно, когда воспроизведение одного потока начинается после завершения воспроизведения предыдущего потока. Альтернативой является одновременное воспроизведение нескольких потоков. Плейер определяет, все ли параллельные потоки были завершены, когда можно воспроизводить следующий поток в последовательности. Параллельный и последовательный подходы могут быть объединены, чтобы дать возможность воспроизвести один длинный поток параллельно с последовательностью более коротких потоков.

Подготовка мультимедийного содержания осложняется тем фактом, что клиенты могут сильно различаться по пропускной способности, размеру экрана, возможностям плейера и предпочтениям пользователя. Например, один пользователь может иметь 21-дюймовый монитор с высоким разрешением и высокопроизводительное подключение к Internet. Другой пользователь может иметь карманный компьютер с беспроводным подключением. Слабослышащий пользователь может предпочесть получать текстовое сопровождение вместо потока аудиоинформации. SMIL может задавать различные параметры презентации, соответствующие различным мультимедийным потокам. Медиаплейер определяет, какие потоки извлекать и как их следует отображать. Это является удобной альтернативой обращения клиента к серверу с целью определить соответствующие параметры для потоков. SMIL может также задавать множество параметров презентации для определенной области, которые указывают, как пользователь может взаимодействовать с презентацией. Например, область может поддерживать увеличение и уменьшение изображения.

12.4. Real Time Streaming Protocol

Протокол Real Time Streaming Protocol (RTSP) дает возможность клиенту запрашивать живые или предварительно записанные потоки с мультимедийных серверов, подобно тому, как HTTP позволяет клиентам выдавать запросы к Web-серверам. Фактически RTSP перенял большую часть своего синтаксиса и семантики от HTTP/1.1, поскольку формально оба протокола выполняют схожие функции. Сходство подчеркивает общий характер многих реализованных в HTTP концепций. Однако протоколы имеют ряд ключевых отличий, которые связаны с уникальными требованиями для мультимедийных потоков и ограниченностью возможностей HTTP/1.1 по передаче мультимедийных данных. В этом разделе мы осуществим сравнение и противопоставление RTSP и HTTP. Затем мы опишем методы запросов RTSP, коды ответов и заголовки в сравнении и противопоставлении с HTTP/1.1. В основу обсуждения положен материал главы 7 (раздел 7.2), описывающий синтаксис HTTP/1.1.

12.4.1. Сходства и различия

Спецификация RTSP в значительной мере перекликается со спецификацией HTTP/1.1. На верхнем уровне оба протокола преследуют одну цель — дать возможность клиенту запрашивать содержание с сервера на основе URI запроса. Исторически разработка RTSP началась после того, как была выполнена часть работ над спецификацией HTTP. Поскольку HTTP послужил основой для RTSP, цели и задачи этих протоколов во многом совпадают. Доставка мультимедийных потоков следует той же модели, что и HTTP: ресурсы, идентифицируемые URI, взаимодействие запрос-ответ, а также возможность передачи данных через одно или несколько промежуточных звеньев на пути между клиентом и сервером. Протоколы имеют сопоставимые требования к безопасности и требования, предъявляемые к серверам-посредникам. Именно эти две области потребовали наиболее значительных усилий при разработке HTTP/1.1. Использование готовых концепций, включение специальных заголовков протокола и кодов ответов уменьшило затраты на разработку и реализацию RTSP. Кроме того, оба протокола играют роль в инициировании передачи мультимедийных потоков через Web. Сходство между двумя протоколами обеспечивает значительную гибкость при принятии решения, какой протокол будет обслуживать данную конкретную функцию.

Любой из рассматриваемых протоколов в принципе может обслужить всю задачу по извлечению описания мультимедийного сеанса и запросу мультимедийных данных. Хотя HTTP, возможно, не лучший протокол для выполнения всех этих действий, он может применяться для передачи описания сеанса. В ответ на щелчок пользователя мышью на гипертекстовой ссылке браузер может выдать HTTP-запрос на информацию с описанием сеанса (например, **http://www.foo.com/bar.sdp**). Ответ Web-сервера будет включать информацию, описывающую сеанс, в формате SDP, как показано ниже:

```
HTTP/1.1 200 OK
Content-Type: application/sdp

v=0
o=- 2890844526 2890842807 IN IP4 192.16.24.202
s=RTSP Session
```

```
m=audio 0 RTP/AVP 0
a=control:rtsp://foo.com/bar/audio
m=video 0 RTP/AVP 31
a=control:rtsp://foo.com/bar/video
```

В этот момент Web-браузер уже может активизировать медиаплейер для выполнения остальных действий, как было описано в разделе 12.3.3. Кроме того, медиаплейер может предоставить пользователю интерфейс для выбора мультимедийных сеансов. В этом случае медиаплейер может напрямую взаимодействовать с RTSP-сервером для извлечения описаний сеансов без привлечения Web-браузера.

Несмотря на многие черты сходства между двумя протоколами, RTSP отличается от HTTP по ряду ключевых моментов.

Отдельные соединения для данных и команд. В противоположность HTTP, RTSP-сервер создает для передачи данных отдельное соединение. В этом смысле RTSP близок к FTP. Клиент и сервер обмениваются RTSP-сообщениями через управляющее соединение. Задачу по передаче данных выполняют другие протоколы, такие как RTP и RTCP. Подобное разделение позволяет клиенту и серверу продолжать обмениваться RTSP-сообщениями во время передачи данных, чтобы адаптироваться к текущей обстановке или инициировать дополнительные передачи. Помимо этого, передача команд и передача данных могут использовать различные транспортные протоколы. RTSP-сообщения обычно передаются через TCP-соединение, хотя может быть использован и UDP. Обмен пакетами RTP и RTCP обычно осуществляется по UDP, хотя возможно и применение TCP.

Различные форматы URI. Для RTSP был зарезервирован порт 554. Выбор протокола (UDP или TCP) может быть задан в схеме URI. Схемы **rtsp** и **rtspu** относятся к TCP и UDP соответственно. Например, **rtsp://foo.com/bar** идентифицирует сеанс, который запрашивается и управляется по RTSP посредством TCP-соединения. С другой стороны, **rtspu://foo.com/bar** указывает, что клиент должен выдать RTSP-запрос посредством UDP. Запрашиваемый URI в RTSP может относиться как ко всему сеансу, так и к отдельным мультимедийным потокам. Строка запроса в RTSP-запросе должна содержать абсолютный путь, включая имя хоста. Это позволяет избежать неопределенности относительно того, какой RTSP-сервер должен получать запрос. В противоположность этому, в HTTP/1.1 для этой цели имеется отдельный заголовок **Host**, призванный сохранить обратную совместимость с HTTP/1.0, как обсуждалось ранее в главе 7 (раздел 7.8).

Протокол с сохранением промежуточного состояния. В противоположность HTTP RTSP-сервер сохраняет информацию между последовательными запросами. Клиент может отправлять несколько RTSP-сообщений, относящихся к одному сеансу или потоку. Это необходимо, поскольку клиент может выполнять функции видеомагнитофона, включая воспроизведение, пауза, ускоренная перемотка и обратная перемотка, в течение одного сеанса. Сервер должен иметь возможность интерпретировать эти запросы в контексте соответствующего потока. Чтобы обработать определенные запросы, серверу может также потребоваться хранить информацию о текущей передаче потока. Допустим, клиент запрашивает паузу в потоке, после чего запрашивает воспроизведение. Сервер должен продолжить передачу с определенного места в потоке. Сохранение промежуточного состояния в протоколе также полезно при передаче и приеме содержания, требующего значительных дисковых и сетевых ресурсов. В RTSP клиент должен выдать запрос серверу на выделение системных ресурсов для мультимедийного сеанса. Это дает возможность серверу заранее определить, достаточно ли у него системных ресурсов.

Различные методы, заголовки и коды состояния. В RTSP имеется иной набор методов запросов, нежели в HTTP, в том числе методов, используемых *сервером* для передачи сообщения-запроса клиенту. RTSP перенял многие коды ответов, хотя для указания ошибочных ситуаций были определены дополнительные коды, отсутствующие в HTTP. В RTSP были заимствованы многие заголовки HTTP с рядом важных добавлений и изъятий. По большей части неприятие ряда заголовков отражает тот факт, что RTSP не передает реальных мультимедийных данных. Большинство сообщений-ответов RTSP содержат только информацию, описывающую сеанс. Новые заголовки, определенные в RTSP, связаны главным образом с (1) параметрами таймирования мультимедийного потока, (2) наличием отдельных протоколов для передачи данных и (3) хранением информации о состоянии на клиенте или на сервере. Эти моменты обусловлены ключевыми различиями между RTSP и HTTP, которые проистекают из уникальных требований, предъявляемых к мультимедийным данным.

12.4.2. Методы запросов RTSP

RTSP-серверы должны поддерживать четыре основных метода, используемых клиентами для извлечения мультимедийных сеансов: **OPTIONS**, **SETUP**, **PLAY** и **TEARDOWN**. На верхнем уровне заголовок **OPTIONS** позволяет клиенту определять функциональные возможности сервера, такие как номер версии RTSP и поддерживаемые методы; этот метод ведет себя так же, как метод **OPTIONS** HTTP/1.1. Остальные три метода манипулируют сохраненной на сервере информацией о состоянии для координации передачи мультимедийных данных. Клиент использует метод **SETUP** для установления транспортного соединения для каждого потока в сеансе. Метод **PLAY** используется для инициирования передачи потока (потоков). Метод **TEARDOWN** используется для завершения передачи. Эти четыре метода описаны в таблице 12.2 вместе с другими RTSP-методами.

Таблица 12.2. Методы запросов RTSP

Направление	Метод	Описание
От клиента к серверу	SETUP	Выделяет системные ресурсы потоку
	PLAY	Начинает передачу данных потока
	TEARDOWN	Освобождает ресурсы, связанные с потоком
	DESCRIBE	Возвращает описание сеанса
	PAUSE	Временно останавливает передачу потока
	RECORD	Начинает запись потока
Двунаправленное	OPTIONS	Возвращает список поддерживаемых методов
	ANNOUNCE	Регистрирует описание презентации
	GET_PARAMETER	Извлекает значение именованного параметра
	SET_PARAMETER	Устанавливает значение именованного параметра
От сервера к клиенту	REDIRECT	Направляет клиента к другому серверу

В противоположность HTTP RTSP сохраняет информацию о состоянии в ответ на клиентские запросы. Помимо запросов **SETUP**, **PLAY** и **TEARDOWN**, методы **RECORD** и **PAUSE** также влияют на информацию о сеансе. Метод **RECORD** предписывает RTSP-серверу принять и сохранить поток с указанным URI для последующего воспроизведения. Метод **PAUSE** временно приостанавливает передачу, не освобождая системных ресурсов сервера. Последующий запрос **PLAY** (или **RECORD**) предписывает серверу продолжить передачу (или запись) потока. И клиент, и сервер сохраняют информацию о состоянии в интересах каждого потока. Методы **PLAY**, **RECORD**, **PAUSE** и **TEARDOWN** могут быть применены к отдельному потоку или ко всему сеансу, в зависимости от того, соответствует ли URI в RTSP-запросе потоку или сеансу. Метод запроса, применяемый к сеансу, воздействует на каждый из составляющих сеанс потоков. Метод **SETUP** применяется только к отдельному потоку.

Механизм управления состоянием для клиента и сервера представлен на рис. 12.1. Сервер изменяет состояние при обработке метода запроса; клиент изменяет состояние после получения ответа от сервера. Метод **SETUP** запускает передачу с начального состояния (*Инициализация*) и вызывает переход к состоянию *Готовность*. Метод **TEARDOWN** возвращает клиента и сервер в состояние *Инициализация*. Методы **PLAY** и **RECORD** переводят в состояния *Воспроизведение* и *Запись*, соответственно, а метод **PAUSE** возвращает в состояние *Готовность*. Передача данных осуществляется только в состояниях *Воспроизведение* и *Запись*. Находясь в одном из этих двух состояний, клиент может инициировать запрос **SETUP** для изменения транспортных параметров потока. Сервер продолжает передачу потоков мультимедийных данных, используя, возможно, другой транспортный протокол или другие порты.

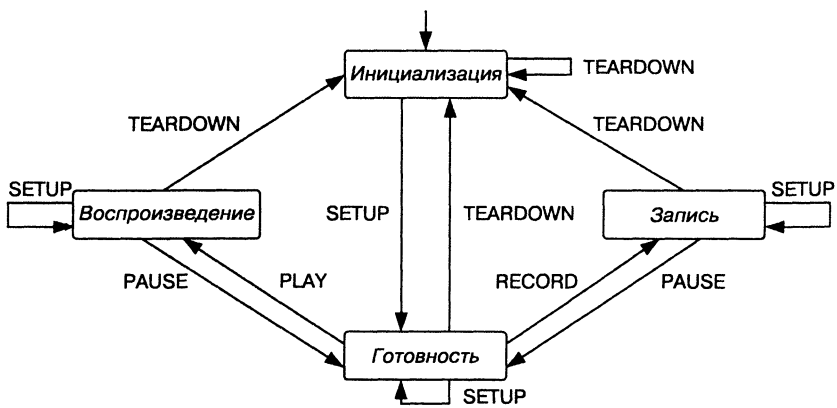


Рис. 12.1. Диаграмма состояний для клиентов и серверов RTSP

Остальные методы RTSP не оказывают влияния на состояния RTSP на клиенте и на сервере, поэтому не представлены на рис. 12.1. Например, клиент может выдать запрос **OPTIONS**, чтобы узнать возможности сервера, не воздействуя на текущую передачу. Клиент в любое время может отправить запрос **DESCRIBE**, чтобы получить описательную информацию о сеансе или потоке. В RTSP также предусмотрен метод **ANNOUNCE**, который дает возможность серверу отправлять клиенту новое описание. Допустим, сервер размещает сеанс реального времени, который первоначально состоит из одного аудиопотока. Предположим, что затем появляет-

ся еще и видеопоток. Сервер может отправить клиентам, прослушивающим текущую аудиоинформацию, запрос **ANNOUNCE**, содержащий обновленное описание сеанса. На основе этой информации клиент может инициировать запрос на видеопоток, отправив запрос **SETUP** с последующим запросом **PLAY**. RTSP также дает возможность клиенту отправлять запрос **ANNOUNCE** серверу, чтобы тот создал новое описание сеанса; здесь имеется аналогия с методом **PUT** HTTP.

Давая возможность серверам посылать запросы клиентам, RTSP отличается от традиционной модели клиент-сервер. Хотя поддержка запросов от сервера к клиенту не является обязательной, эти методы повышают гибкость функционирования сервера. Например, метод **REDIRECT** позволяет серверу проинструктировать клиента связаться с другим сервером по указанному URI. После получения запроса **REDIRECT** клиент должен выдать запрос **TEARDOWN** для текущего сеанса, а затем выдать запрос **SETUP** новому серверу. Сервер может иметь множество причин для переадресации клиента. Переадресация может повысить производительность, если новый сервер находится ближе к клиенту или менее загружен. Кроме того, новый сервер может предоставить другое содержание, отсутствующее на исходном сервере. В этом смысле метод **REDIRECT** схож с классом кодов ответов переадресации в HTTP/1.1. Однако RTSP-сервер может переадресовывать клиента в любое время в процессе передачи мультимедийного потока.

Чтобы выдать запрос, сервер должен знать, какие методы поддерживаются конкретным клиентом. Для этого сервер может отправить клиенту запрос **OPTIONS**. Кроме методов **OPTIONS** и **ANNOUNCE** в RTSP имеются два других метода, которые могут использоваться как клиентами, так и серверами. Методы **GET_PARAMETER** и **SET_PARAMETER** дают возможность отправителю узнать и откорректировать параметры мультимедийного сеанса или потока, ассоциированного с конкретным URI запроса. Эти два метода обеспечивают поддержку операций чтения и записи произвольного набора параметров для конкретного клиента и сервера. Например, сервер может использовать метод **GET_PARAMETER** для запроса количества пакетов данных, полученных клиентом для определенного потока. Тем самым обеспечивается альтернатива протоколу RTCP для получения информации о качестве транспортировки данных. У клиента имеется возможность отправить запрос **SET_PARAMETER** с целью уменьшения сервером скорости передачи. Эти методы обеспечивают клиенту и серверу гибкую и расширяемую возможность по организации взаимодействия в управлении мультимедийным потоком.

Большинство методов запросов связано с поддержкой клиента, который запрашивает воспроизведение потока с мультимедийного сервера. Однако RTSP-сервер может также допускать запись мультимедийного содержания с помощью необязательных методов **ANNOUNCE** и **RECORD**. Допустим, пользователь хочет записать телеконференцию на RTSP-сервере для дальнейшего воспроизведения. В этом случае клиент отправляет описание конференции серверу в сообщении **ANNOUNCE**. Затем клиент должен отправить сообщение **SETUP** для каждого из потоков в сеансе, чтобы проинформировать сервер о транспортных параметрах, таких как IP-адреса и номера портов. URI запроса в сообщении **SETUP** указывает имя, которое клиент хочет ассоциировать с записываемым содержанием. После этого клиенту следует отправить запрос **RECORD** для начала записи потоков. Через некоторое время клиент может связаться с сервером и просмотреть записанную конференцию.

12.4.3. Заголовки RTSP

На момент появления документа RFC 2326 по RTSP информация по HTTP/1.1 содержалась в документе RFC 2068, а не в проекте стандарта RFC 2616. Многие RTSP-заголовки в этой связи были заимствованы из RFC 2068. Как уже рассказывалось в главе 7 (раздел 7.2.2), в HTTP имеются заголовки запросов, заголовки ответов, общие заголовки и заголовки содержимого. В последующих таблицах в этом разделе имеются пометки в виде звездочек (*), указывающие, что заголовки были унаследованы от HTTP/1.1. В конце раздела мы поговорим о заголовках HTTP/1.1, которые не были использованы в RTSP.

ОБЩИЕ ЗАГОЛОВКИ

Общие заголовки могут присутствовать и в сообщениях-запросах, и в сообщениях-ответах. RTSP унаследовал четыре общих заголовка HTTP/1.1, а шесть новых заголовков были добавлены (см. таблицу 12.3). Заголовки **Session** и **CSeq** идентифицируют последовательность RTSP-сообщений, ассоциированных с одним сеансом. RTSP не предъявляет требований к наличию одного выделенного соединения транспортного уровня в ходе сеанса. RTSP-сеанс может иметь несколько TCP-соединений, либо клиент и сервер могут взаимодействовать по UDP. Каждое RTSP-сообщение, которое воздействует на состояние сеанса, содержит идентификатор сеанса —строку, присваиваемую сервером. RTSP-сервер назначает идентификатор после получения запроса **SETUP** и возвращает значение в заголовке **Session**. Последующие сообщения-запросы и сообщения-ответы, относящиеся к данному сеансу, содержат заголовок **Session**. Заголовок **CSeq** указывает порядковый номер, ассоциированный с каждым RTSP-сообщением. Необходимость наличия порядковых номеров связана с двумя ключевыми отличиями RTSP от HTTP. Во-первых, поскольку в RTSP промежуточное состояние сохраняется, с одним сеансом могут быть связаны различные пары запрос-ответ. Во-вторых, поскольку RTSP-сообщения могут передаваться по UDP, протокол не может гарантировать надежной, упорядоченной доставки сообщений. Порядковый номер увеличивается на единицу для каждого следующего запроса; соответствующее сообщение-ответ имеет тот же порядковый номер. При передаче RTSP-сообщений по UDP клиент и сервер ответственны за повторную передачу потерянных UDP-сообщений, если такая передача требуется. Любой повторно переданный запрос имеет тот же порядковый номер, что и оригинальное сообщение.

Заголовок **Transport** используется в сообщениях запросов и ответов **SETUP** для координации выбора транспортного протокола и параметров потока. В сообщении-запросе заголовок

```
Transport: RTP/AVP/UDP;unicast;client_port=18366-18367
```

предписывает серверу доставить поток посредством протоколов RTP и UDP с типом полезной нагрузки AVP (Audio Video Profile) [Sch96]; клиент выбирает порт 18366 для UDP-соединения и порт 18367 для RTCP-соединения. В сообщении-запросе заголовок **Transport** может содержать несколько наборов параметров. Заголовок может включать множество других параметров, таких как адрес назначения для группового вещания, методы RTSP-запросов, поддерживаемые для сеанса, и схему сжатия данных. Заголовок **Transport** в сообщении-ответе сервера подтверждает параметры, выбранные клиентом

```
Transport: RTP/AVP/UDP;unicast;client_port=18366-18367;
           server_port=16970-16971
```

и содержит номера портов сервера.

Таблица 12.3. Общие RTSP-заголовки
(заголовки, имеющиеся в HTTP/1.1, помечены '*')

Заголовок	Описание
Session	Идентификатор сеанса
CSeq	Порядковый номер сообщения
Transport	Согласование параметров транспорта
Range	Выбор интервала времени для воспроизведения
Scale	Согласование скорости воспроизведения
Speed	Согласование скорости передачи
Date*	Дата/время создания сообщения
Via*	Список промежуточных серверов
Connection*	Список заголовков механизма промежуточных передач
Cache-Control*	Директивы кэширования

Заголовки **Range**, **Scale** и **Speed** относятся к временным свойствам мультимедийных потоков. Заголовок **Range** используется для указания определенного места в потоке. Рассмотрим медиаплейер, который дает возможность пользователю осуществлять прокрутку к любой точке в аудиоклипе. Пользователь может захотеть пропустить первые 30 секунд клипа. Вместо того чтобы получать весь поток, можно включить в RTSP-запрос **PLAY** заголовок **Range**, указывающий желаемый интервал времени. Концептуально это похоже на использование заголовка запроса **Range** в HTTP/1.1. Однако запросы на диапазон в HTTP/1.1 идентифицируют подмножество данных в ресурсе, диапазон в RTSP определяет интервал времени для потока. Например, клиентский запрос **PLAY** может содержать

```
Range: npt=30-
```

для указания серверу начать передачу данных с 30-й секунды в потоке (NPT означает нормальное время воспроизведения (Normal Play Time), при этом началу потока соответствует нулевая секунда). На практике индексирование по времени вызывает больше затруднений, чем индексирование по данным, поскольку размеры кадров на протяжении потока могут меняться. 30-ая секунда точка в 60-секундном потоке может не соответствовать точно середине в последовательности байтов. Кроме того, некоторые кадры в потоке кодируются относительно более ранних или более поздних кадров, что затрудняет начало воспроизведения потока в точно заданное время. Поскольку RTSP-сервер может оказаться не в состоянии начать передачу данных с произвольной точки в аудио- или видеопотоке, RTSP разрешает серверу инициировать передачу с близкого момента времени. Сервер указывает выбранный интервал времени в заголовке **Range** сообщения-ответа. Спецификация RTSP не предъявляет каких-либо требований относительно того, насколько значение **Range**, указанное в сообщении-ответе, может отличаться от значения **Range**, запрошенного клиентом.

Заголовки **Scale** и **Speed** поддерживают функции ускоренной и обратной перемотки для методов **PLAY** и **RECORD**. **Scale** относится к скорости воспроизведения медиаплейером, а **Speed** относится к скорости передачи данных сервером. Обычная скорость воспроизведения соответствует значению 1. Большие положительные значения соответствуют ускоренной перемотке, меньшие положительные

значения соответствуют воспроизведению содержания в замедленном режиме. Отрицательное значение указывает, что поток должен быть воспроизведен в обратном направлении. Клиент передает желаемую скорость воспроизведения в заголовке **Scale**. Сервер пытается поддерживать желаемую скорость воспроизведения и возвращает выбранную скорость в заголовке **Scale** сообщения-ответа. Сервер адаптирует передачу потока, чтобы избежать повышения скорости поступления данных. Например, когда клиент запрашивает операцию ускоренной перемотки в видеоклипе, сервер может доставлять не все, а подмножество кадров. Хотя выборка подмножества кадров не предъявляет требований к увеличению скорости передачи, качество ускоренного воспроизведения будет хуже.

Клиент и сервер могут также согласовать скорость передачи, используемую для доставки данных. По умолчанию передача выполняется с потребной для воспроизведения потока скоростью. Это соответствует значению скорости, равному 1. Большее значение соответствует увеличению скорости, а меньшее значение — уменьшению. Клиент запрашивает желаемую скорость передачи в заголовке **Speed** сообщения-запроса, а сервер возвращает выбранную скорость передачи в заголовке **Speed** сообщения-ответа. Заголовок **Speed** может быть использован вместе с заголовком **Scale** для координации передачи потока во время операций ускоренной и обратной перемотки. Клиент может использовать заголовок **Speed** для получения потока с более высокой скоростью передачи, не допуская снижения качества, как это делается при ускоренной перемотке.

Остальные четыре общих заголовка заимствованы из HTTP/1.1. Заголовок **Date** указывает время создания сообщения. Заголовок **Via** используется для идентификации цепочки серверов-посредников на пути между клиентом и исходным сервером. Заголовок **Connection** содержит список заголовков механизма промежуточных передач, которые удаляются следующим получателем на пути. Заголовок **Cache-Control** используется для передачи директив кэширования. Однако интерпретация заголовка **Cache-Control** в RTSP отличается. В отличие HTTP, большинство RTSP-ответов не являются кэшируемыми, за исключением информации с описанием сеанса, возвращаемой в ответ на запрос **DESCRIBE**. В RTSP директивы **Cache-Control** относятся к мультимедийному потоку, который передается другим протоколом. Заголовок **Cache-Control** присутствует только в запросе **SETUP** и сообщениях-ответах, которые устанавливают параметры транспорта. Как и в HTTP/1.1, различные директивы **Cache-Control** предоставляют клиенту несколько способов для выражения требований к актуальности информации, а серверу — для выражения возможностей кэширования данных.

ЗАГОЛОВКИ ЗАПРОСОВ

В RTSP девять заголовков запросов были заимствованы из HTTP/1.1 и появилось пять новых заголовков запросов (см. таблицу 12.4). Новые заголовки запросов связаны с таймированием и выделением системных ресурсов. Клиент использует заголовок **Bandwidth** для предоставления оценки потребных сетевых ресурсов в битах в секунду. Сервер может использовать эту информацию для выбора скорости передачи при доставке мультимедийных данных клиенту с помощью транспортного протокола. Клиент может использовать заголовок **Blocksize** для указания желаемого максимального размера пакета при передаче данных. В размер пакета не входят заголовки нижних уровней, такие как заголовки IP, UDP и RTP. Сервер может выбрать размер пакета меньший или равный размеру, запрошенному клиентом. Заголовки **Bandwidth** и **Blocksize** решают проблемы, которые в HTTP просто не могут возникнуть, поскольку HTTP-сервер передает

данные в виде упорядоченного, надежного потока данных со скоростью, определяемой транспортным протоколом.

Таблица 12.4. Заголовки запросов RTSP
(заголовки, имеющиеся в HTTP/1.1, помечены '*')

Заголовок	Описание
Bandwidth	Оценивает пропускную способность сети, доступную для клиента
Blocksize	Желаемый размер пакета для передачи данных
Conference	Идентификатор конференции, ассоциированный с потоком
Require	Запрос относительно поддержки функции сервером
Proxy-Require	Запрос относительно поддержки функции прокси-сервером
From*	Адрес электронной почты пользователя
User-Agent*	Информация о программном обеспечении агента пользователя
Authorization*	Полномочия агента пользователя
Referer*	URI, от которого получен URI запроса
If-Match*	Проверка соответствия с атрибутами содержимого
If-Modified-Since*	Проверка времени последнего изменения
Accept*	Предпочтительные типы содержания
Accept-Encoding*	Предпочтительное кодирование содержания
Accept-Language*	Предпочтительные языки

Заголовок **Conference** устанавливает логическую связь между потоком RTSP и существующей конференцией по другому протоколу, например, SIP. С точки зрения RTSP-сервера, идентификатор конференции представляет собой кодированную строку. Связывание RTSP-сервера с текущей конференцией полезно для различных целей. Например, клиент хочет воспроизвести предварительно записанный аудиоклип в имеющейся конференции. Клиент может включить заголовок **Conference** в запрос **SETUP**, чтобы идентифицировать соответствующую конференцию. Запрос **SETUP** также включает заголовок **Transport** для передачи информации о текущей конференции. Например, заголовок **Transport** может указывать IP-адрес группового вещания, а также номера портов. После получения ответа на сообщение **SETUP** RTSP-клиент может выдать запрос **PLAY** для инициирования передачи потока аудиоинформации от RTSP-сервера группе вещания.

Заголовки **Require** и **Proxy-Require** предоставляют клиенту эффективный и допускающий расширение способ узнать, какие функциональные возможности поддерживаются сервером. Параметр заголовка представляет собой строку, содержащую названия функций, которые клиент хотел бы использовать (например, **Require: unusual-feature**). Если сервер не распознает или не поддерживает функцию, приведенную в заголовке **Require**, RTSP-ответ возвращает клиенту отрицательное подтверждение ("Unsupported Header"). В противоположность этому, в HTTP нет заголовков запроса для определения функциональных возможностей сервера. HTTP-серверы могут без ущерба игнорировать определенные заголовки, которые они не поддерживают. Заголовок **Require** дает возможность клиенту выдавать RTSP-запрос до того, как он узнает, поддерживает ли сервер нужную функ-

цию. Агент пользователя использует заголовок **Proxy-Require** для указания, что функции должны поддерживаться также всеми прокси-серверами на пути от сервера к клиенту.

В RTSP имеется несколько заголовков запросов HTTP/1.1, связанных с идентификацией и авторизацией клиента (например, **From**, **User-Agent**, **Authorization** и **Referer**), кэшированием (например, **If-Match** и **If-Modified-Since**) и согласованием содержания (например, **Accept**, **Accept-Encoding** и **Accept-Language**). Однако в RTSP нет полного набора заголовков HTTP/1.1, связанных с кэшированием и согласованием содержания. Большинство методов RTSP не возвращают содержимого, поскольку мультимедийные потоки передаются по отдельным транспортным соединениям. Содержимое включается в запрос **ANNOUNCE** и в ответ на запрос **DESCRIBE**. Содержимым является описание сеансов, которое может быть представлено на различных языках (например, английском или швейцарском немецком) и с другим типом содержания (например, SDP).

ЗАГОЛОВКИ ОТВЕТОВ

В RTSP семь заголовков ответов было заимствовано из HTTP/1.1 и введено два новых заголовка ответов (см. таблицу 12.5). Заголовок **RTP-Info** играет важную роль в координации приема мультимедийных потоков. Присутствуя в ответе сервера на запрос **PLAY**, заголовок **RTP-Info** объявляет значения специфических для RTP параметров. Например,

```
RTP-Info: url=rtsp://foo.com/bar/audio;seq=47;rtptime=2894,
          url=rtsp://foo.com/bar/video;seq=184;rtptime=6674
```

Каждый мультимедийный поток состоит из последовательности RTP-пакетов со случайно выбранными значениями для начального порядкового номера и начальной временной метки, как это обсуждалось ранее в разделе 12.3.1. При передаче, инициированной RTSP, клиент узнает эти значения из заголовка **RTP-Info**. Первый RTP-пакет в потоке будет иметь эти начальные значения. Порядковый номер и временная метка увеличиваются для последовательных RTP-пакетов.

Таблица 12.5. Заголовки ответов RTSP
(заголовки, имеющиеся в HTTP/1.1, помечены '*')

Заголовок	Описание
RTP-Info	Специфические для RTP параметры в запросе **PLAY**
Unsupported	Функция не поддерживается сервером
Server*	Идентификация сервера
Location*	Местонахождение ресурса
Retry-After*	Задержка перед попыткой повторного запроса
Vary*	Выбор версии ресурса
Public*	Список поддерживаемых методов (**Allow** в HTTP/1.1)
WWW-Authenticate*	Вызов для аутентификации
Proxy-Authenticate*	Вызов для аутентификации

В RTSP также имеется заголовок ответа **Unsupported** для реакции на сообщения-запросы с заголовками **Require** и **Proxy-Require**. Запрашиваемые опции, не

поддерживаемые сервером, идентифицируются в заголовке **Unsupported** сообщения-ответа. Тем самым серверу предоставляется возможность разъяснить, почему клиентский запрос не может быть выполнен, а не просто возвратить код ошибки. Заголовки ответов, заимствованные из HTTP/1.1, относятся к идентификации сервера (**Server**), переадресации запроса (**Location**), повторному запросу (**Retry-After**), кэшированию (**Vary**) и аутентификации (**WWW-Authenticate** и **Proxy-Authenticate**). Заголовок **Public**, используемый для указания списка опций, поддерживаемых сервером, определен в RFC 2068 и не был включен в RFC 2616. Заголовок **Public** в HTTP был заменен на заголовок **Allow**, как обсуждалось ранее, в главе 6 (раздел 6.2.3). В HTTP/1.1 **Allow** представляет собой заголовок содержимого.

ЗАГОЛОВКИ СОДЕРЖИМОГО

В RTSP девять заголовков содержимого были заимствованы из HTTP/1.1, а новых заголовков содержимого введено не было (см. таблицу 12.6). По сравнению с HTTP, в RTSP содержимое сообщений трактуется достаточно просто. Содержимое RTSP-сообщения, если оно присутствует, представляет собой описание сеанса. В RTSP любое сообщение, которое содержит какое-либо содержимое, должно иметь заголовок **Content-Length**, гарантирующий, что получатель сможет идентифицировать конец сообщения. В HTTP/1.1 же применен ряд других способов для определения конца сообщения; например, содержимое в сообщении HTTP/1.1 может передаваться в закодированном виде или состоять из нескольких фрагментов. RTSP не поддерживает эти функции. Заголовок **Content-Base** предоставляет основной URI, используемый для относительных URI в теле содержимого. Хотя заголовок **Content-Base** был определен в предложении RFC 2068, в проект стандарта HTTP/1.1 (RFC 2616) этот заголовок не вошел. Заголовки **Expires** и **Last-Modified** в RTSP связаны с кэшированием информации, содержащей описание сеанса. Заголовок **Expires** указывает, как долго описание сеанса может быть кэшировано без повторной проверки актуальности информации. Заголовок **Last-Modified** указывает время последнего изменения описания сеанса. Аргумент заголовка **Last-Modified** может быть использован в дальнейших запросах **If-Modified-Since** для проверки актуальности кэшированной копии содержимого. Заголовок **Allow** содержит список методов, которые поддерживаются для запрошенного URI.

Таблица 12.6. Заголовки содержимого RTSP (заголовки, имеющиеся в HTTP/1.1, помечены '*')

Заголовок	Описание
Content-Length*	Длина тела содержимого (обязателен, если имеется содержимое)
Content-Type*	Тип содержания тела содержимого
Content-Language*	Язык, используемый в теле содержимого
Content-Encoding*	Вид кодирования, примененный к телу содержимого
Content-Location*	Альтернативное местоположение ресурса
Content-Base	Основной URI, используемый для относительной адресации внутри тела содержимого (в RFC 2068)
Expires*	Истечение срока хранения описания сеанса
Last-Modified*	Время последней модификации описания сеанса
Allow*	Методы, которые могут быть применены к ресурсу

ЗАГОЛОВКИ, ОТСУТСТВУЮЩИЕ В HTTP/1.1

Несколько заголовков HTTP/1.1 не были включены в спецификацию RTSP. Эти заголовки относятся к ряду категорий, выражающих различия между двумя протоколами (см. таблицу 12.7). Большинство заголовков связано с поддержкой HTTP различного содержимого в сообщениях запросов и ответов. Как говорилось выше, RTSP-сообщения не содержат содержимого, за исключением описаний сеансов. Описания сеансов представляют собой относительно короткие текстовые документы. По сравнению с HTTP, RTSP не имеет мощных механизмов поддержки передачи различных видов содержимого. Клиенту и серверу RTSP не нужно согласовывать набор символов или способ кодирования для описания сеанса. RTSP-сообщения не содержат хэша MD5 для тела содержимого, а после содержимого сообщения отсутствуют какие-либо трейлеры. RTSP также не поддерживает проверку актуальности ресурсов, передаваемых в RTSP-сообщениях.

В RTSP отсутствует ряд заголовков HTTP/1.1, относящихся к запросам на диапазоны. В HTTP/1.1 клиент может использовать заголовок запроса **Range** для того, чтобы запросить подмножество байтов ресурса. Если ресурс с указанным URI не допускает запросов на диапазоны, HTTP-ответ возвращает весь ресурс вместе с заголовком **Accept-Ranges: none**. В отличие от этого, RTSP-сервер, не поддерживающий запросы на диапазоны, не передает клиенту весь мультимедийный поток автоматически. Вместо этого сервер отвечает клиенту кодом ошибки. Таким образом необходимость в заголовке **Accept-Ranges** отпадает. HTTP и RTSP несколько различным образом реагируют на успешный запрос на диапазон. Сервер HTTP/1.1 указывает, какой диапазон байтов возвращается, в заголовке содержимого **Content-Range**. В RTSP же сервер указывает интервал времени, выбранный в заголовке **Range.** Этот диапазон не обязательно должен совпадать с запрашиваемым интервалом. Разница в названии заголовков отражает тот факт, что в RTSP интервал времени относится к мультимедийному потоку, доставляемому по отдельному транспортному соединению, а не к содержимому RTSP.

Таблица 12.7. Заголовки HTTP/1.1, не включенные в RTSP

Категория	Заголовок	Описание
Передача содержимого	**Accept-Charset**	Допустимый набор символов
	TE	Допустимая кодировка при передаче
	Transfer-Encoding	Преобразование, применяемое к телу сообщения
	Content-MD5	Проверка целостности тела содержимого
	Trailer	Список заголовков в конце сообщения
Проверка актуальности	**ETag**	Валидатор
	Age	Время, прошедшее с момента создания ответа
	If-None-Match	Сравнение с атрибутами содержимого
	If-Unmodified-Since	Сравнение со временем последней модификации
	If-Range	Условный запрос на диапазон
Запрос на диапазон	**Accept-Ranges**	Нежелание принимать запросы на диапазоны
	Content-Range	Местоположение диапазона в теле содержимого

Категория	Заголовок	Описание
HTTP/1.0	**Host**	Доменное имя сервера с запрашиваемым ресурсом
	Pragma	Директива сообщения
Разное	**Upgrade**	Переход к другим протоколам
	Warning	Уведомление об ошибке
	Max-Forwards	Ограничение на пересылку запроса
	Expect	Ожидаемая клиентом реакция сервера

В RTSP отсутствуют заголовки **Host** и **Pragma**, которые были включены в HTTP/1.1 для обратной совместимости с HTTP/1.0. Клиент HTTP/1.1 использует заголовок **Host** для указания доменного имени сервера, поскольку эта информация не может содержаться в URI запроса. В отличие от этого, URI запроса в RTSP содержит абсолютный путь, что избавляет от необходимости иметь отдельный заголовок, передающий имя сервера. Заголовок **Pragma** был введен в HTTP/1.0 для директив управления сообщением, таких как **Pragma: no-cache**. В HTTP/1.1 используется другой подход для передачи информации о кэшировании — заголовок **Cache-Control**. Однако для поддержания обратной совместимости с HTTP/1.0 требуется, чтобы клиенты и серверы HTTP/1.1 понимали заголовок **Pragma**. Поскольку в RTSP подобных проблем с обратной совместимостью не существует, заголовки **Host** и **Pragma** ему не нужны. Также не используются заголовки **Upgrade** и **Warning**. В RTSP также отсутствует заголовок **Max-Forwards**, который относится к методам запросов HTTP/1.1, и заголовок **Expect**, который отсутствует в RFC 2068.

12.4.4. Коды состояния RTSP

Каждое сообщение-ответ содержит код состояния, указывающий на результат выполнения запроса. Коды состояния RTSP разделены на те же пять категорий, что и коды ответов HTTP: информационные (1xx), успешного выполнения (2xx), переадресации (3xx), ошибки клиента (4xx) и ошибки сервера (5xx). В RTSP введено несколько новых кодов состояния, приведенных в таблице 12.8. RTSP также заимствовал многие коды состояния, определенные в документе RFC 2068 (см. таблицу 12.9). Во избежание конфликтов с новыми кодами ответов HTTP/1.1, которые могут появиться в будущем, специфические для RTSP коды начинаются с x50. Новые коды относятся к нескольким основным категориям:

- **Системные ресурсы.** Передача или прием мультимедийного потока может потребовать значительных ресурсов сервера. В ответ на запрос **RECORD** код состояния **250 Low on Storage Space** предупреждает клиента, что на сервере отсутствует достаточный объем памяти для хранения записанного потока. Сервер может отклонить клиентский запрос на воспроизведение или запись потока, выдав ответ **453 Not Enough Bandwidth**.
- **Неизвестный идентификатор.** Клиентский запрос может содержать переменную или адрес, которые сервер не распознает. При получении запроса **SET_PARAMETER**, который ссылается на неизвестный параметр, сервер посылает ответ **451 Parameter Not Understood**. Аналогично сервер посылает ответ **452 Conference Not Found** или **454 Session Not Found** при получении запроса с неизвестным идентификатором конференции **Conference** или сеанса **Session**, соответственно. Если сервер не может установить соединение для

доставки потока данных клиенту, сервер отправляет ответ **462 Destination Unreachable**.

- **Неподдерживаемая функция.** Клиентский запрос может содержать метод или заголовок, который не поддерживается для ресурса с указанным в запросе URI. Например, некоторые методы могут быть некорректными для данной ситуации (**455 Method Not Valid in This State**), а некоторые заголовки могут быть некорректными для определенных запрашиваемых URI (**456 Header Field Not Valid for Resource**). Запрос **PLAY** может пытаться осуществить перемещение в несуществующее место в потоке (**457 Invalid Range**), либо запрос **SET_PARAMETER** может пытаться осуществить запись параметра, предназначенного только для чтения (**458 Parameter Is Read-Only**). Заголовок **Transport** в запросе **PLAY** может не содержать поддерживаемой спецификации транспорта (**461 Unsupported Transport**), либо сервер может не поддерживать функцию, содержащуюся в заголовках **Require** или **Proxy-Require** (**551 Option Not Supported**).
- **Операции с сеансом/потоком.** Клиентский запрос может нарушать иерархию потока или сеанса. Например, клиентский запрос может пытаться выполнить операцию потокового уровня над идентифицируемым по URI запроса ресурсом, который является сеансом (**459 Aggregate Operation Not Allowed**), либо операцию сеансового уровня над ресурсом, который является потоком (**460 Only Aggregate Operation Allowed**).

Большинство новых кодов состояния относится к ошибкам клиента (4xx), за исключением кодов **250 Low on Storage space** и **551 Option Not Supported**, которые представляют собой код успешного выполнения и код ошибки сервера, соответственно.

Таблица 12.8. Новые коды состояния RTSP

Категория	Код состояния и краткое описание
Системные ресурсы	250 Low on Storage Space (Недостаточно памяти для хранения)
	453 Not Enough Bandwidth (Пропускная способность недостаточна)
Неизвестный идентификатор	451 Parameter Not Understood (Параметр не может быть распознан)
	452 Conference Not Found (Конференция не найдена)
	454 Session Not Found (Сеанс не найден)
	462 Destination Unreachable (Место назначения недостижимо)
Неподдерживаемые функции	455 Method Not Valid in This State (Неверный метод для данного состояния)
	456 Header Field Not Valid for Resource (Неверное поле заголовка для ресурса)
	457 Invalid Range (Неверный интервал)
	458 Parameter Is Read-Only (Параметр только для чтения)
	461 Unsupported Transport (Неподдерживаемый транспорт)
	551 Option Not Supported (Опция не поддерживается)

Категория	Код состояния и краткое описание
Сеанс/поток	459 Aggregate Operation Not Allowed (Операция агрегирования запрещена)
	460 Only Aggregate Operation Allowed (Разрешена только операция агрегирования)

Резюмируя, можно сказать, что RTSP представляет собой протокол для доставки мультимедийных потоков, производный от HTTP/1.1 (см. таблицу 12.9). Протокол поддерживает воспроизведение мультимедийных данных с сервера и запись мультимедийных данных на сервер. RTSP и HTTP имеют сходную инфраструктуру для применения метода запроса к ресурсу, идентифицируемому URI, в соответствии с которой сообщение-запрос и сообщение-ответ содержат заголовки и необязательное содержимое. Спецификация RTSP во многом следует концепциям HTTP/1.1, связанным с кэшированием, серверами-посредниками и аутентификацией. Эти сходные черты проявляют себя во многих заголовках и кодах состояния RTSP, заимствованных из HTTP/1.1. Тем не менее, RTSP имеет ряд существенных отличий от HTTP. RTSP представляет собой протокол с сохранением промежуточного состояния, а передача данных осуществляется по другим протоколам. Кроме того, в RTSP определено несколько дополнительных заголовков и кодов состояния, которые связаны уникальными для мультимедийных потоков требованиями таймирования, а также к иерархическими отношениями между кадрами, потоками и сеансами. Эти характеристики проявляют себя в добавленных методах запросов, заголовках и кодах состояния.

Таблица 12.9. Коды состояния RTSP, заимствованные из HTTP/1.1

100 Continue	406 Not Acceptable
200 OK	407 Proxy Authentication Required
201 Created	408 Request Time-out
300 Multiple Choices	410 Gone
301 Moved Permanently	411 Length Required
302 Moved Temporaly	412 Precondition Failed
303 See Other	413 Request Entity Too Large
304 Not Modified	414 Request-URI Too Large
305 Use Proxy	415 Unsupported Media Type
400 Bad Request	500 Internal Server Error
401 Authorized	501 Not Implemented
402 Payment Required	502 Bad Gateway
403 Forbidden	503 Service Unavailable
404 Not Found	504 Gateway Time-out
405 Method Not Allowed	505 RTSP Version Not Supported

12.5. Резюме

Аудио- и видеопотоки отличаются от традиционного Web-содержания по трем основным признакам. Во-первых, аудио- и видеоданные имеют параметры таймирования, которые оказывают влияние на то, как медиаплейер получает и воспроизводит данные. Во-вторых, доставка мультимедийных потоков требует от сервера выделения значительной полосы пропускания в течение продолжительного периода времени. В-третьих, для ряда приложений, работающих с мультимедийными потоками, допустима потеря некоторого количества данных. Эти уникальные характеристики вкупе с наличием разнообразных мультимедийных приложений привели к разработке протоколов, пригодных для доставки мультимедийных потоков. Взаимное сближение мультимедийных приложений и Web пока еще только начинается. В дальнейшем приложения для мультимедийных потоков и соответствующие протоколы будут продолжать эволюционировать.

Часть VI
Перспективы исследований

13

Перспективы исследований, связанных с кэшированием

Для исследований, связанных с кэшированием, характерны быстрые изменения — в течение короткого времени были опубликованы сотни документов, проведено множество конференций, образованы десятки компаний. В главе, посвященной кэшированию (глава 11), мы рассмотрели различные технические проблемы, связанные с кэшированием. Большинство обсуждаемых в главе 11 идей было предложено в последние несколько лет и в той или иной степени получило признание. В этой главе мы расскажем о новых идеях, которые могут не выдержать испытания временем и не получить достаточного признания, чтобы найти отражение в официальных документах (RFC) Internet Engineering Task Force (IETF) или в коммерческих продуктах.

Как отмечалось в главе 11, имеются три основных причины, по которым кэширование получило массовое признание: уменьшение времени ожидания на стороне пользователя, уменьшение нагрузки на сеть и уменьшение нагрузки на исходный сервер. Неудивительно, что большинство исследовательских работ, связанных с кэшированием, было направлено на изучение степени воздействия на эти три фактора. Вместо того чтобы попытаться представить обширный обзор по разнообразным исследованиям, связанным с кэшированием, мы решили остановиться лишь на некоторых из них, наиболее интересных с точки зрения авторов. Некоторым исследованиям уделено повышенное внимание. Например, много усилий было потрачено на исследования, связанные с проверкой актуальности кэша, поскольку с этим связана значительная доля Web-трафика. Другое направление связано со снижением затрат на поддержание соединений, а также исследования, посвященные технологиям упреждающей выборки с целью сокращения времени ожидания. Эта глава делится на следующие три части:

- **Проверка актуальности и аннулирование элементов кэша.** При отсутствии явного указания времени истечения срока хранения проверка актуальности кэша необходима, чтобы гарантировать доставку клиентам актуальных ответов. Проверка актуальности ресурса на исходном сервере гарантирует, что кэшированная копия ресурса совпадает с версией, возвращаемой исходным сервером. Аннулирование элементов кэша осуществляется после уведомления, что кэшированный ресурс больше не является актуальным. Действия при проверке актуальности и при аннулировании сопряжены с определенными накладными расходами. Мы рассмотрим новую *технологию совмещения (piggybacking)*, сокращающую затраты на повторную проверку актуальности, в то же время снижающую время ожидания на стороне пользователя. Мы также познакомимся с технологиями аннулирования элементов на сервере, используя которые серверы могут предоставлять сведения об изменении своих ресурсов.

- **Комплексный информационный обмен.** Идея комплексного информационного обмена основана на использовании информации, доступной различным компонентам, участвующим в передаче данных: клиентам, прокси-серверам и исходным серверам. Основу составляет технология совмещения с добавлением рекомендаций от прокси-серверов, помогающих отслеживать и обрабатывать рекомендации, полученные от исходных серверов. Идея состоит в том, чтобы формировать сигнатуру стратегий кэширования в качестве *фильтра* и отправлять ее исходному серверу, который может локально применять фильтр к набору рекомендаций.
- **Упреждающая выборка.** Упреждающая выборка — это технология, предусматривающая извлечение информации до того, как она будет необходима или затребована. Главная побудительная причина — сократить время ожидания на стороне пользователя. В последовательности действий при типовой Web-транзакции, упреждающая выборка может быть осуществлена на уровне DNS (преобразование доменных имен хостов, с которыми наиболее вероятно будет установлена связь), на уровне TCP (предварительное установление соединения) и на уровне HTTP (упреждающая выборка ресурса, который вероятно будет запрошен).

Каждая область исследований будет детально рассмотрена в контексте мотивации, новизны и практической реализации идей.

13.1. Проверка актуальности и аннулирование элементов кэша

Ключевые изменения, принятые в HTTP/1.1, призваны гарантировать актуальность кэшированных ответов. Проверка актуальности кэша подразумевает выполнение проверки, возвратит ли исходный сервер тот же ответ, который соответствует кэшированному содержимому. Уведомление об аннулировании может быть отправлено кэшу в качестве указания, что кэшированный ответ больше не является актуальным. В главе 7 (раздел 7.3) и в главе 11 мы обсуждали различные проблемы, связанные с кэшированием в Web, и ввели несколько относящихся к кэшированию терминов, таких как время *истечения срока хранения*, длительность *актуального состояния* и *возраст*.

Кэш может возвращать кэшированный ответ, являющийся устаревшим. Например, кэш может быть отключен от исходных серверов. Большинство кэшей пытаются придерживаться принципа семантической прозрачности и используют различные технологии для обеспечения того, что кэшированный ответ по-прежнему является актуальным, прежде чем вернуть его в качестве ответа обратившемуся с запросом клиенту. Наиболее известным способом является проверка актуальности на исходном сервере. И в HTTP/1.0, и в HTTP/1.1 проверка актуальности выполняется с помощью запроса **GET** с модификатором запроса **If-Modified-Since**. В HTTP/1.1 имеется модификатор запроса **If-None-Match** для использования его с атрибутами содержимого. При получении такого запроса исходный сервер либо возвращает ответ **304 Not Modified**, не содержащий тела, либо ответ **200 OK** с полноценным телом содержимого.

Тот факт, что проверка актуальности выполняется на регулярной основе, нашел отражение во многих исследованиях. Анализ журналов ряда Web-серверов показывает, что от 10 до 30 процентов трафика составляют запросы на проверку актуаль-

ности. Например, исследование характеристик рабочей нагрузки [AFJ99] прокси-сервера показывает, что около 16% всех ответов составили ответы **304 Not Modified**. Можно считать число ответов **304 Not Modified** в журнале сервера являются *нижней оценкой* числа запросов на проверку актуальности. Некоторые запросы на проверку актуальности порождают ответы **200 OK**, которые содержат тело содержимого. Имеются следующие три причины, по которым запрос на проверку актуальности приводит к выдаче ответа **200 OK**:

- Ресурс был изменен, и сервер посылает новое значение.
- Сервер не знает точное время последней модификации. Например, ресурсом является сценарий, результат выполнения которого может время от времени изменяться, но сервер об этом не осведомлен. Клиенту может быть возвращен заголовок **ETag** вместо времени последней модификации.
- Выдача ответа **304 Not Modified** не является обязательным требованием для серверов (согласно спецификации протокола, оно относится к уровню требований SHOULD (Желательно)).

Далее в этом разделе мы остановимся на составляющих затрат, связанных с проверкой актуальности (и аннулированием) и рассмотрим три метода, позволяющие уменьшить время ожидания на стороне пользователя без увеличения затрат. К этим трем методам относятся: упреждающая проверка актуальности, совмещение и аннулирование под управлением сервера. Упреждающая проверка актуальности состоит в выполнении проверки кэшированных объектов, срок хранения которых мог истечь *до того*, как они будут запрошены клиентами. Упреждающая проверка актуальности может показать, что кэшированный ответ устарел (в этом случае он удаляется из кэша), или же что он является актуальным (в этом случае интервал между проверками может быть увеличен). Совмещение — это технология, впервые предложенная в [KW97], призванная уменьшить затраты на проверку актуальности. Аннулирование под управлением сервера — это способ для исходного сервера уведомить кэш, что кэшированный ответ больше не является актуальным. Проверка актуальности увеличивает трафик, в то время как аннулирование сервером может потребовать установления дополнительных соединений на стороне сервера. Совмещение позволяет избежать выдачи запросов и дополнительного трафика ответов.

13.1.1. Затраты, связанные с проверкой актуальности

Рассмотрим различные составляющие затрат в следующем сценарии:

- Клиент связывается с прокси-сервером, способным посылать запросы напрямую исходным серверам.
- Клиент отправляет запрос. Прокси-сервер, вместо того чтобы возвратить ответ из кэша, посылает исходному серверу запрос на проверку актуальности.
- На запрос возвращается ответ **304 Not Modified**, и прокси-сервер посылает кэшированный ответ клиенту.

Средний размер HTTP-запроса с модификатором запроса **If-Modified-Since** составляет около 200 байтов. HTTP-ответ с кодом ответа **304 Not Modified** имеет размер менее 200 байт и может составлять всего 50 байтов. Запрос на повторную проверку актуальности в HTTP/1.0 требует установления соединения с исходным сервером (или прокси-сервером, в случае его наличия на пути). Несколько пакетов

требуется для установления (трехэтапное TCP-взаимодействие) и завершения (четыре TCP-пакета завершения соединения) HTTP-взаимодействия. Запрос на проверку актуальности с точки зрения исходного сервера является обычным HTTP-запросом; сервер должен создать соединение, выполнить синтаксический анализ запроса, отправить обратно ответ, зарегистрировать обмен (необязательно) и закрыть соединение. Другими словами, затраты для исходного сервера аналогичны тем, которые имеют место при обработке любых других запросов, за исключением дополнительной нагрузки, связанной с выборкой ресурса с диска и записи тела ответа.

Некоторые затраты могут быть минимальны, если HTTP-запрос является частью долговременного соединения. Затраты на установление и закрытие соединения разносятся при этом на множество других запросов и ответов. Однако если мы рассмотрим время ожидания ответа клиентом, то обнаружим лишь небольшое уменьшение. Время ожидания при передаче всего содержимого ресурса с исходного сервера отправителю запроса уменьшается, поскольку возвращается только заголовок HTTP-ответа без тела содержимого. Это, скорее всего, будет составлять лишь малую часть от общих затрат. Если ответ содержался в кэше прокси-сервера, исчезает необходимость в соединении между прокси-сервером и исходным сервером, точно так же, как и обмен запрос–ответ между прокси-сервером и исходным сервером. Соединение клиента с прокси-сервером может отличаться более высоким качеством, чем соединения прокси-сервера с различными исходными серверами, что уменьшает время ожидания при передаче ресурса от прокси-сервера клиенту. Сокращение трафика запросов на проверку актуальности может значительно уменьшить время ожидания на стороне пользователя.

13.1.2. Упреждающая проверка актуальности

Получение информации до того, как она реально будет использована, дает возможность избежать ожидания при выдаче запроса. Упреждающая выборка — достаточно известный прием, который нашел применение в различных областях вычислительной техники. Подобно тому, как ресурсы могут быть выбраны до их запроса, может быть осуществлена упреждающая выборка информации о ресурсах. *Упреждающая проверка актуальности* — это упреждающая выборка информации об актуальности.

Метод **HEAD** HTTP дает возможность осуществлять выборку метаданных ресурса без фактической передачи ресурса. Кэш может выдать запрос **HEAD** и проверить, является ли кэшированный ответ по-прежнему актуальным. Если метаданные, возвращенные в ответе на запрос **HEAD**, свидетельствуют о том, что кэшированный ответ устарел, кэш может отправить запрос **GET**. Однако запрос **HEAD** требует установление HTTP-соединения, т.е. сопровождается теми же затратами, которые имеют место для запроса **GET**, за исключением фактической передачи ресурса. Таким образом, в тех случаях, когда кэшированный ответ является устаревшим, потребуется два отдельных запроса, что приведет к удвоению затрат. Кэш может использовать эвристические методы для принятия решения, какой запрос, **HEAD** или **GET**, отправить. Предварительная проверка актуальности посредством запросов **HEAD** потенциально снижает требования к пропускной способности сети, однако при этом загружает исходный сервер большим количеством запросов. Заметим, что использование кэшем запроса **GET** с заголовком **If-Modified-Since** не является упреждающей проверкой актуальности. Сервер, получающий такой запрос, может отправить ответ типа **200 OK** с содержимым ресурса.

Конвейерная обработка запросов на упреждающую проверку актуальности представляет значительный интерес, поскольку при этом не возникает проблем с блокированием (см. главу 7, раздел 7.5.4), т.к. запросы **HEAD** обычно не требуют сколько-нибудь значительной работы исходного сервера. Кроме того, отсутствует ожидание завершения упреждающей проверки актуальности на стороне агента пользователя. Упреждающая проверка актуальности не будет оказывать какого-либо влияния на общее снижение времени ожидания при передаче ресурса пользователю, за исключением, быть может, увеличения времени ожидания в случаях, когда кэшированный ресурс действительно устарел. Если проверяемый на актуальность ресурс больше не запрашивается, запрос **HEAD** также бесполезно загружает исходный сервер.

13.1.3. Использование совмещения

Запросы **HEAD** могут также быть использованы для определения, не устарел ли ответ, не выполняя передачи самого ресурса. Однако основной составляющей времени ожидания являются время на установление соединений с исходным сервером. Частично это объясняется тем, что фактическое время, затрачиваемое на передачу ответа от исходного сервера прокси-серверу, является относительно небольшим, принимая во внимание, что большинство ответов имеет небольшой размер. Для очень больших или динамически генерируемых ответов время, затрачиваемое на генерирование ресурса, может превысить время передачи.

Один из способов избежать затрат на соединение — передать несколько запросов и ответов по одному соединению. Возможность поддержания долговременных соединений в HTTP/1.1 позволяет уменьшить потребность в новых соединениях. Однако сильно загруженный сервер может закрывать долговременные соединения — протокол это допускает. Если соединение с исходным сервером уже открыто, прокси-сервер может отправлять запросы **HEAD** для проверки актуальности без затрат на установление новых соединений. Однако в связи с тем, что в кэше имеется интервал времени по умолчанию, в течение которого предполагается, что кэшированный ответ по-прежнему актуален, нет смысла выполнять повторную проверку на актуальность до истечения этого интервала.

Рассмотрим технологию, впервые описанную в [KW97], в соответствии с которой исходным сервером отслеживается актуальность кэшированных ресурсов. Предположим, что запрос на ресурс инициирует соединение с одним из исходных серверов. Соединение с этим исходным сервером будет установлено в любом случае, поэтому в нем можно осуществить проверку на актуальность множества ответов исходного сервера, срок хранения которых истек. Такая проверка актуальности выполняется путем *совмещения* этих запросов с запросом на ресурс путем помещения их в конец запроса. В качестве примера предположим, что прокси-сервер имеет десяток ответов исходного сервера **www.a.org**, четыре из них находятся в кэше, время хранения для них истекло. При получении запроса на ресурс **www.a.org/other.html** прокси-сервер формирует запрос **GET** в интересах клиента. Прокси-сервер добавляет информацию о четырех ресурсах, время хранения которых истекло, к запросу на **www.a.org/other.html**. Добавленная информация состоит из имени ресурса, времени его последней модификации и, возможно, его размера. Исходный сервер отвечает на запрос на ресурс **www.a.org/other.html** и может добавить метаданные о четырех других ресурсах.

Исходный сервер, который отправляет информацию о других проверяемых на актуальность ресурсах, выполняет меньше работы, поскольку полноценный обмен

запрос/ответ при этом не осуществляется. Предположим, что прокси-сервер имеет список ресурсов для проверки на актуальность вместе со временем их последней модификации. Исходный сервер отправляет обратно список URL этих ответов, больше не являющихся актуальными, вместе с ответом для ресурса **www.a.org/other.html**. Прокси-сервер возвращает ответ для **other.html** обратившемуся с запросом клиенту после выделения информации о проверке актуальности. Прокси-сервер может отдельно обработать эту информацию, увеличив время хранения для актуальных ответов и удалив устаревшие ответы из кэша. Сутью проблемы, как мы увидим далее в этом разделе, является механизм для определения, отправлять ли добавочную информацию как часть ответа для **other.html**, или же в отдельном сообщении.

На рисунках с 13.1 по 13.4 показаны различные этапы процесса совмещения. Клиент взаимодействует через прокси-сервер с тремя исходными серверами: A, B и C. Прокси-сервер имеет в своем кэше два ресурса (**r2**, **r3**), полученные от исходного сервера B. Теперь предположим, что клиент посылает запрос исходному серверу на ресурс **r1** (рис. 13.1).

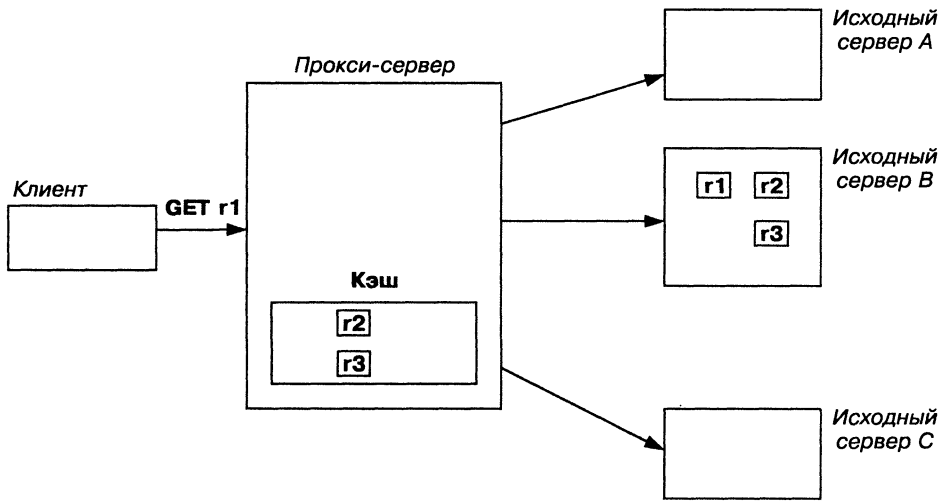


Рис. 13.1. Клиент запрашивает ресурс **r1** у прокси-сервера

В этот момент прокси-сервер не имеет в своем кэше ресурса **r1** и должен передать запрос исходному серверу B, на котором размещен ресурс **r1**. Предположим, что в соответствии со стратегией прокси-сервера должна быть осуществлена проверка актуальности имеющихся в кэше ресурсов **r2** и **r3**. Прокси-сервер совмещает запрос на проверку актуальности этих двух ресурсов с запросом на **r1**, посылаемым исходному серверу B, как показано на рис. 13.2.

Исходный сервер B возвращает ресурс **r1** и информацию об актуальности ресурсов **r2** и **r3**, как показано на рис. 13.3.

Прокси-сервер возвращает ресурс **r1** клиенту и обновляет содержимое своего кэша на основе информации об актуальности ресурсов **r2** и **r3**. Как показано на рис. 13.4, прокси-сервер удалил ресурс **r3** из кэша, поскольку он признан устаревшим на основе информации об актуальности, возвращенной исходным сервером B.

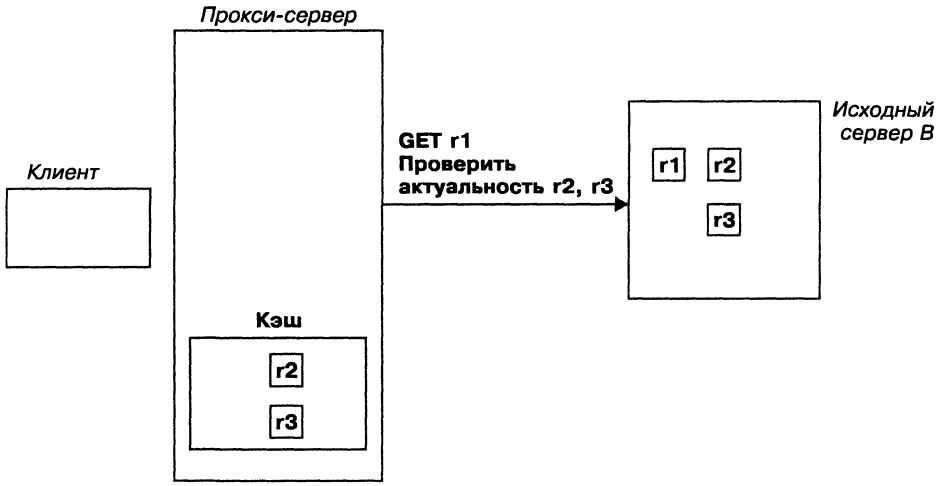


Рис. 13.2. Прокси-сервер запрашивает **r1** и совмещает с запросом проверку актуальности ресурсов **r2**, **r3**

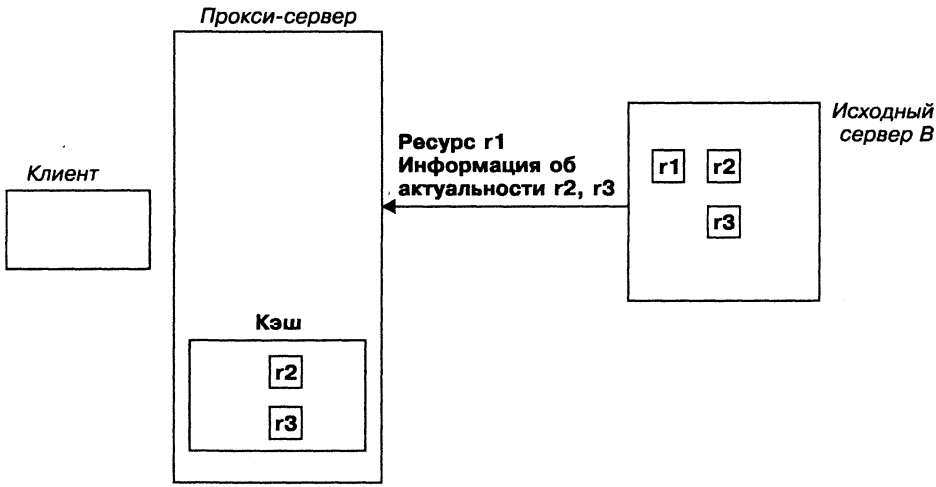


Рис. 13.3. Исходный сервер B возвращает ресурс **r1** и информацию об актуальности ресурсов **r1** и **r2**

Преимущества совмещения очевидны: не устанавливаются специальные соединения для проверки актуальности. Проверяется актуальность только предположительно устаревших ресурсов. Проверяя актуальность кэшированных ресурсов, прокси-сервер увеличивает вероятность избежать повторной проверки актуальности в будущем и сокращает время ожидания на стороне пользователя для этих запросов.

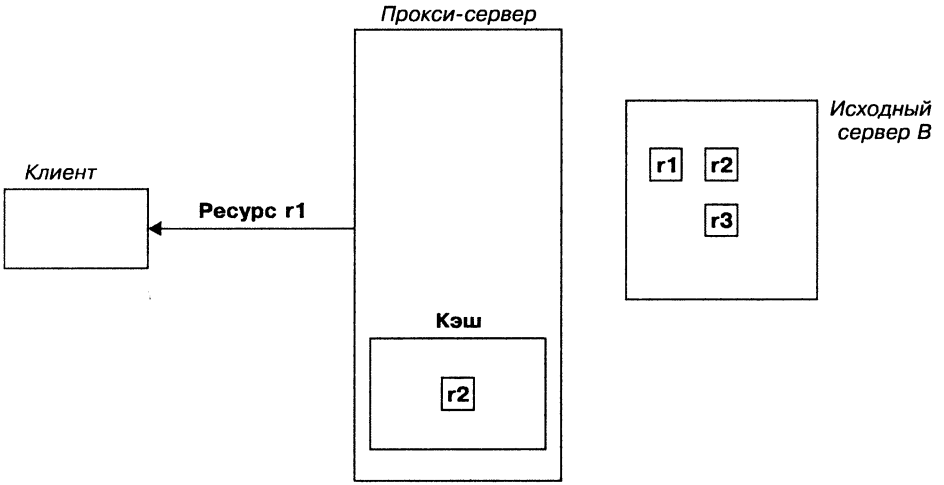


Рис. 13.4. Прокси-сервер возвращает ресурс **r1** и обновляет кэш

РЕАЛИЗАЦИЯ СОВМЕЩЕНИЯ

Теперь мы рассмотрим способы, с помощью которых прокси-серверы могут посылать запросы на проверку актуальности, а также выясним, как исходные серверы могут отвечать на такие запросы. Применительно к прокси-серверу, это просто проверка актуальности группы ответов в его кэше, для которых истекло время хранения. Имеются следующие два способа реализации упреждающей проверки актуальности в составе запроса:

1. Прокси-сервер может отправить группу запросов **HEAD** исходному серверу. Заголовок **HEAD** содержит модификатор запроса **If-Modified-Since**, значение которого берется из заголовка **Last-Modified** кэшированного ответа. Очевидно, что если ответ был сформирован динамически, либо не содержит информации **Last-Modified** по другим причинам, кэш не будет включать такой ответ в список для упреждающей проверки актуальности. В действительности во многих случаях динамически генерируемые ответы вообще не кэшируются. Если HTTP-соединение уже открыто, дополнительных затрат на соединение не возникает. Запросы **HEAD** могут с успехом подвергаться конвейерной обработке. Подобная технология работает и в HTTP/1.0, но в HTTP/1.1 она реализована лучше. Однако для каждого дополнительного запроса сервер должен возвратить ответ с информацией о проверке актуальности, даже если ресурс не был изменен.
2. Прокси-сервер формирует простую структуру данных, содержащую URL устаревших ответов и метки времени, соответствующие заголовку **Last-Modified** в кэшированном ответе. Эта структура данных добавляется в регулярный запрос, как было сказано выше. Прокси-сервер предполагает, что исходному серверу известно, как извлечь данные, т.е. должны быть использованы серверы, которые знают, как реагировать на подобные запросы с совмещением. С другой стороны, если HTTP будет модернизирован с учетом возможностей по обработке совмещений, прокси-серверы смогут без проблем работать с совместимыми с данной версией HTTP-серверами.

При получении запроса на проверку актуальности исходный сервер может действовать одним из следующих двух способов:

1. При первом способе проверки актуальности, когда посылаются отдельные запросы **HEAD**, исходный сервер просто возвращает соответствующие ответы. Никакого изменения поведения от него не требуется.
2. При втором способе, при котором запросы на проверку актуальности совмещаются с регулярным запросом, исходный сервер должен быть способен извлекать добавленные в запрос структуры данных. Исходный сервер может проверять на актуальность каждую из записей в этой структуре и добавлять результат к регулярному ответу. Более интеллектуальный исходный сервер может просто отправлять назад список неактуальных ресурсов, подразумевая, что остальные ресурсы по-прежнему являются актуальными. Если исходный сервер поддерживает HTTP/1.1, а прокси-сервер способен обрабатывать ответы HTTP/1.1, исходный сервер может воспользоваться заголовком **Trailer** HTTP/1.1 и совместить дополнительную информацию с ответом для **other.html**.

В качестве примера использования второго способа рассмотрим следующий обмен с совмещением.

Прокси-сервер включает заголовок запроса **TE:Trailer**, а сервер отправляет дополнительную информацию в трейлере ответа. Прокси-сервер посылает запрос:

```
Get /other.html HTTP/1.1
Host: www.piggy.com
TE: trailers

URL1 LMT: ...
URL2 LMT: ...
URL3 LMT: ...
URL4 LMT: ...
CRLF
```

Исходный сервер отправляет обратно ответ

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 2100
Transfer-Encoding: chunked
Trailer: Piggy
...
<тело ответа>
...
Piggy: <здесь располагается дополнительно отправляемая информация>
```

В разделе 13.2.3 мы разъясним синтаксические элементы протокола, использованные в этом примере. Технически прокси-серверу не обязательно включать заголовок **TE**, а серверу использовать трейлер для включения дополнительной информации. Сервер может просто включить запрашиваемую информацию об актуальности в заголовок, предшествующий телу содержимого. При этом подразумевается, что ответ не будет отправлен, пока не будет выполнена требуемая проверка актуальности, а результаты ее не будут помещены в соответствующий заголовок ответа. При включении дополнительной информации в трейлер прокси-сервер может немедленно переслать тело ответа и избежать, тем самым, увеличения времени ожидания на стороне пользователя. Прокси-сервер может обработать содержимое трейлера после пересылки тела сообщения пользователю.

13.1.4. Аннулирование, управляемое сервером

У кэша имеется информация, какие ресурсы требуется проверять на актуальность. С другой стороны, исходный сервер — это компонент, который знает, когда ресурс на сайте изменяется. Таким образом, вместо отправки запроса на проверку актуальности клиентом или прокси-сервером, исходный сервер может уведомлять заинтересованные стороны об изменениях ресурсов. Подобное уведомление называется *аннулированием, управляемым сервером*. Аннулирование сходно с технологией *проталкивания (pull technology)*, в то время как проверка актуальности аналогична модели *принудительной загрузки информации (push technology)*.

Механизм аннулирования, управляемый сервером, требует ответа на ряд вопросов:

- Как сервер посылает аннулирующую информацию?
- Каким клиентам и прокси-серверам адресована аннулирующая информация?
- С какой частотой посылается аннулирующая информация?

Ответ на первый вопрос очевиден, так как в HTTP исходный сервер не может инициировать передачу сообщений. Прежде чем сервер сможет ответить, должен поступить запрос. Таким образом, аннулирование может быть послано только в ответ на некий запрос. В исследовании, описанном в [KW98], в основу которого легло развитие технологии проверки актуальности содержимого кэша с использованием совмещения (см. раздел 13.1.3), был изучено совмещение с ответами информации об аннулировании. Как и в случае совмещения с запросом информации о проверке актуальности, это позволяет избежать установления новых соединений.

Второй вопрос связан с сохранением промежуточного состояния на сервере, который посылает аннулирующую информацию. Сильно загруженный исходный сервер может получать запросы от тысяч прокси-серверов и клиентов. Чтобы хранить сведения о них для передачи информации об аннулировании, исходному серверу потребуется хранить значительный объем информации о состоянии. В некоторых случаях исходный сервер должен *знать*, какие прокси-серверы заинтересованы в получении аннулирующей информации. Сервер может использовать подход, сходный с основанным на фиксированном времени хранения, который был описан в разделе 11.7, но при этом по-прежнему требуется сохранение информации о состоянии. Прокси-серверы могут быть не заинтересованы в получении информации об аннулировании для определенных ресурсов, но получать такую информацию. Бесполезная информация об аннулировании ресурсов перегружает исходные серверы и прокси-серверы, которые должны обрабатывать дополнительную информацию, совмещенную с ответами.

Третий вопрос, связанный с частотой отправки информации об аннулировании, требует анализа частоты изменений ресурса. Исходный сервер может быть сильно заинтересован в том, чтобы прокси-серверы не хранили устаревшие данные. Несмотря на риск получить большее количество адресованных им запросов, исходные серверы могут требовать актуальности ответов. Если исходный сервер уверен, что прокси-сервер четко следует указаниям по аннулированию, он может установить большее время истечения сроков хранения для ответов и посылать аннулирования, как только ресурс изменяется. Для часто меняющихся ресурсов это может существенно увеличить объем трафика.

Подход, основанный на фиксированном времени хранения и описанный в [LC97], отличается от подхода с совмещенной отправкой аннулирований. При его использовании требуется устанавливать HTTP-соединения между сервером и клиентом/прокси-сервером, если интересующий ресурс изменяется в течение периода

фиксированного времени хранения. При подходе с совмещением аннулирования посылаются с последующими запросами от клиента к прокси-серверу.

Один из уроков, извлеченных из идеи аннулирования сервером, состоит в том, что не следует хранить информацию об аннулировании на уровне отдельных ресурсов из-за большого объема данных о состоянии и увеличения трафика. Другой урок состоит в том, что исходные серверы и прокси-серверы не обладают полной информацией. Исходный сервер знает, когда ресурс изменяется, но не имеет представления, какие клиенты заинтересованы в ресурсе. Даже если исходный сервер ведет подробный журнал в течение длительного периода времени, он не знает об ответах, которые были возвращены из кэша, и, таким образом, никогда не узнает о реальной популярности ресурса у клиентов. Прокси-серверы, с другой стороны, осведомлены об интересах клиентов к ресурсам, но не имеют понятия, когда и как часто изменяются ресурсы. Подробность информации и необходимость двунаправленного информационного обмена подводит нас к следующей теме: комплексному обмену информацией.

13.2. Комплексный обмен информацией

В предыдущем разделе обсуждались методы, предоставляющие прокси-серверу информацию об актуальности кэшированных ответов до того, как поступили клиентские запросы. В этом разделе мы обобщенно дадим вам представление об информационном обмене между прокси-серверами и исходными серверами, обеспечивающем поддержку функций прокси-серверов, включая согласованность кэша, замещение ресурсов в кэше и упреждающую выборку. Эффективность работы прокси-сервера может быть повышена за счет использования информации о ресурсах, которые с большой вероятностью будут затребованы в будущем. Однако исходный сервер имеет наиболее полное представление о частоте доступа к запрашиваемым ресурсам. В соответствии с рассмотренным в предыдущем разделе подходом, сервер может совместить *рекомендации* относительно этих ресурсов с ответами на запросы от прокси-серверов. Сервер формирует эти рекомендации путем группировки ресурсов, которые наиболее вероятно будут затребованы, в *тома*.

Однако некоторые ресурсы в томе могут не иметь отношения к прокси-серверу. Прокси-сервер один знает, какую стратегию кэширования он проводит, а также какие ресурсы в настоящий момент имеются в его кэше. Эта информация сводится в *фильтр*, который применяется на сервере для составления рекомендаций обратившемуся с запросом прокси-серверу. Прокси-сервер может совместить фильтр с сообщением-запросом. После этого сервер применяет фильтр к тому, ассоциированному с запрашиваемым ресурсом, чтобы сформировать список рекомендаций, которые совмещаются с сообщением-ответом. Двунаправленный обмен информацией дает возможность сократить число рекомендаций, получаемых прокси-сервером. В этом разделе мы познакомимся с исследованием, в котором предлагается совместно использовать серверные тома и фильтрацию. В нем также содержится детальная информация, связанная с реализацией совмещенной доставки фильтров и рекомендаций, а также приведены алгоритмы для построения томов [CKR98, CKR99]. Будет также представлено сравнение алгоритмов построения томов на основе информации, полученной из журналов серверов.

13.2.1. Серверные тома

В предложенной схеме рекомендации совмещаются с сообщениями-ответами в зависимости от ресурса, запрошенного прокси-сервером. В ответ на запрос ресурса *r* сервер возвращает рекомендации относительно ресурсов, связанных с *r*. В предыдущих исследованиях, посвященных файловым системам, было введено понятие *тома* как группы взаимосвязанных ресурсов. В работе [KW98], посвященной совмещенной доставке сведений об актуальности элементов кэша и развившей тематику работы [KW97], понятие тома было применено к аннулированиям, управляемым Web-сервером. Построение томов требует от сервера использования эвристических процедур для группировки связанных ресурсов, к которым относятся:

- Ресурсы, доступ к которым осуществляется близко по времени (временной шаблон доступа).
- Ресурсы, имеющие общий синтаксический префикс (применительно к UNIX — это ресурсы, расположенные в одном каталоге).
- Ресурсы, имеющие общие атрибуты (схожий тип содержания, размеры и т.д.).
- Ресурсы, которые модифицируются вместе (например, ресурсы, которые изменились в течение последних пяти минут).

Сервер может ассоциировать идентификатор с каждым томом, а прокси-сервер может отслеживать группу кэшированных ресурсов по их идентификаторам томов. К примеру, предположим, что прокси-сервер знает, какие ресурсы принадлежат тому V1 на сервере. При отправке рекомендаций относительно тома V1 сервер может передать лишь небольшой объем информации о том, как изменились ресурсы в V1. При получении рекомендаций, относящихся к V1, прокси-сервер может выполнить групповые операции над кэшированными ресурсами, принадлежащими V1.

Выбор соответствующей эвристики при построении тома зависит от конкретного применения. Первая эвристика, основанная на временном шаблоне доступа (последовательные обращения осуществляется близко по времени), хорошо подходит для таких приложений, как упреждающая выборка или замещение ресурсов в кэше, которые зависят от точного предсказания будущих клиентских запросов. На основе имевшихся в прошлом обращений сервер может идентифицировать, какие ресурсы запрашивались сразу же после обращения к ресурсу *r*. Если много клиентов запросили ресурс *s* через короткое время после запроса ресурса *r*, возможно, что клиенты следуют одному шаблону поведения. Наиболее очевидный случай, когда *r* представляет собой HTML-файл, а *s* — одно из встроенных в него изображений. Включая *s* в том для *r*, сервер может совместить информацию о *s* с ответом прокси-серверу на запрос для *r*. Пересылая ответ для *r* обратившемуся с запросом клиенту, прокси-сервер может осуществить упреждающую выборку встроенного изображения или обновить метаданные о своей кэшированной копии *s*. В других случаях ресурс *s* может представлять собой чрезвычайно популярную гипертекстовую ссылку в ресурсе *r*, или ресурс, отстоящий на «пару щелчков мышью» от *r*. Подробнее этот подход к построению томов мы рассмотрим в разделе 13.2.4.

Второй способ построения томов основан на предположении, что содержание Web-сайта организовано в виде каталогов в файловой системе сервера. Связанные ресурсы могут принадлежать одному каталогу, что дает возможность серверу группировать ресурсы, просматривая имена файлов в каталогах. Рассмотрим Web-сайт, имеющий три подкаталога *dotty*, *lefty* и *neato*. Каждый из этих каталогов может содержать несколько файлов. Одноуровневый том может содержать URL в виде **/dotty/index.html**, **/dotty/src/graph.c** и **/dotty/src/Makefile**. Однако информация

о ресурсах **/lefty/editor.c** и **/neato/download.html** не будет включена в одноуровневый том. Каталоги с большим количеством ресурсов приведут к созданию очень больших томов. Чтобы уменьшить размер тома, сервер может сгруппировать ресурсы, которые относятся к одному уровню каталогов. Например, ресурсам в каталогах, начинающихся с **/dotty/src/**, может быть назначен один том.

Третий способ построения тома основан на интуитивном представлении, что ресурсы, имеющие сходные физические атрибуты, могут быть сгруппированы вместе. Прокси-сервер может применять стратегию, которая ограничивает кэширование ресурсов определенного размера или с определенным типом содержания. Например, прокси-сервер может принять решение никогда не кэшировать ресурсы, подверженные частым изменениям. Подобный метод имеет тот недостаток, что решения принимаются, только исходя из информации исходного сервера. Если кэширующий прокси-сервер каким-то образом уведомил исходный сервер, что он не кэширует ресурсы с определенным типом содержания, исходный сервер может использовать эту информацию для формирования рекомендации, вместо того, чтобы *априори* делать предположения о стратегии кэширования прокси-сервера. Подобно типу содержания, ограниченный размер кэша прокси-сервера может привести к решению, никогда не кэшировать ресурсы, превышающие определенный размер. Формируя тома на основе размеров, исходные серверы могут решать, включать или не включать в рекомендации определенные тома.

В четвертом методе построения томов отбираются ресурсы, модифицированные в течение определенного интервала времени. В основу этой эвристики положен тот факт, что прокси-серверам нужна информация о том, какие ресурсы были изменены за определенный период времени. Группировка ресурсов, которые изменились в течение этого периода, помогает серверу ограничить число рекомендаций. Кроме того, ресурсы, изменяющиеся в одно время, могут быть связанными друг с другом. Запрос на один из этих ресурсов может сопровождаться запросом на другие ресурсы в том же томе. Назначая эти ресурсы одному и тому же тому, сервер может обеспечить прокси-серверы рекомендациями о ресурсах, не дожидаясь запросов на них.

Формирование томов увеличивает нагрузку сервера. Расчет тома в ходе обработки каждого запроса может вызвать задержку в передаче сообщения-ответа и перегрузить сервер. На практике тома могут быть рассчитаны заранее на основе предыдущих обращений к серверу. Например, отдельная программа, выполняющаяся на другом компьютере, может формировать тома на основе информации из журнала сервера. Тома, рассчитанные в один период времени, могут быть применены сервером для следующего периода. Есть и альтернативный метод: сервер может строить и обновлять тома в фоновом режиме. Тем самым сервер способен предоставить прокси-серверам актуальные рекомендации, но ценой увеличения нагрузки. Выбор компромиссного варианта определяется объемом вычислений при построении томов и частотой изменений томов.

13.2.2. Фильтры прокси-серверов

Построение томов осуществляется на основе информации, имеющейся на сервере. Стандартный прокси-сервер может быть не заинтересован во *всех* ресурсах тома. Например, возьмем прокси-сервер, который обслуживает группу пользователей, с беспроводным подключением к Internet. Такой прокси-сервер может принять решение не кэшировать изображения или какие-либо другие ресурсы, размер которых превышает 1 мегабайт. Отправка рекомендаций относительно этих ресурсов приведет к ненужной загрузке Web-сервера и прокси-сервера. Однако исход-

ный сервер не в состоянии знать о стратегиях кэширования, используемых множеством прокси-серверов, посылающих запросы Web-серверу. Прокси-сервер может включить информацию о собственной стратегии кэширования в свои сообщения-запросы. Прокси-сервер отправляет фильтр, а сервер применяет этот фильтр для отбора соответствующих рекомендаций для тома. Связывание фильтра с сообщением-запросом избавляет сервер от необходимости хранить фильтры для большого набора прокси-серверов между последовательными запросами.

Фильтр прокси-сервера служит нескольким целям. Во-первых, фильтр может быть использован для ограничения числа рекомендаций и выдачи их только для тех ресурсов, которые представляют интерес, например, ресурсов с определенным типом содержания или с определенным размером. Во-вторых, фильтр может указывать, какие метаданные должны быть включены в каждую из рекомендаций. Например, информация о времени последней модификации дает возможность прокси-серверу избавиться от устаревших ресурсов в кэше и подтвердить актуальность ресурсов, которые не были изменены. С другой стороны, прокси-серверу, осуществляющему упреждающую выборку ресурсов, требуется информация о размерах и степени популярности ресурсов. В-третьих, фильтр может быть использован для управления издержками путем ограничения числа рекомендаций. Фильтр может использовать бит «включить-выключить» для разрешения или запрещения рекомендаций, давая тем самым прокси-серверу возможность воздействовать на включение сервером рекомендаций в сообщение-ответ. Прокси-сервер может отслеживать частоту выдачи рекомендаций сервером и решать, выполнить ли рекомендацию для определенного запроса. Кроме того, фильтр прокси-сервера может ограничивать число рекомендаций, совмещаемых с определенным сообщением-ответом. Например, прокси-сервер с помощью фильтра может потребовать, чтобы сервер посылал только первые десять рекомендаций, если рекомендации упорядочены в порядке их важности.

Поскольку сообщения-запросы могут содержать фильтры, исходному серверу не нужно хранить информацию о состоянии для тысяч прокси-серверов, посылающих запросы. Прокси-сервер, с другой стороны, может использовать один и тот же фильтр для множества исходных серверов. Следовательно, хранение фильтра прокси-сервером и передача его по мере необходимости является более эффективным путем, чем возложение этой обязанности на исходные серверы. Прокси-сервер и исходный сервер должны согласовывать форматы фильтров, чтобы одинаковым образом их интерпретировать. Кроме того, применение фильтров воздействует на структуры данных сервера, относящиеся к рекомендациям. Например, если тома допускают фильтрацию по размерам ресурсов, сервер может разделить ресурсы в томе по их размеру. Имея отдельный список для каждого диапазона размеров ресурсов, сервер может быстро применить фильтр, который ограничивает выдачу рекомендаций ресурсами, имеющими размер, меньший заданного. Такая эффективная структура данных для хранения томов облегчает применение фильтров при обработке запросов от прокси-серверов.

13.2.3. Тома и фильтры. Практическое использование

Практическое использование серверных томов и фильтров прокси-серверов требует более строгой спецификации томов, фильтров и технологии совмещения. Том состоит из уникального идентификатора и списка элементов, каждый из которых соответствует определенному ресурсу на Web-сайте. Каждый элемент содержит URL и набор атрибутов, таких как время последней модификации, размер и тип

содержания ресурса. Сервер может хранить элементы тома в одном или нескольких списках типа «первый вошел, первый обработан», разбитых на основе ключевых атрибутов ресурса. Например, сервер может иметь отдельные списки в зависимости от размеров ресурсов или типов содержания, чтобы уменьшить издержки при применении фильтров прокси-серверов. Помимо элементов тома, сервер может отправлять прокси-серверу идентификатор тома. Обладая статистикой относительно последних доставленных методом совмещения томов (RPV — Recently Piggybacked Volumes), прокси-сервер может проинструктировать сервер не передавать эти рекомендации повторно, включив идентификаторы тома в фильтр, совмещенный с последующими запросами. При получении этого фильтра сервер пропустит рекомендации, связанные с данными идентификаторами томов.

Совмещение прокси-сервером фильтра с сообщением-запросом избавляет сервер от необходимости хранить информацию о каждом прокси-сервере. В противном случае система не смогла бы работать с большим количеством прокси-серверов. Фильтр прокси-сервера состоит из различных полей, представленных в таблице 13.1. Список может быть расширен, чтобы учесть другие параметры фильтрации. Любые дополнительные параметры фильтрации должны следовать двум основным принципам. Во-первых, параметр должен быть полезен для настройки рекомендаций, посылаемых обратившемуся с запросом прокси-серверу. Во-вторых, сервер должен иметь возможность эффективно применять фильтр. В таблице 13.1 выделены две группы элементов фильтра: управляющие параметры и атрибуты ресурсов. Управляющие параметры дают возможность прокси-серверу указывать, как часто серверу следует посылать рекомендации, сколько рекомендаций посылать вместе с сообщением-ответом. Атрибуты ресурсов указывают, какие виды ресурсов должны включаться в рекомендации. Например, прокси-сервер может захотеть, либо не захотеть получать рекомендации относительно ресурсов определенного размера, с определенным типом содержания или временем последней модификации. Кроме того, прокси-сервер может захотеть получать рекомендации на основе наличия или отсутствия определенного HTTP-заголовка, например, **Accept-Language**.

Таблица 13.1. Элементы фильтров

Элемент фильтра	**Примеры значений**	**Назначение**
Hints	On/Off	Посылать ли рекомендацию?
RPV	{1, 2, 4}	Сервер не должен отправлять перечисленные тома
MaxPiggy	5, 10	Максимальное число рекомендаций в сообщении
MinAccess	100	Минимальное число обращений к ресурсу
MinProb	0.8	Минимальная вероятность следующего обращения
Level	0, 1, 2	Рекомендации, совместно использующие префиксы с указанным уровнем
Size	1 MB, 10 KB	Ограничение на размер элементов тома
OKType	HTML	Ограничение выдачи рекомендаций указанными типами содержания
NotOKType	CGI	Не посылать рекомендации относительно данного типа содержания

Элемент фильтра	Примеры значений	Назначение
Lastmod	1 min	Минимальное время, прошедшее после последней модификации
HTTP method	PUT	Ресурсы должны допускать применение указанного метода
HTTP header	ETag, Accept	Ресурс должен содержать данный заголовок

Обмен фильтрами и рекомендациями требует эффективного способа совмещения информации с сообщениями запросов и ответов HTTP. Фильтры и рекомендации могут быть включены в виде дополнительных строк в заголовки ответов и запросов HTTP. Однако такой подход требует, чтобы сервер генерировал рекомендации до передачи тела сообщения-ответа. Любая задержка в применении фильтра прокси-сервера вносит задержку в обработку клиентского запроса. Вместо этого рекомендации могут быть отправлены в *конце* сообщения-ответа. Это может быть достигнуто за счет посылки метаданных в трейлере сообщения HTTP/1.1. Трейлеры могут включаться в сообщения, использующие разделение на фрагменты передаваемых данных и заголовок **Trailer**, о чем говорилось ранее в главе 7 (раздел 7.6). Сообщение-запрос должно указывать, что прокси-сервер может обработать запросы с трейлерами, а заголовок сообщения-ответа должен указывать, что сообщение заканчивается трейлером.

Например, прокси-сервер мог отправить серверу запрос:

```
GET /random.html HTTP/1.1
Host: smartserver.com
TE: trailers
Piggy-filter: maxpiggy=5; content-type=gif
```

Заголовок **Piggy-filter** содержит фильтр прокси-сервера, а заголовок **TE: trailers** указывает, что прокси-сервер может обрабатывать ответ, разбитый на фрагменты, и с трейлером. Фильтр в данном случае указывает, что прокси-сервер хочет получать максимум пять рекомендаций и заинтересован в рекомендациях относительно ресурсов, имеющих тип содержания GIF. Сервер, который не способен предоставить том с рекомендациями, будет игнорировать заголовок **Piggy-filter**. Сервер, участвующий в обмене с совмещением, может отправить следующее сообщение-ответ:

```
HTTP/1.1 200 OK
Trailer: P-volume
Transfer-encoding: chunked
 < Размер фрагмента >
      <данные>
   ...
 0
P-volume: vol=7; pe="/random-img1.gif,895527629,546";
    pe="/random-img3.gif,891527021,192";
    pe="/random-img21.gif,821993421,900"
CRLF
```

Заголовок ответа имеет заголовок **Trailer** со значением **P-volume**, указывающим, что заголовок с этим именем присутствует в конце сообщения. Ответ разбит на фрагменты, а трейлер располагается после фрагмента нулевой длины в конце сооб-

щения. Трейлер содержит идентификатор тома (7) и три элемента тома, включая имя, время последней модификации и размер каждого ресурса.

Механизм совмещения не требует, чтобы все прокси-серверы и исходные серверы распознавали новые заголовки. HTTP-компоненты игнорируют заголовки, которые они не воспринимают. Сервер, который не способен посылать рекомендации, будет игнорировать незнакомый заголовок **Piggy-filter** и генерировать обычный ответ. Не участвующий в данной схеме обмена прокси-сервер будет генерировать обычные запросы, которые не содержат заголовок **Piggy-filter**. Для прокси-серверов и серверов, участвующих в схеме обмена, механизм совмещения предусматривает добавление небольшого объема данных в сообщениях запросов и ответов. Например, рекомендация состоит из имени и нескольких атрибутов ресурсов. Средний URL состоит примерно из 50 однобайтных символов (не включая доменное имя сайта), а время последней модификации и атрибуты размера занимают по восемь байт. Следовательно, для каждой рекомендации потребуется 66 байтов. Отправка пяти рекомендаций добавит к сообщению-ответу около 300 байтов. Это относительно немного в сравнении со средним размером Web-ресурса — около 8 Кбайтов, о чем говорилось в главе 10 (раздел 10.4.2). Отправка нескольких рекомендаций самое большее потребует от сервера передать один дополнительный TCP-пакет. На деле рекомендации могут размещаться в том же пакете в конце тела сообщения, что избавляет от необходимости отправлять дополнительный пакет. Фильтры прокси-сервера имеют еще меньшие размеры, чем серверные рекомендации, и все сообщение-запрос может поместиться в одном пакете.

13.2.4. Алгоритмы построения томов

Выигрыш от совместного использования серверных томов и фильтров прокси-сервера определяется тем, насколько рекомендации, посылаемые сервером, будут полезны для прокси-сервера. Последнее, в свою очередь, зависит от алгоритма построения томов.

ПОКАЗАТЕЛИ ЭФФЕКТИВНОСТИ

Разработка и оценка алгоритмов построения томов основываются на достижении конкретных, подлежащих измерению характеристик эффективности. В идеале сервер должен отправлять небольшое число рекомендаций, чтобы каждая рекомендация была полезной, а каждый клиентский запрос предсказуем. При этом учитываются следующие показатели, влияющие на производительность:

- **Объем рекомендаций.** Среднее число рекомендаций, совмещенных с сообщением-ответом, является показателем дополнительной загрузки сервера, сети и прокси-сервера. Небольшой объем рекомендаций обусловливает незначительную дополнительную нагрузку ценой предоставления меньшей информации прокси-серверу.
- **Коэффициент попадания (recall).** Коэффициент попадания учитывает, какая часть клиентских запросов прокси-серверу была предсказана рекомендацией, полученной от сервера. Высокий коэффициент попадания означает, что большое количество запросов получили выигрыш от рекомендаций сервера, в то время как низкий коэффициент попадания свидетельствует, что большинство запросов не было предсказано заранее.
- **Точность.** Показатель точности оценивает долю рекомендаций сервера, которые успешно предсказали будущий запрос. Высокая точность означает, что

много рекомендаций оказались полезными, в то время как низкая точность свидетельствует, что сервер отправил много рекомендаций, которые не были полезными для прокси-сервера.

- **Доля обновлений.** Доля обновлений оценивает вероятность того, что клиентский запрос был предсказан заранее *и* обращен к ресурсу, который был затребован прокси-сервером в прошлом. Высокая доля обновлений означает, что рекомендация позволила прокси-серверу обновить кэшированную копию ресурса до запроса клиента.

Коэффициент попадания и точность широко применяются при оценке информации поисковыми системами, рассмотренными в главе 2 (раздел 2.7.3). В контексте построения томов коэффициент попадания и точность отражают противоречие между желанием предсказать большее число запросов заранее и необходимостью избежать некорректных предсказаний. Например, чтобы достичь высокого коэффициента попадания, серверу может потребоваться отправлять большее число рекомендаций. В свою очередь это ведет к снижению точности, поскольку многие из рекомендаций могут оказаться бесполезными.

Коэффициент попадания, точность и доля обновлений призваны оценить эффективность серверных рекомендаций. На практике полезность рекомендации может зависеть от времени, прошедшего между доставкой рекомендации и временем получения предсказанного запроса. Предсказание запроса за малое время до его поступления может не дать прокси-серверу достаточного запаса времени, чтобы среагировать на предсказание. Например, прокси-сервер может не иметь достаточно времени, чтобы осуществить упреждающую выборку ресурса с сервера до получения запроса на ресурс от клиента. Точно так же, рекомендация, доставленная задолго до запроса клиента, может оказаться не слишком полезной. Например, упреждающая выборка ресурса за час до поступления клиентского запроса может не принести большой выгоды. Заранее выбранный ресурс будет занимать место в кэше прокси-сервера. Кроме того, ресурс может быть модифицирован исходным сервером до поступления клиентского запроса. При поступлении клиентского запроса прокси-серверу придется повторно проверить актуальность кэшированного ответа и, возможно, получить новую копию ресурса с исходного сервера.

Подобные ограничения по времени могут быть учтены путем назначения фиксированного интервала времени для каждой рекомендации. Предположим, прокси-сервер получает рекомендацию относительно ресурса s. Если клиент запрашивает ресурс s в течение промежутка времени, не превышающего τ секунд, рекомендация не будет полезной. Если клиент запрашивает ресурс s по истечении T секунд после выдачи рекомендации, где $T > \tau$, рекомендация не будет полезной. Другими словами, рекомендация будет полезна только для запросов, поступивших на интервале времени между τ и T секунд после получения рекомендации прокси-сервером. Алгоритм построения томов формирует том v_r для каждого ресурса r, где v_r представляет собой набор ресурсов, посылаемых как рекомендации при запросе ресурса r. Применение этих томов к последовательности запросов приводит к различным значениям показателей размера рекомендации, коэффициента попадания, точности и доли обновлений. Достижение баланса между четырьмя показателями предполагает несколько подходов к построению томов. Алгоритм может пытаться добиться максимума или минимизировать один из показателей с учетом ограничений на другие показатели. Например:

- Увеличивать коэффициент попадания до достижения верхнего предела среднего объема рекомендации.

- Уменьшать объем рекомендации до достижения нижней границы коэффициента попадания.
- Увеличивать коэффициент попадания до достижения нижней границы точности.
- Увеличивать точность до достижения нижней границы коэффициента попадания.

В первом алгоритме процесс начинается с целевого размера рекомендации и попыток предсказать как можно больше запросов. Для второго алгоритма, наоборот, процесс начинается с заданного коэффициента попадания, причем делаются попытки отправить как можно меньше рекомендаций с учетом установленного ограничения. Аналогично, последние два алгоритма предполагают достижение оптимального соотношения между точностью и коэффициентом попадания. Однако ни одна из этих задач оптимизации не имеет эффективного решения [CKR99], что заставляет искать эффективные эвристические алгоритмы.

АЛГОРИТМЫ, ОСНОВАННЫЕ НА ВРЕМЕННОЙ ЛОКАЛИЗАЦИИ

Первая эвристическая процедура, рассмотренная в разделе 13.2.1, предлагает способ группировки ресурсов, предполагающий совместное обращение ко всем этим ресурсам. Эвристика предполагает, что том v_r должен содержать ресурс s, если вслед за запросом на r обычно следует запрос на s от того же самого клиента; запрос на s должен поступить не раньше τ секунд, и не позже T секунд после запроса на r. Для принятия решения, включать ли s в v_r, алгоритм подсчитывает число запросов на r, вслед за которыми следуют запросы на s. Это число делится на общее число запросов на ресурс r. Если полученное отношение превышает некоторое пороговое значение, то s включается в v_r; в противном случае этого не происходит. В действительности алгоритм может подсчитать число запросов на каждый из ресурсов и интервалы между парами запросов, последовательно проанализировав журнал запросов сервера. Затем, после обработки журнала, алгоритм вычисляет отношения с целью определить, какие ресурсы включать в каждый из томов. Подобный подход был рассмотрен в исследованиях, посвященных упреждающей выборке в Web [PM96, Bes95, JK98].

Хотя алгоритм эффективен с точки зрения объема вычислений, подобный подход имеет две слабые стороны. Во-первых, он требует наличия большого числа счетчиков, особенно для Web-сайтов, имеющих тысячи уникальных ресурсов. Хранение этих счетчиков требует большого объема памяти. Введенные в базовый алгоритм усовершенствования могут уменьшить нагрузку. В частности, можно сфокусировать внимание на наиболее популярных ресурсах и не использовать счетчики для пар ресурсов, которые редко фигурируют вместе [CKR99]. Во-вторых, алгоритм включает в том некоторые ресурсы, которые не способны послужить основой для новых предсказаний. Если сервер совмещает рекомендации с каждым сообщением-ответом, прокси-сервер может получать множество рекомендаций для одного и того же ресурса. Это дополнительные рекомендации не предоставляют никакой новой информации. Удаление ресурса из одного или нескольких томов позволяет избежать создания этих лишних рекомендаций. Это сокращает размер рекомендации и способствует повышению точности. Удаление ресурса из тома повышает точность, поскольку позволяет избежать ситуаций, в которых рекомендация не предсказывает будущий запрос.

Расширение исходного алгоритма дает возможность удалять неэффективные рекомендации за счет *сокращения* томов. Начав с томов, созданных исходным алгоритмом, двухпроходный алгоритм во второй раз просматривает журнал, чтобы смо-

делировать создание рекомендаций на основе этих томов. Затем алгоритм сокращения томов вычисляет, насколько часто каждый из элементов тома генерирует *новое* предсказание будущего запроса. Получив эти статистические данные, алгоритм удаляет неэффективные элементы из каждого тома. Идея удаления неэффективных предсказателей может быть обобщена в алгоритме, который несколько раз последовательно просматривает журнал. На первой итерации этот многопроходный алгоритм идентифицирует предсказатель, который является наиболее эффективным в предвидении запросов на ресурс. Затем, на следующей итерации, алгоритм определяет следующий наилучший предсказатель для остальных запросов. Повторяя этот процесс несколько раз, алгоритм формирует тома, которые дают минимальное количество избыточных предсказаний.

Исходный однопроходный алгоритм наименее сложен с вычислительной точки зрения, но генерирует избыточные рекомендации. Двухпроходный алгоритм удаляет неэффективные рекомендации, но требует дополнительных вычислительных затрат. Многопроходный алгоритм создает очень эффективный набор рекомендаций, но требует еще больших вычислительных затрат. Чтобы найти наиболее выгодный режим, требуется осознать, насколько более сложный алгоритм повышает эффективность и увеличивает время, необходимое для построения томов. В некоторых случаях увеличение времени вычислений может не иметь значения. Решение зависит от частоты формирования томов, а также различий в эффективности и объеме вычислений для этих трех алгоритмов. Кроме того, важно осуществлять сравнение этих трех алгоритмов с более простыми подходами, например, с эвристическими процедурами, описанными в разделе 13.2.1, в которых ресурсы группируются на основе структуры их URL. Эти важные проблемы не могут быть решены без оценки алгоритмов с учетом реальной работы серверов.

13.2.5. Оценка алгоритмов построения томов

Эффективность и объем вычислений зависят от количества ресурсов на сервере, а также от поведения клиентов. Эксперименты, описанные в [CKR98, CKR99], оценивают алгоритмы на основе данных из журналов серверов нескольких Web-сайтов, а также на основе данных из журналов прокси-серверов крупных корпораций. Интервал времени наблюдений варьировался от одной до семи недель, а количество запросов — от 180000 до 13000000 при обращении до 218000 клиентов к 94–30000 уникальным ресурсам. В журналах прокси-серверов ответ **304 Not Modified** давался на 16–19% запросов. В журналах серверов около 85% запросов пришлось примерно на 10% уникальных ресурсов. Эти статистические данные показывают, что правильное предсказание запросов на эти популярные ресурсы может повысить эффективность работы прокси-сервера. Например, рекомендации, содержащие время последней модификации **Last-Modified** ресурсов сервера, позволят прокси-серверу обновить время истечения актуальности кэшированных ответов, которые не были модифицированы, либо удалить ответы, которые были изменены.

В ходе экспериментальных исследований предпринималась попытка установить, может ли сервер эффективно доставлять полезные рекомендации прокси-серверам. В ходе экспериментов сравнивались различные алгоритмы построения томов с учетом четырех показателей: объема рекомендации, коэффициента попадания, точности и доли обновлений. В исследованиях использовался единственный фильтр прокси-сервера, ограничивающий число рекомендаций, совмещенных с каждым ответом. С сообщениями-ответами для ресурса r сервер совмещал рекомендации для ресурсов, которые наиболее вероятно могли быть запрошены после ре-

сурса *r*. Число рекомендаций на одно сообщение-ответ в ходе экспериментов варьировалось. При построении томов алгоритмы учитывали все ресурсы, доступ к которым осуществлялся не менее десяти раз. Это значительно снизило сложность алгоритмов построения томов за счет удаления большого числа менее популярных ресурсов. Построение томов для таких ресурсов не привело бы к выдаче достаточно большого числа полезных рекомендаций.

В результате исследований было сделано заключение, что каждый из алгоритмов построения томов дает хорошие коэффициенты попадания и доли обновлений. Однако для томов, построенных на основе структур каталогов, высокие коэффициенты попадания достигаются ценой излишне большого размера рекомендаций и низкой точности. Сервер посылает много избыточных рекомендаций. Алгоритмы, основанные на анализе доступа клиентов к ресурсам, дают гораздо большую точность и меньший объем рекомендаций. В общем случае отправка большего числа рекомендаций ведет к снижению точности при весьма незначительном увеличении коэффициента попадания. За счет удаления неэффективных рекомендаций, двухпроходные и многопроходные алгоритмы предлагают дальнейшее уменьшение объема рекомендаций и увеличение точности. Двухпроходный алгоритм дает почти столь же хороший эффект, что и многопроходный алгоритм, требуя при этом гораздо меньших затрат. Для множества журналов в результате применения двухпроходного алгоритма был получен коэффициент попадания от 60% до 80%, точность от 80% до 88% и число рекомендаций от 2 до 10 рекомендаций на сообщение. Эти результаты действительны для предсказаний, соответствующих значениям T в пределах от 60 секунд до 5 минут и $\tau = 0$.

Применение алгоритмов к большим серверным журналам с миллионами записей требует больших затрат времени. Затраты можно уменьшить, выполнив алгоритм по формированию томов над усеченной версией журнала. Создание усеченной версии журнала предусматривает случайную выборку клиентов и анализ запросов, поступивших от этих клиентов. Усеченные журналы содержат от 3% до 50% запросов исходного журнала. Эксперименты показали, что построение томов на основе этих усеченных журналов не приводят к снижению эффективности. В другой группе экспериментов тома создавались на основе запросов одного периода, а применялись эти тома к другому периоду из того же журнала. Использование одних и тех же томов за час или день к последующему часу или дню не искажало результатов. Однако важно периодически пересчитывать тома, чтобы адаптироваться к изменениям в поведении пользователей, например, учесть особенности трафика в рабочие и выходные дни, либо чтобы учесть изменения в содержании Web-сайта. Наблюдения показали, что периодическое построение томов на основе выборок из журналов может быть вполне приемлемым для большинства серверов.

Эксперименты не предлагают идеального решения для достижения оптимального соотношения между различными показателями эффективности. Степень влияния одного показателя на другие зависит от того, каким образом прокси-сервер использует рекомендации сервера. Например, предварительная выборка содержания на основе рекомендаций сервера будет вносить существенную дополнительную нагрузку, если точность рекомендаций низка. Низкая точность подразумевает, что прокси-сервер будет отбирать много ресурсов, которые окажутся невостребованными в дальнейшем. С другой стороны, прокси-сервер может использовать рекомендации для проверки актуальности (или аннулирования) кэшированных ресурсов. Низкая точность в этом случае не будет приводить к большой перегрузке, если прокси-сервер готов смириться с низкой точностью в обмен на больший коэффициент попадания. Различные прокси-серверы могут выбирать различные варианты компромиссов на основе своих фильтров. Прокси-сервер, выполняющий упреж-

дающую загрузку ресурсов, может предпочесть более жесткую фильтрацию, в то время как прокси-сервер, осуществляющий проверку актуальности кэша, может использовать относительно менее строгий фильтр.

13.2.6. Комплексный информационный обмен. Резюме

В этом разделе был описан комплексный подход к повышению производительности прокси-серверов на основе рекомендаций от исходных серверов. Сервер группирует взаимосвязанные ресурсы в тома и применяет фильтры для формирования рекомендаций обратившимся с запросами прокси-серверам. Эксперименты с журналами серверов показали, что эффективные алгоритмы построения томов могут позволить выдавать небольшое число рекомендаций, предсказывающих значительную часть будущих запросов. Наиболее перспективные алгоритмы вычисляют вероятность того, что ресурсы будут затребованы совместно, и опускают рекомендации, которые генерируют дублирующие предсказания. Объединение фильтров прокси-серверов и серверных рекомендаций может быть реализовано без модификации HTTP/1.1. Фильтр прокси-сервера может быть отправлен в новом заголовке запроса, а серверные рекомендации могут быть отправлены в трейлере сообщения-ответа. Использование существующего стандарта в значительной мере устраняет барьеры на пути практического внедрения данной схемы, поскольку при этом не требуется длительного процесса стандартизации. Выразительные средства HTTP/1.1 облегчают введение новых возможности без внесения изменений в протокол.

В ходе изучения проблемы были введены понятия серверных томов и фильтров прокси-серверов, описаны и сопоставлены различные алгоритмы построения томов. Однако остается ряд проблем. Прокси-серверы могут посылать множество разнообразных фильтров с различными параметрами. Поиск эффективного способа применения этих фильтров к серверным томам представляет собой интересную алгоритмическую задачу. Было предложено на начальном этапе хранить элементы тома в отдельных списках типа «первым пришел — первым обработан» в зависимости от атрибутов ресурсов. Однако это может оказаться не самым эффективным способом обработки большого числа параметров фильтрации. Кроме того, прокси-серверы отличаются по важности, придаваемой ими коэффициенту попадания, точности и доли обновлений. Определение влияния этих показателей на эффективность работы приложений задает направление для будущих исследований. Результатом может стать разработка новых технологий для построения томов и выбора параметров фильтров прокси-серверов.

Другая открытая проблема связана со способностью сервера создавать тома, объединяющие ресурсы, доступ к которым осуществляется совместно. Серверу необязательно видеть все запросы на ресурсы. Во-первых, некоторые запросы обрабатываются прокси-сервером без обращения к исходному серверу. Это ограничивает возможность сервера иметь полную информацию о доступе клиентов. Чтобы решить проблему, прокси-сервер может информировать сервер об этих запросах. Так, предположим, что клиент запрашивает ресурс *s* после обращения к ресурсу *r*, и что прокси-сервер обрабатывает запрос на *r*, но должен связаться с сервером относительно ресурса *s*. В этом случае запрос на *s* может содержать информацию о предыдущем запросе на *r*. Это даст возможность серверу ассоциировать запросы на ресурс *s* с ресурсом *r*. Аналогично, сервер не обрабатывает все запросы, которые относятся к встроенным изображениям и гиперссылкам HTML-файлов. Некоторые из этих ресурсов находятся на других серверах. При условии определенного сотрудничества между Web-серверами рекомендации, отправляемые прокси-серверам, могут содержать информацию о ресурсах, расположенных на других серверах.

13.3. Упреждающая выборка

Обработка Web-запроса состоит из нескольких ключевых действий, включая преобразование доменного имени, установление TCP-соединения и обмен запросами-ответами HTTP. Web-клиенты, включая агенты пользователей и прокси-серверы, могут выполнять некоторые из этих задач с *упреждением* пользовательского запроса. Тем самым сокращается задержка, испытываемая пользователем, но вносится дополнительная нагрузка на сеть и на сервер. В течение последних нескольких лет исследователи предложили и произвели оценку разнообразных методов упреждающей выборки. До недавнего момента внедрение этих идей сдерживалось по причинам, связанным с увеличением затрат. Однако желание повысить эффективность работы пользователей в конечном итоге может перевесить эти ограничения.

13.3.1. Упреждающее преобразование доменных имен

Прежде чем установить соединение с Web-сервером, клиент должен преобразовать доменное имя запрашиваемого URL в IP-адрес. Клиент инициирует системный вызов, такой как *gethostbyname()*, который посылает запрос локальному DNS-серверу, который, в свою очередь, связывается с другими DNS-серверами, как описывалось ранее в главе 5 (раздел 5.3). Ожидание ответа от локального DNS-сервера вносит задержку при обработке пользовательского запроса. Чтобы избежать этой задержки, Web-клиент может инициировать преобразование имени в адрес до пользовательского запроса. Например, пользователь посещает Web-страницу, содержащую несколько гипертекстовых ссылок на другие Web-серверы. Клиент может определить IP-адреса этих серверов, пока пользователь читает страницу. Если пользователь щелкает на одной из этих гипертекстовых ссылок, клиент может немедленно установить TCP-соединение с Web-сервером. Это уменьшает время ожидания пользователя при загрузке новой страницы.

Сокращение времени ожидания зависит от времени, которое требуется для обработки DNS-запроса. DNS-запрос не вносит большой задержки, если локальный DNS-сервер уже имеет кэшированную копию IP-адреса Web-сервера. Однако задержка может быть значительной, если локальный DNS-сервер должен связаться с другими DNS-серверами, чтобы обработать запрос. Задержки в несколько секунд — вполне обычное явление [CK00, HW00]. Отсутствие IP-адреса в кэше DNS-сервера — вполне обычное явление для не слишком популярных Web-сайтов или соответствий доменное имя–IP-адрес с небольшими временами жизни (TTL). Небольшие значения TTL достаточно часто имеют место, когда ответы на DNS-запросы используются для распределения нагрузки между репликами Web-сайта. Однако небольшие значения TTL также ограничивают выигрыш от упреждающей выборки информации DNS задолго до получения пользовательского запроса. Хотя упреждающая выборка обеспечивает Web-клиенту и локальному DNS-серверу копию IP-адреса Web-сервера, кэшированная информация может устареть до того, как пользователь запросит содержание с этого Web-сервера. Рост популярности сетей распределения содержания, которые используют небольшие значения TTL с целью переадресации запросов наименее загруженной реплике ресурса, осложняют проблему упреждающего преобразования доменных имен DNS.

Упреждающая выборка IP-адреса Web-сервера уменьшает время ожидания на стороне пользователя, но несет риск дополнительной загрузки сети и DNS-серверов. Web-клиент выполняет упреждающее преобразование доменного имени в IP-адрес в надежде, что пользователь со временем посетит Web-сайт. Однако поведение поль-

зователя нельзя предугадать заранее. Если пользователь никогда не щелкнет на гипертекстовой ссылке, DNS-запрос окажется излишним. Запрос загружает локальный DNS-сервер и, возможно, другие DNS-серверы. Обработка этого запроса может задержать обработку DNS-серверами других запросов, что увеличивает время ожидания для этих запросов. Кроме того, кэш DNS может содержать имена серверов, которые редко используются. Если большая часть Web-клиентов выполняет упреждающее преобразование доменных имен, это может значительно увеличить число запросов, обрабатываемых DNS-серверами. Эти запросы также приводят к увеличению трафика. Преимущества от упреждающего преобразования доменных имен должны быть сопоставлены с возникающими при этом издержками.

13.3.2. Упреждающее установление соединений

Составной частью обработки HTTP-запроса является установление TCP-соединения с Web-сервером или с промежуточным компонентом, таким как прокси-сервер. Открытие TCP-соединения требует трехэтапного обмена между клиентом и сервером, о чем рассказывалось в главе 5 (раздел 5.2). Время задержки на этом этапе зависит от времени прохождения пакетов в обоих направлениях между компьютерами, а также от задержки в очереди на сервере и дополнительной задержки, обусловленной утерей пакетов. Чтобы скрыть эти задержки от пользователя, Web-клиент может заранее установить TCP-соединение с сервером до выдачи пользовательского запроса [CK00]. Затем, когда пользователь щелкает мышью на гиперссылке, клиент может немедленно отправить HTTP-запрос через открытое TCP-соединение. Если пользователь не инициирует запрос, либо клиент, либо сервер со временем закрывает TCP-соединение.

Задержка в обработке запроса для статического содержания по основном вызвана временем установления TCP-соединения. Для небольших сообщений-ответов дополнительное время прохождения пакетов в прямом и обратном направлении составляет ощутимую долю общей задержки при открытии TCP-соединения. Однако успех в сокрытии этой задержки от пользователя зависит от двух факторов. Во-первых, чтобы достичь максимального выигрыша в производительности, установление соединения должно закончиться до того, как пользователь щелкнет мышью на гиперссылке; в противном случае клиенту придется ждать, прежде чем отправить HTTP-запрос. Во-вторых, пользовательский запрос должен поступить до того, как клиент или сервер решат закрыть соединение. Web-серверы обычно устанавливают ограничение на время нахождения TCP-соединения в бездействующем состоянии. Пользовательский запрос, инициированный после того, как сервер закрыл соединение, либо в то время, когда сервер осуществляет закрытие соединения, не дает выигрыша в производительности. Наличие открытого бездействующего TCP-соединения может вызвать увеличение ожидания в очереди на создание новых соединений. Что еще хуже, увеличение числа открытых соединений может уменьшить значение таймаута для выполняющихся сеансов.

Открытие TCP-соединения с сервером снижает время ожидания на стороне пользователя ценой дополнительной нагрузки на сеть, клиента и сервер. Трехэтапный обмен предполагает передачу через сеть трех IP-пакетов без учета какой-либо повторной передачи пакетов и дополнительных пакетов, необходимых для закрытия соединения. Даже если клиент закрывает TCP-соединение, не отправляя TCP-запрос, сервер потребляет ресурсы на открытие и поддержание соединения. Web-сервер должен сохранять состояние соединения и может назначить процесс для обработки ожидаемого HTTP-запроса (запросов). В действительности Web-

сервер обычно сохраняет состояние в течение определенного периода времени после закрытия соединения. Сохранение данного TCP-соединения может помешать серверу устанавливать TCP-соединения с другими клиентами. Открытие и закрытие TCP-соединения без отправки HTTP-запроса может рассматриваться как атака отказа от обслуживания (DoS) на Web-сервер. Следует сопоставлять увеличение нагрузки на сеть и на сервер с потенциальным повышением производительности на стороне пользователя.

13.3.3. Упреждающая выборка в HTTP

После установления TCP-соединения Web-клиент выдает HTTP-запрос Web-серверу. Даже если клиент имеет кэшированную копию запрашиваемого ресурса, у него может возникнуть необходимость связаться с сервером для проверки актуальности кэшированной версии. Время ожидания получения HTTP-ответа зависит от множества факторов, в том числе от времени создания ответа на сервере, размера сообщения-ответа и пропускной способности сети. Клиент может замаскировать задержку на стороне пользователя, выдав заранее HTTP-запрос и кэшировав ответ. Если пользователь запрашивает ресурс, клиент может предоставить ресурс непосредственно из своего кэша. Предполагая, что кэшированный ответ не устарел, клиент может удовлетворить пользовательский запрос без определения IP-адреса сервера, открытия соединения с сервером и отправки HTTP-запроса. Пользователь получит ответ практически немедленно. Кроме того, упреждающая выборка может сделать использование TCP-соединений более эффективным, нежели при традиционных Web-передачах, управляемых действиями пользователя. Клиент с упреждающей выборкой может осуществить конвейерную обработку ряда запросов в ходе одного постоянного соединения, а совместная передача сообщений-ответов позволяет избежать дополнительных затрат, связанных с повторением фазы медленного старта при управлении скользящим окном.

Упреждающей выборке HTTP-ответов посвящены многочисленные исследования [Bes95, PM96, JK98, CB98, Duc99]. Несмотря на потенциальное уменьшение времени ожидания на стороне пользователя, извлечение с упреждением Web-ресурсов может вызвать значительное увеличение нагрузки на Web-сервер и сеть. Упреждающая выборка не принесет пользы, если запросы пользователя на ресурс и выбираемый с упреждением ответ не будут оставаться актуальными. Кроме того, пропускная способность сети и сервера, выделенная под передачу ответа, испытывает конкуренцию со стороны других текущих передач. Например, предположим, что пользователь осуществляет доступ в Web через модем с низкой пропускной способностью [HL96, LB97]. С одной стороны, упреждающая выборка в моменты бездействия позволит более эффективно использовать ограниченную пропускную способность. С другой стороны, упреждающая выборка в момент загрузки пользователем другой Web-страницы увеличит время ожидания для текущей страницы. Кроме того, упреждающая выборка ресурса может привести к вытеснению других ресурсов из кэша клиента, что в будущем увеличит время ожидания ответов на другие запросы.

Вместо упреждающей выборки ресурса сервер может осуществить выборку метаданных ресурса. Например, клиент может отправить запрос **HEAD** серверу, как говорилось ранее в разделе 13.1.2. HTTP-ответ будет иметь те же заголовки, что и ответ на запрос **GET**. Эта информация особенно полезна, если клиент уже имеет кэшированную копию ресурса. В зависимости от того, был ли ресурс изменен исходным сервером, клиент может аннулировать кэшированный ресурс, либо обновить информацию об актуальности ресурса. Заголовки в сообщении-ответе могут

быть полезными даже в том случае, если клиент не имеет кэшированной копии ресурса. Например, клиент может просмотреть заголовки, относящиеся к кэшированию, такие как **Last-Modified**, **Cache-Control** или **Content-Type**, чтобы определить, стоит ли осуществлять упреждающую выборку ресурса. Клиент может принять решение не выбирать с упреждением ресурс, который не кэшируется или часто изменяется. Кроме того, клиент может просмотреть заголовок **Content-Length** с целью определения размера ресурса. Клиент может решить осуществить упреждающую выборку ресурса, если заголовок **Content-Length** указывает, что размер ресурса меньше некоторого порогового значения. Это позволит избежать упреждающей выборки ресурсов, отвлекающих на себя значительную часть сетевого трафика.

Вместо того чтобы выдавать запрос **HEAD** для определения размера ресурса, клиент может непосредственно управлять объемом данных, выбираемых с упреждением, путем передачи запроса **GET** с заголовком **Range** (в предположении, что Web-сервер допускает запросы на диапазоны). Например, клиент может запросить первые 1000 байтов ресурса. Это позволит клиенту загружать небольшой ресурс целиком, а для большого ресурса загружать его начальную часть. В некоторых случаях может оказаться полезной упреждающая выборка *префикса* ресурса [Hor98, SRT99]. Например, первые несколько байтов GIF-файла содержат размер изображения и могут представлять собой версию рисунка с меньшим разрешением. Имея префикс изображения до запроса пользователя, браузер может начать воспроизведение изображения. Аналогично префикс аудио- или видеоклипа может содержать достаточное количество кадров, чтобы начать воспроизведение потока [SRT99, GRB00]. Как только пользователь запросит ресурс, браузер может отправить второй запрос **Range**, чтобы осуществить выборку оставшегося содержимого. С точки зрения пользователя содержимое начинает воспроизводиться немедленно, несмотря на то, что браузер заранее осуществил выборку только начальной части ресурса.

13.3.4. Компромиссы при упреждающей выборке

Для упреждающей выборки характерно противоречие между снижением времени ожидания на стороне пользователя и увеличением нагрузки на сеть и компоненты Web. При достижении компромисса между этими двумя факторами следует учитывать следующее:

- **Какие задачи следует выполнять с упреждением.** Упреждающая выборка может включать выполнение DNS-запроса, открытие TCP-соединения, извлечение метаданных и выборку всего ресурса или его начальной части. Выполнение большего числа задач с упреждением предполагает дальнейшее сокращение ожидания на стороне пользователя ценой повышения нагрузки на сеть и компоненты Web.
- **Какие ресурсы учитывать.** Выборка с упреждением наиболее эффективна, если информация вероятнее всего будет использоваться и сохранит свою актуальность. Это подразумевает выявление наиболее популярных ресурсов и серверов, а также ресурсов, которые изменяются не слишком часто. В противном случае упреждающая выборка может сопровождаться лишь увеличением нагрузки, не принося особой выгоды. Клиент может применять эвристические процедуры для идентификации этих ресурсов, либо воспользоваться статистикой о поведении пользователей.
- **Какие компоненты осуществляют упреждающую выборку.** Упреждающую выборку может выполнять любой клиент. В случае упреждающей выборки

агентом пользователя данные перемещаются ближе к пользователю, но при этом потребляется больше сетевых ресурсов, включая пользовательское соединение с Internet. Если агент пользователя осуществляет взаимодействие через прокси-сервер, ответ сервера может кэшироваться на прокси-сервере. В качестве альтернативы прокси-сервер может сам выполнять упреждающую выборку ресурсов. Это снижает нагрузку на каналы между прокси-сервером и агентом пользователя ценой внесения дополнительной задержки для агента пользователя, связанной с извлечением данных с прокси-сервера.

Одной из первых систем выборки с упреждением в HTTP, осуществляющих синтаксический анализ HTML-файлов и упреждающую выборку ресурсов на основе гиперссылок, была система Letizia [Lie95], о которой говорилось в главе 2 (раздел 2.8.2). В Letizia для отображения загруженного с упреждением содержания открывались дополнительные окна. Система допускала настройку, чтобы следовать гиперссылкам в этих документах с целью упреждающей выборки ресурсов, на которые указывают гиперссылки. Однако упреждающая выборка всех ресурсов, на которые указывают гиперссылки на странице, может обусловить слишком большую нагрузку. Применяя эвристическую процедуру, клиент может с упреждением выбрать первые несколько ссылок на странице в предположении, что они являются наиболее популярными. Возьмем Web-страницу, содержащую список URL, возвращенный поисковой системой. Пользователь, скорее всего, будет обращаться к элементам списка по порядку, поскольку поисковая система уже выполнила их ранжирование. Однако в некоторых случаях популярность URL может быть не связана с их расположением на странице.

Агент пользователя может оценить популярность URL на основе предыдущих запросов. Например, Letizia отслеживает предпочтения пользователя и действия при просмотре страниц с целью определения, какие ресурсы выбрать с упреждением. В сравнении с агентами пользователя, прокси-сервер лучше может судить о популярности ресурсов, анализируя запросы групп клиентов. На основе анализа поведения пользователей прокси-сервер может определить, какие гипертекстовые ссылки наиболее популярны. При получении запроса на HTML-файл прокси-сервер может также осуществить упреждающую выборку популярных ресурсов, доступных на Web-странице. Однако прокси-сервер не будет иметь достаточно данных о поведении пользователей, если только несколько клиентов не обратились в прошлом к HTML-файлу [JK98]. Сервер обладает более полной информацией о поведении пользователей. Таким образом сервер может предоставить прокси-серверу рекомендации относительно того, какие ресурсы вероятнее всего будут запрошены в будущем, о чем говорилось ранее в разделе 13.2.

Вместо того чтобы отправлять прокси-серверу рекомендации, можно встраивать статистические данные о популярности ресурсов в HTML-файл. Например, гипертекстовая ссылка может содержать дополнительную информацию о вероятности того, что пользователь выберет эту ссылку после чтения содержимого страницы. Это требует изменения HTML-страницы и периодического обновления статистики для страницы. Подобные подходы наиболее эффективны для специфических приложений, таких как поисковые машины, но не для всех HTML-страниц. В некоторых случаях человек, создавший HTML-файл, может напрямую управлять упреждающей выборкой. Например, предположим, что в HTML-файле **http://www.foo.com/index.html** имеется гипертекстовая ссылка на HTML-файл **http://www.foo.com/neat.html**, содержащий встроенное изображение **http://www.foo.com/pic.jpg**. Наличие гиперссылки на **pic.jpg** на странице **index.html** приводит к тому, что клиент заранее выберет изображение при загрузке начальной Web-страницы. В результате изображение уже будет

находиться в кэше клиента, когда пользователь запросит ресурс **neat.html**. Чтобы не допустить вывода изображения вместе с первой страницей, можно указать для него небольшие размеры, например, один на один пиксел по ширине и высоте.

Большинство исследований, посвященных упреждающей выборке, основное внимание уделяют упреждающей выборке HTTP-ответов с сервера. Результаты этих исследований довольно разнородны. Снижение времени ожидания на стороне пользователя при упреждающей выборке можно снижать до тех пор, пока это не снижает производительности для других запросов. Однако выбранные с упреждением ресурсы, которые никогда не будут запрошены пользователем, способствуют значительному увеличению нагрузки на сеть и на сервер. Упреждающая выборка информации DNS, метаданных HTTP и установление TCP-соединений не слишком увеличивает сетевой трафик. Однако упреждающая выборка информации DNS загружает DNS-серверы, а упреждающее создание TCP-соединений и упреждающая выборка метаданных HTTP загружает Web-серверы. В некоторых случаях упреждающая выборка метаданных HTTP через существующее TCP-соединение может реально уменьшить нагрузку, поскольку в этом случае нет необходимости устанавливать отдельные TCP-соединения в будущем.

Внедрение технологии упреждающей выборки в Web зависит от того, перевесит ли выигрыш в производительности затраты. Упреждающая выборка может стать более популярной, если снизятся затраты на передачу данных и возрастет скорость их передачи, а также если пользователи станут нетерпимы к большим задержкам. Достижение компромисса между затратами и производительностью требует более детального понимания, насколько упреждающая выборка сокращает время ожидания на стороне пользователя и насколько уменьшение задержки улучшит восприятие ресурсов пользователями в сравнении с увеличением нагрузки на Web-серверы и магистральные сети.

13.4. Резюме

В этой главе мы рассмотрели три направления исследований, связанных с кэшированием в Web. В первом разделе речь шла о проверке актуальности элементов кэша как важном факторе, обеспечивающем семантическую прозрачность при кэшировании. За счет технологии совмещения возможно уменьшить затраты на проверку актуальности, в то же время уменьшив время ожидания на стороне пользователя путем использования усовершенствованных алгоритмов проверки актуальности. Двунаправленный подход с применением фильтров и томов позволяет наиболее полно использовать информацию, доступную как прокси-серверам, так и исходным серверам. В результате удается оптимизировать выдачу рекомендаций прокси-серверам при минимальных затратах для исходного сервера и сети. Алгоритмы для построения томов — обширная область для дальнейших исследований. Изучение различных показателей эффективности позволяет строить алгоритмы для применения в различных ситуациях. Изучение упреждающей выборки на уровне DNS, TCP и HTTP раскрывает различные пути к снижению времени ожидания на стороне пользователя. Для каждого метода следует учитывать реальные затраты, которые будут иметь место в конкретной ситуации. При проведении исследований огромную роль играют фактические данные. Хотя обычно трудно получить достаточно большой массив данных, последние достижения в построении хранилищ данных (см. главу 14) могут способствовать тому, что рожденные в результате исследований идеи пройдут практическую проверку с использованием реальных данных измерений.

14

Перспективы исследований, связанных с измерениями

Сбор и анализ данных измерений играет важную роль в оценке Web-протоколов и программных компонентов. Исследователи часто берут за основу данные из журналов регистрации событий при определении характеристик Web-трафика и оценке новых идей по совершенствованию Web. В связи с этим в сообществе исследователей Web ведется интенсивная работа по поиску эффективных способов получения и обработки данных измерений параметров трафика. Проведение исследований, связанных с измерениями в Web, требует понимания принципов, заложенных в HTTP и других сетевых протоколах, поверх которых он работает, а также способности разрабатывать эффективное и устойчивое программное обеспечение. В Web-измерениях обычно участвуют большие массивы данных. Подобно другим сетевым протоколам, HTTP не был предназначен для выполнения измерений. Несмотря на важность сбора данных измерений, генерирование точных и исчерпывающих записей измеренных данных не является главной задачей для большинства реализаций Web-серверов и прокси-серверов. В связи с этим подобные записи могут содержать ошибки и несоответствия, которые осложняют анализ данных. Выявление этих ошибок требует применения специальных программ, проверяющих синтаксис и семантику записей.

Эффективность Web зависит от взаимодействия между рядом протоколов. Время ожидания на стороне пользователя и производительность сервера при HTTP-передачах зависит от транспортного уровня. Эффективность мультимедийных приложений зависит от взаимодействия нескольких протоколов для передачи управляющих сообщений, данных аудио и видео, данных о текущем состоянии сети. Чтобы охарактеризовать взаимодействие между протоколами, необходимо осуществлять детальную трассировку и следить за активностью каждого уровня в стеке протоколов. Традиционные прокси-серверы и Web-серверы не отражают эту информацию в своих журналах. Трассировка пакетов предлагает возможную альтернативу для изучения влияния сети на производительность прикладного уровня. Однако мониторы пакетов обычно собирают данные на сетевом и на транспортном уровнях, не предоставляя информации о протоколах прикладного уровня, такой как HTTP-заголовки, которые могут размещаться в нескольких IP-пакетах. Изучение взаимодействия между прикладным и транспортным уровнями требует наличия программного обеспечения, осуществляющего более сложный мониторинг пакетов, нежели простое извлечение информации о сообщениях прикладного уровня.

В этой главе мы рассмотрим следующие четыре вопроса:

- **Мониторинг пакетов HTTP-трафика.** Большинство исследований характеристик Web-трафика основаны на данных из журналов регистрации прокси-серверов и Web-серверов. Однако эти журналы обычно не содержат подробной

информации о запросах и ответах HTTP, а также о временных параметрах различных этапов Web-транзакций. Трассировка пакетов путем мониторинга сети может предоставить гораздо более подробные данные, как уже говорилось в главе 9 (раздел 9.2.4). В этой главе мы опишем проблемы, связанные с перехватом IP-пакетов в сети, восстановлением потока байтов для каждого TCP-соединения и восстановлением сообщений запросов и ответов HTTP.

- **Анализ журналов Web-серверов.** Данные журналов серверов используются в большинстве исследований. Однако анализ журналов серверов сопряжен с различными практическими проблемами, обусловленными большим числом записей, часто встречающимися ошибками и различиями в форматах журналов. Предварительная обработка журналов может значительно уменьшить сложность программы анализа, как говорилось в главе 9 (раздел 9.4). В этой главе будут описаны программы синтаксического анализа, фильтрации, преобразования и анализа серверных журналов. В ходе обсуждения будут затронуты некоторые подводные камни, встречающиеся при обработке серверных журналов, и описано, как воспользоваться имеющимися библиотечными процедурами для разработки надежного и эффективного программного обеспечения.
- **Общедоступные журналы и результаты трассировки.** Важность измерений параметров Web-трафика для исследований способствовала усилиям по созданию репозиториев общедоступных журналов и результатов трассировки. Данные измерений обычно являются доступными в Web. Мы представим краткий обзор существующих Web-сайтов, предоставляющих доступ к данным измерений. Большинство этих сайтов предоставляют доступ к данным в различных форматах по принципу «как есть». Консорциум World Wide Web Consortium (W3C) разработал официальное описание синтаксиса и семантики серверных журналов и предоставляет доступ к надежным, сертифицированным журналам в одинаковом формате.
- **Измерение параметров мультимедийных потоков.** Хотя большинство работ по измерению в Web посвящены HTTP-трафику, превращение мультимедийных потоков в важное Web-приложение подвигло на ряд исследований по измерению параметров мультимедийного трафика. При измерениях, проводимых на потоках аудио и видео, берутся в расчет те же самые параметры рабочей нагрузки, что и для HTTP-трафика, однако учитываются и уникальные для мультимедийных данных характеристики. Измерение характеристик мультимедийного трафика порождает новые проблемы, вызванные наличием группы различных протоколов для потоков аудио и видео, о чем говорилось ранее в главе 12 (раздел 12.3). Мы познакомимся с четырьмя исследованиями, основанными на статическом анализе мультимедийных файлов в Web, анализе журналов мультимедийных серверов, мониторинге пакетов аудиопередач и методах мониторинга пакетов для сбора управляющих сообщений и пакетов данных.

На протяжении главы будет подчеркиваться важная роль программного обеспечения в сборе и анализе измерений трафика.

14.1. Мониторинг пакетов HTTP-трафика

Мониторинг пакетов — эффективный способ получить подробную информацию о Web-трафике. Монитор пакетов представляет собой компьютер, получающий копии пакетов, передаваемых по одному или нескольким сетевым каналам, и содер-

жащий программное обеспечение, которое обрабатывает пакеты с целью формирования трассы. Трассы могут храниться в памяти, на диске либо на магнитной ленте в мониторе пакетов, а затем копироваться на другой компьютер для архивирования и анализа данных. В этом разделе будут представлены различные этапы трассировки пакетов HTTP-трафика:

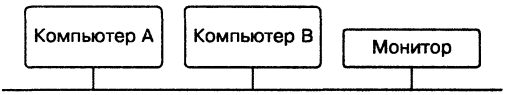
- Подключение к IP-сети.
- Перехват пакетов HTTP-передач.
- Демультиплексирование пакетов в TCP-соединениях.
- Восстановление упорядоченного потока байтов.
- Извлечение HTTP-сообщений из потока байтов.
- Формирование журнала HTTP-сообщений.

При измерении параметров HTTP-трафика с помощью монитора пакетов требуется учитывать все особенности TCP и HTTP. На уровне TCP монитор должен уметь работать с утерянными, передаваемыми в другом порядке и повторяющимися IP-пакетами. На уровне HTTP монитор должен допускать возможность долговременных соединений, которые передают несколько HTTP-сообщений и учитывать прерванные передачи. Помимо этого, монитор должен мириться с HTTP-сообщениями, отправляемыми Web-компонентами, которые не являются совместимыми с этими протоколами. Первая попытка восстановления HTTP-сообщений из IP-трасс в оперативном режиме описывается в [Fel00a].

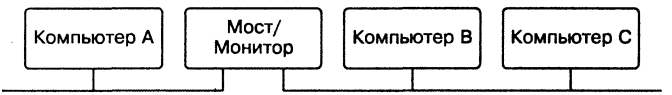
14.1.1. Подключение к каналу

Трассировка пакетов требует эффективного способа подключения к функционирующей IP-сети. Сложности, связанные с подключением, зависят от базовой технологии и конфигурации сети, а именно:

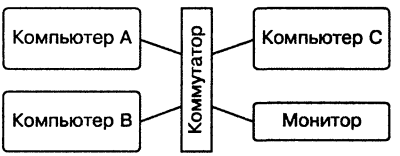
- **Совместно используемая среда передачи данных.** Многие локальные сети включают совместно используемую среду передачи данных, например, Ethernet, кольцо FDDI или беспроводную сеть. Каждая машина в сети видит каждый пакет, передаваемый другими компьютерами. Для перехвата в совместно используемой среде передачи данных требуется всего лишь подключить монитор пакетов к сети, как показано на рис. 14.1 а. Обычный компьютер в сети с совместно используемой средой передачи данных только получает копии направленных ему пакетов. Однако многие сетевые карты допускают настройку для работы в *смешанном режиме*, при котором локально копируется каждый пакет. Пакеты должны копироваться в монитор достаточно быстро, чтобы сетевая карта имела достаточно памяти для сохранения последующих пакетов при их поступлении. Эта операция является пассивной и не оказывает влияния на другие компьютеры в сети. Фактически остальные компьютеры в сети не будут знать, что монитор пакетов собирает данные.
- **Мост.** Альтернативный подход предусматривает мониторинг трафика на мосту, который передает трафик между сегментами сети, как показано на рис. 14.1 б. Мост может быть сконфигурирован для получения копии каждого пакета, передаваемого от одного сегмента другому. Участвуя в пересылке каждого пакета, мост может обеспечить захват всех пакетов. В отличие от смешанного режима мониторинга, перехват пакетов мостом может задержать пересылку трафика в сети. Следует быстро записывать необходимую информацию относительно каждого пакета, чтобы избежать снижения производительности.



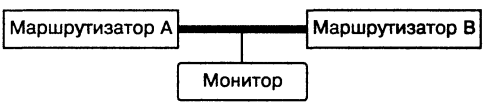
а) Совместно используемая среда передачи данных



б) Мост



в) Коммутатор



г) Расщепление канала

Рис. 14.1. Методы подключения к сети

- **Коммутатор.** Некоторые сети включают в себя коммутаторы, которые направляют пакеты из входных каналов в выходные каналы. Коммутаторы обычно имеют более высокую производительность, чем среда передачи данных, поскольку трафик может одновременно передаваться по каждому из каналов. В сравнении с мониторингом в общей среде передачи данных, перехват пакетов на коммутаторе несомненно вызывает больше затруднений. Ни один из каналов не видит все копии пакетов. Один из способов получить каждый из пакетов — подключить монитор пакетов к каждому входному каналу или к каждому выходному каналу. Однако это может оказаться слишком дорогостоящим. Другая альтернатива — выделить один выходной канал коммутатора для монитора пакетов и заставить коммутатор выполнять *групповую* пересылку, чтобы направлять копии исходящих пакетов монитору, как показано на рис. 14.1 в. Коммутатор будет направлять пакет в предназначенный для него исходящий канал и в монитор. Однако копирование пакетов из каждого входного канала может потребовать дополнительной пропускной способности канала, к которому подключен монитор. В этой связи групповая пересылка больше подходит для выборки части пакетов, проходящих через коммутатор.
- **Канал точка-точка.** Установка моста или коммутатора с возможностями групповой передачи не всегда возможна. Альтернативный подход к мониторингу пакетов предусматривает расщепление канала на две части, как показано на рис. 14.1 г. На рисунке представлен двунаправленный канал, который состоит из двух однонаправленных каналов между парой маршрутизаторов. Каждый

однонаправленный канал расщепляется и подключается к монитору пакетов. В некоторых случаях канал в действующей сети может уже быть расщеплен для целей тестирования и диагностики. Подключение к расщепленному каналу дает возможность монитору пакетов получать копию любого пакета, передаваемого между двумя маршрутизаторами. Однако расщепление канала может ослабить электрический или оптический сигнал, передаваемый между двумя маршрутизаторами. В ряде случаев во избежание снижения уровня сигнала, получаемого маршрутизаторами и монитором пакетов, потребуется установка дополнительного оборудования, например, оптического усилителя.

14.1.2. Перехват пакетов

Кроме подключения к каналу передачи данных, монитор должен иметь эффективный способ идентифицировать, какие IP-пакеты перехватывать при их поступлении. Протоколы прикладного уровня обычно связаны с определенным TCP-портом, например, HTTP использует порт 80. Для отслеживания HTTP-трафика монитор пакетов может ограничиться учетом TCP-трафика на порту 80. Такое решение может быть принято для каждого конкретного пакета путем анализа поля протокола в IP-заголовке и номеров портов источника и места назначения в TCP-заголовке. Однако выделить весь HTTP-трафик достаточно трудно. Некоторые Web-сайты не используют данный порт, а порты 8000 или 8080. Кроме того, некоторые приложения могут взаимодействовать через порт 80, используя другой протокол. Хотя порты выделяются определенным приложениям, разработчик приложения или системный администратор могут игнорировать эти указания и задействовать порт 80 для приложения, использующего другой протокол.

Кроме того, в Web имеется протокол HTTPS, использующий порт 443. Монитор пакетов может захватить HTTPS-трафик путем отслеживания TCP-трафика, содержащего в качестве номера порта источника и места назначения порт 443. Однако данные, передаваемые через HTTPS, зашифрованы с применением Secure Socket Layer (SSL). Монитор не может идентифицировать HTTP-сообщения, передаваемые через эти TCP-соединения. Однако может перехватывать TCP-трафик на порту 443 с целью расчета основных статистических показателей, таких как число соединений и байтов, передаваемых с помощью HTTPS. Другие передачи, инициированные Web-браузером, могут использовать другие протоколы прикладного уровня. В некоторых случаях этот трафик может перехватываться путем мониторинга портов, выделенных этим протоколам. Однако ряд приложений использует протоколы, которые динамически назначают номера портов при передаче данных. Например, клиент и сервер File Transfer Protocol (FTP) могут динамически выбирать номер порта для TCP-соединения, используемого для передачи данных. Аналогично многие протоколы для передачи мультимедийных потоков обычно не используют определенные номера портов для передачи данных аудио и видео, о чем подробнее будет говориться далее в разделе 14.4.4.

Перехват трафика с определенными IP- и TCP-заголовками требует применения фильтра к IP-пакетам по мере их поступления. Как можно более ранняя фильтрация пакетов сокращает издержки, связанные с копированием и обработкой пакетов, которые будут затем отвергнуты. Монитор пакетов может иметь специальное аппаратное обеспечение, которое осуществляет классификацию пакетов при их приеме сетевой картой. Если аппаратная поддержка отсутствует, пакеты могут отфильтровываться операционной системой или приложением. Большое число программ мониторинга пакетов основывается на инструментальном средстве *tcpdump*

[JLM, Tcp] на основе фильтра пакетов Berkeley Packet Filter (BPF) [MJ93]. BPF-фильтры пакетов анализируют комбинации полей заголовков IP и TCP/UDP, а также сохраняют фиксированное число байтов для каждого пакета, прошедшего через фильтр. Например, BPF может быть настроен так, чтобы перехватывать первые 40 байтов всех пакетов, использующих TCP на порту 80 в качестве источника или назначения, либо UDP-пакетов с портом назначения 23. Программное обеспечение BPF может быть установлено на большом числе платформ.

Для расчета основных статистических показателей Web-трафика может оказаться достаточным хранить заголовки IP и TCP. Однако для просмотра HTTP-сообщений в TCP-соединении требуется иметь доступ к данным, следующим после заголовков IP и TCP. HTTP-сообщения могут быть разбиты на IP-пакеты различными способами. Следовательно, информация на уровне HTTP может размещаться в любом IP-пакете TCP-соединения, как показано на рис. 14.2. Первое сообщение HTTP-запроса или ответа в соединении начинается с первого пакета, передаваемого после трехэтапного установления соединения между двумя компьютерами. Однако HTTP-заголовок может размещаться в нескольких пакетах. Кроме того, по одному долговременному соединению могут передаваться несколько HTTP-сообщений. HTTP-заголовки могут встретиться в любом месте TCP-соединения. Фактически заголовок может начинаться даже в середине IP-пакета. Таким образом, перехват каждой из HTTP-передач требует перехвата и анализа содержимого IP-пакетов.

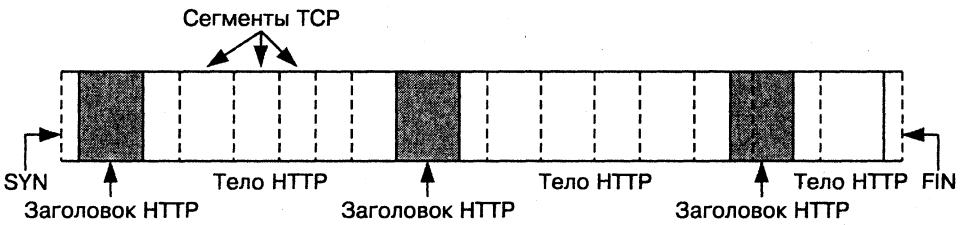


Рис. 14.2. HTTP-сообщения, состоящие из последовательности TCP-сегментов

Мониторинг пакетов HTTP-трафика предусматривает перехват всего содержимого каждого IP-пакета, который содержит TCP-сегмент с портом 80 для источника или получателя. В идеале монитор захватывает каждый такой пакет. Однако монитор может потерять пакеты, если сетевой интерфейс или операционная система не может справиться с большим объемом поступающих пакетов. При перехвате и фильтрации пакетов монитор может ассоциировать каждый пакет с меткой времени, которая указывает, когда пакет был получен монитором. Для защиты конфиденциальности пользователей монитор может скрывать или зашифровывать IP-адреса источника и получателя захваченных пакетов; однако это не препятствует доступу к частной информации в HTTP-заголовках и телах сообщений. Пройдя через стадии перехвата и фильтрации, пакет может быть сохранен в оперативной памяти или на диске для последующей обработки. По мере восстановления HTTP-сообщений из перехваченного трафика монитор может освободить область хранения, отвергнув некоторые IP-пакеты. Монитор пакетов должен эффективно использовать функциональные возможности подсистемы хранения, чтобы с максимальной скоростью перехватывать запросы и ответы HTTP.

14.1.3. Демультиплексирование пакетов

Любой стандартный канал в Internet обычно передает трафик в интересах нескольких TCP-соединений. Монитор должен иметь эффективный способ связывания каждого пакета с соответствующим TCP-соединением. Однако монитор пакетов не является источником или конечной точкой TCP-соединения и может не видеть все пакеты, передаваемые между двумя хостами. Вместо этого монитор должен *подразумевать* наличие TCP-соединения, исходя из информации в заголовках IP- и TCP-пакета. TCP-соединение идентифицируется по IP-адресам и номерам портов источника и получателя. Монитор может группировать пакеты, имеющие одни и те же IP-адреса и номера портов источника и получателя. Для того чтобы различить такую группу пакетов от соответствующего TCP-соединения, мы будем называть группу пакетов *потоком*. Пакеты с портом назначения 80 являются составной частью потока запросов от Web-клиента, а пакеты с портом 80 источника составляют часть потока ответов к Web-клиенту.

Чтобы демультиплексировать пакеты, монитор может использовать хэш-таблицу потоков, как показано на рис. 14.3. Предполагая, что Web-сервер использует порт 80, монитор может воспользоваться ключом, представляющим собой сцепление трех полей: IP-адреса клиента, IP-адреса сервера и номера порта клиента. Каждая запись в хэш-таблице соответствует двум потокам: потоку запросов и потоку ответов. Поступающий пакет ассоциируется с ключом и направлением на основе IP-адресов и номеров портов. В зависимости от своего местоположения монитор может не видеть все пакеты и из потока запросов, и из потока ответов. Если монитор располагается непосредственно перед компьютером клиента или сервером, отслеживаются все пакеты, передаваемые по сети между двумя компьютерами. Это может быть не так для каналов на пути между клиентом и сервером. IP-маршрутизация не гарантирует, что все пакеты между одним и тем же источником и получателем проходят по одному пути, а трафик от клиента к серверу может не проходить по тем же каналам, по которым он идет от сервера к клиенту.

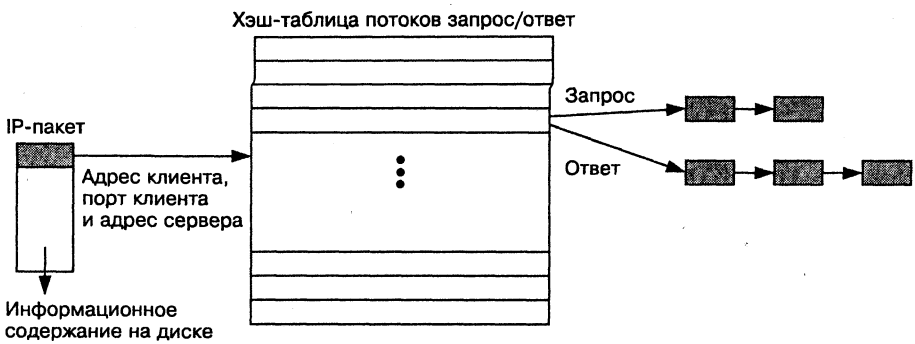


Рис. 14.3. Хэш-таблица для демультиплексирования пакетов на потоки

Следовательно, нельзя полагаться на то, что монитор сможет увидеть все пакеты, передаваемые между двумя компьютерами. На деле, если даже каждый пакет проходит по каналу, монитор может потерять некоторые из пакетов при высокой нагрузке. Необходимо эффективное проектирование монитора пакетов, чтобы избежать потерь пакетов вследствие недостатка оперативной памяти или дискового пространства. С этой целью монитор может присваивать наивысший приоритет не-

рехвату и фильтрации пакетов. Другим задачам, таким как демультиплексирование пакетов на потоки, может быть присвоен более низкий приоритет. Например, монитор может копировать пакеты в память или на диск после их захвата и фильтрации, а второй, менее приоритетный процесс, может выполнять остальные задачи. Для дальнейшего повышения эффективности программное обеспечение, выполняющая эти задачи, по мере возможности должно избегать чтения или копирования содержимого IP-пакетов. Пакеты демультиплексируются только на основе данных из заголовков IP и TCP, но не на основе содержимого пакетов. Доступ к остальной части пакета может быть отложен до следующего этапа либо вообще не осуществляться, о чем пойдет речь ниже.

14.1.4. Восстановление упорядоченного потока

При обработке пакетов в составе потока монитор должен выполнять многие функции получателя TCP-пакетов. В частности монитор должен обрабатывать полученные вне естественного порядка, поврежденные и повторяющиеся пакеты. IP не гарантирует надежной, упорядоченной доставки пакетов, следовательно, пакеты могут быть повреждены или повторяться. Кроме того, монитор может получать повторные пакеты, появившиеся в результате повторной передачи на стороне отправителя TCP-пакетов. Отправитель может ошибочно предположить, что пакет был утерян, и передать данные еще раз. Кроме того, первоначальный пакет может быть утерян где-то между отслеживаемым каналом и TCP-получателем. Таким образом монитор может перехватить пакет, который никогда не достигнет предполагаемого получателя. В любом случае монитор может получить повторные пакеты. Еще более осложняет ситуацию то, что TCP не гарантирует, что повторно передаваемые данные будут сформированы в пакеты таким же образом, как первоначальные данные. Следовательно, монитор не сможет легко обнаружить и удалить пакеты-дубликаты. Вместо этого монитор должен выявлять повторяющиеся области байтов в TCP-потоке.

Как и TCP-получатель, монитор пакетов может использовать информацию TCP-заголовков, чтобы сформировать упорядоченный поток байтов из последовательности пакетов. Поврежденные пакеты могут выявляться и удаляться на основе анализа поля контрольной суммы в TCP-заголовке. Поступившие вне естественного порядка пакеты могут быть переупорядочены на основе значения поля порядкового номера в TCP-заголовке. Повторяющиеся данные могут быть выявлены на основе порядковых номеров и длин сегментов. Например, рассмотрим два пакета с одинаковыми адресами источника, получателя и номерами портов. При этом один пакет имеет порядковый номер 50 и длину 10, а другой пакет имеет порядковый номер 55 и длину 20. Последние пять байтов в первом пакете перекрываются с первыми пятью байтами второго пакета. При обработке данных в этом потоке монитор должен учесть эти пять байтов один раз, отвергнув последние пять байтов первого пакета или первые пять байтов второго пакета. В других случаях, когда один пакет полностью поглощен другим, монитор может отвергнуть избыточный пакет.

В определенный момент монитору потребуется определить, что поток завершен. Возможность поступления пакетов не в том порядке и повторная передача пакетов затрудняют этот процесс. Поток считается завершенным, если он имеет начало, конец и все, что находится между ними. Таким образом, поток должен представлять собой законченное TCP-соединение с пакетом SYN, пакетом FIN или RST и полным объемом данных. Количество байтов, ожидаемое между SYN и FIN/RST, может быть определено на основе порядковых номеров в пакетах SYN и FIN/RST. Однако поток может не удовлетворять этим трем критериям. Монитор мог пропус-

тить пакеты, некоторые пакеты могли проходить по пути, который не включает в себя отслеживаемый канал, какая-либо причина могла заставить TCP-соединение закончиться до того, как будут повторно переданы все потерянные пакеты. В самом крайнем случае TCP-отправитель мог прекратить работу (по причине фатального сбоя) до завершения соединения, в результате чего поток не будет содержать пакетов FIN или RST. В любом из этих сценариев монитор пакетов может быть не в состоянии успешно завершить восстановление упорядоченного, надежного байтового потока на основе исходного потока.

Незавершенные потоки потребляют ресурсы монитора пакетов. Кроме того, в определенный момент в будущем другое TCP-соединение между этими же компьютерами может использовать те же номера портов. Это может заставить монитор ошибочно связать пакеты нового TCP-соединения с потоком старого TCP-соединения. Чтобы этого не случилось, монитор должен прекратить ожидание поступления дополнительных пакетов в поток. Наиболее просто это реализуется в схеме на основе таймаутов. Если поток не получил каких-либо новых пакетов в течение определенного периода времени, монитор может предположить, что TCP-соединение закончено. Выбор соответствующего интервала времени — непростая задача. Большое время таймаута обусловливает высокие требования к ресурсам монитора пакетов и увеличивает вероятность, что новое TCP-соединение продолжит поток закрытого TCP-соединения. В то же время малое значение таймаута повышает вероятность того, что поток будет завершен, когда TCP-соединение еще остается активным. Например, долговременное соединение может находиться в состоянии простоя несколько десятков секунд между последовательными запросами или ответами HTTP.

Стратегия выбора значений таймаута для потоков зависит от деталей реализации TCP и HTTP. Значение таймаута для потока должно превышать допустимые значения любых таймеров, используемых для управления передачей пакетов в TCP и для закрытия бездействующих TCP-соединений клиентами и серверами HTTP. Исследование данных измерений и серверного программного обеспечения могут способствовать выработке рекомендаций по правильному выбору значений таймаута. На практике монитор может завершать бездействующие потоки, периодически проверяя записи в хэш-таблице. Для каждой записи монитор может определять время, прошедшее с момента поступления последнего пакета. Поток завершается, если время превысило заранее установленное пороговое значение. Как и при демультиплексировании пакетов на потоки, восстановление и завершение потоков не требует чтения или доступа к данным в сегментах TCP.

14.1.5. Извлечение HTTP-сообщений

Извлечение HTTP-сообщений из потока требует от монитора просмотра последовательности байтов. При обсуждении процесса извлечения мы изначально предполагаем, что поток был полностью восстановлен. Далее мы обсудим, как учесть потерянные пакеты и как совместить извлечение HTTP-сообщений с восстановлением упорядоченного байтового потока. Каждое HTTP-сообщение состоит из заголовка и необязательного тела, а каждый поток состоит из одного или нескольких HTTP-сообщений, как показано на рис. 14.2. Обработка потока требует идентификации заголовка сообщения и обнаружения начала следующего сообщения. Поток должен начинаться с корректного HTTP-заголовка. Например, поток запроса должен начинаться с метода запроса, запрашиваемого URI и версии протокола; поток ответа должен начинаться с версии протокола и кода ответа. Заголовок должен за-

канчиваться дважды повторяющейся последовательностью символов возврата каретки и перевода строки (CRLF), вслед за которыми идет необязательное тело сообщения.

Выявление границы между двумя последовательными сообщениями зависит от параметров первого сообщения на уровне HTTP. При восстановлении сообщения-ответа монитор пакетов должен иметь дело с теми же ситуациями, что и любой другой компонент HTTP/1.1, а именно:

- **Отсутствие тела сообщения.** В некоторых случаях монитор может предположить, что сообщение не содержит тела, а также следующее сообщение начинается сразу же после двойного CRLF. Например, запросы **GET** и ответы **304 Not Modified** не содержат тела сообщения.
- **Наличие поля Content-Length.** Некоторые сообщения содержат заголовок **Content-Length**, который указывает размер тела сообщения. Это позволяет монитору идентифицировать место в байтовом потоке, с которого начинается следующее сообщение, не просматривая байты в теле первого сообщения.
- **Сообщение, разбитое на фрагменты.** Некоторые HTTP-сообщения содержат метаданные в теле сообщения. При этом содержимое HTTP-сообщения делится на несколько фрагментов, как описывалось ранее в главе 7 (раздел 7.6). Каждый фрагмент содержит заголовок фрагмента, указывающий его длину. Определение конца сообщения требует от монитора учитывать каждый заголовок фрагмента для выявления начала следующего фрагмента. Сообщение заканчивается фрагментом нулевой длины.
- **Multipart/byteranges.** Сообщение-ответ с заголовком типа содержимого **Content-Type:multipart/byteranges** имеет в теле сообщения несколько диапазонов данных, о чем говорилось ранее в главе 7 (раздел 7.4.1). Заголовок HTTP-ответа идентифицирует разграничитель, который помечает начало каждого байтового диапазона в теле. Монитор может выявить начало каждого диапазона и перейти к началу следующего диапазона подобно тому, как это делается в сообщении с разбиением на фрагменты.
- **Конец TCP-соединения.** Последний способ обнаружения монитором конца сообщения-ответа — это выявить, что TCP-соединение было завершено пакетами FIN или RST. Это может случиться, если сервер использует закрытие TCP-соединения для указания конца тела ответа, либо если клиент или сервер прервали TCP-соединение.

Введение в HTTP/1.1 разбиения на фрагменты и сообщений, состоящих из нескольких разделов, усложняет идентификацию границ сообщения, поскольку требует от монитора пакетов просмотра тела сообщения. Для трафика HTTP/1.0 в этом нет необходимости. Кроме того, граница между двумя сообщениями может проходить внутри одного IP-пакета из-за конвейеризации.

После определения конца одного сообщения монитор может начать извлечение следующего сообщения. Процесс повторяется до тех пор, пока все сообщения не будут извлечены из потока. Однако потерянные пакеты вносят дополнительные сложности в процесс извлечения. В действительности, в ряде случаев потерянный пакет может сделать невозможным продолжение обработки потока. Например, восстановление при отсутствии заголовка **Content-Length** или дважды повторяющейся последовательности CRLF может оказаться затруднительным, либо вообще невозможным, поскольку монитор не будет знать, как идентифицировать конец тела данного сообщения и начало следующего сообщения. Утеря заголовка фрагмента

или заголовка раздела внутри тела сообщения затруднит определение конца сообщения. Столкнувшись с такого рода потерями пакетов, монитор может просто записать сообщение об ошибке для указания, что оставшаяся часть потока не может быть обработана. Иметь низкую долю потерянных пакетов чрезвычайно важно. В противном случае значительная часть потоков будет иметь одну или несколько невосполнимых потерь. Вероятность потери пакета зависит от скорости передачи данных и загруженности канала, а также от эффективности работы программных и аппаратных компонентов монитора пакетов.

В то время как потеря пакета с HTTP-заголовком порождает существенные проблемы, потеря пакета в середине тела содержимого не обязательно нарушает обработку потока. Потеря нескольких сотен байтов в большом документе в формате Postscript не мешает монитору находить следующее сообщение в потоке. На самом деле обработка тела содержимого может вообще быть не нужна. Если монитор записывает тело содержимого или статистические данные, вычисленные на основе тела содержимого, то потеря пакета не даст монитору возможности получить информацию для этого запроса. Тем не менее, монитор может без особых затруднений перейти к следующему сообщению в потоке. Если монитор регистрирует только HTTP-заголовки, тело содержимого вообще не подлежит обработке. В этом случае монитор может избежать дополнительных затрат, связанных с чтением содержимого большинства захваченных IP-пакетов. Типовое сообщение HTTP-ответа содержит около 300 байтов в заголовке и около 10 килобайтов в теле сообщения. Игнорирование тела сообщения избавляет от необходимости просматривать примерно 97% перехваченных данных.

14.1.6. Создание HTTP-трасс

В конечном счете, монитор может записывать самую разнообразную информацию об HTTP-трафике. Соответствующий набор полей зависит от того, для какого вида анализа будут использоваться эти записанные значения (трассы). В противоположность традиционным журналам прокси-серверов и Web-серверов, трасса пакетов может содержать информацию на нескольких протокольных уровнях (IP, TCP, HTTP), а также метки времени для ключевых этапов каждой HTTP-транзакции. Образцы полей приведены в таблице 14.1. Для форматов трасс измерений Web-трафика на пакетном уровне не существует никаких официальных или неофициальных стандартов. В связи с этим трасса пакетов может содержать практически любую информацию, которая может быть извлечена из исходных данных. Однако извлечение, запись и хранение полей создает нагрузку на монитор пакетов, что может сказываться на перехвате и восстановлении HTTP-трафика. Нахождение оптимального компромисса зависит от объема трафика в канале, а также от быстродействия платформы, на которой реализован монитор и от эффективности аппаратных средств восстановления.

На уровне IP и TCP монитор может записывать IP-адреса клиента и сервера, а также число байтов и пакетов, переданных в каждом направлении. TCP-соединение начинается с трехэтапного обмена, состоящего из клиентского SYN, серверного SYN-ACK и клиентского ACK. Метки времени для двух пакетов SYN помечают начало каждого направления TCP-соединения. Аналогично, метки времени пакетов FIN или RST в каждом из направлений помечают конец соединения. Отслеживание, заканчивается ли соединение пакетом FIN или RST, дает возможность идентифицировать прерванные передачи. Кроме того, трасса может идентифицировать, сталкивался ли монитор с потерей пакетов для данного TCP-соединения, и не на-

рушила ли потеря пакетов процесс восстановления. Помимо полей, приведенных в таблице 14.1, монитор может записывать размер и метку времени для каждого пакета в TCP-соединении. Эти сведения могут оказаться полезными для определенных задач анализа, но при этом потребуется запись и хранение дополнительной информации.

Таблица 14.1. Примеры полей в трассе пакетов

Категория	Примеры
TCP/IP	IP-адреса клиента и сервера
	Количество переданных пакетов и байтов
	Метка времени для начала TCP-соединения (SYN)
	Метка времени для конца TCP-соединения (FIN/RST)
	Способ завершения потока (FIN или RST)
	Ошибки (отсутствующий пакет в составе потока)
HTTP	Заголовок запроса HTTP
	Заголовок ответа HTTP
	Метка времени начала HTTP-заголовка
	Метка времени начала тела HTTP
	Метка времени конца тела HTTP
	Ошибки (сообщения с некорректным синтаксисом HTTP)
Содержимое	Контрольная сумма тела содержимого
	Встроенные ссылки

На уровне HTTP монитор пакетов может записывать заголовки запроса и ответа HTTP. Выборочная регистрация подмножества заголовков сократит затраты на запись, хранение и анализ трассы. Оптимальный выбор зависит от компромисса между затратами на сбор данных и желанием анализировать данные разными способами. Каждое HTTP-сообщение занимает определенную часть байтового потока базового TCP-соединения, как было показано ранее на рис. 14.2. Поскольку одно TCP-соединение может обслуживать несколько HTTP-передач, метки времени начала и конца TCP-соединения не обязательно являются хорошими показателями для определения времени отправки каждого HTTP-сообщения. Монитор пакетов может регистрировать метки времени для основных частей HTTP-сообщения: начала заголовка, начала тела сообщения и конца тела сообщения. Чтобы определить эти значения времени, программное обеспечение монитора должно принимать во внимание метки времени, ассоциированные с соответствующими IP-пакетами.

Некоторые HTTP-сообщения содержат тело содержимого. Теоретически монитор пакетов может записывать тело каждого HTTP-сообщения. Однако это приведет к значительным затратам, связанным с записью и хранением данных, поскольку тело сообщения часто имеет гораздо больший размер, чем HTTP-заголовок. Тем не менее, информация о содержимом может оказаться полезной при проведении анализа. Например, контрольная сумма содержимого может использоваться для определения, изменился ли Web-ресурс с последнего обращения к нему. Помимо этого, регистрация гипертекстовых ссылок и встроенных изображений в HTML-файле будет полез-

ной для определения Web-ресурсов, связанных с Web-страницей. Монитор может избирательно записывать тело содержимого, либо информацию о теле содержимого, в зависимости от информации в HTTP-заголовке. Например, просмотр заголовка **Content-Type** позволит монитору идентифицировать HTML-ресурсы, которые могут иметь гипертекстовые ссылки и встроенные ресурсы. Кроме того, монитор может с помощью синтаксического анализатора HTML извлечь URL и записать их в трассу. Или же монитор может записать весь HTML-файл и отложить синтаксический анализ данных на более позднее время.

14.2. Анализ журналов Web-серверов

Необходимость обработки больших и разнообразных наборов данных измерений порождает ряд практических проблем, о чем уже говорилось в главе 9 (раздел 9.4). В этом разделе представлен обзор технологий, связанных с обработкой журналов Web-серверов. Особое внимание при этом уделяется ключевым программным компонентам, используемым для синтаксического анализа, фильтрации, преобразования и анализа данных [KR98].

14.2.1. Синтаксический анализ и фильтрация

При оценке новых методов повышения эффективности Web требуется особое внимание уделять сбору данных о рабочих нагрузках компонентов Web. Однако журналы Web-серверов могут содержать ошибки и несоответствия, которые осложняют анализ данных. Кроме того, результаты анализа любого серверного журнала могут быть недостаточно репрезентативны, чтобы оценить по ним ожидаемую производительность других серверов. Анализ множества журналов различных Web-сайтов помогает отделить общие тенденции, связанные с производительностью, от характеристик трафика, свойственных конкретному сайту. На первый взгляд, выполнение одного и того же анализа для множества серверных журналов не труднее, чем анализ одного журнала. Однако Web-серверы ведут журналы в различных форматах с разным количеством и типами полей, о чем говорилось в главе 9 (раздел 9.3). Часы различных серверов обычно не синхронизированы друг с другом, что приводит к разночтению меток времени из различных журналов. Кроме того, синтаксис полей может варьироваться от одного сайта к другому. Преобразование серверных журналов к универсальному формату — эффективный способ скрыть эти детали от программного обеспечения, осуществляющего анализ данных.

Синтаксический анализ журнала — необходимый начальный этап. Он требует идентификации формата записей применительно к количеству полей и их синтаксису. На основе управляющей строки, которая задает формат записи, программа считывает каждую строку ввода и назначает для каждого фрагмента данных переменную определенного типа. Например, коду ответа HTTP в серверном журнале может быть назначена целочисленная переменная, в то время как для метода запроса HTTP может использоваться переменная строкового типа. Однако на практике серверные журналы могут содержать ошибочные записи, которые приводят к ошибкам при синтаксическом анализе входных данных. Запись может не иметь необходимого числа полей, или же в полях могут быть синтаксические ошибки. Программные средства обработки ошибок являются важной частью анализа записей в журнале. Программа может проверять число полей во входной строке и соответствие каждого поля требуемому синтаксису.

Хотя большинство полей и записей выводятся корректно, изредка могут возникать проблемы, связанные с конкретной реализацией Web-сервера. Некоторые Web-серверы порождают несколько процессов для одновременной обработки HTTP-запросов, о чем говорилось в главе 4 (раздел 4.4). Эти процессы конкурируют при записи данных в журнал сервера. Если сервер не осуществляет специальных мер, некоторые записи могут накладываться друг на друга. Несмотря на возможные ошибки, сервер может разрешить процессам осуществлять запись независимо, чтобы избежать затрат, связанных с координацией операций записи. Корректное формирование серверных журналов не обязательно является существенно важным при обработке HTTP-запросов. Даже если сервер пытается упорядочить операции регистрации в журнале, ошибки при реализации могут привести к появлению спорадических ошибок. Аналогично, в ряде случаев сервер может не присваивать значений всем без исключения полям в записи, что способно привести к ошибочному результату. Как следствие, большие серверные журналы часто содержат ошибки. Программа, осуществляющая синтаксический анализ журналов, должна проверять записи на наличие ошибок.

Записи, которые приводят к ошибкам в ходе синтаксического анализа, могут быть просмотрены вручную, или же программа может автоматически пропускать запись и переходить к следующей. В некоторых случаях более тщательное изучение записи может выявить источник проблемы. Например, URL может содержать символ новой строки (ctrl-M), который приводит к ошибочному обнаружению конца строки синтаксическим анализатором. Запись может быть исправлена путем удаления символа новой строки. Другая проблема связана с длиной URL. URL может иметь неограниченно большую длину. В связи с этим любое предположение о максимальной длине URL может оказаться неверным; на практике иногда встречаются URL длиной более 4000 символов. В некоторых языках программирования процедуры стандартного ввода/вывода подразумевают, что память для каждой входной переменной выделяется заранее (например, функция *scanf()* в языке программирования C). Если длина URL превышает ожидаемую, процедура синтаксического анализа осуществит запись за пределы выделенного пространства памяти, что приведет к повреждению других данных в памяти. Ошибки подобного рода особенно трудно обнаружить и исправить. Чтобы не допустить возникновения таких проблем, программа может выделять пространство для каждой строки в процессе синтаксического анализа записи. Альтернативой является наличие в программе отдельной процедуры для обработки длинных URL в случае их обнаружения.

Другая практическая проблема связана с различием формата полей меток времени в различных серверных журналах. При анализе набора реальных серверных журналов нами было выявлено не менее 14 различных способов представления даты и времени. Создание программы для синтаксического анализа и интерпретации различных форматов — хлопотное дело. Кроме того, такая программа, скорее всего, будет иметь внутренние ошибки. Очень полезно иметь отдельную библиотеку процедур для синтаксического анализа и преобразования полей меток времени в серверных журналах. Такие процедуры могут быть тщательно отлажены и многократно использоваться множеством различных приложений. Кроме того, процедура может преобразовывать метку времени к универсальному представлению времени, соответствующему, например, количеству секунд, прошедших с полуночи 1 января 1970 г. (представление времени в UNIX). Это позволит остальным программам игнорировать различия в представлении времени в различных журналах. Однако это не решает проблем, связанных с различием показаний часов для различных Web-серверов, которые создают эти журналы. На практике довольно трудно установить соотноше-

ние значений времени между различными журналами. Кроме того, назначение меток времени в журнале может быть различным для различных серверов, о чем говорилось ранее в главе 9 (раздел 9.3.1).

После синтаксического анализа записи и проверки синтаксических ошибок, каждое поле просматривается на предмет семантических ошибок. Каждому полю должен соответствовать определенный диапазон значений. Например, коды ответов HTTP должны представлять собой трехзначные целые числа, начинающиеся с 1, 2, 3, 4 или 5, а все метки времени должны лежать внутри интервала времени, соответствующего времени существования журнала. Записи с неправильными полями могут быть просмотрены вручную и удалены. Тщательный контроль ошибок также упрощает процесс написания и отладки программ для анализа измеренных данных. Помимо этого, операции над журналами в универсальном формате позволяют таким программам игнорировать различия в исходном представлении каждого серверного журнала. Например, для анализа может быть использована усеченная версия журнала, содержащая имя клиента (IP-адрес), запрашиваемый URI, код ответа и время запроса.

14.2.2. Преобразования

Удаление ошибочных записей и преобразование в универсальный формат решает некоторые проблемы при обработке серверных журналов. Однако написание анализирующих программ, которые оперируют строками URL, — весьма утомительное занятие. Для URL допустимо множество форматов, в результате чего один и тот же ресурс в Web может иметь множество представлений. Рассмотрим ресурс **foo.html**, размещенный на Web-сайте **www.xyz.com**. Записи в журнале для этого ресурса могут иметь множество различных URL, таких как **http://www.xyz.com/foo.html**, **www.xyz.com/foo.html**, **foo.html** или **www.xyz.com///foo.html**. Преобразование этого URL к канонической форме требует манипулирования строками для поиска и удаления символов. Такого рода операции обычно довольно утомительно программировать на таких языках, как C, хотя на языках сценариев это сделать достаточно просто. Наличие отдельной библиотеки процедур для синтаксического анализа и работы со строками может избавить от хлопот, связанных с написанием программ для обработки журналов.

URL может быть приведен к сжатому, каноническому формату путем выполнения операций над строками, имитирующих преобразования, выполняемые Web-сервером при обработке запроса. Например:

- Удаление "http://" и доменного имени компьютера, если оно присутствует. В противном случае анализирующая программа может не распознать, что **http://www.xyz.com/foo.html** и **foo.html** ссылаются на один и тот же ресурс. Имя сервера может иметь псевдоним, это означает, что **http://www.xyz.com/foo.html** и **http://xyz.com/foo.html** относятся к одному и тому же ресурсу.
- Двойные символы "/" могут быть удалены. Например, **/bar//foo.html** может быть преобразовано в **/bar/foo.html**.
- Могут быть извлечены составные части URL. Например, суффикс URL часто идентифицирует тип содержимого ресурса (например, **.html** или **.gif**). Идентификация ресурсов, которые расположены по одному и тому же пути, предусматривает удаление имени ресурса в конце URL; например, **/files/bar/foo.html** имеет путь **/files/bar**.

Хотя многие URL могут быть преобразованы указанным образом, в некоторых случаях для этого требуется дополнительная информация о настройках сервера. Например, **http://www.xyz.com/bar** может ссылаться на ресурс **bar** или же соответствовать документу **http://www.xyz.com/bar/index.html**, если **bar** — каталог. Кроме того, сервер может быть настроен таким образом, что один URL трактуется как псевдоним другого URL. Распознать, что два URL соответствуют одному и тому же ресурсу, при отсутствии информации о настройках сервера будет затруднительно. Для обработки доменного имени обратившегося с запросом клиента также полезны операции со строками. Например, извлечение домена верхнего уровня в доменном имени компьютера пригодится для идентификации запросов, поступивших от образовательных учреждений (например, **.edu** в **users.berkley.edu**).

Для большинства задач анализа знание конкретных URL и имен клиентов или IP-адресов необязательно. Может оказаться достаточным знать, какие записи имеют одни и те же URL или относятся к одному клиенту. Для упрощения анализа поля, относящиеся к URL и клиенту, могут быть преобразованы в целочисленные значения. Тем самым снижаются требования к памяти при хранении данных и исчезает необходимость иметь дело со строками переменной длины для остальных программных компонентов. Кроме того, целочисленное представление предотвращает раскрытие информации о пользователях. Преобразование строк в целые числа может быть выполнено за один проход журнала, вместе с приведением URL к универсальному формату. Программа ассоциирует целое число с каждым URL в порядке их вхождений, выстраивая хэш-таблицу. Отдельная хэш-таблица может быть использована для связывания каждого доменного имени клиента или IP-адреса с уникальным целым числом.

Хэш-таблица хранит целое число, ассоциированное с каждым преобразованным к каноническому виду URL. Для каждой новой записи программа проверяет, существует ли в хэш-таблице приведенный к каноническому виду URL, чтобы определить, какое целое число соответствует этому ресурсу. В противном случае программа ассоциирует новый URL со следующим по порядку целым числом и помещает запись в хэш-таблицу. Для обработки журналов с большим числом различных URL требуется эффективная реализация хэш-таблиц. Достаточно часто встречаются журналы Web-серверов с десятками тысяч уникальных URL, а количество уникальных клиентов может быть еще больше. Хэш-таблицы, созданные с помощью языков сценариев, таких как Perl и ему подобных, обычно требуют много оперативной памяти. Если потребность в памяти превышает доступный объем, операционная система периодически осуществляет свопинг части хэш-таблицы на диск или с диска. Это может привести к значительным задержкам при обработке журнала. Эффективные процедуры хэширования на таких языках, как C или Java, обычно требуют гораздо меньше памяти, и преобразование данных осуществляется гораздо быстрее.

После преобразования клиентских имен и URL в целые числа каждая запись состоит из небольшого числа целых чисел. Анализ журналов предполагает последовательный просмотр запросов для вычисления определенных показателей, моделирования определенной стратегии поведения или механизма. В некоторых случаях анализирующая процедура трактует каждого клиента или каждый ресурс независимо. Например, рассмотрим задачу определения интервала времени между последовательными обращениями клиента. Естественный подход заключается в последовательном просмотре журнала по одному запросу за раз и обновлении информации об обратившемся с запросом клиенте. Однако это требует выделения памяти для каждого клиента. Альтернатива состоит в переупорядочении журнала

с целью группировки запросов по клиентам. Переупорядоченный журнал будет содержать все запросы клиента 1, за которыми следуют все запросы клиента 2, и т.д. Сортировка журнала по идентификаторам клиентов намного упрощает задачу подсчета статистических показателей. В то же время, в ряде случаев более полезной может оказаться сортировка по идентификаторам ресурсов.

Сортировка может выполняться по нескольким полям журнала. Например, изучение поведения клиентов может потребовать сортировки записей по идентификаторам клиентов с разбивкой по времени выдачи запроса. Это гарантирует, что полученный в результате журнал будет содержать запросы от каждого из клиентов в порядке поступления запросов. Однако в некоторых случаях несколько записей могут иметь один и тот же идентификатор клиента и одну и ту же отметку времени. Например, клиент может выдать несколько запросов на загрузку HTML-файла и встроенных изображений. Эти запросы поступают близко во времени, возможно, в одну и ту же секунду. Серверные журналы обычно фиксируют время поступления запроса с точностью до секунды. В неупорядоченном журнале записи, ассоциированные с этими запросами, появляются в том порядке, в котором они были записаны Web-сервером. После проведения сортировки журнал должен сохранять этот порядок. Обеспечить, чтобы процедура упорядочения выполняла такую «стабильную» сортировку, весьма важно для избежания изменений порядка расположения таких записей в журнале.

Предварительная сортировка записей позволяет значительно уменьшить сложность анализирующей программы и снизить требования к оперативной памяти. Кроме того, отсортированный журнал может быть использован для различных задач анализа, что позволяет разложить начальные затраты на сортировку на эти задачи. Поскольку сортировка является основной задачей для различных приложений, процедура сортировки должна быть оптимизирована. Во многих ситуациях сортировка данных оказывается более эффективной, чем использование сложной анализирующей программы, написанной «с нуля». Самое важное, что предварительная сортировка данных может уменьшить время, необходимое для написания анализирующей программы, а также облегчить проведение многих экспериментов на ранних этапах анализа данных измерений.

Резюмируя, можно сказать, что анализ больших серверных журналов различных Web-серверов требует решения нескольких проблем, связанных с программным обеспечением. Предварительная обработка данных с целью удаления записей с ошибками, нежелательных полей, а также преобразование строк в целые числа, существенно уменьшает сложность анализирующего программного обеспечения. Кроме того, сортировка записей журнала по одному или нескольким полям может значительно облегчить выполнение задач анализа. Программа предварительной обработки и анализа данных может предусматривать эффективную поддержку чтения и записи файлов, работу с регулярными выражениями, преобразование форматов представления времени, хэширование и сортировку. Использование эффективных реализаций этих функций позволяет сэкономить усилия, связанные с разработкой, отладкой и оптимизацией программного обеспечения. Например, в нашей работе по анализу журналов Web-серверов главным образом использовались библиотеки *sfio* (safe/fast I/O) и *libast* [KV91, FKV95], которые позволили написать эффективные и надежные программы на C для предварительной обработки и анализа измеренных данных.

14.3. Общедоступные журналы и трассы

Многие исследования параметров Web-трафика основываются на общедоступных коллекциях журналов клиентов, прокси-серверов и серверов, а также трасс пакетов. Было предпринято несколько попыток сформировать централизованное хранилище журналов и трасс. В их числе Internet Traffic Archive [ITA], хранилище, созданное группой Web Characterization Group консорциума World Wide Web Consortium [WCA], коллекция NLANR [IRC] и CAnet Squid Logs [Can].

Журналы обычно предоставляются «как есть» — в форме, в которой они были записаны прокси-сервером или Web-сервером. В некоторых случаях журналы могут предоставляться в сжатом виде, чтобы уменьшить требования к объему памяти. Такие поля, как имя/адрес обратившегося с запросом клиента, могут быть удалены или преобразованы, чтобы сохранить конфиденциальность. Организация, предоставляющая журналы, может не иметь серьезного стимула для преобразования журналов в стандартный формат и для выявления явных несоответствий в записях измеренных данных. Пользователям данных приходится расплачиваться за применение недостаточно подготовленных данных; о возникающих при этом проблемах упоминалось в разделе 14.2. Альтернативой является выполнение пользователями различных семантических проверок, которые диктуются конкретным применением данных. Однако проверки, выполняемые после того, как пользователь загрузит журнал, не принесут пользы всем остальным лицам, использующим эти же данные из хранилища.

В идеале журналы должны проверяться на наличие ошибок *до* того, как они станут доступными. Помимо предоставления доступа к журналам, хранилище может содержать список процедур предварительной обработки и их спецификации. Наличие хранилища сертифицированных журналов дает возможность осуществлять сравнение приложений, использующих данные. Хранилище также предоставляет доступ к исходным журналам, чтобы дать возможность пользователям убедиться в корректности программ, осуществляющих предварительную обработку.

Группой Web Characterization Group была предпринята попытка определить обобщенный формат для записей журналов и создать XML-схему для него. Схема представляет собой краткое семантическое описание, как следует интерпретировать поля в записи. Одной из причин использования XML в качестве языка описания является то, что для проверки корректности записей журнала могут быть использованы стандартные инструментальные средства. Если журнал был представлен в формате XML, синтаксический анализатор XML может выполнять различные действия, а именно:

- Обеспечивать, чтобы каждая запись с результатами измерений имела нужное количество полей. Это облегчает написание простого синтаксического анализатора для журнала, не беспокоясь об ошибочных записях.
- Обеспечивать, чтобы каждое поле имело ожидаемый тип и диапазон значений. Например, эти проверки могут выявлять все коды ответов, которые не являются допустимыми.
- Явным образом исправлять любые найденные ошибки, чтобы облегчить преобразование исходного журнала в семантически безупречный журнал, имеющий записи в ожидаемом формате.

14.4. Измерение параметров мультимедийных потоков

Измерение параметров и анализ Web-трафика играет важную роль в оценке Web-протоколов и программных компонентов. Однако измерению характеристик мультимедийного трафика было уделено сравнительно мало внимания. В этом разделе вкратце рассматриваются четыре метода сбора данных измерений параметров мультимедийных потоков: загрузка мультимедийных ресурсов с Web-сайтов, регистрация запросов к мультимедийным серверам, получение трасс пакетов непосредственно перед сервером и получение детализированных трасс пакетов управляющих сообщений и данных. Ниже будет описано, как эти методологии применяются в реальных исследованиях.

14.4.1. Статический анализ мультимедийных ресурсов

Первоначально HTTP был доминирующим протоколом для передачи имеющихся в Web потоков аудио и видео. Одно из первых исследований мультимедийного содержания в Web было посвящено изучению характеристик видеофайлов, размещенных на Web-серверах [AS98]. Помимо традиционных параметров рабочей нагрузки, таких как размер ресурса, учитываются дополнительные свойства видеоданных, такие как формат кодирования, число кадров в секунду, размеры изображения и требования к средней пропускной способности. Анализ имеющихся в Web видеоресурсов сопровождается тремя проблемами: обнаружение видеоресурсов на различных Web-сайтах, получение копий файлов и вычисление статистических показателей на основе содержимого файлов. Теоретически для решения каждой их этих задач может потребоваться разработка нового программного обеспечения. К счастью, для упрощения задач обнаружения видеофайлов и анализа их содержания может быть использовано существующее программное обеспечение.

Для поиска видеофайлов в Web исследователи используют существующие поисковые машины, которые дают возможность пользователям искать определенные подстроки в составе URL. Поиск подстроки, например, ".mpg" или ".avi", дает в результате список Web-страниц, содержащих одну или несколько гипертекстовых ссылок с ".mpg" или ".avi" в составе URL. Запрос к поисковой машине выдает список URL для Web-страниц. Затем исследователи написали отдельную программу для загрузки Web-страниц и извлечения всех гиперссылок с нужными расширениями файлов. Поскольку гиперссылки на некоторые видеофайлы могли присутствовать на нескольких Web-страницах, необходимо обработать список для удаления повторяющихся URL.

На основе списка URL отдельная программа пытается извлечь ресурсы путем отправки HTTP-запроса **GET** для каждого URL. Около половины запросов по той или иной причине оказываются безуспешными. Некоторые запросы возвращают ответ **404 Not Found**, если запрошенный ресурс отсутствует на Web-сайте. В других случаях Web-сервер возвращает код **200 OK**. Однако это не обязательно подразумевает, что ответ содержит нужный видеофайл. Например, файловое расширение в URL может не соответствовать реальному формату файла. Кроме того, некоторые файлы могут иметь свойства, не отвечающие разумным требованиям. Например, некоторые файлы имеют длительность менее одной секунды или частоту кадров, меньшую, чем четыре кадра в секунду. В ряде случаев попытка воспроизвести содержимое файла показывает, что ресурс представляет собой статическое изображение, а не видеоклип. Отсев некорректных URL и файлов, которые представляются некорректными, приводят к удалению примерно половины URL из исходного списка.

Для обнаружения некорректных файлов и вычисления статистических показателей о корректных файлах требуется выполнять синтаксический анализ данных. Создание программного обеспечения для декодирования видеоданных на практике достаточно затруднительно. К счастью, для типовых форматов видео, включая MPEG, QuickTime и AVI, имеется общедоступное программное обеспечение. В некоторых случаях программное обеспечение, осуществляющее декодирование, может оказаться не в состоянии обработать данный входной файл из-за проблем с форматом данных. В других случаях программа выдает множество статистических показателей, таких как частота кадров, ширина и высота окна и длительность. Эти параметры могут быть использованы для вычисления других показателей, таких как требования к средней пропускной способности (битов в секунду) или отношение ширины к высоте. Файлы с нетипичными параметрами могут быть исключены из дальнейшего рассмотрения. При анализе учитываются значения различных статистических параметров и их представления в разных форматах кодирования. Кроме того, анализирующая программа учитывает, содержит ли файл данные и аудио, и видео.

Хотя исследование дает широкое представление об имеющемся в Web наборе видеоресурсов, статический анализ имеет ряд ограничений. Во-первых, любая поисковая машина обладает неполным представлением об имеющихся в Web-ресурсах. Некоторые Web-страницы со ссылками на видеоресурсы могли быть не подвергнуты индексированию. Кроме этого, некоторые URL для видеофайлов могут иметь файловые расширения, не использованные при поиске. Во-вторых, что весьма существенно, исследование не может определить популярность различных видеофайлов. Для определения популярности ресурсов требуется наличие трассы или журналов клиентов, извлекающих видеоданные с сервера, или результаты отдельного исследования, выполненного рейтинговой компанией. В-третьих, появление новых протоколов для мультимедийных потоков ограничивает эффективность методов выборки видеофайлов. Для потокового видео вместо HTTP все большее применение находят протоколы Real-time Transport Protocol (RTP) и Real Time Streaming Protocol (RTSP). Гиперссылки на Web-страницах часто являются указателями на описания сеансов, сценарии или мультимедийные презентации, которые не раскрывают, что URL соответствуют мультимедийному содержанию.

14.4.2. Журналы мультимедийных серверов

Для анализа поведения пользователей при запросах мультимедийных потоков требуется доступ к журналу или трассе. При этом могут учитываться разнообразные параметры рабочей нагрузки, характерные для традиционного Web-содержания. Такие свойства, как размер ресурса, размер ответа, популярность ресурса, применимы как к мультимедийному содержимому, так и к традиционному Web-содержанию. Однако при анализе содержимого аудио и видео следует принимать в расчет и дополнительные показатели, которые являются уникальными для потокового мультимедиа. При работе с мультимедийным потоком пользователь может активизировать функции видеомагнитофона, такие как воспроизведение, останов, обратная перемотка, ускоренная перемотка. Функции видеомагнитофона оказывают существенное влияние на выделение ресурсов прокси-серверами и исходными серверами в сети. Если большинство пользователей просматривают потоки целиком, то требования к ресурсам становятся известны сразу же, как только сервер получает запрос. В противоположность этому, частое использование функций видеомагнитофона сопровождается меняющимися требованиями к пропускной способности и ограничивает эффективность упреждающей отправки клиенту большого числа кадров.

В то время как Web-сервер записывает информацию о каждом HTTP-запросе, мультимедийный сервер генерирует запись для каждого действия пользователя, такого как обращение к функциям видеомагнитофона. Для форматов журналов Web-серверов имеются определенные стандарты. Для мультимедийных серверов каких-либо строгих требований к форматам нет. Кроме того, журналы мультимедийных серверов не являются широко доступными. Существующие исследования журналов мультимедийных серверов основаны на данных измерений, собранных академическими учреждениями. Многие университеты, имеющие Web-сайты, предоставляют доступ к учебным материалам, таким как курсы лекций, для просмотра в оперативном режиме. Журналы таких сайтов предоставляют возможность анализировать поведение пользователей при запросах мультимедийных потоков для ограниченного сообщества пользователей, обращающихся за содержимым определенного типа. Большое разнообразие мультимедийных приложений затрудняет получение обобщенных результатов о характеристиках ресурсов и шаблонах доступа. Тем не менее, методология обработки серверных журналов и получения характеристик рабочей нагрузки для этих сайтов может быть применена к другим приложениям и сообществам пользователей.

Исследования двух сайтов дистанционного обучения иллюстрируют проблемы, связанные со сбором данных и анализом журналов мультимедийных серверов [PK99, ASP00]. Различаясь форматами, оба журнала содержат информацию одного и того же вида, а именно, традиционные HTTP-запросы на HTML-файлы и изображения вместе с запросами на начало и завершение мультимедийной передачи. Каждая запись содержит метку времени, идентификатор клиента, запрашиваемое действие и имя, ассоциированное с мультимедийным ресурсом. В одном из исследований серверный журнал был подвергнут предварительной обработке для удаления запросов, поступивших от определенного клиента, использованного для целей тестирования; в противном случае запросы от этого клиента могли помешать анализу поведения пользователей [ASP00]. Кроме того, записи журнала, такие как повторяющиеся команды «стоп» от одного и того же клиента, были сжаты до одной записи.

Традиционные параметры рабочей нагрузки имеют характеристики, сходные с соответствующими характеристиками HTTP-трафика. Оба исследования выявили, что пользователи редко просматривают весь поток целиком. Некоторые пользователи прекращают просмотр потока после просмотра его начальной части; это наводит на мысль, что кэширование первых нескольких минут потока на прокси-сервере — достаточно эффективный способ снизить нагрузку на сервер и повысить производительность на стороне пользователя. Другие пользователи часто обращаются к функциям видеомагнитофона, таким как переход, обратная перемотка и ускоренная перемотка. При некоторых операциях обратной перемотки перемещение осуществляется на небольшое число кадров. Это наводит на мысль сохранять небольшое число кадров в буфере клиента, что позволит клиенту обрабатывать эти операции без извлечения кадров с сервера. Операции ускоренной перемотки вперед на небольшое число кадров могут быть осуществлены без задержки, связанной с воспроизведением пользователем кадров, которые уже имеются на клиенте. Однако возможны переходы к произвольному месту в потоке. Такие запросы обычно требуют загрузки новых данных с исходного сервера ценой дополнительной задержки на стороне пользователя.

14.4.3. Мониторинг пакетов мультимедийных потоков

Анализ серверных журналов дает возможность определить характерные параметры поведения пользователей. Однако серверные журналы не содержат доста-

точно данных для изучения транспортировки мультимедийных данных через Internet. В традиционных Web-передачах сервер записывает сообщение-ответ в буфер сокета и оказывается зависимым от базового TCP-соединения при доставке данных. В противоположность этому, доставка мультимедийных потоков требует от сервера чередовать во времени передачу сэмплов аудио и видеокадров, в результате чего трафик может сильно меняться. Получение характеристик мультимедийного трафика на транспортном уровне требует сбора и анализа детально измеренных параметров потока. В недавно проведенном исследовании был представлен подробный анализ транспортировки аудиопотоков с сервера RealAudio онлайновой радиостанции [MH00]. В ходе исследования были собраны данные трасс пакетов, проанализированы параметры трафика на уровне пакетов и потоков, разработана модель, описывающая основные параметры рабочей нагрузки.

Для сбора трасс пакетов требуется перехват пакетов и генерирование записей с данными измерений. Трассы пакетов в исследовании собирались на коммутаторе Ethernet путем направления копий пакетов, передаваемых аудиосерверу или от него, в канал, подключенный к монитору, в соответствии с подходом, представленном на рис. 14.1 в. Аудиосервер может направлять потоки данных клиенту с использованием UDP, TCP или HTTP поверх TCP. Перехват пакетов требует настройки монитора пакетов, включающей задание фильтра, который идентифицирует IP-адрес, протоколы и номера портов, как описывалось ранее в разделе 14.1.2. Хотя сервер всегда выбирает порт 80 для передач HTTP поверх TCP, при передаче аудиопотоков по UDP или TCP может использоваться любой незарезервированный порт (т.е. порты с номерами от 1024 до 65535). На практике сервер использует небольшое, фиксированное множество номеров портов, которые известны лицам, проводящим исследование. Монитор пакетов был настроен для перехвата пакетов с этими номерами портов. Монитор записывает первые 90 байт каждого аудиопакета, что гарантирует включение в трассу заголовков IP и TCP/UDP, а также заголовка данных аудио. Такая подробная трасса дает возможность сосредоточиться при анализе на низкоуровневых транспортных проблемах, связанных с временами передач пакетов от сервера клиентам.

Программа анализа группирует пакеты, принадлежащие одному TCP- или UDP-сеансу; т.е. пакеты с одними и теми же IP-адресами источника, получателя и номерами портов считаются частью одного аудиопотока. Транспортный механизм потока (UDP, TCP или HTTP поверх TCP) может быть определен на основе протокола и номеров портов. Поле протокола указывает, использовался ли протокол UDP или TCP, а номер порта TCP сервера позволяет отличить HTTP-трафик (порт 80) от остального TCP-трафика. По большей части трафик состоит из UDP-пакетов. Это важно, поскольку приложения, использующие UDP, необязательно корректируют скорость передачи в качестве реакции на заторы в сети. Вместо этого аудиосервер передает данные на периодической основе. Чтобы разработать модель рабочей нагрузки, анализирующая программа особое внимание должна уделять размерам пакетов и интервалам времени между началом одного потока и запуском следующего потока, а также длительности потоков. Вариации каждого из этих параметров оцениваются с учетом распределений вероятностей.

Распределения вероятностей, полученные из данных измерений, образуют модель трафика аудиоданных на потоковом и пакетном уровнях. Такая модель способствует проведению экспериментов, оценивающих воздействие трафика аудиоданных на сервер и на сеть. Значения параметров зависят от природы мультимедийного приложения. Онлайновая радиостанция может иметь иные свойства по сравнению с сайтом, предназначенным для прослушивания пользователями аудиоклипов. Немного клиентов прослушивает более одного аудиопотока, а половина потоков имеет длительность

более 45 минут. По большей части потоки имеют несколько различных скоростей передач данных в зависимости от типа аудиоресурсов. Для стереомузыки требуется более высокая скорость передачи, чем для ток-шоу и спортивных репортажей, поскольку речь сжимается лучше, чем музыка. Во всех трех случаях скорость передачи аудиоинформации ограничена 20 Кбит/с, чтобы гарантировать получение потоков пользователями, подключенными к Internet с помощью модемов со скоростью передачи данных 28,8 Кбит/с. Сервер также использует особые размеры пакетов для UDP-передач. Изменения в мультимедийном содержании, перемены в сообществе пользователей или появление новых реализаций серверов могут привести к изменениям характеристик рабочей нагрузки.

14.4.4. Многоуровневый мониторинг пакетов

В отличие от традиционных HTTP-передач, большинство потоковых приложений используют отдельные сеансы для передачи команд и данных. Например, клиент и сервер могут обмениваться RTSP-сообщениями для координации передачи одного или нескольких RTP-потоков; каждый RTP-поток может иметь соответствующий RTCP-поток (RTP Control Protocol) для обмена сигналами обратной связи относительно качества передачи. Детальное определение характеристик мультимедийного трафика может зависеть от наличия информации о каждом из этих протоколов. Запрашиваемые URL, запросы на управление видеопотоком на стороне клиента передаются в RTSP-сообщениях, тогда как данные аудио и видео передаются в RTP-пакетах. Измерение трафика команд и трафика данных несет в себе две проблемы. Во-первых, содержимое управляющих сообщений должно восстанавливаться из базового потока пакетов, подобно восстановлению HTTP-сообщений (см. раздел 14.1). Во-вторых, монитор должен идентифицировать трафик данных, ассоциированный с управляющими сообщениями. На практике это сопряжено со значительными проблемами, поскольку при передаче данных используются номера портов UDP и TCP, которые могут не быть известны заранее. Вместо этого, номера портов для трафика данных доставляются через управляющие сообщения. Например, клиент отправляет RTSP-сообщение на мультимедийный сервер, чтобы установить сеанс, состоящий из одного аудиопотока, использующего RTP для передачи данных и RTCP для обратной связи. После получения описания сеанса клиент отправляет RTSP-сообщение **SETUP** для создания двух транспортных соединений. Сообщение-ответ сервера содержит заголовок **Transport**, который идентифицирует протокол (UDP или TCP) и номера портов на клиенте и на сервере, как описывалось ранее в главе 12 (раздел 12.4.3). Клиент, запросивший сеанс с отдельными потоками аудио и видео, выдает два запроса **SETUP** для создания двух групп соединений, по одной для каждого потока. Каждое соединение уникально идентифицируется по группе из пяти чисел: IP-адресам клиента и сервера, номерам портов клиента и сервера и протоколу.

Идентификация этих соединений довольно затруднительна. IP-адреса клиента и сервера соответствуют адресам, используемым для RTSP-соединения, в то время как номера портов и протокол зависят от ответа сервера на сообщение **SETUP**. Простой подход к решению этой проблемы состоит в том, что монитор захватывает *все* пакеты в канале или все пакеты, использующие незарезервированные номера портов (т.е. от 1024 до 65535). Далее отдельная программа может идентифицировать, какие передачи данных ассоциированы с каждым из командных сеансов. Однако это может потребовать от монитора захватывать и хранить чрезвычайно большой объем данных, особенно для высокоскоростных каналов. Например, монитор может захватывать пакеты из передач больших файлов по FTP и сеансов IP-телефонии, не связанных с какими-либо командными сеансами RTSP.

Вместо этого монитор может просматривать управляющие сообщения при их поступлении и определять, какие номера портов отслеживать. При этом при поступлении пакетов от монитора требуется выполнение двух основных задач: синтаксический анализ управляющих сообщений для определения номеров портов и изменение фильтра пакета для перехвата пакетов данных, использующих эти порты. Эти задачи выполняет инструментальное средство *mmdump* [vCCS00]. Оно перехватывает полное содержимое пакетов, ассоциированных с протоколом управления, идентифицируемого по известному номеру порта (например, 554 для RTSP), и восстанавливает управляющие сообщения. Средство содержит программу для синтаксического анализа различных управляющих протоколов для мультимедиа, таких как RTSP и H.323. После извлечения информации о протоколе, номерах портов и потоках данных, *mmdump* обновляет фильтр пакета для перехвата пакетов данных. Инструментальное средство может перехватывать полное содержимое этих пакетов или быть настроенным для записи определенного числа байтов из начала каждого пакета данных.

Мониторинг передач команд и данных создает условия для выполнения различного вида анализа. Например, рассмотрим мультимедийную презентацию, в состав которой входят изображения, аудио, видео и текст. Управляющие сообщения содержат URL для различных ресурсов, а пакеты данных несут информацию о размере или скорости передачи для каждого потока. Знание URL также дает возможность осуществлять анализ популярности различных мультимедийных ресурсов в Internet. Наличие управляющих сообщений и пакетов данных полезно для изучения воздействия степени загруженности сети на приложения для доставки мультимедийных потоков. Некоторые управляющие сообщения включают клиентские запросы на изменение скорости передачи, что может привести к изменениям в скорости передачи потока данных. Подобные эффекты, связанные с обратной связью, очень трудно учитывать без совместного перехвата трафика команд и трафика данных.

14.5. Резюме

Программное обеспечение анализа Web-трафика играет важную роль в оценке эффективности Web. Мониторинг пакетов обеспечивает детальную информацию о Web-передачах на уровнях HTTP и TCP. Однако для сбора трасс пакетов Web-трафика требуется эффективное программное обеспечение для восстановления HTTP-сообщений из потока IP-пакетов. Многие исследования Web-трафика основываются на данных журналов прокси-серверов и серверов. Эффективное программное обеспечение для синтаксического анализа, фильтрации и преобразования журналов облегчает процесс анализа данных. Несколько хранилищ данных измерений Web-трафика предоставляют исследователям доступ к журналам и трассам. Устранение ошибок и преобразование данных в универсальный формат помогает исследователям использовать хранилище для выполнения различных видов анализа. В сравнении с исследованиями Web-трафика, технология измерений и анализа мультимедийных потоков находится на начальном этапе развития. Мультимедийные потоки порождают новые проблемы в измерении их параметров, поскольку для передачи управляющих сообщений и данных используются отдельные протоколы, имеется множество различных форматов кодирования. Определение характеристик рабочей нагрузки становится более сложным из-за наличия новых параметров, связанных с размерами пакетов, скоростью передачи данных и использованием функций видеомагнитофона.

15

Перспективы исследований протоколов

В главах 13 и 14 рассматривались перспективы исследований, связанных с кэшированием и измерениями. В этой главе мы поговорим об исследованиях некоторых проблем, связанных с Web-протоколами, главным образом с TCP и HTTP. На практике HTTP выполняется поверх TCP, используемого в качестве транспортного протокола. Мы рассмотрим несколько предложений, относящихся к TCP, которые могут оказать существенное влияние на Web. Мы также рассмотрим несколько идей, связанных с изменениями и расширениями HTTP. Поскольку Web-трафик занимает доминирующее положение в Internet, было предпринято несколько попыток усовершенствовать HTTP. В главе 7 мы рассмотрели различные попытки исправить присущие HTTP/1.0 недостатки, которые в итоге привели к появлению HTTP/1.1. Хотя протокол в процессе создания прошел через несколько этапов (проект стандарта, предложение по стандарту), ряд проблем остался нерешенным. Вот несколько причин, по которым в процессе стандартизации не был решен ряд проблем:

- Проблемы не были признаны достаточно серьезными, чтобы задерживать из-за них стандартизацию протокола.
- Срочная необходимость выпустить улучшенную версию протокола, чтобы устранить имеющиеся в HTTP/1.0 проблемы.
- Появление многочисленных версий Web-компонентов, *якобы* поддерживающих HTTP/1.1 до завершения процесса стандартизации. Такие компоненты объявлялись согласующимися с документом RFC 2068 [FGM+97], который далек от стандарта.

Поскольку основой деятельности рабочих групп IETF является достижение консенсуса, отдельные проблемы не должны препятствовать стандартизации протокола в целом. Решение ряда проблем было передано рабочим группам с целью определить, не смогут ли менее масштабные документы содействовать в достижении консенсуса. Несколько дискутируемых предложений, таких как измерение числа посещений и прозрачная доставка содержания, не прошли процесс стандартизации и были либо сняты с рассмотрения, либо выделены в отдельные стандарты. Кроме того, возник ряд новых проблем, которые не проявились в процессе перехода от HTTP/1.0 к HTTP/1.1. В этой главе мы рассмотрим некоторые из таких проблем. Одни проблемы уже решаются в процессе стандартизации, тогда как другие пока обсуждаются неофициально. Новые идеи по совершенствованию протокола могут быть предложены каждым в форме рабочего проекта. Заинтересованные лица могут присоединиться к обсуждению и предложить конструктивные решения.

В этой главе будут обсуждаться следующие темы:

- **Мультиплексирование HTTP-передач.** Взаимодействие между HTTP и TCP оказывает существенное влияние на производительность Web в частности и на эффективность Internet в целом. Web-клиент обычно устанавливает несколько TCP-соединений с одним и тем же сервером для параллельной загрузки встроенных изображений и уменьшения времени ожидания на стороне пользователя. Однако параллельные TCP-соединения порождают ряд проблем, связанных с тем, что активно работающий клиент получает большую часть пропускной способности сети, чем другие клиенты. Кроме этого, многие мультимедийные приложения используют UDP и не реагируют на заторы в сети так, как это делают TCP-соединения. Для решения этих проблем необходимо вновь обратиться к взаимодействию протоколов прикладного уровня, включая HTTP, с транспортным уровнем. Мы познакомимся с тремя предложениями, которые предполагают изменения способа мультиплексирования HTTP-передач: WebMux, TCP Control Block Interdependence и Integrated Congestion Management.
- **Добавление механизма отличий (дельта-механизма) в HTTP/1.1.** Предположим, в кэше содержится экземпляр ресурса, а изменившийся экземпляр на исходном сервере несколько отличается от кэшированной версии. Вместо того, чтобы передавать новую версию ресурса целиком, исходный сервер может отправить имеющиеся по сравнению с предыдущей версией отличия в предположении, что они небольшие. Передача только отличий уменьшает время ожидания на стороне пользователя и объем данных, передаваемых по сети. Мы обсудим побудительные причины для введения механизма отличий (или «дельты»), поговорим об имеющихся алгоритмах, об оценке алгоритмов на основе измерений в Web, а также рассмотрим соображения, связанные с внедрением механизма в HTTP/1.1.
- **Совместимость с протоколом HTTP.** Совместимость с протоколом подразумевает, что реализации компонентов отвечают требованиям MUST (Обязательно), SHOULD (Желательно) и MAY (Возможно), изложенным в спецификации. Хотя стандартизация протокола требует тестирования на совместимость компонента каждой из функций, общего механизма проверки на совместимость не существует. В распоряжении разработчиков отдельных компонентов имеются перечни тестов, помогающих проверить совместимость различных функций. Отсутствие надежного, исчерпывающего и, что самое важное, *обязательного* механизма тестирования на совместимость является слабым местом процесса стандартизации. Несовместимые реализации могут привести к возникновению проблем для пользователей, владельцев Web-сайтов и сети. HTTP в этом смысле не является исключением — большинство разработанных IETF протоколов формально не тестировались на совместимость. Мы обсудим методологию проверки совместимости HTTP (под названием PRO–COW) и рассмотрим результаты обширного исследования, в ходе которого тестировались серверы большой группы популярных Web-сайтов.
- **Комплексное измерение параметров для изучения воздействия усовершенствований протоколов на производительность Web.** Несколько изменений в протоколе HTTP были обсуждены Рабочей группой и внедрены без каких-либо измерений показателей производительности. На практике детальное измерение показателей производительности при внедрении невозможно без доработки компонентов. Комплексное (от одной конечной точке к другой) измерение производительности Web потребует измерения параметров ответов у

клиентов, находящихся в различных точках и связывающихся с различными Web-серверами. Будет представлена методика оценки производительности и результаты экспериментов, в ходе которых были проведены активные измерения параметров на Web-серверах для большого числа различных местоположений клиентов. Принимая во внимание асимметричный характер трафика запросов к Web-сайтам (несколько тысяч Web-сайтов получают значительную часть всех запросов), можно изучить воздействие внесения изменений в протокол путем рассмотрения поведения трафика запросов и ответов для этих сайтов.

- **Другие расширения HTTP.** Ряд предложенных расширений HTTP не нашел широкого признания. Однако некоторые из них показали себя более перспективными, чем другие. Мы рассмотрим в этой главе три таких расширения, начиная с интегрированной инфраструктуры расширения HTTP, предложенной в RFC 2774. Затем мы рассмотрим концепцию прозрачной доставки содержания (Transparent Content Negotiation), исключенную из HTTP/1.1 в ходе дискуссии, результатом которой стало появление документа RFC 2616, представляющего собой проект стандарта для HTTP/1.1. Наконец, мы рассмотрим протокол Distributed Authoring and Versioning (WebDAV), призванный дать возможность нескольким авторам совместно редактировать ресурсы.

15.1. Мультиплексирование HTTP-передач

При загрузке Web-страниц Web-клиент обычно открывает несколько TCP-соединений с одним и тем же сервером. Параллельные соединения порождают проблемы, связанные с производительностью и равномерным распределением ресурсов, о чем говорилось ранее в главе 8 (раздел 8.3). В этом разделе мы рассмотрим три предложения по изменению взаимоотношений сеансов прикладного уровня с базовыми механизмами транспортного уровня. Предложение WebMux описывает способ мультиплексирования нескольких HTTP-сеансов в одно TCP-соединение. Предложение TCP Control Block Interdependence предусматривает более тесное взаимодействие между TCP-соединениями с одним и тем же удаленным компьютером. Предложение Integrated Congestion Management основное внимание уделяет выделению полосы пропускания для множества потоков пакетов на транспортном уровне и способам предоставления приложениям возможности адаптироваться к изменениям пропускной способности.

15.1.1. WebMux — экспериментальный протокол мультиплексирования

При загрузке Web-страницы браузер может устанавливать несколько параллельных соединений с Web-сервером для получения и отображения встроенных изображений. Параллельные соединения дают возможность браузеру преодолевать ограничения, присущие взаимодействию HTTP с транспортным уровнем. В HTTP сообщения передаются последовательно через сокет, который напрямую связан с базовым TCP-соединением. Наличие нескольких одновременных передач требует от браузера иметь несколько сокетов, что, в свою очередь, требует установки нескольких TCP-соединений с сервером. Один из способов решения этой проблемы — расширить HTTP, чтобы разрешить чередование нескольких сообщений в одном и том же транспортном соединении. Однако это потребует внесения значительных изменений

в протокол, а также изменений реализаций Web-клиентов, прокси-серверов и Web-серверов. Процесс стандартизации и большая база имеющихся программных компонентов Web делает это решение неприемлемым.

В качестве альтернативного подхода WebMux предлагает отказаться от однозначной связи между сокетами и TCP-соединениями. WebMux — экспериментальный протокол мультиплексирования, который не требует внесения изменений в HTTP и программные компоненты Web [GN98]. Вместо этого WebMux предоставляет нескольким сокетам прикладного уровня возможность отправлять и принимать данные через одно и то же базовое соединение транспортного уровня. Данные, записанные в сокет, делятся на один или несколько фрагментов. WebMux делит данные из каждого сокета на фрагменты и передает фрагменты через TCP-соединение. Каждый фрагмент имеет заголовок WebMux с идентификатором сеанса, который дает возможность принимающей стороне TCP-соединения направлять данные соответствующему получателю. Заголовок WebMux также содержит различные флаги для открытия и закрытия сеанса, подобные флагам TCP. Базовое TCP-соединение трактует фрагменты WebMux как обычные данные, подлежащие доставке из одной конечной точки в другую.

Мультиплексирование нескольких сеансов в одно TCP-соединение сопровождается проблемами, связанными с совместным использованием полосы пропускания и памяти. Отправка большого количества фрагментов в интересах одного сеанса может привести к дискриминации других сеансов. Чтобы предотвратить эту дискриминацию, отправитель WebMux чередует передачу данных сеансов по кругу. Хотя порядок пересылки определяет отправитель, базовое TCP-соединение управляет передачей данных с помощью скользящего окна. На принимающей стороне TCP-получатель реконструирует входящие пакеты в упорядоченный поток данных. Получатель WebMux связывает его с соответствующим сокетом на основе идентификатора сеанса. Однако получатель WebMux должен сохранять фрагмент до того, как принимающее приложение прочитает данные из соответствующего сокета. Если один сеанс занимает весь буфер, другие принимающие сокеты, связанные с тем же TCP-соединением, должны быть заблокированы, ожидая данные.

Чтобы избежать блокирования других сокетов, получатель WebMux выделяет каждому сеансу пространство в памяти. Имея отдельные области памяти, получатель WebMux может считывать и осуществлять буферизацию каждого сеанса независимо. Однако это не решает проблему полностью. Расположенный на транспортном уровне TCP-получатель также имеет ограниченный объем буфера, что является препятствием при передаче данных TCP-отправителем. Один сеанс может занять весь входной буфер TCP, что помешает TCP-отправителю передавать данные в интересах других сеансов. Чтобы решить эту проблему, в WebMux вводятся *кредиты*, связанные с каждым сеансом. Каждый кредит дает возможность отправителю WebMux передавать определенный объем данных в интересах сеанса. За счет ограничения количества ожидающих обработки кредитов получателем WebMux обеспечивается, что сеанс не будет потреблять слишком много памяти. Схема управления потоками на базе кредитов похожа на использование подтверждений и окна приема в TCP. Однако кредиты ассоциируются с сеансами, тогда как окно приема ассоциируется с TCP-соединением.

WebMux решает ряд проблем, связанных с ассоциированием нескольких сеансов прикладного уровня с одним транспортным соединением. Главным преимуществом предлагаемого подхода является то, что WebMux не требует внесения изменений в TCP, HTTP и программные компоненты Web. Кроме того, поддержка мультимедийных сеансов может быть полезна для других протоколов прикладного

уровня. Хотя WebMux разрабатывался применительно к HTTP, предлагаемые им идеи не зависят от используемого протокола прикладного уровня. Однако при полноценной реализации WebMux требуется уделить особое внимание совместному использованию сеансами пропускной способности сети и пространства буфера. В случае принятия официального стандарта WebMux должен будет допускать встраивание в основные операционные системы и развертывание на компьютерах по всему Internet. Протокол WebMux был принят в качестве рабочего проекта в конце 1998 г. [GN98] в результате усилий организации HTTP Next Generation (HTTPng) [SJ00]. Однако дальнейшие работы были остановлены, и протокол WebMux не прошел дальнейших этапов стандартизации IETF.

15.1.2. TCP Control Block Interdependence

WebMux предлагает схему для мультиплексирования нескольких сеансов прикладного уровня в одном TCP-соединении. Альтернативный подход предусматривает учет взаимодействия между несколькими TCP-соединениями. Традиционно каждое TCP-соединение функционирует независимо даже в том случае, если несколько соединений установлено между одной парой компьютеров. Каждое соединение само определяет время передачи в прямом и обратном направлении (RTT) и максимальный размер сегмента (MSS) для передаваемых данных. Кроме того, каждое соединение осуществляет управление скользящим окном TCP с целью выбора подходящей скорости передачи, не используя при этом информацию, накопленную в результате функционирования других соединений. Каждое соединение начинается с небольшого начального размера окна на этапе медленного старта, даже если предыдущее соединение уже нашло оптимальные размеры скользящего окна. Помимо этого несколько соединений, действующих независимо, могут потреблять гораздо большую часть пропускной способности сети, чем клиент с одиночным TCP-соединением.

Предложение TCP Control Block Interdependence [Tou97] подразумевает пересмотр предположения, что TCP-соединения функционируют независимо, установив более тесную связь между соединениями с одним и тем же удаленным компьютером. Каждое TCP-соединение на локальном компьютере ассоциируется с управляющим блоком, который хранит информацию о состоянии соединения. Управляющий блок содержит информацию о локальном процессе, включая указатели на буферы передачи и приема сокета прикладного уровня, а также информацию о состоянии соединения, такую как номера портов, размеры окон передачи и приема, значения различных таймеров. Эти данные управляющего блока не могут быть совместно использованы различными TCP-соединениями. Однако другая информация о состоянии соединения, включая оценки значений RTT, MSS и размер скользящего окна, относится к паре взаимодействующих между собой хостов. Эти параметры могут совместно использоваться различными соединениями. Совместное использование информации соединениями координируется операционной системой, что не требует внесения изменений в протокол прикладного уровня (например, HTTP) или в приложения (например, Web-браузер или Web-сервер). Предложение учитывает два возможных сценария взаимодействия между TCP-соединениями с одним удаленным компьютером: *согласованное (ensemble)* и *временное (temporal)* совместное использование.

Согласованное использование (ансамбль) имеет место, когда два или более активных соединения сотрудничают при совместном доступе к информации о состоянии. Простейший пример — инициализация управляющего блока нового со-

единения с параметрами, взятыми из текущего соединения. После этого соединения функционируют независимо. Точная начальная оценка значений RTT поможет установить для нового соединения время таймаута повторной передачи (RTO). Более сложный пример — сотрудничество при определении размера скользящего окна каждого соединения. Ансамбль может трактоваться как одно TCP-соединение для задач управления скользящим окном. Результирующее скользящее окно может быть поделено ансамблем между индивидуальными соединениями. Тем самым агрегированный трафик будет вести себя так, как если бы передавался по одному TCP-соединению. Подобная стратегия позволяет Web-браузеру иметь несколько соединений с Web-сервером без дискриминации других клиентов в использовании пропускной способности сети. WebMux также решает эту проблему. Главным отличием является то, что WebMux мультиплексирует несколько передач в одном TCP-соединении, тогда как TCP Control Block Interdependence координирует совместное использование пропускной способности путем группировки отдельных TCP-соединений в ансамбль.

Временное совместное использование предусматривает инициализацию управляющего блока нового соединения с параметрами, взятыми из уже завершившегося соединения. При закрытии TCP-соединения значение MSS, размер скользящего окна, среднее значение и дисперсия RTT могут быть сохранены для последующего использования. Например, рассмотрим Web-браузер, который устанавливает TCP-соединение с Web-сервером для извлечения встроенного изображения после загрузки HTML-файла, в котором оно содержится. Если реализация TCP на сервере поддерживает временное совместное использование, новое соединение может начаться со скользящего окна большего размера и при более точной оценке значения RTT. Это уменьшит время ожидания на стороне пользователя при загрузке изображения. В отличие от согласованного совместного использования, временное совместное использование требует кэширования информации о состоянии предыдущих соединений с распределением по компьютерам, с которыми были установлены соединения. Кроме того, информация из кэшированного управляющего блока может устареть; это зависит от того, насколько давно было завершено соединение. Смягчить эти проблемы можно за счет ограничения длительности кэширования информации о состоянии.

Предложение TCP Control Block Interdependence является попыткой оптимизировать функционирование TCP в переходных состояниях без изменения долговременных характеристик. Особенно актуально это для HTTP-трафика, поскольку при извлечении Web-страниц используется большое количество TCP-соединений. Кроме того, время ожидания на стороне пользователя для большого числа коротких Web-передач сильно зависит от поведения TCP в переходных режимах. Эффективные методы управления группами TCP-соединений могут также избавить от необходимости использования протоколов, например, WebMux, которые чередуют несколько сеансов в одном TCP-соединении. Предложение Control Block Interdependence основано на предыдущих работах, посвященных проблемам совместного использования параметров RTT и MSS, а также методах снижения затрат на открытие и закрытие TCP-соединений [Bra92, Bra94]. Внедрение предложения требует внесения изменений в операционные системы, под управлением которых работают компьютеры в Internet, а также дальнейших исследований стратегий совместного использования параметров управляющего блока, таких как размер скользящего окна. Предложение TCP Control Block Interdependence было опубликовано в качестве информационного документа IETF в 1997 г. и не проходило последующих этапов процесса стандартизации IETF.

15.1.3. Integrated Congestion Management

WebMux и TCP Control Block Interdependence предлагают более тесную связь между сеансами прикладного уровня, осуществляющими обмен данными с одним и тем же удаленным компьютером. Однако успех Web и коммерциализация Internet привели к возникновению ряда дополнительных проблем, связанных с управлением загруженностью. Некоторые коммерческие продукты повышают эффективность коммуникационного взаимодействия за счет других пользователей, изменяя базовые алгоритмы управления TCP. Как говорилось в главе 12, многие приложения для работы с потоками аудио и видео используют UDP вместо TCP. Этим приложениям нет нужды адаптироваться к загруженности сети таким же образом, как это делается в TCP. Рост объема трафика, для которого не осуществляется управление загруженностью, поставил под угрозу стабильность Internet. Например, Web-сервер может посылать данные более агрессивно, чем предполагает механизм управления загруженностью TCP, чтобы сократить время ожидания при передаче сообщений-ответов TCP. Эти передачи неравномерно распределяют пропускную способность сети в пользу ряда соединений, что приводит к увеличению количества заторов в сети. Кроме того, традиционный программный интерфейс сокетов не предоставляет приложениям явной обратной связи для адаптации передачи к доступной пропускной способности. Например, мультимедийное приложение может передавать низкокачественную версию видеопотока в периоды снижения пропускной способности.

В предложении Integrated Congestion Management предусматривается использование интегрированной среды для решения перечисленных проблем, а также проблем справедливого распределения ресурсов при параллельных TCP-соединениях. Помимо эффективного мультиплексирования трафика предлагается архитектура для управления загруженностью и интерфейс для приложений, позволяющий им реагировать на изменения пропускной способности [BRS99, BS00]. В состав пакета Congestion Manager входят несколько программных модулей, которые управляют передачей и приемом потоков пакетов между конечными компьютерами. Реализация TCP может быть выстроена поверх этих модулей. На передающей стороне Congestion Manager определяет пропускную способность, доступную для передачи данных удаленному компьютеру, применяя специальный алгоритм. Этот алгоритм основан на использовании информации о предыдущих передачах данных и периодических тестовых передачах между отправителем и получателем. Периодические тестовые передачи гарантируют, что Congestion Manager будет обладать своевременной информацией о загруженности сети, даже если принимающее приложение не предоставляет ее. Например, медиаплейер, принимающий UDP-пакеты, может не предоставлять такую информацию мультимедийному серверу.

Отправитель сочетает явную обратную связь с получателем и статистику, полученную в результате периодических тестовых передач для формирования точного представления о пропускной способности. Алгоритм управления загруженностью по своему принципу схож с механизмом, используемым в TCP, с некоторыми отличиями. Подобно TCP, Congestion Manager передает данные с помощью скользящего окна, размер которого линейно увеличивается при отсутствии потерь пакетов и мультипликативно (в геометрической прогрессии) уменьшается в случае потерь пакетов. Размер окна сокращается до небольшой величины в ответ на продолжительный затор аналогично тому, как это происходит на этапе медленного старта в TCP. В отличие от традиционных реализаций TCP, Congestion Manager задает скорость передачи пакетов, чтобы избежать резкого увеличения трафика в сети.

Congestion Manager передает данные медленнее при отсутствии информации от получателя, чтобы избежать перегрузки сети. Таким образом Congestion Manager отправляет данные с меньшей скоростью, если точная информация о загруженности сети отсутствует.

После определения доступной пропускной способности соединения с удаленным компьютером Congestion Manager выделяет ресурсы различным потокам, передающим данные на этот удаленный компьютер; эта идея схожа с согласованным совместным использованием в TCP Control Block Interdependence. Это решает проблему с равноправным распределением ресурсов, возникающую при взаимодействии Web-клиента и сервера через несколько TCP-соединений одновременно. При простейшей стратегии выделения ресурсов имеющаяся пропускная способность делится поровну между конкурирующими потоками. Затем Congestion Manager уведомляет приложение, ответственное за поток, о выделенной пропускной способности. Приложение, например Web-сервер, решает на основе выделенной пропускной способности, какие данные отправить. Использование явной обратной связи с приложением относительно доступных ресурсов требует нового интерфейса прикладного программирования. Коммуникационная модель отличается от традиционной абстракции сокетов. Приложение выражает заинтересованность в передаче данных. Через некоторое время Congestion Manager уведомляет приложение, что разрешение на передачу данных предоставлено. Затем приложение решает, какие данные отправлять. Congestion Manager может также уведомлять приложение об изменениях в скорости передачи данных. Это полезно для мультимедийных серверов, которые могут динамически приспосабливать качество потока аудио или видео к доступной пропускной способности.

Предложение Integrated Congestion Management пересматривает роль, которую играют хосты в управлении загруженностью в Internet и обеспечении требований приложения. Таким образом предложение затрагивает гораздо более широкий круг проблем и решает более амбициозные задачи, чем WebMux и TCP Control Block Interdependence. Предложенный подход требует внесения изменений в интерфейс прикладного программирования, что осложняет его широкое внедрение. С другой стороны, возможно постепенное внедрение, поскольку передающий компонент Congestion Manager взаимодействует с традиционными получателями (например, получателями, выполняющимися на компьютерах, на которых отсутствует Congestion Manager). Congestion Manager был принят в качестве рабочего проекта в июле 2000 г. [BS00] в результате работы группы Endpoint Congestion Management Working Group IETF. Оценить окончательные результаты усилий на этом начальном этапе процесса стандартизации довольно трудно.

15.2. Добавление дельта-механизма в HTTP/1.1

За последние несколько лет возросло беспокойство, связанное с ростом объема трафика в Internet и требованием снижения времени ожидания на стороне пользователя при получении ответов на Web-запросы. В этом разделе мы обсудим предложение, связанное с добавлением в HTTP/1.1 дельта-механизма для решения этой проблемы. Предложению применить данный механизм предшествовали два других подхода: сжатие данных и селективная загрузка ответов. Сокращение объема ненужных данных, передаваемых через сеть, является важной задачей, в этом разделе описываются шаги, предпринятые в этом направлении сообществом, занимающимся совершенствованием протокола HTTP. Обзор эволюции протокола так-

же проливает свет на то, как возникают определенные идеи, которые требуют внесения изменений в протокол, а также на то, как протокол продолжает развиваться с учетом таких изменений.

Типичным механизмом, с помощью которого можно достичь двух целей, заключающихся в уменьшении числа пакетов, передаваемых через Internet, и времени ожидания на стороне пользователя, является сжатие данных. Ответ может быть сжат отправителем (как правило, исходным сервером) и подвергнут декомпрессии получателем. При этом передается меньший объем данных, а время, затрачиваемое на передачу этих данных, обычно уменьшается. В процессе эволюции протокола HTTP/1.1 было замечено, что многие Web-ресурсы передавались в несжатом виде. Сжатие данных функционально доступно и в HTTP/1.0. Текст, который хорошо поддается сжатию, по-прежнему составляет значительную часть Web-трафика. Текстовые ресурсы обычно имеют меньший размер по сравнению с изображениями, а популярные форматы изображений уже являются сжатыми (например, GIF, JPEG). Наличие и доступность алгоритмов быстрой компрессии и декомпрессии для текстовых документов сделало этот подход еще более привлекательным. Кроме того, можно использовать тот или иной алгоритм сжатия в зависимости от типа содержания. Например, могут быть использованы специализированные словари, которые учитывают частотные характеристики появления символов алфавита данного языка. Такие словари создаются для оптимизации поиска часто используемых в языке слов. Помимо компрессии сообщений программным путем, модемы могут выполнять свое собственное сжатие. Однако с учетом того, что Web-сообщение проходит по множеству каналов, модемы представляют лишь один из многих сегментов пути передачи данных. Поскольку меньший объем данных обычно передается за меньшее время, можно достичь совокупной экономии, учитывая, что ответ перемещается от отправителя к получателю через множество сегментов. Эксперименты показали, что традиционное сжатие данных дает значительно больший эффект, чем сжатие, выполняемое модемами [Nie97].

Особая проблема связана с использованием протоколом HTTP в качестве транспортного механизма протокола TCP. Затраты на ожидание первого пакета TCP выше, чем для последующих пакетов. При передаче контейнерного документа, состоящего из текста (HTML) и изображений, сначала загружается HTML, а затем изображения. Сжатие HTML-страницы (которая часто находится в первом пакете) приведет к тому, что последующие изображения будут поступать быстрее. Общее время ожидания контейнерного документа снижается. Браузер может быстрее начать отображать документ, сокращая тем самым время ожидания на стороне пользователя.

Сжатие является общим механизмом уменьшения объема данных, передаваемых между отправителем и получателем. Для задания способа сжатия могут быть использованы существующие механизмы HTTP, такие как заголовки **Accept-Encoding** и **Content-Encoding**. Другой способ уменьшить объем данных состоит в использовании условных запросов в HTTP, например, заголовок **If-Modified-Since** используется для извлечения ресурса только в том случае, если он был модифицирован после определенного момента времени, указанного в запросе. Это полезно при наличии кэшей, в которых хранятся предыдущие версии ответов.

Механизм запросов на диапазоны в HTTP/1.1 был шагом в этом направлении и дал возможность передавать только части ресурса. Однако механизм запросов на диапазоны годится для отправки только непрерывных фрагментов ресурса. Предположим, что ресурс изменяется так, что с помощью запроса на диапазон могут быть запрошены только новые фрагменты ресурса. Подобный запрос приведет

к передаче минимального объема данных. Однако ресурсы часто изменяются таким образом, что клиентам не известно, в каком месте произведены изменения, и они не могут запросить только эти измененные фрагменты, воспользовавшись запросами на диапазоны.

Если обобщить принципы применения сжатия и выборочной загрузки фрагментов, то можно сказать, что идеальным является передача только *необходимых* для конкретной Web-транзакции данных. При этом снижаются требования к пропускной способности и уменьшается время ожидания на стороне пользователя. При наличии кэшированных экземпляров предыдущих ответов, возникает естественное желание определить, может ли быть передана только разность между кэшированным экземпляром и текущим экземпляром. Идея генерировать разности между экземплярами достаточно давно используется в информатике и компьютерных технологиях. Имеется множество алгоритмов, которые постоянно совершенствуются применительно к различным форматам данных. Средства определения разности для традиционных файлов достаточно популярны. Однако применительно к Web механизм вычисления разности (дельты) должен учитывать определенные ограничения, которые не учитывались в алгоритмах, предназначенных для моделей файловых систем.

Мы начнем этот раздел с рассмотрения побудительных причин для применения дельта-механизма для HTTP-сообщений. Далее мы поговорим о выборе алгоритмов вычисления дельты, познакомившись с посвященным этой проблеме исследованием. После этого мы посмотрим, как дельта-механизм может быть внедрен в HTTP/1.1. В конце мы познакомимся с текущим состоянием дел по добавлению дельта-механизма в HTTP/1.1.

Немного о терминологии: в HTTP/1.1 нет термина для описания значения, которое будет возвращено в ответ на запрос **GET** в текущий момент времени для выбранной версии данного ресурса. Для этой цели вводится термин *экземпляр*.

15.2.1. Побудительные причины для использования дельта-механизма для HTTP-сообщений

Как описывалось ранее в главе 10 (раздел 10.4), ресурсы в Web часто изменяются, и эти измененные ресурсы часто снова запрашиваются тем же клиентом [DFKM97]. Можно предположить, что разность между экземплярами ресурса, выраженная в байтах, является небольшой. Например, ответ может содержать число обращений к ресурсу. Если два различных экземпляра этого ресурса отличаются только значением этого числа, разность между экземплярами ничтожна в сравнении с самим ресурсом. Исходный сервер может отправить либо весь экземпляр, либо разность между кэшированным экземпляром у получателя и текущим экземпляром на исходном сервере. Отправка разности и быстрое воссоздание нового экземпляра на стороне получателя положительно сказывается на всех участниках транзакции. Исходный сервер может потратить меньше времени на формирование и отправку разности, чем в случае формирования и отправки всего экземпляра целиком. Помимо этого исходный сервер может кэшировать разность, а не различные экземпляры, что приведет к амортизации затрат на формирование разности. Сокращается загрузка сети, поскольку разность, как правило, имеет гораздо меньший размер, чем полный экземпляр. В действительности, если разность велика, сервер может отправить полный экземпляр. Принимающий клиент или прокси-сервер должен обработать ответ и воссоздать полный экземпляр из кэшированного экземпляра и разности. Если это может быть сделано быстро, пользователь вряд ли заметит какую-либо дополни-

тельную задержку. Задержка может стать ощутимой только для первого пользователя, для которого инициируется обновление. Модифицированный экземпляр кэшируется, и последующие обращения к ресурсу будут браться из кэша. Если на пути следования запроса-ответа имеется прокси-сервер, разность будет сохранена и отправлена дальше, если прокси-сервер участвует в механизме вычисления разности. Прокси-сервер может также применять дельта-механизм к своей кэшированной записи и использовать обновленный кэшированный ответ как полный ответ для будущих запросов от клиентов, не поддерживающих дельта-механизм. Кроме того, последующие запросы от таких клиентов могут преобразовываться в дельта-запрос (запрос на получение разности) перед его передачей исходному серверу.

Периодические изменения в ресурсах вызывают необходимость применения дешевого механизма, который может формировать разность между версиями, и способа транспортировки разности по протоколу HTTP. Ниже мы рассмотрим различные способы вычисления разности.

15.2.2. Сравнение дельта-алгоритмов

Для вычисления разности между версиями ресурса существует множество алгоритмов. В традиционных файловых системах для создания разности между двумя текстовыми файлами используется команда UNIX *diff*. Разность может быть представлена пользователю способом, при котором выделяются различные виды разности: добавление, удаление и изменение. Использование опции *–e* команды *diff* дает возможность генерировать сценарий. Этот сценарий может использоваться программой-редактором *ed* для автоматического применения изменений к одному варианту с целью получения другого варианта. Таким образом, разность может быть вычислена на одной стороне и передана другой стороне в виде сценария, а этот сценарий применяется для создания модифицированной версии. Предполагается, что разность имеет гораздо меньший размер по сравнению с новой версией.

Чтобы оценить алгоритмы вычисления разности для ресурсов в Web, следует иметь в виду следующие факторы:

- Время вычисления разности между вариантами ресурса.
- Размер разности.
- Частота изменения ресурса.

Указанные факторы не обязательно приводят к однозначному выбору алгоритма вычисления разности. Например, алгоритм, который способен быстро вычислять разность, может выдавать разность большого размера, жертвуя объемом данных ради скорости. Такой алгоритм менее полезен для ресурсов, которые часто изменяются. Подобные обстоятельства привели к использованию алгоритмами вычисления текстовой разности, такими как Revision Control System (RCS) [Tic85] и Source Code Control System (SCCS) [Roc75], принципа *опережающей разности*. Опережающая разность накапливает разности по сравнению с последней версией. Разности хранятся вместе, что позволяет приложению быстро создать любую версию. При рассмотрении второго фактора следует помнить, что объем информации увеличивается медленно от версии к версии, на любом этапе разность между двумя ближайшими версиями обычно невелика. Главное наблюдение состоит в том, что в зависимости от приложения, величина разности между версиями и суммарная разность могут оказаться различными для различных алгоритмов. Третий фактор учитывает, что не все типы ресурсов изменяются с одинаковой частотой. Например, текстовые ресурсы изменяются чаще изображений [DFKM97].

В оставшейся части этого раздела мы познакомимся с исследованием, проведенным в 1997 г., целью которого было оценить, надо ли расширять протокол, добавляя в него дельта-механизм. Сначала мы рассмотрим методологию и детали проведения эксперимента, а также его результаты. Затем мы обсудим, где уместно применять дельта-механизм, а также познакомимся с общими соглашениями по внедрению расширений в инфраструктуру HTTP/1.1. В конце мы поговорим о текущем положении дел, связанных с внедрением дельта-механизма в протокол HTTP/1.1.

МЕТОДОЛОГИЯ ИССЛЕДОВАНИЯ

Чтобы оценить алгоритмы вычисления разности на предмет их применения в Web, при проведении исследования следует учитывать следующие методологические соображения:

- Чтобы обеспечить репрезентативность различных атрибутов, таких как тип содержания и размер, должно быть изучено большое количество различных ресурсов.
- Следует вычислять разность для различных версий ресурсов во времени.
- Эксперимент должен повторяться через определенный период времени для множеств ресурсов, отобранных из самых разнообразных мест.

При проведении исследования необходимо также учитывать следующие соображения:

- Частота, с которой изменяются ресурсы.
- Время вычисления разности между версиями ресурсов.
- Экономия в объеме данных при передаче разности в сравнении с передачей полного ответа.

Результаты исследования могут помочь при определении механизма, который будет использоваться всеми браузерами, прокси- и Web-серверами для вычисления разности. Любые изменения в протоколе, необходимость которых обусловливается результатами исследования, должны подвергаться стандартизации.

Теперь мы представим краткий отчет об исследовании, который впервые был опубликован в [MDFK97a], а более подробный отчет можно найти в [MDFK97b]. Исследование проводилось несколько лет назад, но единственный основной фактор, который с тех пор мог подвергнуться изменениям, — это состав трафика (меньше текстовых документов, больше динамических ответов и больше изображений в различных форматах). Никаких значительных изменений, связанных с ускорением вычисления разности или с алгоритмами сжатия, не произошло.

Главным отличием этого исследования от проводившихся ранее (например, [HL96]), было использование потока реальных запросов и ответов в Web, вместо выбора коллекции документов и рассмотрения возможности применения к ним методов вычисления разности. Исследование начинается с рассмотрения, какие ресурсы выбраны путем просмотра журналов прокси-серверов и трасс пакетов. В ходе исследования было отобрано два набора ресурсов из двух различных мест, расположенных в США. Один источник данных представлял собой корпоративный прокси-сервер. Прокси-сервер отслеживал все исходящие запросы. В течение всего двух дней было записано достаточно много данных (около 9 гигабайтов, составивших примерно полмиллиона трасс примерно 8000 различных клиентов). Второй источник представлял собой трассы мониторинга пакетов исследовательской орга-

низации. Было записано 19 гигабайтов данных в течение 17 дней, что составило около миллиона трасс примерно 7500 клиентов. Трассировка пакетов перехватывает данные на низком уровне и реконструирует их в запросы и ответы HTTP, как говорилось в главе 14 (раздел 14.1). Множество ресурсов для оценки дельта-механизма было выбрано из этих двух наборов.

Для определения полезности дельта-механизма должны анализироваться только запросы на ресурсы, которые изменялись в период проведения тестирования. Идентифицировались ответы для одного и того же ресурса, имеющие код статуса **200 OK** для более чем одного экземпляра, а значения в заголовке ответа **Last-Modified** использовались для определения, были ли ресурсы изменены. После того, как такие пары экземпляров были выявлены, к каждой паре применялись различные механизмы вычисления разности, чтобы определить величину разности второго экземпляра относительно первого. Если подходящий дельта-механизм имелся, исходный сервер отправлял соответствующий код. Такие пары экземпляров называются *дельта-приемлемыми*.

В ходе исследования изучались три различных алгоритма кодирования разности с учетом их популярности и доступности. Первый из них заключается в использовании команды UNIX *diff* с опцией *-e* для построения сценария, который может быть применен на другой стороне к первому экземпляру ресурса для воссоздания второго экземпляра. Второй алгоритм состоит в сжатии выходного результата команды *diff -e* с помощью программы *gzip* для уменьшения размера разности. Третий метод использует новый алгоритм вычисления разности под названием *vdelta* (позднее переименованный в *vcdiff* [KV00]), который не только находит разность между экземплярами, но и осуществляет сжатие полученной разности.

Ряд ресурсов был исключен из рассмотрения, поскольку их типы содержания плохо подходили для вычисления разности. Например, изображения **GIF** и **JPEG** уже являются предварительно сжатыми, и вычисление разности для них принесет мало пользы. Программы, подобные *diff*, предназначены только для текстовых ресурсов и не будут работать с двоичными ресурсами. В то же время алгоритм *vcdiff* работает со любыми форматами данных, поэтому он использовался в исследовании для всех ресурсов, включая изображения.

РЕЗУЛЬТАТЫ ИССЛЕДОВАНИЯ

Здесь мы вкратце подытожим результаты исследования. Показатель дельта-приемлемости позволяет оценить полезность дельта-механизма в целом. Около 38% ресурсов в исследуемых трассах прокси-сервера оказались дельта-приемлемыми, в то время как только около 10% ресурсов оказались дельта-приемлемыми в трассах пакетов. Главная причина такого различия заключается в том, что при исследовании трасс прокси-сервера исключалось несколько типов содержания — главным образом нетекстовые типы, такие как GIF, JPEG, MPEG и аудиоформаты. При более широком исследовании никакие форматы не исключались, а данные собирались в различных местах, чтобы гарантировать репрезентативность распределения типов содержания, имеющихся в Web.

Были рассчитаны две группы показателей: количество байтов, которое будет сэкономлено, если передавать только разности между экземплярами, и примерная экономия времени, которая достигается за счет отказа от извлечения полного ответа. При подсчете экономии в исследовании не учитывались затраты на кодирование (и декодирование) разности. Поскольку разность вычисляется только при появлении изменений, затраты на кодирование и декодирование малы, если раскладываются на все содержание.

Исследование показало, что около 30% объема полных ответов, имеющих тела, не будут отправляться, если в качестве механизма кодирования разности используется *vcdiff*. Если получатель уже имел кэшированную копию предыдущего экземпляра, 83% объема тел ответов нет необходимости передавать. В ходе исследования также были определены затраты на вычисление разности и применение дельта-механизма к предыдущему экземпляру. Кроме того, исследование показало, что из всех протестированных алгоритмов *vcdiff* в большинстве случаев дал наилучший результат.

Один из результатов состоял в том, что библиотечная реализация (что является наиболее вероятным способом реализации вычисления разности для реальных Web-серверов) *vcdiff* работает с производительностью линии T-1 (193 Кб/с). Таким образом, все пользователи, пропускные способности соединений которых меньше или равны пропускной способности линии T-1, могут получить преимущество от кодирования разности и сжатия. Исследование подтвердило, что *vcdiff* является наиболее эффективным (с точки зрения затрат времени) для кодирования разности и сжатия. Исследование также показало, что кодирование разности и сжатие предпочтительнее сжатия, реализуемого современными модемами, особенно для больших файлов.

В результате исследования был сделан вывод, что широкое применение дельта-механизма может привести к значительному сокращению объема данных, передаваемых в Web, не увеличивая время ожидания при вычислении разности и воссоздании ответов. Применительно к трассам прокси-сервера исследование показало, что объем передаваемых данных ответа может быть сокращен почти на треть, а время передачи ответов, подвергнутых вычислению разности, сокращено почти на 40%. Для трасс пакетов с разнообразными типами содержания (нетекстовые типы не игнорировались) может быть достигнут очень большой выигрыш в объеме передаваемых данных (около 85%), если ресурсы дельта-приемлемы. Однако включение всех типов содержания снижает общую эффективность примерно на 9%. В результате исследования делается вывод, что дельта-приемлемые ответы должны подвергаться дельта-кодированию, а для других следует осуществлять лишь сжатие, чтобы по крайней мере уменьшить объем данных, передаваемых по сети.

ДРУГИЕ СООБРАЖЕНИЯ, СВЯЗАННЫЕ С ДЕЛЬТА-МЕХАНИЗМОМ

После выбора дельта-механизма, следует выбрать место его применения. Для применения дельта-механизма необходимо иметь как можно более раннюю версию ресурса. Естественным выбором будет прокси-сервер, который может хранить несколько версий ресурсов, чтобы добиться максимальной выгоды для расположенных за ним клиентов. Однако и браузер, который имеет в своем собственном кэше предыдущие версии набора больших, очень часто запрашиваемых ресурсов, также может оказаться подходящим местом для применения дельта-механизма. Размер кэша браузера может ограничивать число различных экземпляров, которые могут быть сохранены в браузере, особенно если ответ достаточно велик. Низкоскоростной канал пользовательского подключения к Internet может быть основным узким местом, а применение дельта-механизма ближе к пользователю способно значительно сократить время ожидания. Поскольку задержка в результате применения дельта-механизма меньше, чем время на выборку всего ресурса целиком, пользователь выигрывает за счет уменьшения времени ожидания. За счет кэширования результатов применения дельта-механизма кэширующий прокси-сервер может возвращать последнюю полную версию в ответ на последующие клиентские запросы. Конечно, если дельта-механизм может быть применен и на клиенте, и на про-

кси-сервере, клиент может запросить разность и также получить ее от прокси-сервера, даже если прокси-сервер имеет полную версию ресурса. В этом случае следует учитывать задержку при передаче большого ресурса между клиентом и прокси-сервером, которые расположены близко друг к другу, а также задержку, связанную с применением дельта-механизма на клиенте.

15.2.3. Проблемы, связанные с внедрением дельта-механизма в HTTP/1.1

Внедрение дельта-механизма требует изучения изменений, которые необходимо будет внести в протокол HTTP/1.1. Клиент или прокси-сервер должен иметь возможность заявить о своей поддержке дельта-механизма. Сервер, принимающий запрос, должен уметь генерировать разности для различных версий ресурсов и идентифицировать соответствующие разности. Рабочий проект [MKD+01] для этого предложения прошел несколько этапов стандартизации и на момент издания этой книги имел статус *предложения по стандарту*. Поскольку проект обсуждался в течение двух лет, итоговые предложения оказались достаточно зрелыми. Некоторые дополнения, не относящиеся к главному принципу применения дельта-механизма, были выделены в другой проект стандарта [MDH].

Далее мы представим краткое изложение сути предложения в трех частях:

1. Основные проблемы, связанные с внедрением дельта-механизма.
2. Обзор этапов преобразования HTTP-сообщения в процессе дельта-кодирования.
3. Краткое рассмотрение некоторых основных этапов преобразования.

Соображения, связанные с внедрением. К основным проблемам, связанным с внедрением дельта-механизма в HTTP/1.1, относятся следующие:

- **Идентификация экземпляров.** Клиент должен идентифицировать экземпляр (экземпляры), для которого (которых) он хотел бы вычислить разность. Клиент, который имеет несколько кэшированных экземпляров, может применить эвристику для определения экземпляров, которые он планирует включить в свой запрос. Кэшированный экземпляр и соответствующая этому экземпляру разность должны быть правильно идентифицированы, чтобы получить на сервере наиболее свежий экземпляр. Не все экземпляры, кэшированные клиентом, могут оставаться доступными на исходном сервере.
- **Синтаксис.** Должен быть определен синтаксис, относящийся к спецификации содержания и преобразования кодирования для клиента и сервера, который согласуется с выбором способа кодирования/декодирования разности. Клиент может поддерживать только некоторые из алгоритмов вычисления разности и иметь определенные предпочтения для этих алгоритмов. Клиент должен иметь возможность выразить свои предпочтения, чтобы повысить вероятность получения разности в том формате, который наиболее ему подходит. Решение при выражении предпочтений может зависеть от типа содержания и размера кэшированного ответа.
- **Реализация.** Формирование, передача, применение и кэширование разности составляют практические проблемы реализации дельта-механизма. Исходный сервер может оценивать затраты, связанные с применением дельта-механизма, на основе времени, которое требуется для формирования разностей для набора экземпляров. Если сформированная разность больше, чем текущий экземп-

ляр, исходный сервер может просто отправить сам экземпляр. Сервер может согласовать выбранный им образ действий с клиентскими предпочтениями, если при этом затраты для исходного сервера не увеличиваются. Например, исходный сервер может посчитать для себя более выгодным генерировать разность в одном избранном формате, предпочтя его всем другим. Если исходный сервер имеет кэшированные разности для предыдущих экземпляров, использующих определенный дельта-механизм, он может выбрать этот механизм, даже если данный конкретный механизм не является наиболее предпочтительным для обратившегося с запросом клиента. Принятие решения, какие разности кэшировать и как долго хранить кэшированные значения, представляют собой дополнительные задачи, которые должен решать исходный сервер.

Преобразование HTTP-сообщения. К основным действиям по преобразованию HTTP-сообщений, описанным в [MKD+01], относятся следующие:

- Идентификация запрашиваемого ресурса сервером.
- Выбор варианта ресурса.
- Генерирование тела, ассоциированного с *экземпляром* (но не с содержимым) ресурса.
- Формирование дельта-кодированного содержимого.
- Передача дельта-кодированного сообщения.

На каждом из этапов сервер может принять решение не генерировать разность: клиент может не иметь возможности принимать разности; экземпляр, для которого создается разность, может быть недоступен для клиента, и т.д. В проекте стандарта большое внимание уделено работе с диапазонами, поскольку диапазоны являются одной из форм манипулирования экземплярами. Это важно, поскольку клиент, допускающий использование дельта-механизма, может запросить разность для диапазона или диапазон дельта-кодированного экземпляра. При наличии возможности работы с диапазонами важное значение приобретает порядок кодирования разности, а клиент должен четко задавать последовательность, чтобы ответ не был двусмысленным. Чтобы позволить клиенту задавать порядок выполнения манипуляций над экземплярами, был введен новый заголовок запроса. Чтобы сервер мог задавать порядок, в котором следует выполнять манипуляции над двумя или более экземплярами, был введен новый заголовок ответа.

Исследование основных этапов обработки разности. Существуют следующие основные этапы обработки запроса клиента и ответа с кодированием разности:

- Клиент выражает свой интерес в применении дельта-кодирования.
- Сервер вычисляет и передает соответствующую разность.
- Осуществляется взаимодействие.
- Клиенту становятся доступными несколько экземпляров ресурса.

Сначала клиент выражает свою заинтересованность и возможность применения дельта-механизма с помощью заголовка **A-IM** (Accept-Instance-Manipulation), который задает множество приемлемых алгоритмов вычисления разности. Например,

```
GET /chap15.html HTTP/1.1
Host: www.vcdiff.com
If-None-Match: "38432-s8-13"
A-IM: vcdiff, diffe, gzip
```

означает, что клиент предпочитает в первую очередь кодирование *vcdiff*, а во вторую очередь кодирование *diffe* для разности и *gzip* в качестве формата сжатия, даже если ответ не был кодирован с применением дельта-кодирования. Заголовок **If-None-Match** идентифицирует экземпляр, к которому следует применять дельта-механизм. Как и во всех других случаях, сервер может возвращать ответ **304 Not Modified**, если текущий экземпляр ресурса **chap15.html** совпадает с вариантом, связанным с атрибутом содержимого "38432-s8-13". Если текущий ресурс имеет другой атрибут содержимого, сервер, допускающий применение дельта-механизма, может вычислить разность между текущим экземпляром и экземпляром, на который ссылается атрибут содержимого "38432-s8-13". Клиент предпочитает использовать алгоритм *vcdiff*, и сервер должен попытаться применить именно этот механизм вычисления разности, если такое возможно.

Сервер может принять решение не вычислять разность. На это есть несколько причин. Например, сервер имеет основание предположить, что ответ, содержащий разность, будет иметь больший размер, чем текущий экземпляр ресурса. Сервер может использовать эвристику, основанную на размере текущего экземпляра или на прошлом опыте применения механизма кодирования. Сервер может иметь кэшированную разность для пары экземпляров, если они часто запрашиваются, и может передать кэшированное значение. В любом случае серверный ответ будет иметь вид, подобный следующему:

```
HTTP/1.1 226 IM Used
Date: Sun, 2 Jul 2000 23:35:35 GMT
ETag: "192-qpt-899"
IM: vcdiff
...
```

Сервер использует новый код ответа **226 IM Used** и заголовок ответа **IM**, указывающий на выбор алгоритма кодирования разности. Экземпляр ресурса имеет новый атрибут содержимого (**192-qpt-899**), который будет предположительно сохранен прокси-сервером или клиентом для дальнейшего использования.

Предположим, клиент заинтересован в получении только части разности между двумя экземплярами. Клиент может попросить сервер вычислить разность, а затем извлечь часть разности:

```
GET /foo.html HTTP/1.1
Host: www.vcdiff.com
If-None-Match: "38432-s8-13"
A-IM: vcdiff, range
Range: bytes=0-200
```

Сервер может отправить ответ, подобный следующему:

```
HTTP/1.1 226 IM Used
Date: Mon, 20 Nov 2000 11:48:45 GMT
ETag: "192-pqr-99"
IM: vcdiff, range
...
```

который подразумевает, что сервер вычислил разность с использованием *vcdiff* для текущего экземпляра на основе копии экземпляра с атрибутом содержимого "38432-s8-13", а затем извлек первые 201 байт этого кодированного с помощью *vcdiff* экземпляра. Обратите внимание, что заголовок Instance Manipulation (IM)

указывает порядок, в котором осуществлялось манипулирование экземплярами: кодирование разности с последующим извлечением диапазона.

Наличие прокси-серверов, которые не воспринимают дельта-механизм, заставляет ввести новый код ответа. Поскольку большинство существующих на сегодняшний день прокси-серверов не поддерживают даже последнюю версию HTTP, маловероятно, что в ближайшем будущем прокси-серверы смогут поддерживать дельта-механизм. Большинство прокси-серверов игнорирует коды ответов, которые они не понимают, и пересылают эти ответы, не кэшируя их; следовательно, они будут вести себя аналогичным образом и для ответов с новым кодом **226 IM Used**. Прокси-сервер, который не поддерживает дельта-механизм, но ошибочно кэширует ответ с разностным кодом, может вернуть такой ответ клиенту, который не способен обработать разность. При использовании нового кода ответа **226 IM Used** прокси-сервер, не воспринимающий дельта-механизм, скорее всего не будет кэшировать ответ, поскольку он не понимает код ответа. Таким образом, маловероятно, что прокси-сервер возвратит неверный ответ клиенту, обратившемуся с запросом на этот же ресурс позднее.

Клиент может иметь различные кэшированные экземпляры, для которых он хотел бы получить разность, а сервер, вычисляющий разность, может отправлять разности для одного из этих нескольких экземпляров. В таких случаях клиент может воспользоваться существующим заголовком **If-None-Match** для указания *всех* имеющихся экземпляров, для которых допускается получение разности. Клиент может отправить запрос, подобный следующему:

```
GET /chap15.html HTTP/1.1
Host: www.vcdiff.com
If-None-Match: "38432-s8-13", "97-ru486-v", "2-dumbya-1f"
A-IM: vcdiff, diffe, gzip
```

который указывает, что клиент имеет три основных экземпляра, для которых допускается получение разности. Сервер может воспользоваться эвристикой при принятии решения, для какого из этих трех экземпляров следует сформировать разность, а затем указать выбранный базовый экземпляр, например:

```
HTTP/1.1 226 IM Used
Date: Sun, 2 Jul 2000 23:35:35 GMT
ETag: "282-ela-899"
IM: vcdiff
Delta-Base: "97-ru486-v"
...
```

Сервер выбирает экземпляр с атрибутом содержимого "97-ru486-v" в качестве базы вычисления разности для текущего экземпляра ответа. Выбор экземпляра указывается в новом заголовке **Delta-Base**. Поле **ETag** в ответе с разностью идентифицирует текущий экземпляр ответа. Клиент теперь знает, как реконструировать последнюю версию ресурса. Если экземпляр, для которого создавалась разность, не был указан, то подразумевается, что запрос должен иметь возможность уникально идентифицировать базовый экземпляр. Сервер просто возвращает разность для этого базового экземпляра и не обязан предоставлять какую-либо дополнительную информацию.

На рис. с 15.1 по 15.4 представлены упрощенные версии этапов обработки разности. Клиент имеет в своем кэше экземпляр **v1** ресурса **r** с ассоциированным с ним атрибутом содержимого "xyz", полученным от исходного сервера A. Теперь предположим, что клиент отправляет запрос на ресурс исходному серверу A, а в за-

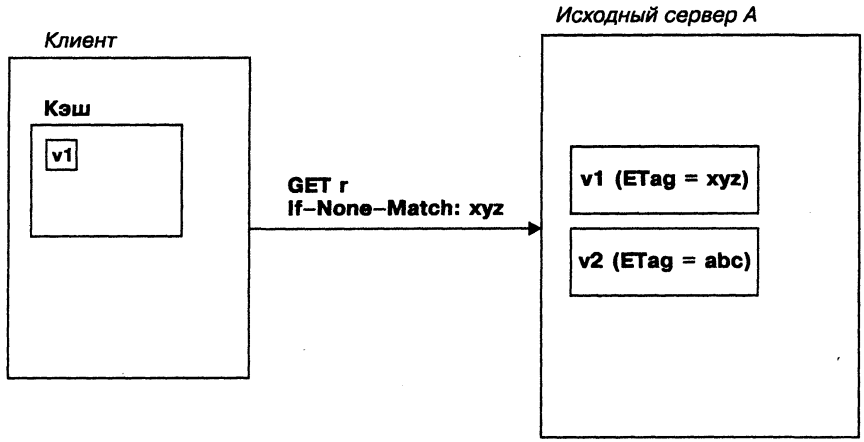


Рис. 15.1. Клиент запрашивает ресурс **r** с исходного сервера

головке **If-None-Match** запроса указан атрибут содержимого ETag экземпляра, который он имеет в своем кэше (рис. 15.1).

Исходный сервер имеет новый экземпляр ресурса, а именно **v2**, с ассоциированным с ним атрибутом содержимого "abc". Исходный сервер A формирует разность между экземплярами **v1** и **v2** и возвращает ее клиенту вместе с атрибутом содержимого нового экземпляра, а именно, "abc" (рис. 15.2). Клиент реконструирует текущий экземпляр ресурса **r** на основе кэшированного экземпляра **v1** и разности.

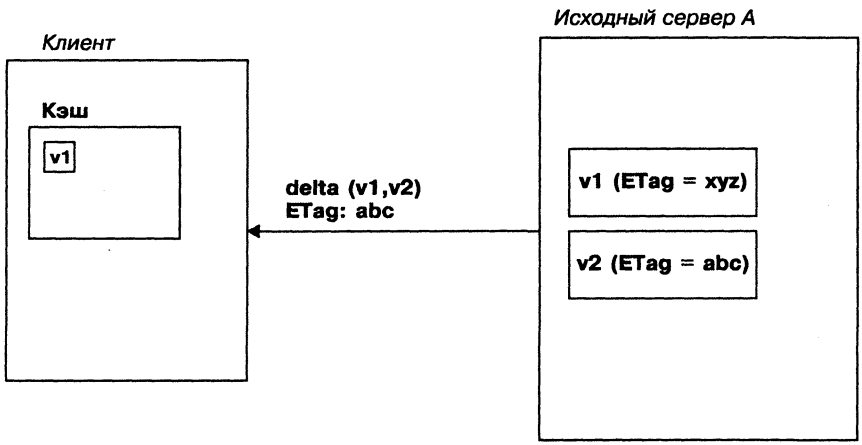


Рис. 15.2. Исходный сервер возвращает разность относительно экземпляра **v1**

На рис. 15.3 представлен клиентский запрос на этот же ресурс, когда клиент имеет два кэшированных экземпляра **v1** и **v2**.

Текущим является экземпляр **v3**. Исходный сервер вычисляет разность между экземплярами **v2** и **v3**, значение атрибута содержимого для нового экземпляра "pqr". Он также включает заголовок **Delta-Base** со значением "abc", чтобы уведомить клиента, что базовым экземпляром, относительно которого была вычислена разность, был **v2** (рис. 15.4).

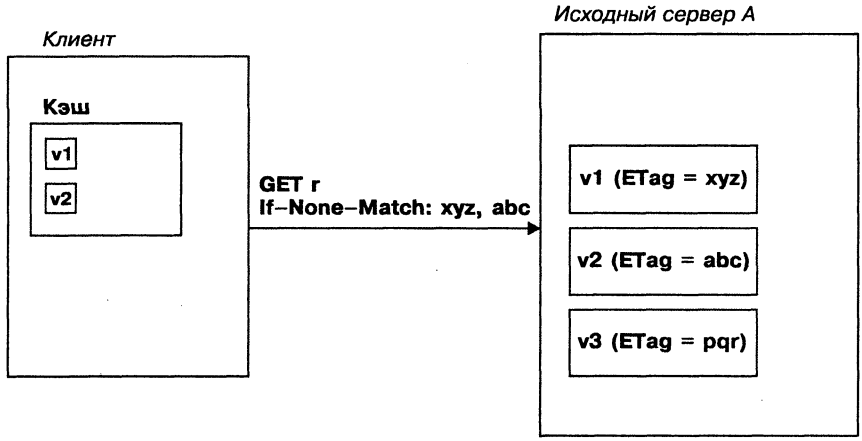


Рис. 15.3. Клиент запрашивает ресурс **r**, указывая свои кэшированные экземпляры

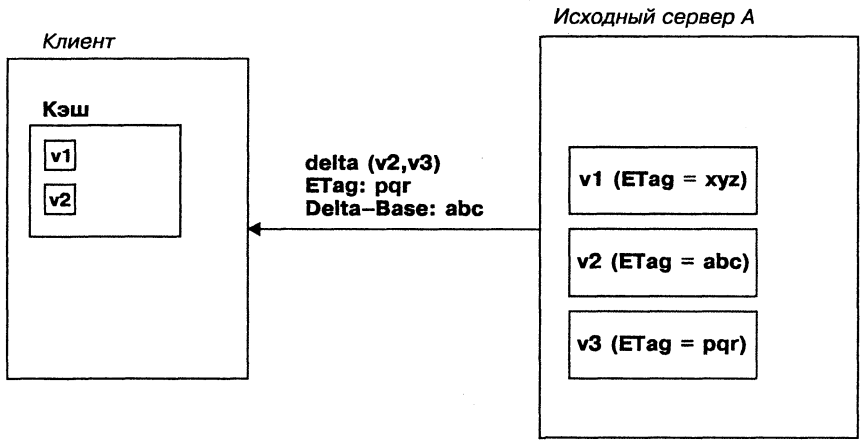


Рис. 15.4. Исходный сервер возвращает разность между текущим экземпляром и экземпляром **v2**

Сервер должен решить, сколько базовых экземпляров хранить, а также как долго. Такое решение, подобно решению по замещению содержимого кэша, принимается на основе частоты использования, времени создания и т.д. Для каждого базового экземпляра, хранимого сервером, в зависимости от сделанного клиентом выбора сервер должен принять ряд решений. Например, сервер должен принять решение, должен ли он вычислять несколько разностей и отправлять наилучший вариант, либо выбирать один вариант случайно. Сервер также должен выбрать стратегию замещения, позволяющую уничтожать устаревшие экземпляры, подлежащие замещению, на новые экземпляры. Проблемы, связанные с такой стратегией, не рассматриваются в данном предложении, не обсуждались при подготовке рабочего проекта по использованию дельта-механизма.

15.2.4. Текущее состояние дел с внедрением дельта-механизма в HTTP/1.1

Предложенная версия стандарта на момент издания этой книги рассматривается IETF. Имеются, по меньшей мере, две известные реализации предложения. Функционально совместимые реализации пока не созданы, хотя несколько университетов работают над этим.

15.3. Совместимость с протоколом HTTP/1.1

По мере распространения HTTP/1.1 стали появляться многочисленные реализации браузеров, прокси-серверов и Web-серверов. В достаточно короткое время перестройка рынка программных Web-компонентов привела к появлению нескольких доминирующих продуктов. В конце 1999 года три Web-сервера: Apache, Microsoft/IIS и Netscape-Enterprise занимали свыше 95% рынка. Годом позже, в конце 2000 г., эти три Web-сервера по-прежнему доминировали, хотя их доли на рынке несколько изменились. Среди браузеров популярными остаются два основных (Internet Explorer и Netscape Navigator), каких-либо изменений здесь не предвидится.

Проблемы, связанные со следованием спецификации протокола, выражены достаточно прямолинейно: являются ли реализации совместимыми с требованиями, которые присутствуют в протоколе? В главе 6 (раздел 6.5) мы говорили о различных уровнях требований, предъявляемых к программным компонентам при реализации протокола. Чтобы реализация считалась совместимой со спецификацией протокола, она должна удовлетворять различным требованиям уровней **MUST** (Обязательно), **SHOULD** (Желательно) и **MAY** (Возможно) для синтаксиса и семантики. Любая реализация, которая не удовлетворяет всем требованиям уровня **MUST**, считается *несовместимой* со спецификацией протокола. Реализация, удовлетворяющая всем требованиям уровней **MUST** и **SHOULD**, является *безусловно* совместимой. Если соблюдаются не все требования уровня **SHOULD**, реализация считается *условно* совместимой.

Как ни странно, какие-либо механизмы для выяснения, отвечают ли программные реализации (хотя бы несколько из наиболее популярных) требованиям совместимости с HTTP/1.1, отсутствуют. В этом разделе мы рассмотрим, как проверить, какие из требований HTTP, описанные в документе RFC 2616, учитываются в реализации сервера. Мы также поговорим, к чему может привести несовместимость с требованиями спецификации. Материал этого раздела основан на данных отчета [KA01] по результатам проведения первого масштабного и продолжительного исследования совместимости со спецификацией HTTP, проведенного в течение 16 месяцев с июня 1999 г. по сентябрь 2000 г. Мы рассмотрим методологию тестирования, а затем обсудим результаты исследования. Следует заметить, что на протяжении всех исследований не обязательно тестировались одни и те же сайты. Это было вызвано тем, что популярность сайтов во время проведения исследований изменялась. Результаты, однако, не расходились с теми, которые были получены при выполнении тестирования для сайтов, популярность которых оставалась неизменной во время проведения исследований.

15.3.1. Мотивация для проведения исследований совместимости со спецификацией протокола

Информация о следовании спецификации протокола позволяет оценить степень освоения протокола и полезность новых функциональных возможностей протокола. Определение показателей совместимости необходимо для оценки доли Web-трафика, использующего определенную версию протокола. Администраторы сайтов могут воспользоваться информацией о степени распространения версий протокола, чтобы принять решение, следует ли переходить на новую версию. Исследования совместимости помогают разработчикам протокола. Данные о том, что определенная функция не получила широкого распространения, обеспечивают полезную обратную связь. Разработчики могут попытаться понять, почему некоторые функции были быстро восприняты, а другие нет.

Взаимная совместимость различных реализаций клиентов, прокси-серверов и серверов основывается на совместимости компонентов с протоколом. Поскольку компоненты создаются различными группами разработчиков, единственным объединяющим фактором, гарантирующим должное функционирование обменов данными в Web, является способность компонентов соответствовать ожидания со стороны других компонентов. Если браузер не уверен, что определенный запрос будет воспринят сервером, функционирующим под определенной версией протокола, он не сможет предложить универсальный интерфейс пользователям, работающим с ним. Задача прокси-сервера и так достаточно сложна, принимая во внимание, что он должен действовать как сервер для клиентов и как клиент для серверов. Если, помимо этого, прокси-сервер должен приспосабливаться к различным уровням совместимости с протоколом для различных клиентов и серверов, сложность выполняемых им задач может значительно возрасти. Хотя совместимые реализации могут быть уязвимыми для атак, связанных с отказом от обслуживания, или других попыток нарушения безопасности, для несовместимого сервера риск намного выше.

15.3.2. Тестирование совместимости клиентов и прокси-серверов

Проблемы совместимости для серверов поднимают вопрос и о совместимости двух других основных Web-компонентов: клиентов и прокси-серверов. Тестирование совместимости клиентов обычно более простая операция, чем тестирование совместимости серверов. Браузеры дают возможность настраивать некоторые из своих параметров. Например, IE 5.0 позволяет пользователю выбирать, какую версию протокола использовать (HTTP/1.0 или HTTP/1.1). Выбор при этом делается не между HTTP/1.0 и HTTP/1.1 в целом, но между подмножеством функций, характерных для HTTP/1.1. Что еще более важно, реализованы ли доступные функции совместимым образом. Например, браузер может объявить о реализации функции запроса диапазона **Range** HTTP/1.1, но посылать запросы без указания диапазонов байтов и получать в ответ полное тело ответа вместо только тех областей, которые необходимы. Однако с учетом того, что пользователь имеет возможность управлять браузером, локально установленный монитор пакетов может отображать группы заголовков, включенных в запросы. Группа заголовков ответов также может быть легко перехвачена. Требования протокола применительно к клиентам обычно менее строгие. Выполнить проверку их соблюдения для клиентов, как правило, проще, чем для серверов.

Тестирование совместимости прокси-серверов — гораздо более сложная задача. Спецификация протокола по большей части умалчивает о специфических правилах для прокси-серверов, за исключением общих замечаний, что поскольку прокси-сервер является и клиентом, и сервером, совместимый прокси-сервер должен представлять собой и совместимый клиент, и совместимый сервер. Хотя это обстоятельство и увеличивает сложность прокси-сервера как программного компонента, тестирование его на совместимость затруднено по другим причинам. Совместимость прокси-сервера с протоколом должна тестироваться и при работе его в качестве клиента, и при работе его в качестве сервера. Для направления запроса в Internet через прокси-сервер требуется разрешение. Хотя имеется ряд прокси-серверов, которые достаточно снисходительны и позволяют прохождение запросов, это может рассматриваться как несоответствующее использование их вычислительных ресурсов. Даже если разрешение на выполнение тестирования совместимости прокси-сервера получено, фактическое содержимое запроса, который прокси-сервер посылает серверу, не будет известно. Конфигурации различных прокси-серверов существенно различаются — системные администраторы часто модифицируют конфигурацию прокси-серверов для кэширования. Что еще более важно, пока лишь малое число прокси-серверов соответствуют HTTP/1.1. Еще более осложняет ситуацию то обстоятельство, что некоторые прокси-серверы HTTP/1.0 реализуют ряд функций HTTP/1.1. Путь HTTP-сообщения между клиентом и исходным сервером может проходить через несколько прокси-серверов, некоторые из которых совместимы с HTTP/1.0, а другие — с HTTP/1.1. Прокси-серверы, совместимые с HTTP/1.1, могут выявляться по наличию заголовков **Via**, а прокси-серверы, совместимые с HTTP/1.0, идентифицировать невозможно. Имеет смысл тестировать прокси-серверы на совместимость в том случае, если путь сообщения от одной конечной точки до другой главным образом состоит из компонентов, совместимых с HTTP/1.1.

В оставшейся части раздела основное внимание уделяется тестированию совместимости серверов.

15.3.3. Методология тестирования на совместимость

Ниже перечислены способы тестирования на совместимость с протоколом:

- Разработчики компонентов могут обратиться к таблице [Fea], сформированной Консорциумом Всемирной паутины (W3C) и содержащей различные функции, чтобы выявить уже реализованные функции. Как правило, тем самым можно обеспечить соответствие реализации требованиям, задаваемым протоколом. Однако в реальности это не всегда так.
- Программное обеспечение может подвергаться многократным тестам, часть из которых представляет собой тесты на совместимость. На практике это не слишком широко распространено. Большинство тестов проверяют, способен ли компонент справиться с большим потоком запросов (для серверов) и способен ли отображать произвольные ответы (для браузеров). Однако нельзя сказать, что тесты на совместимость являются составной частью используемых тестов.
- Серверы могут быть проверены в *оперативном* режиме путем отправки запросов с удаленных клиентов. Это хороший способ тестирования различных серверов. Каждый Web-сервер можно тестировать путем создания запросов группой клиентов и контроля корректности ответов.

Последний подход наиболее пригоден для тестирования совместимости, поскольку тестирование работающего Web-сервера само по себе дает достаточно реалистичный ответ на вопрос о совместимости. Web-серверы, как мы видели в главе 4 (раздел 4.6), настраиваются по многими параметрами, каждый из которых может иметь различные значения в зависимости от конкретного сайта. Популярный Web-сервер Apache имеет около 700 комбинаций параметров настройки [KA01]. Таким образом, выполнять тесты локально с перебором всех возможных комбинаций значений параметров едва ли возможно. Причина, по которой сайты настраиваются различным образом, может быть связана с конкретным сочетанием ресурсов на сайте, средней скоростью поступления запросов и временем обработки запросов. Кроме того, некоторые запросы могут иметь побочные эффекты. Многие запросы изменяют содержание сайта, а сценарии требуют для своего выполнения определенного окружения. Таким образом, идея воссоздавать все эти условия для тысяч сайтов с помощью локальных настроек не может быть реализована.

Приняв решение тестировать работающие Web-серверы, мы должны выбрать, какие серверы из миллионов имеющихся следует тестировать. Ниже приведены некоторые способы выбора серверов:

- **Выборочный метод.** Выборочный метод предполагает выбор случайного подмножества серверов из нескольких миллионов имеющихся. Однако при этом приходится выбирать различные реализации и различные конфигурации для каждого типа сервера. Исчерпывающее тестирование одного сервера во всех возможных конфигурациях является непрактичным решением.
- **Функциональное назначение.** Ранее, в главе 4 (раздел 4.1), мы познакомились с различными способами классификации сайтов на основе их функционального назначения. Выбор серверов может также осуществляться по тому же принципу: внутренний сайт компании, новостной сайт, портал, сайт поисковой системы, сайт электронной коммерции и т.д. Степень совместимости серверов может варьироваться в зависимости от функционального назначения сайта. Еще более важно, что несоответствие сервера определенным требованиям может иметь большее значение для одних сайтов по сравнению с другими. Например, для сайта электронной коммерции отсутствие совместимости может оказаться более критичным, чем для внутренних корпоративных сайтов. Это связано с тем, что сервер электронной коммерции с большей вероятностью может подвергнуться атакам отказа от обслуживания, чем внутренний корпоративный сайт. Таким образом сделать разумный выбор на основе только функционального назначения довольно трудно.
- **Популярность.** Другой подход учитывает популярность *Web-сайтов*, реализацию Web-сервера и параметры настройки. Исследования популярных сайтов показали, что несколько сотен сайтов получают около половины всех запросов. Принимая во внимание это обстоятельство, можно предположить, что для тестирования совместимости серверов достаточно будет рассмотреть несколько сотен наиболее популярных сайтов. Среди множества компаний, подсчитывающих степень популярности Web-сайтов, наиболее известными являются MediaMetrix [Med], Hot 100 [Hot], Alexa1000 [Ale] и Netcraft [Netc]. Каждая компания использует различные методы для определения степени популярности и периодически обновляет свои рейтинги.

В исследовании, с которым мы познакомимся ниже, используется метод, учитывающий популярность, хотя последующие исследования могут осуществлять выбор сайтов для тестирования на основе другой методологии. Чтобы обеспечить

большую широту охвата, в исследовании были объединены результаты, взятые из двух рейтинговых списков: Fortune 500 [For99] и Global 200 [Glo98]. Объединенный набор сайтов в определенном смысле представляет канонический перечень наиболее популярных сайтов.

15.3.4. PRO-COW. Масштабное исследование совместимости

Методология исследования совместимости в Web (PRO-COW — Protocol Compliance on the Web) заключается в генерировании множества запросов, охватывающих различные функции HTTP/1.1, и автоматической проверке ответов с целью определить, удовлетворяют ли они требованиям, содержащимся в спецификации. Например, если предполагается, что каждый ответ имеет заголовок **Date**, ответы сервера, в которых этот заголовок отсутствует, считаются несовместимыми. Более подробное описание исследования PRO-COW можно найти в [KA01].

Описание исследования совместимости поделено на четыре части и начинается с различных категорий тестов. После рассмотрения, имеет ли значение местоположение клиента для выполнения тестирования, мы вкратце рассмотрим среду для выполнения тестирования, программное обеспечение, используемое для выполнения тестов. Наконец, мы познакомимся с реальными тестами и их результатами. С учетом эволюции Web-технологий особое значение приобретает долговечность результатов тестирования. За счет выполнения трех отдельных групп измерений за период в 16 месяцев снижается риск непостоянства результатов.

КАТЕГОРИИ ТЕСТОВ НА СОВМЕСТИМОСТЬ

Поскольку не все нарушения требований совместимости одинаково серьезны, следует оценить важность различных тестов на совместимость. В этой связи тесты на совместимость делятся на три части в порядке уменьшения значимости:

1. **Тестирование соответствия требованиям уровня MUST.** Эти требования четко сформулированы, и компоненты должны полностью им следовать. Любая реализация, которая не отвечает *всем* требованиям MUST, считается несовместимой. Разработчики протокола подчеркивают, что неспособность функции подчиняться требованиям уровня MUST может привести к семантическим нарушениям. Например, проект стандарта HTTP/1.1 настаивает, что во всех сообщениях в долговременном соединении ДОЛЖНА (MUST) быть указана длина. Предположим, что сервер, обслуживающий долговременное соединение, отправляет сообщение без явного указания его длины. Получатель не знает, получил ли он все сообщение целиком. С учетом того факта, что может быть отправлено дополнительное сообщение (поскольку соединение является долговременным), получатель не будет знать, как отделить два сообщения друг от друга. Аналогично, прокси-сервер НЕ ДОЛЖЕН (MUST NOT) посылать сообщение со значением указателя версии протокола, превышающим его фактическую версию. Версия протокола является индикатором функциональных возможностей отправителя. Получатель может ошибочно предположить, что отправитель реализует определенные функции. Например, получатель может подумать, что отправитель способен принимать и декодировать сообщения, разделенные на фрагменты.
2. **Тестирование новых функций HTTP/1.1.** HTTP/1.1 присущи несколько новых функций (о различиях между HTTP/1.0 и HTTP/1.1 говорилось в главе 7). Некоторые из функций, такие как долговременные соединения, запросы

на диапазоны, являются более важными, чем другие, такие как уведомление об ошибке. Однако полный комплект тестов на совместимость должен проверять все новые функции новой версии протокола. Например, важным добавлением в HTTP/1.1 является запросы на диапазоны. Однако, как обсуждалось в главе 7 (раздел 7.4.1), возможность принимать запрос на диапазон в HTTP/1.1 не является обязательной. Серверам рекомендуется принимать такие запросы, поскольку это уменьшает число пакетов, передаваемых по сети. Точно так же, способность использовать долговременные соединения, хотя и подразумевается по умолчанию, но не является требованием уровня MUST в HTTP/1.1.

3. **Тестирование других изменений.** Как указывалось в главе 7, количество изменений, введенных в HTTP/1.1, достаточно велико. После разбиения изменений на категории и упорядочения можно выполнить их тестирование. Например, в спецификацию были введены предупреждения, связанные с обработкой сообщений произвольной длины. Сервер может оказаться неспособным обрабатывать длинные URI запросов. Хотя спецификация протокола не содержит конкретного ограничения длины, она вводит код ответа **414 Request-URI Too Long**. Реализация сервера должна быть достаточно интеллектуальной, чтобы контролировать переполнение буфера при получении сообщения-запроса с очень длинным URI.

МЕСТОПОЛОЖЕНИЕ КЛИЕНТОВ, С ПОМОЩЬЮ КОТОРЫХ ВЫПОЛНЯЕТСЯ ТЕСТИРОВАНИЕ

После того, как набор тестов выбран, должно быть определено местоположение клиентов, на стороне которых выполняется тестирование. Технически местоположение клиентов, выдающих запрос, не должно иметь значения. Web-сервер, который не использует в ответах информацию о местоположении клиента и его идентификационные данные, будет отвечать одним и тем же образом, независимо от того, кем был выдан запрос. При выполнении теста, при котором запросы передаются от клиента непосредственно исходному серверу, это не является проблемой. Однако если запрос проходит через один или нескольких прокси-серверов, местоположение клиента имеет значение. Местоположение клиента может повлиять на то, какая реплика сервера получает запрос и отвечает на него. Исследование игнорирует роль прокси-серверов — все запросы передаются исходному серверу напрямую. При проведении более широкого исследования совместимости роль прокси-серверов учитывается — осуществляется тестирование, является ли прокси-сервер совместимым с протоколом, а также оказывает ли прокси-сервер влияние на выполнение теста на совместимость при пересылке запросов исходному серверу. Если исходный сервер имеет заместителя или имеются другие локальные серверы (например, с изображениями), они также подлежат тестированию. Такие серверы могут функционировать под другой версией серверного программного обеспечения, отличающейся от версии исходного сервера. Серверы-заместители могут работать не под HTTP/1.1, поэтому от них нельзя требовать совместимости с HTTP/1.1. Для проверки гипотезы, оказывает ли влияние местоположение клиентов на результаты проверки на совместимость, использовались клиенты, местоположение которых было различно.

СРЕДА ДЛЯ ТЕСТИРОВАНИЯ СОВМЕСТИМОСТИ

После того, как выбор тестов и местоположения был обоснован, должна быть выбрана программа для тестирования. Для тестирования базовой совместимости можно связаться с Web-сервером с помощью *telnet* (через порт 80), но эта возмож-

ность не подходит для выполнения обширных тестов в автоматическом режиме. Основной программой, используемой для генерирования групп запросов, является клиентское инструментальное средство *httperf* [MJ98], которое представляет собой настраиваемое средство измерения производительности с тремя основными логическими компонентами: ядро, которое посылает HTTP-запросы и осуществляет синтаксический анализ HTTP-ответов, генератор рабочей нагрузки, который формирует потоки HTTP-запросов, и средство сбора статистики, который собирает статистические данные о сделанных запросах. Средство *httperf* поддерживает HTTP/1.1 и доступно в исходных кодах, что облегчает модификацию. Выбор расположения клиентов для выполнения тестов — более сложная задача. В идеале желательно выбирать репрезентативный набор клиентов, подобно тому, как это делается при выборе серверов. Однако в отличие от статистики популярности серверов статистические данными о распределении местоположения клиентов отсутствуют. Как оказалось, местоположение клиентов особой роли не играет. Однако как мы увидим в резюме (раздел 15.3.5), последующие более глубокие исследования уже не смогут игнорировать местоположение клиентов. В ходе данного исследования тесты выполнялись как в организациях, где работали авторы, так и в различных странах.

ТЕСТЫ И РЕЗУЛЬТАТЫ ИССЛЕДОВАНИЯ СОВМЕСТИМОСТИ

Исследование совместимости выполнялось трижды в течение 16 месяцев. Во время выполнения тестов тремя наиболее популярными серверами[1] являлись Apache, Netscape-Enterprise и Microsoft-IIS. Указанные серверы составляют около 95% от всех тестируемых серверов. Относительная популярность серверов для выполняемой группы тестов не обязательно распространяется на Web в целом. Например, доля Web-сервера Apache устойчиво составляла 60% [Netc], в то время как в процессе тестирования выборка серверов Apache составила около 30%. Для серверов Apache и Netscape-Enterprise информация о конфигурации может быть получена из заголовка ответа **Server**. Вот несколько примеров из нескольких сотен различных вариантов конфигураций, которые могут быть получены для серверов Apache и Netscape:

```
Server: Apache/1.3b3 mod_perl/1.07
Server: Netscape-Communications/1.1+SiteTrack 1.10i26mck
Server: Apache/1.2.5 FrontPage/3.0.3
```

Однако серверы Microsoft-IIS не сообщают каких-либо параметров конфигурации за исключением идентификационных номеров версий программного обеспечения (например, IIS/4.0 и IIS/5.0). Результаты исследования представлены с разбивкой по серверам и не могут быть обобщены на все имеющиеся версии.

К первой категории тестов относится тестирование методов **GET** и **HEAD**, а также проверка правильности реакции на отсутствие заголовка запроса **Host**. Напомним, что правильная обработка заголовка **Host** (см. главу 7, раздел 7.8) является требованием уровня MUST, а все серверы, поддерживающие HTTP/1.1, обязаны возвращать ошибку, если заголовок отсутствует в каком-либо запросе HTTP/1.1. Около трети серверов не являются совместимыми в отношении обработки заголовка **Host**.

Условная совместимость имеет место в случае отсутствия заголовков **Content-Length** и **Transfer-Encoding: chunked**. Около 80% серверов работают с методом **GET** должным образом. Совместимость с запросами **GET** требует от серверов по-

---

[1] Здесь и далее мы используем термин «сервер» для обозначения функционирующего на тестируемом сайте программного обеспечения Web-сервера.

сылать код состояния **200 OK** и включать в ответы соответствующие заголовки, такие как **Date** и либо **Content-Length**, либо **Transfer-Encoding: chunked**. Свыше 70% серверов являются безусловно совместимыми при работе с методом **HEAD**. При этом требуется, чтобы ответ содержал те же заголовки содержимого, что и ответ на запрос **GET**, а также, чтобы ответ имел надлежащий заголовок **Date**.

Серверы Apache и Microsoft-IIS показали себя с наилучшей стороны: они выдержали тесты. Около 20% серверов Netscape-Enterprise не выдержало всех тестов. Последним версиям Netscape-Enterprise свойственно большее количество проблем, чем более ранним версиям. Хотя результаты были агрегированы по всем тестированным серверам, между группами серверов Apache, Microsoft-IIS и Netscape-Enterprise имеются значительные различия. Свыше одной пятой всех серверов не прошло, по меньшей мере, один из трех тестов. На момент проведения последнего исследования осенью 2000 года доля серверов, не прошедших тестов, составляла еще около 20%, хотя количество серверов, не выдержавших все три теста, несколько уменьшилось.

Тесты второй категории предусматривали проверку важных новых функций, появившихся в HTTP/1.1, таких как долговременные соединения, конвейеризация и запросы на диапазоны. Как обсуждалось в главе 7 (раздел 7.5.2), долговременные соединения относятся к требованиям уровня SHOULD (Желательно). Однако из-за выгоды, которую приносят долговременные соединения, они приняты в HTTP/1.1 по умолчанию. Другими словами, сервер должен быть явным образом настроен, если необходимо *запретить* долговременные соединения. Около 70% серверов поддерживают долговременные соединения. Примерно столько же серверов способны обслуживать конвейерные запросы. Однако лишь половина из подвергнутых тестированию серверов оказалась способна обслуживать запросы на диапазоны. Опять-таки, возможность обработки запросов на диапазоны является требованием уровня SHOULD (Желательно) в спецификации протокола. В целом при проведении тестов второй категории только 40% серверов оказались полностью совместимыми, а 20% серверов не прошли все тесты. Третье исследование, выполненное осенью 2000 г., не показало каких-либо значительных улучшений относительно двух предыдущих исследований.

Объединенные тесты первой и второй категорий прошло примерно 30% серверов, а 7% серверов провалило все тесты.

Наконец, тесты третьей категории ориентированы на менее значимые функции, такие как редко используемые методы (**OPTIONS**, **TRACE**), потенциально рискованные ситуации (длинные URL) и различные форматы представления дат. Хотя тестируемые методы пока еще используются мало, в будущем ситуация может измениться. Метод **OPTIONS** (см. главу 7, раздел 7.7.1) может использоваться для получения информации о функциональных возможностях сервера. Отправитель может, например, проверить, способен ли сервер обслуживать передачу с сообщений, разделенных на фрагменты, прежде чем отправить запрос.

Наиболее показательный тест состоял в проверке обработки длинных URI запросов. Ряд серверов не проверял переполнение буфера, что приводило к ненормальному функционированию серверного программного обеспечения. К счастью, эти ошибки были быстро устранены после того, как авторы исследования уведомили разработчиков программного обеспечения серверов. Принимая во внимание, что для многих компаний, занимающихся электронной коммерцией, требуется, чтобы их Web-сайты были постоянно доступными, подобная простая проблема не должна остаться без внимания. Наличие таких проблем заключается в том, что разработчики, как правило, уделяют требованиям совместимости уровня SHOULD меньше внимания, а возмож-

ность обработки длинных URI относится именно к требованиям уровня SHOULD. Разработчики протокола могут принимать решения о включении требований в тот или иной уровень совместимости на основе дополнительных критериев, таких как потенциальная возможность возникновения катастрофических ситуаций в результате того, что определенная функция не была подвергнута тщательному тестированию. Не все программные ошибки могут быть устранены за счет повышения строгости требований. Однако в рискованных ситуациях следует принимать во внимание относительно разрешительный характер требований уровня SHOULD. Альтернативной возможностью является добавление потенциально рискованных ситуаций в раздел Security Considerations (Соображения по безопасности) спецификации протокола.

15.3.5. Совместимость с протоколом. Резюме

Исследования, выполнявшиеся в течение 16 месяцев, выявили повышение степени совместимости, но процесс тестирования должен быть более строгим. В различных списках рассылки проводилась дискуссия о проведении *не афишируемых* тестов, результаты которых не подлежали бы публичному распространению, чтобы не приводить в замешательство разработчиков программных компонентов. Хотя эта идея достаточно интересна, публикация результатов проведения стандартного набора тестов на совместимость основными производителями программного обеспечения над своими продуктами, представляется значительно лучшим вариантом. Пользователи и администраторы серверов могут быстро оценить уровни совместимости различных Web-компонентов. Большая степень открытости дает возможность разработчикам сделать их программное обеспечение более совместимым со спецификацией протокола.

Более широкое тестирование на совместимость должно охватывать больший набор местоположений клиентов и обеспечивать проведение тестирования прокси-серверов наряду с исходными серверами. К концу 2000 г. прокси-серверы, совместимые с HTTP/1.1, еще не получили широкого распространения, хотя в этом направлении ожидаются скорые изменения. Помимо этого, техника выбора серверных сайтов для тестирования может основываться на других критериях, таких как назначение и информационное содержимое сайта (например, сайт новостей, сайт электронной коммерции и т.д.). Тестированием могут также охватываться популярные на местном уровне сайты; т.е. если тестирование выполняется из Италии, могут подвергаться тестированию несколько популярных в Италии сайтов, даже если они не входят в перечень глобально популярных сайтов.

15.4. Комплексное измерение показателей эффективности Web

После рассмотрения вопроса совместимости со спецификацией протокола, было бы естественно выяснить, действительно ли изменения в протоколе ведут к повышению эффективности Web. Эффективность Web определяется несколькими факторами, включая время ожидания, испытываемое пользователем при загрузке Web-ресурсов, нагрузку на сеть в виде количества пакетов, которое требуется для передачи ответов, и нагрузку на Web-сервер, связанную с обработкой запросов.

Простой способ проверить эффективность — сформировать группу запросов серверам, работающим под двумя различными версиями протокола, и сравнить их

производительность. Более общепринятый подход состоит в создании глобальной группы тестирования, состоящей из клиентов с различным местоположением. Группа может использоваться для генерирования синтетической рабочей нагрузки в виде запросов, направленных репрезентативной группе Web-сайтов. Группу репрезентативных Web-сайтов легко сформировать, принимая во внимание, что списки популярных сайтов регулярно публикуются, а также учитывая, что большая часть запросов направляется довольно небольшому числу сайтов. Однако идентификация группы репрезентативных клиентов — гораздо более сложная задача.

В этом разделе мы поговорим об измерении комплексных показателей ответов Web-серверов с точки зрения различных клиентов. Начнем мы с определения факторов, влияющих на комплексную эффективность Web, и сосредоточим внимание на проблемах, связанных с протоколом. Затем мы познакомимся с отчетом одного комплексного исследования [KW00], вкратце остановившись на некоторых деталях и результатах этого исследования. Объединенное исследование, которое учитывало бы все факторы одновременно, провести довольно затруднительно. Необходимо считаться с большим количеством факторов, независимое управление которыми не всегда возможно. Однако стоит рассмотреть *влияние* различных факторов друг на друга, чем мы и займемся в заключительном разделе. Любое исследование, поставившее целью определить комплексную производительность, должно четко выделить исследуемые параметры и предположить, какие факторы можно игнорировать.

15.4.1. Определение факторов, влияющих на комплексную эффективность

На эффективность Web-транзакций оказывают влияние несколько факторов. Среди них пропускная способность соединения с конечным пользователем, наличие прокси-серверов на пути между клиентом и исходным сервером, потери пакетов в сети, а также нагрузка на исходный сервер. Объем доступных данных о местах происхождения запросов или характеристиках соединений клиентов с Internet достаточно ограничен.

В таблице 15.1 представлены некоторые компоненты, влияющие на эффективность Web. Таксономия не отражает фактических задержек, вносимых каждым из факторов. На это есть две причины: отсутствие убедительных масштабных исследований и изменчивость, присущая любому большому исследованию комплексной эффективности. Хотя проведение всестороннего исследования, которое учитывало бы все факторы, — непростая задача, можно начать с рассмотрения некоторых ключевых факторов, ограничив воздействие других. Кроме того, таблица не является исчерпывающей — в ней приведены не все факторы, которые могут оказывать влияние на суммарную задержку. В таблице сведены *наиболее характерные* источники задержки.

В первом столбце таблицы 15.1 представлены различные этапы Web-транзакции, начиная со щелчка мышью пользователем на гиперссылке в браузере до окончательного воспроизведения ответа браузером. Во втором столбце приведены примеры потенциальной изменчивости, которые могут повлиять на этап, приведенный в первом столбце (избежать его или осложнить его выполнение). Примером вариабельности является пропуск этапа преобразования доменного имени с помощью DNS, если IP-адрес сервера содержится в кэше DNS. Другой пример — дополнительные запросы на встроенные изображения после синтаксического анализа HTML-документа.

Таблица 15.1. Компоненты задержки

Этап Web-транзакции	Факторы, влияющие на задержку
Щелчок мышью в браузере	–
Просмотр кэша браузера	Обращение/отсутствие обращения к кэшу DNS, быстродействие клиентского компьютера
Построение HTTP-запроса	Версия протокола
DNS-запрос на IP-адрес прокси-сервера	Обращение к кэшу DNS, обращение к корневому серверу DNS
Перехватывающий прокси-сервер	Быстродействие аппаратных средств/группы прокси-серверов
TCP-соединение с прокси-сервером	Характеристики соединения с провайдером / долговременное соединение
	Нагрузка на прокси-сервер/кэш
Иерархия кэшей	UDP/ICP
Доменное имя расположенного выше по потоку сервера	Обращение к DNS
Обращение к следующему прокси-серверу в цепочке	Версии протоколов, задержка в сети
TCP-соединение с исходным сервером	Наличие сервера-заместителя
Переадресация запроса сервером	Соединение с новым сервером
Синтаксический анализ HTTP-запроса сервером	Статический/динамический ресурс, задержка при выполнении CGI-сценария
Нагрузка на сервер	Задержка на стороне сервера
Создание ответа сервером	Размер выходного буфера сокета
Синтаксический анализ и отображение ресурса браузером	Выдача дополнительных запросов

Рассмотрим типовую Web-транзакцию, которая начинается с выбора пользователем URL и заканчивается отображением ответа браузером. В главе 2 (раздел 2.3.1) мы обсуждали классический пример. Мы игнорировали потенциальное присутствие ряда других компонентов, таких как перехватывающие прокси-серверы, а также другие возможные действия, такие как переадресация запроса сервером. Мы также игнорировали особенности соединения с провайдером Internet и/или с Internet, а также потенциальные задержки, вызванные заторами в сети. Таблица 15.1 отражает попытку дать полную картину различных этапов Web-транзакции. Щелчок мышью пользователем на гиперссылке приводит к обращению к кэшу браузера, чтобы проверить, доступен ли ресурс локально. Если нет, щелчок обусловливает HTTP-запрос.

Предположим, что пользователь соединился с провайдером для доступа к Internet. Соединение может быть низкоскоростным коммутируемым соединением, быстродействующим соединением с помощью кабельного модема или DSL. Сам провайдер может быть подключен к Internet через каналы, совместно используемые с другими провайдерами. Провайдеры могут использовать один или несколько прокси-серверов для различных целей, включая пересылку сообщений, совместное

использование ресурсов и кэширование. Браузеры пользователей могут быть явно настроены на использование прокси-сервера провайдера.

Обращение к прокси-серверу на пути между клиентом и исходным сервером может быть настроено непосредственно на клиенте путем указания IP-адреса или доменного имени. Если настройка прокси-сервера произведена указанием доменного имени (например, **proxy01.aol.com**), посылается DNS-запрос для получения IP-адреса прокси-сервера. Как говорилось в главе 5 (раздел 5.3.3), доменное имя прокси-сервера передается локальному DNS-серверу, который может иметь кэшированный ответ, в этом случае задержка минимальна. Если же нет, то придется обращаться к другим DNS-серверами, может пройти несколько секунд, прежде чем IP-адрес будет получен. В большинстве случаев такой DNS-запрос посылается только один раз, поскольку информация будет кэшироваться либо браузером, либо локальным DNS-сервером.

Даже если прокси-сервер явно не указан, запрос может быть перехвачен перехватывающим прокси-сервером (см. главу 11, раздел 11.10.2). Заметим, что браузер не осведомлен о таких перехватах. Если ответ пришел от одного из кэширующих прокси-серверов, общее время ожидания может быть небольшим. Если же ответ не был найден на кэширующем прокси-сервере на пути к исходному серверу, запрос должен быть отправлен выше по потоку.

Предположим, что осуществляется обращение к явно указанному в браузере прокси-серверу, который является частью иерархии кэшей. Прокси-сервер может обратиться к иерархии кэшей, если он не способен найти ресурс в своем кэше. Запрос к кэшу обычно представляет собой UDP-сообщение, посылаемое с использованием протокола Internet Cache Protocol (ICP) другим кэшам на том же уровне иерархии. Успешный ответ от одного из кэшей на данном уровне иерархии сохранит малое значение общей задержки. Однако если ответ от кэша на данном уровне иерархии не получен, запрос должен быть отправлен серверам, находящимся выше по потоку в направлении исходного сервера. Типичная задержка при проверке в иерархии кэшей Squid составляет несколько миллисекунд в предположении, что UDP-сообщения не были потеряны.

Как в случае явно указанного прокси-сервера, так и в случае наличия перехватывающего прокси-сервера, нагрузка на прокси-серверы будет также иметь значение. Если прокси-серверы заняты обработкой других запросов, может возникнуть дополнительная задержка.

До сих пор мы обсуждали роль браузеров и прокси-серверов. Теперь мы рассмотрим роль, которую играет протокол HTTP применительно к соединениям и кэшированию. Если клиент уже установил долговременное соединение с прокси-сервером, новый запрос не требует преобразования доменного имени прокси-сервера или установления TCP-соединения. Предположим, что запрос не был удовлетворен из кэша браузера, ближайшим к пользователю прокси-сервером или каким-либо еще прокси-сервером из группы прокси-серверов провайдера. В этом случае запрос должен быть отправлен расположенному выше по потоку серверу. При этом начинают играть роль характеристики соединения между сетью провайдера и Internet. Дополнительным фактором являются задержки, обусловленные заторами в сети и потерями пакетов. Обычно задержки в сети не представляют проблемы при взаимодействии пользователя с ближайшим к нему прокси-сервером. Если запрос пересылается другому прокси-серверу, расположенному дальше от пользователя, начинают сказываться факторы, связанные с обращением к DNS и нагрузкой на расположенный выше по потоку прокси-сервер. Общая задержка между границей сети провайдера и конечным прокси-сервером (который может пред-

ставлять собой сервер-заместитель перед исходным сервером) равна сумме задержек между компонентами. Эта часть измерений (от границы сети провайдера до конечного прокси-сервера перед исходным сервером) склонна к большой изменчивости результатов, а исследования в этой области широко не проводились.

При поступлении запроса комплексу Web-серверов, где располагается исходный сервер, первым запрос может получить сервер-заместитель перед исходным сервером. Сервер-заместитель может поддерживать свой собственный кэш часто запрашиваемых ресурсов. Если ресурса в кэше нет, заместитель переадресует запрос соответствующему серверу согласно принятой стратегии выравнивания нагрузки. Заместитель или исходный сервер в зависимости от типа запроса могут решить переадресовать запрос на другой сайт. Такая переадресация может быть обусловлена выравниванием нагрузки, наличием более подходящего зеркального сайта или тем, что ресурс был перемещен.

Если предположить, что ресурс не был перемещен и является доступным, то может быть сформирован ответ. Нагрузка на исходный сервер является важным фактором, способствующим возникновению задержки при генерировании ответа. Независимо от нагрузки на исходный сервер, обработка запроса может потребовать значительного времени в зависимости от действий, которые должны быть выполнены. Например, для формирования ответа может оказаться необходимым обращение к серверу баз данных, и сервер должен поддерживать соединение до завершения запроса к базе данных. Ответ сервера может ожидать очереди на отправку и, в конце концов, достигает браузера после прохождения через различные промежуточные звенья. Только после этого браузер может начать синтаксический анализа ответа. Синтаксический анализ и воспроизведение ответа — еще один фактор, влияющий на общее время ожидания. Воспроизведение ответа может потребовать выполнения сценариев JavaScript и взаимодействия с локальными подключаемыми модулями перед окончательным отображением ответа пользователю. Если в ответе имеются встроенные ресурсы, браузер должен также осуществить их загрузку. Эти встроенные ресурсы могут размещаться на сервере, отличном от исходного сервера. Если мы будем считать исходный сервер базовым сервером для документа-контейнера, доставку встроенных изображений или другого содержания, связанного с документом-контейнером, например, рекламы, могут осуществлять альтернативные серверы.

Рост популярности исходных серверов, распространяющих содержание, размещенное на других компьютерах, должен приниматься во внимание любым исследованием комплексной эффективности. Причинами применения такой альтернативной стратегии доставки является снижение нагрузки на исходный сервер и потенциально более эффективная доставка содержания альтернативными серверами. Одновременно со снижением нагрузки, исходный сервер сохраняет соединения и обслуживает большее число пользователей. С точки зрения пользователя, изначальный запрос был сделан к исходному серверу, а все содержимое было доставлено с исходного сервера. Сети распространения содержания (см. главу 11, раздел 11.13) играют все более важную роль в Web.

Потеря пакетов может иметь различную степень влияния на общую задержку. Если был потерян DNS-запрос, повторная передача такого запроса может значительно увеличить время ожидания. Точно так же, потеря пакета SYN в процессе установки TCP-соединения может внести существенную дополнительную задержку, о чем говорилось в главе 8 (раздел 8.1.1). Другие виды потерь не столь важны.

15.4.2. Отчет по исследованию комплексной эффективности

Исследование [KW00], поставившее своей целью изучить факторы, влияющие на комплексную эффективность Web, проводилось с конца 1999 года в течение всего 2000 года. Инфрастуктура и методология этого исследования основывались на предыдущем исследовании совместимости, которое описывалось в разделе 15.3.4. Группа Web-сайтов была увеличена, чтобы включить популярные сайты, работающие под HTTP/1.0.

В этом разделе мы вкратце расскажем об исследовании, начав с его методологии, деталях проведенных экспериментов и результатах исследования. Хотя данные, приведенные в [KW00], могли измениться, главная наша цель при знакомстве с этим исследованием — сделать акцент на этапах проведения экспериментов. Помимо этого сосредоточим внимание на обобщенных аспектах исследования и на задачах последующих, более широких исследований.

МЕТОДОЛОГИЯ КОМПЛЕКСНОГО ИССЛЕДОВАНИЯ

Комплексное исследование представляло собой активное исследование по измерению характеристик — запросы генерировались и отправлялись на многие сайты. Группы клиентов находились по всему миру и отбирались главным образом по принципу доступности для авторов исследования, а не по географическому расположению и репрезентативности клиентов. В идеале множество клиентов следует выбирать таким образом, чтобы они представляли различные группы клиентов в Web, например, характерные свойства браузера (версия, функциональные возможности, различные настройки кэша), характеристики соединения с Internet и наличие прокси-серверов. Для действительно глобального тестирования важными факторами также являются учет различных часовых поясов и деятельности в сети других клиентов, находящихся поблизости.

Исследование комплексной эффективности игнорирует роль, которую могут играть прокси-серверы. Вместо этого генерировался трафик с использованием *httperf* [MJ98]. Назначение *httperf*, как уже говорилось в разделе 15.3.4, состоит генерации HTTP-запросов, установлении соединений, передаче запросов и получении информации о различных стадиях обработки запросов. Инструментальное средство имеет несколько изъянов, не позволяющих ему действовать в качестве полноценного клиента в реальном исследовании комплексной эффективности. Средство передает запрос клиента непосредственно исходному серверу и не предусматривает наличие одного или нескольких прокси-серверов на пути. Хотя ряд свойств *httperf*, таких как способность поддерживать долговременные соединения и предоставлять информацию о времени на различных этапах обработки запросов, делает его привлекательным клиентским инструментальным средством, оно не является клиентом, совместимым HTTP/1.1. Например, оно не выполняет функций браузера по воспроизведению ответа и поэтому позволяет получать лишь нижнюю оценку величины задержки. Репрезентативным клиентом может быть либо полностью совместимый с HTTP/1.1 клиент, который также способен предоставлять информацию о времени выполнения этапов запросов, либо совместимый браузер, который подвергся модификации с целью регистрации необходимой статистической информации.

После того, как было выбрано местоположение клиентов и сами клиенты, следующим шагом является выбор группы серверов. Для локальных Web-серверов или серверов в составе высшего учебного заведения, как правило, проводят небольшие исследования. Более масштабные исследования пытаются собрать данные для множества серверов, выбранных на основе популярности сайтов. По-настоящему

репрезентативное исследование должно принимать во внимание множество факторов, относящихся к Web-сайтам.

- **Популярность.** Общая популярность сайта является мерой, насколько в исследовании отражены привычки пользователей. Исследование не должно сосредотачивать внимание на неизвестных Web-сайтах, которые редко посещаются. Оно должно уделять внимание наиболее популярным сайтам, которые обслуживают значительную долю трафика запросов.
- **Репрезентативность.** Набор сайтов, включаемых в исследование, должен содержать новостные сайты, порталы, ежедневно получающие десятки миллионов запросов от сотен тысяч различных клиентов. Сайты электронной коммерции должны включать разнообразные формы, в ходе одного Web-сеанса требуется последовательно отображать несколько страниц, при этом велика вероятность использования cookies.
- **Версия и параметры настройки.** Версии протоколов сервера могут быть различными (HTTP/1.0 или HTTP/1.1). Для конкретной версии протокола имеется множество реализаций, включая Apache или Netscape. Для одной программной реализации может иметься множество различных способов настройки, например, подключения или отключения протокола Secure Socket Layer (SSL).
- **Конфигурация сайта.** Сайты конфигурируются различными способами: несколько сайтов на одном компьютере, один сайт, размещенный на нескольких компьютерах, зеркальные сайты, множество серверов позади сервера-заместителя и т.д.
- **Архитектура сервера.** Следует принимать во внимание различные архитектуры серверов: с управлением по событиям, с одним серверным процессом, с множеством программных потоков, с множеством процессов и т.д. (см. главу 4, раздел 4.4).
- **Содержание сервера.** Имеются значительные вариации содержания Web-сайтов — некоторые сайты содержат в основном изображения, другие —текстовые ресурсы, а на третьих преобладают динамически генерируемые ресурсы. К другим атрибутам ресурсов относятся возможность кэширования и частота изменений.
- **Роль сетей распространения содержания.** На некоторых сайтах значительная часть ресурсов доставляется через сети распространения содержания. Включать в рассмотрение такие сайты важно, чтобы учесть роль сетей распространения содержания.

Перечисленные выше факторы не являются исчерпывающими. Однако этот перечень следует принимать во внимание при выполнении любого комплексного исследования эффективности. Разнообразие и сложность факторов делают задачу проведения действительно репрезентативного исследования весьма трудной.

После выбора клиентов и серверов, а также идентификации клиентского программного обеспечения, следующим шагом является разработка набора тестов. Типичными целями комплексного исследования являются: измерение времени ожидания, испытываемого пользователем, объема передаваемых данных и задержки в пересчете на байт, учет вариантов соединений (например, одно долговременное соединение для загрузки нескольких ресурсов), оценка влияния функциональных особенностей протокола, времени дня, роли сетей распространения содержания и т.д. Исследование должно считаться со всеми этими факторами, особое внимание следует уделять особенностям протокола, времени суток, в которое проводился

сбор данных. Исследование состоит из двух частей: передачи группы запросов от множества клиентов множеству серверных сайтов и более подробного исследования с учетом различного времени суток.

КОМПЛЕКСНОЕ ИССЛЕДОВАНИЕ. ДЕТАЛИ ПРОВЕДЕНИЯ ЭКСПЕРИМЕНТОВ

В этом разделе мы познакомимся с деталями экспериментов, выполненных для исследования комплексной эффективности Web. Для отправки разнообразных запросов было выбрано одиннадцать клиентов в различных странах. Группа серверных сайтов была выбрана на основе двух приведенных выше критериев: популярности и версии протокола. Основное внимание при проведении исследования уделялось выбору репрезентативного «среза» сайтов в отношении популярности и поддерживаемой версии протокола. Набор серверов и клиентов, используемый в комплексном исследовании, был взят из предыдущего исследования PRO-COW [KA01], описанного ранее в разделе 15.3.1. На деле многие решения при проведении комплексного исследования были обусловлены решениями, принятыми в ходе исследования PRO-COW. Это дало возможность повторно использовать программное обеспечение, построить разумную инфраструктуру и воспользоваться готовой методологией. Программа *httperf*, используемая в PRO-COW, была модифицирована для получения дополнительной информации. Набор тестируемых сайтов был расширен за счет включения сайтов HTTP/1.0, не рассматривавшихся в исследовании PRO-COW (целью которого было изучение совместимости с HTTP/1.1). Всего было протестировано 711 серверных сайтов. Однако использование инфраструктуры и методологии PRO-COW имело и недостатки. Выбор сайтов для тестирования мог быть и иным, если протокол и популярность являются не единственными факторами, которые следует учитывать. Например, могли быть включены другие Web-реализации серверов, отсутствующие в выборке популярных сайтов.

Комплексное исследование эффективности основное внимание уделило четырем факторам:

- **Возможности протокола.** Исследование изучало различия между HTTP/1.0 и HTTP/1.1 с точки зрения сохранения соединения между отдельными парами запрос-ответ. В исследовании не рассматривается подмножество серверов HTTP/1.0, поддерживающих долговременные соединения с помощью заголовка **Keep-Alive**.
- **Кэширование.** Исследовались вариации эффективности кэширования, обусловленные использованием различных возможностей протокола.
- **Запросы на диапазоны.** Были рассмотрены различия времени ожидания при получении части ресурса вместо всего ресурса целиком.
- **Распространение содержания.** Были исследованы различия времени получения ответа при выборке встроенных изображений в случае, если не все ресурсы в контейнерном документе возвращались с основного сервера.

В более широком исследовании комплексной эффективности могут учитываться и другие факторы, такие как задержки, связанные с DNS-запросами; запросы, переадресованные другим серверам на уровне HTTP; наличие серверов-заместителей и разбиение на фрагменты для динамически генерируемых ресурсов.

Выбор URI был ограничен домашней страницей каждого из сайтов, участвующих в тестировании. Опять-таки этот выбор нельзя назвать идеальным, поскольку домашняя страница может не отражать популярности всего сайта. Выборка ресурса, отстоящего от домашней страницы на несколько «щелчков мышью», — вполне обыч-

ное дело. Другими словами, хотя сайт **www.vpopular.com** может быть очень популярен, эта популярность достигается за счет ресурса **http://www.vpopular.com/top-ten.html**. Время загрузки домашней страницы **http://www.vpopular.com/** может отличаться от времени, необходимого для загрузки ресурса **http://www.vpopular.com/topten.html**.

В исследовании использовалось средство *httperf* для получения базового URI и извлечения встроенных ресурсов. Для серверов, имеющих встроенные ресурсы, один и тот же набор документов запрашивался с применением четырех различных механизмов соединений.

1. **Последовательный.** Первый метод состоит в последовательном извлечении контейнерного документа и встроенных в него ресурсов с использованием протокола HTTP/1.0 через четыре отдельных соединения.
2. **Пакетный (burst).** Модификация первого метода, при котором до четырех соединений выполняются параллельно.
3. **Долговременный.** Третий метод состоит в использовании долговременного соединения *без* конвейеризации запросов.
4. **Долговременный с конвейеризацией.** Четвертый метод состоит в использовании долговременного соединения HTTP/1.1 с конвейерной обработкой запросов в этом соединении.

Для оценки дисперсии результатов, обусловленной разным временем суток, была выбрана более узкая группа серверных сайтов. Исследование было повторено в различное время в течение суток (четыре раза с интервалом в шесть часов), чтобы определить дисперсию.

В части исследования, посвященной кэшированию, за основу был взят кэш на стороне клиента в предположении, что отсутствие изменения размера ресурса между последовательными ответами указывает, что ресурс не был модифицирован. Использование факта изменения размера представляет собой простой эвристический подход — могут учитываться также заголовки ответов, такие как **ETag** и **Last-Modified**. При более широком тестировании более реалистичным будет использовать для индикации изменения ресурса контрольную сумму. Тем самым охватываются ситуации, когда ресурс изменяется без изменения длины содержания. Вместе с исследованием эффективности кэширования были также исследованы затраты на проверку актуальности и возврат кода ответа **304 Not Modified**.

РЕЗУЛЬТАТЫ КОМПЛЕКСНОГО ИССЛЕДОВАНИЯ

Исследование комплексной эффективности проводилось из 11 различных мест, а тестированию подверглось 711 серверных сайтов. Ниже мы кратко опишем результаты исследования.

Не все серверы успешно отвечали на запросы, а ответившие не всегда возвращали все встроенные ресурсы. Некоторые ресурсы не были найдены (ответ содержал код **404 Not Found**) или запрос на них переадресовывался (с кодом ответа **302 Found**). Один из неожиданных результатов состоял в том, что при запросе группы ресурсов через долговременное соединение с использованием конвейеризации, возвращались не все ресурсы. Реализации серверов используют различные эвристические процедуры для принятия решения, когда закрывать долговременные соединения. В главе 7 (раздел 7.5.5) мы рассмотрели различные причины, по которым сервер закрывает существующие долговременные соединение. К ним относятся: длительный период бездействия (если запросы направляются по существующему долговременному соединению), перегрузка сервера, превышение допустимого чис-

ла запросов в соединении, величина таймаута между запросами и код ответа, требующий закрытия соединения.

В исследовании все запросы в долговременном соединении посылались один за другим, после чего соединение закрывалось. Закрытие долговременных соединений осуществлялось в зависимости от числа запросов, обрабатываемых в одном соединении. В исследовании соединение называется *совершенным* долговременным соединением, если все запрошенные объекты были получены через одно соединение. Серверы способны передавать в долговременном соединении только определенное число запрашиваемых ресурсов. Исследование показало, что две трети серверов проявляли определенную степень совершенства при поддержании долговременных соединений, лишь около четверти тестируемых серверов проявили полное совершенство в поддержании долговременных соединений. Четверть серверов не сообщали о завершении долговременных соединений; т.е. они явно не отправляли заголовок ответа **Connection: close**, но закрывали соединение. Эффективность, выраженная через общее время ожидания на стороне пользователя, представляется максимальной, если серверы проявляли совершенство при поддержании долговременных соединений. Частично ухудшение эффективности для серверов, не проявлявших совершенства при поддержании долговременных соединений, определяется затратами, связанными с установлением нового соединения. С другой стороны, использование пакетных соединений HTTP/1.0 (несколько параллельных соединений) дает лучший результат, чем несовершенное поддержание долговременных соединений серверами. Главная причина этого состоит в том, что затраты на повторное соединение сводят на нет повышение эффективности за счет долговременных соединений.

Относительная эффективность протоколов незначительно менялась в зависимости от времени суток. Частично причина этого заключается в том, что группа серверов выбиралась на основе популярности, а не на основе географического местоположения. Практически все популярные серверы находились в Северной Америке. Единственное видимое различие было связано с периодами наибольшей активности в Северной Америке. Клиент, обладающий качественным соединением с Internet, имел лучшие показатели общей производительности за все периоды времени, а различие показателей за различные периоды времени не были значимыми. В исследовании рассматривалось лишь одно подмножество серверных сайтов для ограниченного подмножества местоположений клиентов. Если бы в множестве тестируемых сайтов были представлены сайты из других регионов, можно было бы оценить, повсеместно ли сказывается эффект периодов максимальной активности. При более широком тестировании следует включать в рассмотрение серверы с учетом их популярности по странам и регионам, а также тестировать эти сайты с клиентов, находящихся в различных странах. В расширенном исследовании также должно быть задействовано большее число местоположений клиентов.

Исследование кэширования не было по-настоящему репрезентативным, поскольку большинство ресурсов являлись статическими, а длительность тестирования составила всего одну неделю. В исследовании время проверки нахождения ресурса в кэше принималось равным нулю. При этом было выявлено, что повторное использование кэшированного ответа может значительно уменьшить время ожидания ответа. Кэширование давало максимальную выгоду, когда ресурсы запрашивались последовательно через соединение HTTP/1.0, которое имело наибольшее время ожидания среди четырех способов соединения. Соответственно в наибольшем выигрыше оказался тот способ соединения, который сопровождался наибольшими затратами на загрузку объектов. Анализ влияния затрат на проверку актуальности (с выдачей за-

просов **GET If-Modified-Since**) показал, что для последовательных соединений HTTP/1.0 общее время на извлечение ресурса было гораздо выше. Результаты проверки актуальности были подвергнуты перекрестному тестированию путем включения заголовка **If-None-Match: *** (см. главу 7, раздел 7.3.3) в запросы **GET** для получения ответа **304 Not Modified**. Была отмечена более высокая относительная эффективность последовательных запросов HTTP/1.1 в сравнении с другими опциями, предусматривающими проверку актуальности кэша. В целом исследование кэширования показало, что при извлечении меньшего количества объектов наблюдается меньше различий между различными возможностями протокола.

Следует провести более широкое исследование, охватывающее больший период времени и более разнообразный состав статических и динамических ресурсов. Следует также проанализировать снижение затрат благодаря выдаче запросов на повторную проверку актуальности в составе одного долговременного соединения, аналогично описанной ранее технологии совмещения.

Касательно содержания, размещенного на нескольких серверах, в исследовании было уделено внимание рассмотрению влияния альтернативных серверов, каждый из которых отвечает за определенную часть содержания. В исследовании исходный сервер назван базовым сервером содержания, а альтернативные серверы поделены на следующие четыре категории:

- **Вспомогательные серверы.** Серверы, обычно размещаемые вместе с базовым сервером. Если альтернативный сервер и базовый сервер имеют одинаковый суффикс (например, серверы **images.cnn.com** и **www.cnn.com** имеют один и тот же суффикс, **cnn.com**), считается, что серверы расположены в одном месте.
- **Рекламные серверы.** Серверы, используемые для распространения рекламы (их имена обычно начинаются со строки **ad**).
- **Серверы сетей распространения содержания.** Серверы, используемые для распространения определенного содержания (например, изображений) с зеркальных сайтов, расположенных ближе к пользователю. Однако в исследовании число серверов, относящихся к серверам распространения содержания, не очень велико.
- **Смешанные.** Другие серверы, не подпадающие ни под одну из трех указанных выше категорий.

Хотя разбиение по категориям может оказаться полезным, точно отнести каждый сервер к определенной категории не всегда оказывается возможным. Например, рекламный сервер может распознаваться по некоторым хорошо известным именам или префиксу **ad** в имени сервера. Аналогично, серверы распространения содержания могут распознаваться по известным префиксам, таким как **akamai**, в URI встроенного ресурса. Однако не все URI, содержащие префикс **ad**, могут соответствовать рекламным сайтам, а ряд ресурсов, обслуживаемых сайтами распространения содержания, не содержат имени сети распространения содержания.

Исследование показало, что среди популярных серверных сайтов 13,5% хранят некоторую часть своего содержания вне базовых серверов. Если базовыми серверами обрабатывалась лишь небольшая доля встроенных объектов, общее количество HTTP-запросов, которые базовый сервер должен обработать, может быть уменьшено. Прием HTTP-запроса, его синтаксический анализ и возврат ответа составляют значительную часть общей работы, которую приходится выполнять исходным серверам. Таким образом, уменьшение количества объектов, обрабатываемых базовым сервером, поможет снизить нагрузку на исходный сервер.

Сети распространения содержания (рассмотренные в главе 11, раздел 11.13) работают по принципу размещения зеркальных серверов ближе к пользователям, что не обязательно означает географическую близость. Определению ближайших серверов был посвящен ряд исследований [JCDK00]. Данное комплексное исследование также показало, что скорость передачи данных при доставке содержания была гораздо выше, когда ресурсы загружались с серверов распространения содержания, нежели с базовых серверов, хотя имеет значение и местоположение клиента. Наиболее известной сетью распространения содержания, с которой осуществлялось взаимодействие в ходе комплексного исследования, была сеть Akamai [Aka]. Пользователи Akamai преобразуют URI встроенных ресурсов в URI, идентифицирующий сайт Akamai, включая в него строку **akamai**. Тем самым обеспечивается доставка ресурсов с зеркальных серверов Akamai, а не базового сервера. Например, **http://www.cnn.com/foo.gif** должен быть преобразован в **http://a138g.akamaitech.net/0923/sdh2/www.cnn.com/foo.gif**. В исследовании встроенные ресурсы загружались и с зеркального сервера сети распространения ресурсов, и с базового сервера с целью сравнить скорость загрузки данных.

15.4.3. Резюме комплексного исследования эффективности

В целом комплексное исследование подтвердило, что долговременные соединения с конвейеризацией повышают эффективность. Однако серверы, обрабатывающие небольшое количество объектов, могут не почувствовать какого-либо значительного повышения эффективности. Серверы, проявившие совершенство при обслуживании долговременных соединений, когда все конвейеризированные запросы получают ответы в одном и том же TCP-соединении, показали наилучшую эффективность. Распространение содержания также продемонстрировало значительное увеличение эффективности, хотя не многие фигурировавшие в исследовании сайты использовали удаленные серверы распространения содержания. За последнее время в этой сфере наметился заметный рост.

Однако исследование не содержало подробного сравнения между получением ресурсов с базовых серверов и с альтернативных серверов. Для этого потребовалось бы сравнить затраты на установку соединений с альтернативными серверами и оценить стратегии серверов в отношении закрытия долговременных соединений (с конвейеризацией и без нее). Браузер, использующий несколько параллельных соединений с базовым или вспомогательными серверами, может достичь общего снижения времени ожидания ценой более высокой загрузки сети. Исследование заложило основу для более широкого комплексного исследования, которое учитывало бы дополнительные факторы.

Рассмотренное нами комплексное исследование является лишь начальным этапом более масштабного исследования, учитывающего все параметры, приведенные в таблице 15.1. Расширенное комплексное исследование должно принимать во внимание различные виды клиентов (например, характеристики их соединений с Internet и совместимость клиентов с протоколом) и, помимо всего прочего, выбирать серверные сайты с учетом различных факторов, рассмотренных в главе 4, раздел 4.1. Некоторые этапы, приведенные в таблице 15.1, могут быть разбиты на составные части и проанализированы более детально. Так оценка нагрузки на прокси-сервер может потребовать рассмотрения влияния разрывов долговременных соединений, установленных между клиентом и прокси-сервером, а также между прокси-сервером и расположенным выше по потоку сервером. Последнее соединение может быть долговременным соединением, а может не быть им. Применительно к содержанию альтернативных серверов, должны быть исследованы затраты на установку соединений с ними.

15.5. Другие расширения HTTP

Вопросы перехода от HTTP/1.0 к HTTP/1.1 обсуждались в течение пяти лет. В это же время предпринимались попытки расширения HTTP в других направлениях. Поскольку расширение протокола осуществляется на основе консенсуса, ряд предложений по внесению дополнений в HTTP в ходе обсуждения их рабочей группой HTTP Working Group был отвергнут. Некоторые предложения запоздали, другие не были одобрены рабочей группой, а некоторые предложенные дополнения потребовали бы значительной корректировки HTTP. Однако несколько предложений были оформлены в виде рабочих проектов, а некоторые из предложений прошли несколько итераций, прежде чем были сняты с рассмотрения и не включены в окончательную версию HTTP/1.1. Например, первоначально предполагалось, что предложение Transparent Content Negotiation [HM98], которое описывало расширяемый механизм для выбора наиболее подходящей версии ресурса, станет частью HTTP/1.1. Аналогично, предложение HTTP Extension Framework [NLL00], которое являлось попыткой скоординировать расширения HTTP, не стало частью предложения по стандарту HTTP/1.1. Попытка реализовать совместное обновление Web-ресурсов и решить проблему согласования версий привела к появлению отдельного проекта под названием WebDAV — Web Distributed Authoring and Versioning [GMF+99].

15.5.1. Transparent Content Negotiation

Идея Transparent Content Negotiation (TCN) [HM98] возникла в ходе обсуждения рабочей группой HTTP Working Group проблемы выбора варианта содержания в зависимости от требований клиента. Вопросы выбора варианта содержания подробно обсуждались в главе 7 (раздел 7.9). Клиенту предоставляется возможность выразить свои предпочтения при выборе варианта ресурса из числа доступных на исходном сервере. В качестве примера таких предпочтений можно привести выбор языка или набора символов.

В целом процесс выбора варианта содержания достаточно очевиден: клиент выражает предпочтение и, если предпочтение приемлемо для исходного сервера, ресурс возвращается в соответствующем формате. В процессе обсуждения перехода от HTTP/1.0 в HTTP/1.1 возможности выбора варианта содержания HTTP/1.0 были значительно расширены. Поддержка протоколом HTTP/1.0 выбора варианта содержания требовала от клиентов указания приемлемых языков, наборов символов и т.д., чтобы сервер мог выбрать наилучший возможный вариант ресурса. Предоставляя клиенту возможные значения (или диапазон значений), такой механизм не допускает расширения. Как говорилось ранее в главе 7, в HTTP/1.1 были введены две формы выбора варианта содержания: с управлением на стороне агента пользователя и с управлением на стороне сервера. TCN представляет собой комбинацию этих двух способов с целью использования кэшей, расположенных на пути между клиентом и исходным сервером. Дружественность по отношению к кэшам является важным добавлением в HTTP/1.1, реализованным через заголовки **Vary** и **If-None-Match**. TCN использует дополнительную поддержку кэширования в HTTP/1.1 для выбора варианта содержания с помощью агента пользователя. Такое решение дает пользователю возможность явного выбора из множества вариантов.

Основным заголовком, используемым в TCN, является заголовок ответа **Alternates**, который передает список вариантов, ассоциированных с ресурсом. Например, копия этой книги может быть доступна в различных форматах при обращении

к URI **http://www.research.att.com/~bala/books/kandr**. Одна версия книги может быть представлена в формате PostScript, другая — в виде большого контейнерного HTML-документа, а еще одна, переведенная на санскрит, доступна на Devanagari Script в виде документа в формате Portable Document Format (PDF). Поскольку ответ доступен в различных форматах, используется код ответа **300 Multiple Choices**. Если сделан запрос на книгу, исходный сервер может включить заголовок ответа **Alternates** в свой ответ, например, следующим образом:

```
HTTP/1.1 300 Multiple Choices
Date: Sun, 2 Jul 2000 17:45:43 GMT
Alternates: {"book.html" 1.0 {type text/html} {language en}},
  {"book_ss.pdf" 1.0 {type application/pdf} {language ss}},
  {"book.ps" 0.9 {type application/postscript} {language en}},
  {"book.doc" 0.5 {type application/word} {language en}}
Vary: negotiate, accept-language
ETag: "krbk2921103-2311-dtee"
Content-Type: text/html
Content-Length: 212

<h2>Multiple Choices:</h2>
<ul>
<li><a href=book.ps>Postscript, English version</a>
<li><a href=book.html>HTML, English version</a>
<li><a href=book_ss.pdf>PDF, Spoken Sanskrit version</a>
</ul>
```

Качество источника, заданное в заголовке **Alternates**, представляется значением качества, или *qvalue* (см. главу 7, раздел 7.9), которое указывает на качество альтернативного представления. В приведенном выше примере принято, что версия книги в формате Word имеет значение качества, меньшее 1 (0.5), т.е. уступает представлениям в форматах HTML, PostScript и PDF.

Заголовок **Vary** (см. главу 7, раздел 7.3.3) используется для указания, что выбор кэшированного ресурса должен осуществляться с учетом списка заголовков. Значение поля, следующее за *negotiate*, указывает, какой атрибут будет использоваться прокси-серверами для выбора варианта содержания. В примере таким значением является **accept-language**, которое указывает, что серверам следует использовать язык ресурса. Другими возможными атрибутами являются: тип ресурса, набор символов и набор характеристик, которые влияют на качество варианта, такие как цветовое разрешение, ширина или высота страницы.

Агент пользователя HTTP/1.1, который не поддерживает TCN, может отображать тело содержимого. Если приведенный выше ответ содержит заголовок **Location**, агент пользователя может переадресовать пользователя в указанное место. Агент пользователя HTTP/1.1, который поддерживает TCN, должен уметь автоматически извлекать и отображать приемлемый вариант после того, как такой вариант выбран. Если ни один из указанных вариантов неприемлем, агент пользователя может отобразить сообщение об ошибке.

Хотя несколько серверных реализаций [Hol] уже имеется, на стороне клиента таких реализаций пока нет, что не позволяет говорить о распространении TCN в Web. В IETF была сформирована рабочая группа Content Negotiation Working Group для исследования проблем выбора вариантов содержания как внутри протокола HTTP, так и вне его.

15.5.2. WebDAV — Web Distributed Authoring and Versioning

Первоначально поток Web-содержания был направлен главным образом от исходных серверов клиентам. Web может выступать в качестве средства коллективного создания и обновления содержания. WebDAV [Weba] представляет собой расширение протокола HTTP/1/1 для поддержки распределенного редактирования ресурсов в Web и управления версиями. Протокол WebDAV был описан в документе RFC 2518 [GWF+99]. В отличие от двух предыдущих расширений HTTP, WebDAV достаточно популярен и входит в состав программных продуктов, таких как Web-серверы Microsoft Internet Information Services (IIS) 5.0, Office 2000 Web Folders и Exchange 2000 Web Storage System. WebDAV с успехом использует расширяемый язык разметки Extensible Markup Language (XML) в качестве ключевого расширения HTTP/1.1. Подробное описание функций WebDAV не входит в задачу этой книги, мы представим здесь лишь краткий обзор на основе часто задаваемых вопросов (Frequently Asked Questions) по WebDAV [Webb].

Целью WebDAV Working Group целью было:

> Определить расширения HTTP, которые необходимы для содействия широкому применению средств редактирования ресурсов в Web.

В этом плане DAV призван сделать Web средством коллективного взаимодействия, а не только как однонаправленным средством, работающим только с одним пользователем за раз. Помимо простого средства редактирования Web-страниц, DAV можно рассматривать как сетевую файловую систему для сред с большими временами ожидания, способную работать с несколькими файлами одновременно и обладающую высокой производительностью. DAV представляет собой протокол для манипулирования содержанием документов через Web.

К целям DAV относятся следующие:

- Служить в качестве основного протокола, поддерживающего широкий спектр приложений для коллективной работы.
- Поддерживать удаленную разработку программного обеспечения коллективами разработчиков.
- Содействовать превращению HTTP в стандартное средство доступа для широкого класса хранилищ информации путем дополнения возможностей HTTP новыми возможностями записи/чтения данных.

Для поддержки коллективного обновления содержания должен быть определен перечень лиц, которые имеют доступ к содержанию, время последней модификации содержания и другие аналогичные атрибуты. Содержание — это не единичный ресурс, а коллекция ресурсов, которые организованы некоторым осмысленным образом. Коллекции могут создаваться, удаляться или копироваться вместе. Могут вызвать интерес совокупные показатели множества ресурсов. Если несколько авторов не только просматривают документ, но и, возможно, редактируют его, необходимо иметь механизм для внесения исправлений и управления доступом, например, возможность блокировать ресурсы и объединять изменения. В соответствии с этим в WebDAV было введено несколько новых методов, заголовков запросов и ответов, а также форматов тела содержимого. Для представления множественных состояний ответов (поскольку методы WebDAV могут оперировать с несколькими ресурсами) и совершенствования классификации кодов состояния по категориям были добавлены новые коды ответов. Расширен также набор свойств ресурсов.

15.5.3. Инфраструктура расширений HTTP

В процессе эволюции протокола от HTTP/1.0 к HTTP/1.1 было предложено несколько различных расширений. В результате усилий по координации различных предложений был сформулирован унифицированный механизм расширений. Предложенные расширения относились к коллективной разработке (об этом говорилось в разделе 15.5.2, посвященном WebDAV), механизмам удаленного вызова процедур и публикации в Internet. Предложенное расширение протокола может потребовать внесения изменений в один или несколько Web-компонентов. Инфраструктура расширений HTTP Extension Framework предлагает простой способ для обмена информацией между авторами предлагаемых расширений и лицами, заинтересованными в использовании расширений. Разработчики расширения должны сделать доступной через глобальный уникальный URI всю информацию, связанную с расширением. Каждый, кто пользуется расширением, должен указать, что он использует расширение, включив в HTTP-сообщение заголовок с URI расширения.

Основной идеей при создании инфраструктуры расширений было включать унифицированный указатель на описание расширения, которое может быть избирательно использовано. Инфраструктура предоставляет собой способ определения подходящего синтаксиса и семантики, который является достаточно гибким, чтобы отделять обязательные правила расширений от необязательных. Определения расширений включают следующую документацию: спецификацию, которая определяет семантику расширения, и код, который реализует семантику конкретного расширения. Расширения могут создаваться двумя заинтересованными сторонами, отстоящими друг от друга на один «переход» HTTP, либо могут быть сквозными. Чтобы сторона могла указать, что она не поддерживает расширение, был предложен новый код ответа **510 Not Extended**.

Хотя инфраструктура расширений HTTP не стала частью стандарта HTTP/1.1 из-за отсутствия консенсуса, она нашла отражение в документе RFC 2774 [NLL00], относящегося к классу RFC, описывающему экспериментальные технологии.

15.6. Резюме

Протокол HTTP используется в течение десяти лет, на него приходится значительная часть трафика Internet. Неудивительно, что предпринимаются попытки усовершенствовать HTTP и базовый протокол транспортного уровня TCP. В этой главе мы рассмотрели несколько направлений исследований от мультиплексирования HTTP-передач на уровне TCP до расширений собственно HTTP для улучшения взаимодействия между Web-компонентами. При усовершенствование существующего и широко используемого протокола требуется уделять повышенное внимание обратной совместимости, что особенно характерно для добавления в HTTP/1.1 дельта-механизма. Дискуссия по внедрению дельта-механизма показала, что для анализа реального снижения времени ожидания на стороне пользователя и снижения загрузки сети без значительных затрат на стороне исходного сервера, необходимо проведение экспериментов. Исследование совместимости продемонстрировало необходимость изучения широко используемых Web-компонентов на предмет следования ими спецификации протокола. Практическая полезность протокола определяется корректностью реализаций спецификации протокола в популярных Web-компонентах. Большинство пользователей Web вряд ли смогут воспользоваться преимуществами протокола, если не соблюдаются требования совместимо-

сти. Если в популярной реализации спецификации протокола не соблюдаются требования уровня MUST, нельзя ожидать, что приложение, построенное поверх HTTP, будет вести себя, как должно. Комплексное исследование изучает различные факторы, влияющие на эффективность Web. Хотя данное исследование носило предварительный характер, оно выявило ряд причин, влияющих на задержки в Web. Мы также познакомились с некоторыми другими проектами, связанными с выбором варианта содержания и инфраструктурой расширения HTTP. Распределенная система редактирования и управления версиями WebDAV является весьма многообещающей при переходе к следующему этапу организации взаимодействий в Web — несколько пользователей могут осуществлять доступ к ресурсам и, возможно, их модификацию, вместо простой загрузки ресурсов с серверов.

Литература

[AAL+92] Bob Alberty, Farhad Anklesaria, Paul Linder, Mark McCahill and Daniel Torrey. The Internet Gopher protocol: A distributed document search and retrieval protocol, Spring 1991; Revised Spring 1992. **http://boombox.micro.umn.edu/pub/gopher/gopher_protocol/protocol.txt**.

[Abb99] Janet Abbate. *Inventing the Internet.* MIT Press, June 1999. ISBN 0262011727.

[ABCdO96] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing Reference Locality in the WWW. In *Proc. Parallel and Distributed Information Systems*, December 1996. **http://www.cs.bu.edu/techreports/1996-011-www-reference-locality.ps.Z.**

[AC98] J. Alameida and P. Cao. Measuring Proxy Performance with the Wisconsin Proxy Benchmark. *Computer Networks and ISDN Systems*, 30 (22-23): 2179-2192, November 1998. **http://www.cs.wisc.edu/~cao/papers/cao-wpb/index.html**.

[AD99] Mohit Aron and Peter Druschel. TCP Implementation Enhancements for Improving Web Server Performance. Technical Report 99-335, Department of Computer Science, Rice University, 1999. **http://www.cs.rice.edu/~aron/papers/rice-TR99-335.ps.gz.**

[Ade] Adero. **http://www.adero.com.**

[Adi94] C. Adie. A Status Report On Networked Information Retrieval: Tools and Groups, RFC 1689, IETF, August 1994. **http://rfc-editor.org/rfc/rfc1689.txt**.

[AFJ99] Martin Arlitt, Rich Frederich, and Tai Jin. Workload Characterisation of Web Proxy in Cable Modem Environment. *ACM Performance Evaluation Review*, 27(2):25-36, August 1999. Also aviable as HPL Technical Report HPL-1999-48. **http://www.hpl.hp.com/techreports/1999/HPL-1999-48.html**.

[AFP98] M. Allman, S.Floyd, and C. Partridge. Increasing TCP's Initial Window. RFC 2414, IETF, September, 1998. **http://www.rfc-editor.org/rfc/rfc2414.txt.**

[AJ00] M. Arlitt and T. Jin. Workload Characterization of the 1998 World Cup Web Site. *IEEE Network Magazine*, 14(3):30-37, May/June 2000. Also available as HPL Technical Report HPL-1999-35. **http://www.hpl.hp.com/techreports/1999/HPL-1999-35R1.html.**

[Aka] Akamai. **http://www.akamai.com.**

[AKMS95] K. Andrews, F. Kappe, H. Maurer, and K. Schmaranz. On second generation hypermedia systems. In *Proc. ED-MEDIA 95, World Conference on Educational Multimedia and Hypermedia*, June 1995.

[Ale] Alexa. **http://www.alexa.com**.

[Alt] Alta Vista. **http://www.altavista.com.**

[Ami99] Amit Gupta and Geoffrey Baehr. Ad insertion at proxies to improve cache hit rate. In *Proc. 4th International Web caching Workshop*, March/April 1999.
http://www.ircache.net/Cache/Workshop99/Papers/gupta-final.ps.gz.

[AML+93] F. Anklesaria, M. McCahill, P. Lindner, D. Johnson, D. Torrey, and B. Alberti. The Internet Gopher Protocol. RFC 1436, IETF, March 1993.
http://www.rfc-editor.org/rfc/rfc1436.txt.

[Apa] Apache Software Foundation. **http://www.apache.org.**

[APS99] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581, IETF, April 1999. **http://www.rfc-editor.org/rfc/rfc2581.txt.**

[AS98] Soam Acharya and Brian Smith. An Experiment to Charaterize Videos on the World Wide Web. In *Proc. Multimedia Communications and Networking*, January 1998.
http://www.cs.cornell.edu/zeno/Papers/webvideo/Paper.pdf.

[ASA+95] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Stephen Williams, and Edward A. Fox. Caching Proxies: Limitations and Potentials. In *Proc. World Wide Web Conference*, December 1995.
http://www.w3.org/pub/Conferences/WWW4/Papers/155/.

[ASP00] Soam Acharya, Brian Smith, and Peter Parnes. Characterizing User Access to Videos on the World Wide Web. In *Proc. Multimedia Communication and Networking*, January 2000.
http://www.cs.cornell.edu/home/soam/papers/drafts/videoaccess.pdf.

[AW97] Martin F. Arlitt and Carey L. Williamson. Internet Web Servers: Workload Characterization and Performance Implications. *IEEE/ACM Transactions on Networking*, 5(5): 631-645, October 1997.
http://www.cs.usask.ca/faculty/carey/papers/ton97.ps.

[BBBC99] Paul Barford, Azer Bestavros, Adam Bradley, and Mark Crovella. Changes in Web Client Access Patterns: Characteristics and Caching Implications. *WWW Journal*, 2(1/2):15-28, June 1999.
http://cs-www.bu.edu/faculty/crovella/paper-archive/traces98.ps.

[BC94] H. Braun and K. Claffy. Web Traffic Characterization: An Assessment of the Impact of Caching Documents from NCSA's Web Server. In *Proc. World Wide Web Conference*, October 1994.
ftp://oceana.nlanr.net/papers/2iwwwc.cache.ps.gz.

[BC95] Steven M. Bellovin and William R. Cheswick. *Firewalls and Internet Security*. Addison-Wesley, 1995, ISBN 0201633574.

[BC98] Paul Barford and Mark Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proc. ACM SIGMETRICS*, June 1998.
http://cs-www.bu.edu/faculty/crovella/paper-archive/sigm98-surge.ps.

[BC99] Paul Barford and Mark Crovella. Measuring Web Performance in the Wide Area. In *Proc. ACM SIGMETRICS Performance Evaluation Review*, August 1999.
http://www.cs.bu.edu/techreports/1999-004-wide-area-web-measurement.ps.Z.

[BCF+99] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web Caching and Zipf-Like Distributions: Evidence and Implications. *In Proc. IEEE INFOCOM*, March 1999.
http://www.ieee-infocom.org/1999/papers/01d_03.pdf.

[BD99] G. Banga and P. Druschel. Measuring the capacity of a Web server. *WWW Journal*, 2(1-2):69-83, June 1999.
http://**www.cs.rice.edu/~druschel/wwwjsi99.ps.gz**.

[BDH+94] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Scwartz. The Harvest Information Discovery and Access System. In *Proc. Second International World Wide Web Conference*, pages 763-771, October 1994.

[Bes95] Azer Bestavros. Using Speculation to Reduce Server Load and Service Time on the WWW. In *Proc. ACM International Conference on Information and Knowledge Management*, November 1995.
http://www.cs.bu.edu/faculty/best/res/papers/cikm95.ps.

[BL] Tim Berners-Lee. Web Architecture from 50,000 feet.
http://www.w3.org/DesignIssues/Architecture.html.

[BL90] Tim Berners-Lee. Information Management: A Proposal, May 1990.
http://www.w3.org/History/1989/proposal.html.

[BL92a] Tim Berners-Lee. Is there a paper which describes the www protocol, January 9 1992. WWW-talk mailing list.
http://lists.w3.org/Archives/Public/www-talk/1992JanFeb/0000.html.

[BL92b] Tim Berners-Lee. MIME, SGML, UDIs, HTML and W3, June 1992. WWW-talk mailing list.
http://lists.w3.org/Archives/Public/www-talk/1992MayJune/0038.html.

[BL92c] Tim Berners-Lee. WorldWideWeb news: New software includes Gopher, News, Telnet access, January 24 1992, WWW-talk mailing list.
http://lists.w3.org/Archives/Public/www-talk/1992JanFeb/0001.html.

[BL93a] Tim Berners-Lee. Hypertext Transfer Protocol (HTTP): A Stateless Search, Retrieve and Manipulation Protocol, November 1993.
http://ftp.std.com/obi/Networking/WWW/draft-ietf-iiir-http-00.txt.

[BL93b] Tim Berners-Lee. Uniform Resource Locators (URL): A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network, October 1993. Expired Internet Draft.
http://ftp.std.com/obi/Networking/WWW/url-spec.txt.

[BL94] Tim Berners-Lee. Universal Resource Identifiers in WWW, March 1994.
http://www.w3.org/Addressing/URL/uri-spec.html.

[BLC93] Tim Berners-Lee and Dan Connoly. Hypertext Markup Language (HTML): A Representation of Textual Information and MetaInformation for Retrieval and Interchange, June 1993.
http://www.w3.org/MarkUp/draft-ietf-iiir-html-01.

[BLCGP92] Tim Berners-Lee, Robert Caillau, Jean-Francois Groff, and Bernd Pollerman. World Wide Web: The Information Universe. *Electronic Networking: Research, Applications, and Policy*, 1(1), Spring 1992.
http://www.w3.org/History/1992/ENRAP/Article_9202.ps.

[BLFF96] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945, IETF, May 1996. Defines current usage of HTTP/1.0. **http://www.rfc-editor.org/rfc/rfc1945.txt.**

[BLFM98] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. RFC 2396, IETF, August 1998. **http://www.rfc-editor.org/rfc/rfc2396.txt.**

[BLFN95] Tim Berners-Lee, R. T. Fielding, and H. Frystyk Nielsen. Hypertext Transfer Protocol – HTTP/1.0, March 1995. Expired Internet Draft. **ftp://www.ics.uci.edu/pub/ietf/http/history/ draft-ietf-http-v10-spec-00.txt.**

[BLMM94] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). RFC 1738, IETF, December 1994. **http://www.rfc-editor.org/rfc/rfc1738.txt.**

[BM98] G. Banga and J. Mogul. Scalable Kernel Performance for Internet Servers Under Realistic Loads. In *Proc. USENIX*, June 1998. **http://www.cs.rice.edu/~gaurav/papers/usenix98.ps.**

[BPS+98] H. Balakrishnan, V. Padmanabhan, S. Seshan, M. Sremm, and R. Katz. TCP Behavior of a Busy Internet Server: Analysis and Improvements. In *Proc. IEEE INFOCOM*, March 1998. **http://www.ieee-infocom.org/1998/papers/02d_3.pdf.**

[Bra92] R. Braden. Extending TCP for Transactions – Concepts. RFC 1379, IETF, November 1992. **http://www.rfc-editor.org/rfc/rfc1379.txt.**

[Bra94] R. Braden. Extending TCP for Transactions – Functional Specifications. RFC 1644, IETF, July 1994. **http://www.rfc-editor.org/rfc/rfc1644.txt.**

[Bra96a] S. Bradner. Keywords for use in RFCs to Indicate Requirement Levels. RFC 2119, IETF, November 1996. **http://www.rfc-editor.org/rfc/rfc2119.txt.**

[Bra96b] S. Bradner. The Internet Standards Process –Revision 3. RFC 2026, IETF, October 1996. **http://www.rfc-editor.org/rfc/rfc2026.txt.**

[BRS99] H. Balakrishnan, H. Rahul, and S. Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *Proc. ACM SIGCOMM*, September 1999. **http://www.acm.org/sigcomm/sigcomm99/papers/session5-2.html.**

[BS00] Hari Balakrishnan and Srinivasan Seshan. The Congestion Manager, November 2000. Expired Internet Draft. **http://www.ietf.org/internet-drafts/draft-ietf-ecm-cm-03.txt.**

[Bus45] Vannevar Bush. As We May Think. Atlantic *Monthly*, July 1945. **http://www.theatlantic.com/unbound/flashbks/computer/bushf.htm.**

[Can] CA*net II Caching Hierarchy Project. **http://ardnoc41.canet2.net/cache/.**

[CAR] Cache array routing protocol and MS proxy server version 2.0. **http://www.microsoft.com/technet/Proxy/technote/prxcarp.asp.**

[Cat92] V. Cate. Alex – A Global FileSystem. In *Proc. USENIX File System Workshop*, pages 1-12. USENIX Association, May 1992.

[CB97] Mark E. Crovella and Azer Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. *IEEE/ACM Transactions on Networking*, 5(6):835-846, December 1997.

http://www.cs.bu.edu/fac/crovella/paper-archive/self-sim/journal-version.ps.

[CB98] Mark Crovella and Paul Barford. The Network Effects of Prefetching. In *Proc. IEEE INFOCOM*, April 1998. **http://www.ieee-infocom.org/1998/papers/10a_4.pdf.**

[CBC95] Carlos R. Cunha, Azer Bestavros, and Mark E. Crovella. Characteristics of WWW Client-based Traces. Technical Report BU-CS-95-010, Computer Science Department, Boston University, 1995. **http://www.cs.bu.edu/techreports/1995-010-www-client-traces.ps.Z.**

[CGC+00] I. Cooper, P. Gauthier, J. Cohen, M. Dunsmuir, and C. Perkins. The Web Proxy Auto-Discovery Protocol, November 2000. Expired Internet Draft. **http://wrec.org/Drafts/draft-cooper-webi-wpad-00.txt.**

[CHW99] Jon Crowcroft, Mark Handley, and Ian Wakeman. *Internetworking Multimedia.* Morgan Kaufmann. October 1999. ISBN 1558605843.

[CI97] Pei Cao and Sandy Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Proc. USENIX Symposium on Internet Technologies and Systems.* USENIX Association, December 1997. **http://www.usenix.org/publications/library/proceedings/usits97/full_papers/cao/cao_html/cao.html.**

[Cis] Cisco Content Delivery Networks. **http://www.cisco.com/go/cdn.**

[CK00] Edith Cohen and Haim Kaplan. Prefetching the Means for Document Transfer: A New Approach for Reducing Web Latency. In *Proc. IEEE INFOCOM, March 2000.* **http://www.ieee-infocom.org/2000/papers/438.ps.**

[CKR98] Edith Cohen, Balachander Krishnamurthy, and Jennifer Rexford. Improving End-to-End Performance of the Web Using Server Volumes and Proxy Filters. In *Proc. ACM SIGCOMM*, September 1998. **http://www.acm.org/sigcomm/sigcomm98/tp/abs_20.html.**

[CKR99] Edith Cohen, Balachander Krishnamurthy, and Jennifer Rexford. Efficient Algorithms for Predicting Requests to Web Servers. In *Proc. IEEE INFOCOM*, March 1999. **http://www.research.att.com/~bala/papers/infocomm99.ps.gz.**

[Cla88] David D. Clark. The Design Philosophy of DARPA Internet Protocols. In *Proc. ACM SIGGCOMM*, pages 106-114, August 1988. **http://www.acm.org/sigs/siggcomm/ccr/archive/1995/jan95/ccr-9501-clark.html.**

[Cle] Clever. **http://www.almaden.ibm.com/cs/k53/clever.html.**

[CMT98] K. Claffy, Greg Miller, and Kevin Thompson. The Nature of the Beast: Recent Traffic Measurements from an Internet Backbone. In *Proc. INET*, July 1998. **http://www.caida.org/outreach/papers/Inet98/.**

[Con92] Dan Connolly. MIME for global hypertext, June 1992. WWW-talk mailing list. **http://lists.w3.org/Archives/Public/www-talk/1992MayJun/0029.html.**

[Coo] Cookie Central. **http://www.cookiecentral.com.**

[CP95] L. D. Catledge and J. E. Pitkow. Characterizing browsing strategies in the World Wide Web. *Computer Networks and ISDN Systems*, 26(6):

1065-1073, 1995.
ftp://ftp.cc.gatetech.edu/pub/gvu/tr/1995/95-13.ps.Z.

[Cro82] David H. Crocker. Standard for Format of ARPA Internet Text Messages. RFC 822, IETF, August 1982.
http://www.rfc-editor.org/rfc/rfc822.txt.

[CSSa] World Wide Web Consortium, Cascading Style Sheets.
http://www.w3.org/Style.

[CSSb] Cascading Style Sheets.
http://www.htmlhelp.com/reference/css/properties.html.

[DA98] T. Dierks and C. Allen. The TLS Protocol, Version 1.0. RFC 2246, IETF, January 1998. **http://www.rfc-editor.org/rfc/rfc2246.txt.**

[Dav99] Brian Davison. Web Traffic Logs: An Imperfect Resource for Evaluation. In *Proc. INET*, June 1999.
http://www.cs.rutgers.edu/~davison/pubs/inet99/imperfect.html.

[DFKM97] Fred Douglis, Anja Feldman, Balachrander Krishnamurthy, and Jeffrey Mogul. Rate of Change and other Metrics: A Live Study of the World Wide Web. In *Proc. USENIX Symposium on Internet Technologies and Systems*, pages 147-158, December 1997.
http://www.research.att.com/~bala/papers/ros-usits97.ps.gz.

[DH98] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6). RFC 2460, IETF, December 1998.
http://www.rfc-editor.org/rfc/rfc2460.txt.

[Dig] Digital Island. **http://www.digitalisland.net.**

[Din95] Adam Dingle. HTTP should be able to transfer part of document, March 1995. HTTP-WG Mailing List Archives.
http://www.ics.uci.edu/pub/ietf/http/hypermail/1995q1/0105.html.

[DMF97] Bradley Duska, David Marwood, and Michael Feeley. The Measured Access Characteristics of World-Wide-Web Proxy Caches. In *Proc. USENIX Symposium on Internet Technologies and Systems*, December 1997.
http://www.usenix.org/publications/library/proceedings/usits97/full_papers/duska/duska_html/.

[DOK92] Peter B. Danzig, Katia Obraczka, and Anant Kumar. An Analysis of Wide-Area Name Server Traffic. In *Proc. ACM SIGCOMM*, pages 281-292, August 1992.
http://www.acm.org/pubs/articles/proceedings/comm/144179/p281-danzig/p281-danzig.pdf.

[Dor95] Tim Dorcey. CU-SeeMe Desktop VideoConferencing Software. *Connexions, 9(3)*, March 1995.
http://www.cu-seeme.net/squeek/tech/DorceyConnexions.html.

[Dro97] Ralph Droms. Dynamic Host Configuration Protocol. RFC 2131, IETF, March 1997. **http://www.rfc-editor.org/rfc/rfc2131.txt.**

[DS86] R. B. D'Augostino and M. A. Stephens, editors. *Goodness-of-Fit Techniques*. Marcel Decker, Inc., June 1986. ISBN 0824774876.

[Duc99] Dan Duchamp. Prefetching Hyperlinks. *In Proc. USENIX Symposium on Internet Technologies and Systems*, October 1999.
http://www.usenix.org/publications/lubrary/proceedings/usits99/duchamp.html.

[ED92] A. Emtage and P. Deutch. Archie — An Electronic Directory Service for the Internet. In *Proc. Winter USENIX Conference Proceedings*, pages 93-110, January 1992.

[EE68] Douglas C. Engelbart and William K. English. A Research Center for Augmenting Human Intellect. In *AFIPS Conference, Proc. of the 1968 Fall Joint Computer Conference*, pages 395-410, December 1968. **http://sloan.stanford.edu/mousesite/Archive/ResearchCenter1968/ ResearchCenter1968.html.**

[Eri94] Hans Eriksson. MBone: The Multicast Backbone. *Communications of the ACM*, 37(8):54-60, August 1994.

[Exo] Exodus, **http://www.exodus.net.**

[FB96a] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. RFC 2045, IETF, November 1996. Defines format of MIME message bodies. **http://www.rfc-editor.org/rfc/rfc2045.txt.**

[FB96b] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. RFC 2046, IETF, November 1996. Defines MIME mulipart types. **http://www.rfc-editor.org/rfc/rfc2046.txt.**

[FCAB98] Li Fan, Pei Cao, Jussara Almedia, and Andrei Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. In *Proc. ACM SIGCOMM*, pages 254-265, September 1998. **http://www.acm.org/sigcomm/sigcomm98/tp/abs_21.html.**

[Fcg] Fast CGI. **http://www.fastcgi.com.**

[Fea] HTTP/1.1 Feature List Report Summary. **http://www.w3.org/Protocols/HTTP/Forum/Reports.**

[Fel00a] Anja Feldmann. BLT: Bi-Layer Tracing of HTTP and TCP/IP. In *Proc. World Wide Web Conference*, pages 321-335, May 2000. **http://www.cs.uni-sb.de/~anja/feldmann/papers/blt_httptrace.ps.**

[Fel00b] Anja Feldmann. Characteristics of TCP Connection Arrivals. In K. Park and W. Willinger, editors, *Self-Similar Network Traffic and Performance Evaluation.* John Wiley, 2000.

[FF96] K. Fall and S. Floyd. Simulation-based comparison of Tahoe, Reno, and SACK TCP. *ACM Computer Communication Review*, 26(3):5-21, July 1996. **http://www.acm.org/sigs/sigcomm/ccr/archive/ 1996/jul96/ccr-9607-fall.html.**

[FF99] K. Fall and S. Floyd. Promoting the Use of End-to-End Congestion Control in the Internet. *IEEE/ACM Transactions on Networking*, 7(4): 458-472, August 1999. **http://www.aciri.org/floyd/papers/collapse.may99.ps.**

[FFBL95] R. Fielding, H. Frystik, and T. Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1, November 1995. Expired Internet Draft. **http://www.w3.org/Protocols/ HTTP/1.1/draft-ietf-http-v11-spec-00.txt.**

[FGM+97] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, and Tim Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. RFC 2068, IETF, January 1997. Proposed Standard of HTTP/1.1. **http://www.rfc-editor.org/rfc/rfc2068.txt.**

[FGM+99] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. RFC 2616, IETF, June 1999. Draft Standard of HTTP/1.1. **http://www.rfc-editor.org/rfc/rfc2616.txt.**

[FHBH+99] J. Franks, P. Hallman-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, E. Sink and L. Srewart. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617, IETF, June 1999. **http://www.rfc-editor.org/rfc/rfc2617.txt.**

[Fie94] R. T. Fielding. Maintaining Distributed Hypertext Infostructures: Welcome to MOMspider's Web. *Computer Networks and ISDN Systems*, 27(2): 193-204, November 1994. **http://www.ics.uci.edu/pub/websoft/MOMspider/docs/www94_paper.ps.**

[Fie95] Roy Fielding. Relative Uniform Resource Locators. RFC 1808, IETF, June 1995. **http://www.rfc-editor.org/rfc/rfc1808.txt.**

[Fie97] Roy Fielding. Content encoding problem, February 1997. HTTP-WG Mailing List Archives. http://**www.ics.uci.edu/pub/ietf/http/hypermail/1997q1/0151.html.**

[FKV95] Glenn Fowler, David Korn, and Kiem-Phong Vo. Libraries and file system architecture. In Balachander Krishnamurphy, editor, *Practical Reusable UNIX Software*, chapter 2. John Wiley, New York, NY, 1995. **http://www.research.att.com/library/books/reuse.**

[FL95] John Franks and Ari Luotonen. Byte ranges — formal spec proposal, May 1995. HTTP-WG Mailing List Archives. **http://www.ics.uci.edu/pub/ietf/http/hypermail/1995q2/0122.html.**

[Flo94] Sally Floyd. TCP and Explicit Congestion Notification. *ACM Computer Communication Review*, 24(5):10-23, October 1994. **http://www.aciri.org/floyd/papers/tcp_ecn.4.pdf.**

[Flo00] Sally Floyd. Congestion Control Principles. RFC 2914, IETF, September 2000. **http://www.rfc-editor.org/rfc/rfc2914.txt.**

[FLYV93] V. Fuller, T. Li, J. Y. Yu, and K. Varadhan. Classless InterDomain Routing (CIDR): An Address Asignment and Aggregation Strategy. RFC 1519, IETF, September 1993. **http://www.rfc-editor.org/rfc/rfc1519.txt.**

[For99] Fortune 500 Companies, 1999. **http://www.fortune.com/fortune/fortune500/.**

[Fou] Foundry Networks. **http://www.foundrynet.com.**

[Fra94a] John Franks. An MGET Proposal for HTTP, October 1994. WWW-talk mailing list. **http://www.webhistory.org/www.lists/www-talk.1994q4/0479.html.**

[Fra94b] John Franks. Proposal for HTTP MGET Method, December 1994. HTTP-WG Mailing List Archives. **http://www.ics.uci.edu/pub/ietf/http/hypermail/1994q4/0260.html.**

[Fra95] John Franks. Re: HTTP should be able to transfer part of a document, March 1995. HTTP-WG Mailing List Archives. **http://www.ics.uci.edu/pub/ietf/http/hypermail/1995q1/0107.html.**

[FRB93] P. S. Ford, Y. Rekhter, and H-W. Braun. Improving the Routing and Addressing of IP. *IEEE Netwotk Magazine*, 7(3):10-15, May 1993.

[FTY99] Theodore Faber, Joe Touch, and Wei Yue. The TIME-WAIT state in TCP and Its Effect on Busy Servers. In *Proc. IEEE INFOCOM*, pages 1573-1583, March 1999.
http://www.isi.edu/~touch/pubs/infocomm99/.

[Get97] Jim Gettys. HTTP Connection Management, March 1997. HTTP-WG Mailing List Archives.
http://www.ics.uci.edu/pub/ietf/http/hypermail/1997q1/0656.html.

[Glo98] Global 500 Companies, 1998. **http://fortune.com/fortune/global500/.**

[GN98] J. Gettys and H. F. Nielsen. The WebMUX protocol, August 1998. Expired Internet Draft.
http://www.w3.org/Protocols/MUX/WD-mux-980722.html.

[Goo] Google. **http://www.google.com.**

[GRB00] Stephane Gruber, Jennifer Rexford, and Andrea Basso. Protocol Considerations for Prefix-Caching Proxy for Multimedia Streams. In *Proc. World Wide Web Conference*, pages 657-668, May 2000.
http://research.att.com/~jrex/papers/www00.ps.

[Gun96] Shishir Gundavaram. *CGI Programming on the World Wide Web*. O'Reilly, March 1996. **http://www.oreilly.com/openbook.cgi.**

[GWF+99] Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen. HTTP Extensions for Distributed Authoring – WEBDAV. RFC 2518, IETF, February 1999. **http://www.rfc-editor.org/rfc/rfc2518.txt.**

[HBD97] Mor Harchol-Balter and Allen Downey. Exploiting Process Lifetime Distributions for Dynamic Balancing. *ACM Transactions on Computer Systems*, 15(3):253-285, August 1997.
http://www.cs.cmu.edu/~harchol/Papers/TOCS.ps.

[HBMT99] R. Herriot, S. Butler, P. Moore, and R. Turner. Internet Printing Protocol/1.0: Encoding and Transport. RFC 2565, IETF, April 1999.
http://www.rfc-editor.org/rfc/rfc2565.txt.

[Hei97] J. Heidemann. Performance Interactions Between P-HTTP and TCP-Implementations. *ACM Computer Communication Review*, 27(2):65-73, April 1997.
http://www.acm.org/sigs/sigcomm/ccr/archive/1997/apr97/ccr-9704-heidemann.html.

[HJ98] M. Handley and V. Jacobson. SDP: Session Description Protocol. RFC 2327, IETF, April 1998. **http://www.rfc-editor.org/rfc/rfc2327.txt.**

[HL96] Barron C. Housel and David B. Lindquist. WebExpress: A System for Optimizing Web Browsing in the Wireless Environment. In *Proc. ACM/IEEE MOBICOM*, pages 419-431, October 1996.
http://www.baltzer.nl/monet/articlesfree/1998/3-4/mnt078.pdf.

[HM98] K. Holtman and A. Mutz. Transparent Content Negotiation in HTTP. RFC 2295, IETF, March 1998.
http://www.rfc-editor.org/rfc/rfc2295.txt.

[HM00] Sam Halabi and Daniel McPherson. *Internet Routing Architectures*. Cisco Press, second edition, 2000. ISBN 157870233X.

[HN99] Allan Heidon and Mark Najork. Mercator: A scalable, extensible Web clawler. *World Wide Web*, 2(4), 1999. **http://www.research.compaq.com/SRC/mercator/papers/www/paper.pdf.**

[Hol] Koen Holtman. Transparent Content Negotiation in HTTP. **http://gewis.win.tue.nl/~koen/conneg/.**

[Hor83] Mark R. Horton. Standard for Interchange of USENET Messages. RFC 850, IETF, June 1983. **http://www.rfc-editor.org/rfc/rfc850.txt.**

[Hor98] Eric Horwitz. Continual Computation Policies for Utility-Directed Prefetching. In *Proc. ACM Conference on Information and Knowledge Management*, pages 157-184, November 1998. **http://www.research.microsoft.com/~horvitz/ccfetch.htm.**

[Hos00] P. Hoschka. The application/smil Media Type, October 2000. Expired Internet Draft. **http://www.ietf.org/internet-drafts/draft-hoschka-smil-media-type-06.txt.**

[Hot] 100hot Web Rankings. **http://100hot.com.**

[HPW00] M. Handley, C. Perkins, and E. Whelan. Session Announcement Protocol. RFC 2974, IETF, October 2000. **http://www.rfc-editor.org/rfc/rfc2974.txt.**

[HRW98] Martin Hamilton, Alex Rousskov, and Duane Wessels. Cache Digest, December 1998. **http://www.squid-cache.org/CacheDigest.**

[HTT94] Minutes of HyperText Transfer Protocol Working Group, December 1994. 31st IETF Meeting. **ftp://ftp.ietf.cnri.reston.va.us/ietf-online-proceedings/94dec/area.and.wg.reports/app/http/http-minutes-94dec.txt.**

[Hui98] Cristian Huitema. *IPv6: The New Internet Protocol.* Prentice Hall, second edition, January 1998. ISBN 0138505055.

[Hui00] Cristian Huitema. *Routing in the Internet.* Prentice Hall, second edition, January 2000. ISBN 0130226475.

[HW00] Cristian Huitema and Sam Weeranhandi. Internet Measurements: The Rising Tide and DNS Snag. In *Proc. ITC Seminar on IP Traffic Measurement, Modeling and Management*, September 2000.

[ICA] The Internet Corporation for Assigned Names and Numbers. **http://www.icann.org**

[ICA01] ICAP Protocol Group. ICAP: The Internet Content Adaptation Protocol, February 2001. Work in progress. **http://www.i-cap.org/icap/media/draft-elson-opes-icap-01.txt.**

[IMA] The IMAP Connection. **http://www.imap.org.**

[Ink] Inktomi Search. **http://www.inktomi.com.**

[IRC] A Distributed Testbed for National Information Provisioning. **http://ircache.nlanr.net/Cache/.**

[ISO] Internet Histories. **http://www.isoc.org/internet-history.**

[ITA] Internet Traffic Archive. http:**//www.acm.org/sigcomm/ita/.**

[J. 81] J. Postel, Editor. Transmission Control Protocol. RFC 793, IETF, September 1981. **http://www.rfc-editor.org/rfc/rfc793.txt.**

[Jac] Van Jacobson. Traceroute. **ftp://ftp.ee.lbl.gov/traceroute.tar.gz.**

[Jac88] Van Jacobson. Congestion Avoidance and Control. In *Proc. ACM SIGCOMM*, pages 314-329, August 1988. **http://www.acm.org/sigcomm/ccr/archive/1995/jan95/ccr-9501-jacobson.html.**

[JCDK00] Kirk L. Johnson, John F. Carr, Mark S. Day, and M. Frans Kaashoek. The Measured Performance of Content Distribution Networks. In *Proc. Fifth Web Caching Workshop*, May 2000. **http://terena.nl/conf/wcw/Proceedings/S4/S4-1.pdf.**

[JK98] Zhimei Jiang and Leonard Kleinrock. An adaptive prefetching scheme. *IEEE Journal on Selected Areas in Communications*, 16(3):358-368, April 1998. **http://www.research.att.com/~jiang/Research/Publication/prefetch_jsac98.pdf.**

[JLM] Van Jacobson, C. Leres, and S. McCanne. TcpDump. **ftp://ftp.ee.lbl.gov/tcpdump.tar.Z.**

[Joh99] John Dilley. The Effect of Consistency on Cache Response Time. Technical Report HPL-1999-107, Hewlett Packard Laboratories, September 1999. **http://www.hpl.com.hp.com/techreports/1999/HPL-1999-107.html**

[Joh00] Kevin Johnson. *Internet Email Protocols: A Developers Guide*. Addison-Wesley, January 2000. ISBN 0201432889.

[Jon81] Jon Postel, Editor. Internet Protocol. RFC 791, IETF, September 1981. **http://www.rfc-editor.org/rfc/rfc791.txt.**

[Jun] Junkbusters. **http://www.junkbusters.com**.

[KAO1] Balachander Krishnamurphy and Martin Arlitt. PRO-COW: Protocol Compliance on the Web — A Longitudinal Study. In *Proc. USENIX Symposium on Internet Technologies and Systems*, March 2001. **http://www.research.att.com/~bala/papers/usits01.ps.gz**.

[KBM94] Eric Dean Katz, Michelle Butler, and Robert McGraph. A Scalable HTTP Server: The NCSA Prototype. In *Proc. World Wide Web Conference*, May 1994. **http://www.ncsa.uiuc.edu/InformationServers/Conferences/CERNwww94/www94.ncsa.html.**

[Kes97] Srinivsan Keshav. *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network*. Addison-Wesley, January 1997. ISBN 0201634422.

[Key] Keynote Systems. **http://www.keynote.com**.

[KL86] Brian Kantor and Phil Lapsley. Network News Transfer Protocol. RFC 977, IETF, February 1986. **http://www.rfc-editor.org/rfc/rfc977.txt.**

[KL00] R. Khare and S. Lawrence. Upgrading to TLS Within HTTP/1.1. RFC 2817, IETF, May 2000. **http://www.rfc-editor.org/rfc/rfc2817.txt.**

[Kle75] Leonard Kleinrock. *Queuing Systems, Volume I: Theory*. John Wiley, January 1975. ISBN 0471491101.

[KM87] C. A. Kent and J. C. Mogul. Fragmentation Considered Harmful. In *Proc. ACM SIGCOMM*, pages 390-401, August 1987. **http://www.acm.org/sigs/sigcomm/ccr/archive/1995/jan95/ccr-9501-mogulf1.html.**

[KM91] B. Kahle and A. Medlar. An Information System for Corporate Users: Wide Area Information Servers, November 1991. Connexions — The Interoperability Report, 5(11).

[KM00] David Kristol and Lou Montulli. HTTP State Management Mechanism. RFC 2965, IETF, October 2000.
http://www.rfc-editor.org/rfc/rfc2965.txt.

[KMK99] Balachander Krishnamurphy, Jeffrey C. Mogul, and David M. Kristol. Key Differences between HTTP/1.0 and HTTP/1.1. In *Proc. Eighth International World Wide Web Conference*, May 1999.
http://www.research.att.com/~bala/papers/h0vh1.html.

[KMR95] T. Kwan, R. McGrath, and D. Reed. NCSA's World Wide Web Server: Design and Performance. *IEEE Computer Magazine*, 28(11):68-74, November 1995.

[KR98] Balachander Krishnamurphy, and Jennifer Rexford. Software Issues in Characterizing Web Server Logs. In *W3C Web Characterization Group Workshop*, November 1998.
http://www.research.att.com/~bala/papers/ew3c.html.

[KR00] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet.* Addison-Wesley, July 2000. ISBN 0201477114.

[KV91] David G. Korn and Kiem-Phong Vo. SFIO: Safe/Fast String/File I/O. In *Proc. Summer USENIX Conference*, pages 235-256, 1991.

[KV00] David G. Korn and Kiem-Phong Vo. The VCDIFF Generic Differencing and Compression Data Format, November 2000. Expired Internet Draft.
ftp://ftp.ietf.org/internet-drafts/draft-korn-vcdiff-02.txt.

[KW97] Balachander Krishnamurphy and Craig Wills. Study of Piggyback Cache Validation for Proxy Caches in World Wide Web. In *Proc. USENIX Symposium on Internet Technologies and Systems*, pages 1-12, December 1997.
http://www.research.att.com/~bala/papers/pcv-usits97.ps.gz.

[KW98] Balachander Krishnamurphy and Craig E. Wills. Piggyback Server Invalidation for Proxy Cache Coherency. In *Proc. World Wide Web Conference*, April 1998.
http://www.research.att.com/~bala/papers/psi-www7.ps.gz.

[KW00] Balachander Krishnamurphy, and Craig E. Wills. Analyzing Factors that influence end-to-end Web performance. In *Proc. World Wide Web Conference*, pages 17-32, May 2000.
http://www.research.att.com/~bala/papers/www9.html.

[LA94] Ari Luotonen and Kevin Altis. World-Wide Web Proxies. In Proc. First International Conference on the World-Wide Web, WWW '94, May 1994.
http://cern.ch/PaoersWWW94/luotonen.ps.

[LB97] Tong Sau Loon and Vaduvur Bharghavan. Alleviating the Latency and Bandwidth Problems in WWW Browsing. In *Proc. USENIX Symposium on Internet Technologies and Systems*, December 1997.
http://www.usenix.org/publications/library/proceedings/usits97/tong.html.

[LC97] Chengjie Liu and Pei Cao. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proc. International Conference on Distributed Com-*

puting Systems, pages 326-334, May 1997.
http://www.cs.wisc.edu/~cao/papers/icache.html.

[LCC+97] Barry M. Leiner, Vinton G. Cerf, David D. Clark, Robert E. Kahn, Leonard Kleinrock, Daniel C. Lynch, Jon Postel, Lawrence G. Roberts, and Stephen S. Wolff. The Past and Future History of the Internet. *Communications of ACM*, 40(2):102-108, February 1997.

[LDV99] Henry Lieberman, Neil Van Dyke, and Adriana Vivacqua. Let's Browse: A Collaborative Browsing Agent. In *Proc. International Conference on Intelligent User Interfaces*, January 1999.
http://lieber.www.media.mit.edu/people/lieber/Lieberary/Lets-Browse/Lets-Browse.html.

[LG99] Steve Lawrence and C. Lee Giles. Accessibility of Information on Web. *Nature*, 400(6740):107-109, 1999.
http://www.neci.nj.nec.com/homepages/lawrence/.

[Lie95] H. Lieberman. Letizia: An Agent That Assists Web Browsing. In *Proc. International Joint Conference on Artificial Intelligence*, August 1995.
http://lieber.www.media.mit.edu/people/lieber/Lieberary/Letizia/Letizia-AAAI/Letizia.html.

[LK99] Averill Law and David Kelton. *Simulation, Modeling, and Analysis.* McGraw Hill, third edition, December 1999, ISBN 0070592926.

[LL99] Ben Laurie and Peter Laurie. *Apache: The Definitive Guide.* O'Reilly, February 1999. ISBN 1565925289.

[LNJV99] Z. Liu, N. Niclausse, and C. Jalpa-Vilanueva. Web Traffic Modeling and Performance Comparison Between HTTP1.0 and HTTP1.1. In Erol Gelenbe, editor, *System Performance Evaluation: Methodologies and Applications.* CRC Press, August 1999.
http://www-sop.inria.fr/mistral/personnel/Zhen.Liu/Papers/wagon_perf99.ps.gz.

[LTWW94] Will. E. Lelend, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the Self-Similar Nature of Ethernet Traffic (Extended Version). *IEEE/ACM Transactions on Networking*, 2(1):1-15, February 1994.

[Luo97] Ari M. Luotonen. *Web Proxy Servers.* Prentice Hall, 1997. ISBN 0-13680-612-0.

[Mah97] Bruce Mah. An Empirical Model of HTTP Network Traffic. In *Proc. IEEE INFOCOM*, April 1997. **http://www.ieee-infocom.org/1997/papers/bmah.pdf.**

[MCS98] S. Manley, M. Courage, and M. Seltzer. A Self-Scaling and Self-Configuring Benchmark for Web Servers. In *Proc. ACM SIGMETRICS*, pages 270-271, June 1998.
http://www.eecs.harvard.edu/~margo/papers/hbench-web.ps.

[MD90] J. Mogul and S. Deering. Path MTU Discovery. RFC 1191, IETF, November 1990. **http://www.rfc-editor.org/rfc/rfc1191.txt.**

[MDFK97a] Jeffry C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proc. ACM SIGCOMM*, pages 181-194, August 1997.
http://www.acm.org/sigcomm/sigcomm97/papers/p156.html.

[MDFK97b] Jeffry C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. Technical Report 97/4, DEC, July 1997. **http://gatekeeper.dec.com/pub/Digital/WRL/research-reports/WRL-TR-97.4.pdf.**

[MDH] Jeffrey C. Mogul, Fred Douglis, and Daniel Hellerstein. HTTP Delta Clusters and Templates. Work in progress. **http://www.ietf.org/internet-drafts/draft-mogul-http-dcluster-00.txt.**

[Med] Media Metrix. **http://www.mediametrix.com.**

[MF00] Keith Moore and Ned Freed. Use of HTTP State Management. RFC 2964, IETF, October 2000. **http://www.rfc-editor.org/rfc/rfc2964.txt.**

[MFGF97] J. Mogul, R. Fielding, J. Gettys, and H. Frystyk. Use and Interpretation of HTTP Version Numbers. RFC 2145, IETF, May 1997. **http://www.rfc-editor.org/rfc/rfc2145.txt.**

[MFH00] A. Mockus, R.F. Fielding, and J. Herbsleb. A Case Study of Open Source Development: The Apache Server. In *Proc. International Conference on Software Engineering*, pages 263-272, June 2000. **http://dev.acm.org/pubs/citations/proceedings/soft/337180/p263-mockus/.**

[MH00] Art Mena and John Heidemann. An Empirical Study of Real Audio Traffic. In *Proc. IEEE INFOCOM*, March 2000. **http://www.isi.edu/~johnh/PAPERS/Mena00a.html.**

[Mil92] David L. Mills. Network Time Protocol (Version 3): Specification, Implementation and Analysis. RFC 1305, IETF, March 1992. **http://www.rfc-editor.org/rfc/rfc1305.txt.**

[Mir] Mirror Image. **http://www.mirror-image.com.**

[MJ93] Steve McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *Proc. Winter USENIX Technical Conference*, January 1993. **ftp://ftp.ee.lbl.gov/papers/bpf-usenix93.ps.Z.**

[MJ98] D. Mosberger and T. Jin. httpref — A Tool for Measuring Web Server Performance. In *Proc. Workshop on Internet Server Performance*, pages 59-67, June 1998. **http://www.hpl.hp.com/personal/David_Mosberger/httpref.**

[MKD+01] Jeffrey Mogul, Balachander Krishnamurphy, Fred Douglis, Anja Feldmann, Yaron Goland, Arthur van Hoff, and Daniel Hellerstein. Delta encoding in HTTP, March 2001. Work in progress. **http://www.ietf.org/intenet-drafts/draft-mogul-http-delta-08.txt.**

[ML97] J. Mogul, P. Leach. Simple Hit-Metering and Usage-Limiting for HTTP. RFC 2227, IETF, October 1997. **http://www.rfc-editor.org/rfc/rfc2227.txt.**

[MMFR96] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgement Options. RFC 2018, IETF, April 1996. **http://www.rfc-editor.org/rfc/rfc2018.txt.**

[Moc87a] P. Mockapetris. Domain Names — Concepts and Facilities. RFC 1034, IETF, November 1987. **http://www.rfc-editor.org/rfc/rfc1034.txt.**

[Moc87b] P. Mockapetris. Domain Names — Implementation and Specification. RFC 1035, IETF, November 1987. **http://www.rfc-editor.org/rfc/rfc1035.txt.**

[Mog95] Jeffrey Mogul. A modest proposal, August 1995, HTTP-WG Mailing List Archives. **http://www.ics.uci.edu/pub/ietf/http/hypermail/1995q3/0360.html.**

[Mos] NCSA Mosaic. A WWW Browser. **http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/help-about.html.**

[MSMV99] G. Minshall, Y. Saito, J. Mogul, and B. Verghese. Application Performance Pitfalls and TCP's Nagle Algoriphm. In *Proc. Workshop on Internet Server Performance*, May 1999. **http://www.cc.gatech.edu/fac/Ellen.Zegura/wisp99/papers/minshall.ps.**

[Muf] Muffin: World Wide Web Filtering System. **http://muffin.doit.org.**

[NA93] B. Clifford Neuman and Steven Seger Augart. Prospero: A Base for Building Information Infostructure. In *Proc. INET*, August 1993. **http://www.isi.edu/people/bcn/papers/pdf/9308_prospero-bii.pdf.**

[Nag84] John Nagle. Congestion Control in IP/TCP Internetworks. RFC 896, IETF, January 1984. **http://www.rfc-editor.org/rfc/rfc896.txt.**

[NBK99] E. Nahum, T. Barzalai, and D. Kandlur. Performance issues in WWW servers. Technical report, IBM Research, February 1999. **ftp://gaia.cs.umass.edu/pub/nahum/Nahu99:Performance.ps.**

[Nel67] T. H. Nelson. Getting It Out of Our System. In G. Schechter, editor, *Critique of Information Retrieval*, pages 191-210. Thompson Books, 1967.

[Neta] Network Appliance. **http://www.netapp.com.**

[Netb] Netcaching. **http://www.netcaching.com.**

[Netc] The Netcraft Web Server Survey. **http://netcraft.co.uk/survey.**

[Netd] Persistent client state HTTP cookies. **http://www.netscape.com/newsref/std/cookie_spec.html.**

[Nie97] Henrik Frystik Nielsen. Pipelining and compression effect on HTTP/1.1 proxies, April 1997. HTTP-WG Mailing List Archives. **http://www.ics.uci.edu/pub/ietf/http/hypermail/1997q2/0165.html.**

[NLL00] H. Nielsen, P. Leach, and S. Lawrence. An HTTP Extension Framework. RFC 2774, IETF, February 2000. **http://www.rfc-editor.org/rfc/rfc2774.txt.**

[Nov] Novell. **http://www.novell.com.**

[P3P] World Wide Web Consortium, Platform for Privacy Preferences (P3P) Project. **http://www.w3.org/P3P/.**

[PA00] V. Paxson and M. Allman. Computing TCP's Retransmission Timer. RFC 2988, IETF, November 2000. **http://www.rfc-editor.org/rfc/rfc2988.txt.**

[Pad95] V. N. Padmanabhan. Improving World Wide Web Latency. Technical Report UCB/CSD-95-875, University of California, Berkley, May 1995. **http://www.research.microsoft.com/~padmanab/papers/masters-tr.ps.**

[Pax97a] Vern Paxson. Automated Packet Trace Analysis of TCP Implementations. In *Proc. ACM SIGGCOMM*, September 1997. **ftp://ftp.ee.lbl.gov/papers/vp-tcpanaly-sigcomm97.ps.Z.**

[Pax97b] Vern Paxson. End-to-End Routing Behavior in the Internet. *IEEE/ACM Transactions on Networking*, 5(5):601-615, October 1997. **ftp://ftp.ee.lbl.gov/papers/vp-routing-TON.ps.Z.**

[Pax99] Vern Paxson. End-to-End Internet Packet Dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277-292, June 1999. **ftp://ftp.ee.lbl.gov/papers/vp-pkt-dyn-ton99.ps.gz.**

[PDZ99] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An Efficient and portable Web server. In *Proc. USENIX*, June 1999. **http://www.cs.princeton.edu/~vivek/flash99/flash.ps.gz.**

[Pea97] Oscar Pearson. Squid users guide, September 97. **http://www.squid-cache.org/Doc/Users-Guide/Welcome.html.**

[Pel91] Nicola Pellow. linemode, June 1991. **http://www.w3.org/Talks/Seminar_LM.html.**

[PFG+94] M. St. Pierre, J. Fullton, K. Gamiel, J. Goldman, B. Kahle, J. Kunze, H. Morris, and F. Schiettecatte. WAIS over Z39.50-1988. RFC 1625, IETF, June 1994. **http://www.rfc-editor.org/rfc/rfc1625.txt.**

[PH97] J. Palme and A. Hopmann. MIME E-mail Encapsulation of Aggregate Documents, such as HTML (MHTML). RFC 2110, IETF, March 1997. **http://www.rfc-editor.org/rfc/rfc2110.txt.**

[PH99] J. Palme and A. Hopmann. MIME E-mail Encapsulation of Aggregate Documents, such as HTML (MHTML). RFC 2557, IETF, March 1999. Obsoletes RFC2110. **http://www.rfc-editor.org/rfc/rfc2557.txt.**

[Pit99] James E. Pitkow. Summary of WWW characterizations. *WWW Journal*, 2:3-13, 1999. **http://www.baltzer.nl/www/contents/1999/2-1,2/www024.pdf.**

[PK99] J. Padhye and J. Kurose. An Empirical Study of Client Interactions of a Continuous-Media Courseware Server. *IEEE Internet Computing*, April 1999. **ftp://gaia.cs.umass.edu/pub/Padh97:Empirical.ps.gz.**

[PM95] Venkata N. Padmanabhan and Jeffrey C. Mogul. Improving HTTP Latency. *Computer Networks and ISDN Systems*, 28(1/2):25-35, December 1995. **http://www.research.microsoft.com/~padmanb/papers/www-fall94.ps.**

[PM96] Venkata N. Padmanabhan and Jeffrey Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. *ACM Computer Communication Review*, 26(3):22-36, July 1996. **http://www.acm.org/sigs/sigcomm/ccr/archive/1996/jul96/ccr-9607-pad.html.**

[Pol] Web Polygraph: Proxy performance benchmark. **http://polygraph.ircache.net/.**

[Pos81] J. Postel. Internet Control Message Protocol. RFC 792, IETF, September 1981. **http://www.rfc-editor.org/rfc/rfc792.txt.**

[Pos82] Jonothan B. Postel. Simple Mail Transfer Protocol. RFC 821, IETF, August 1982. **http://www.rfc-editor.org/rfc/rfc821.txt.**

[Pos94] J. Postel. Domain Name System Structure and Delegation. RFC 1591, IETF, March 1994. **http://www.rfc-editor.org/rfc/rfc1591.txt.**

[PQ00] Venkata Padmanabhan and Lili Qiu. The Content and Access Dynamics of Busy Web Site: Findings and Implications. In *Proc. ACM SIGGCOM,*

August/September 2000.
http://www.acm.org/sigcomm/sigcomm2000/conf/paper/sigcomm2000-3-3.ps.gz

[PR85] Jon Postel and Joyce Reynolds. File Transfer Procol. RFC 959, IETF, October 1985. **http://www.rfc-editor.org/rfc/rfc959.txt**.

[PR93] J. Postel and J. Reynolds. Telnet Protocol Specification. RFC 854, IETF, May 1993. **http://www.rfc-editor.org/rfc/rfc854.txt**.

[Rag] Raging Search. **http://www.raging.com**.

[RES] A Standard for Robot Exclusion.
http://info.webcrawler.com/mak/projects/robots/norobots.html.

[RF99] K. K. Ramakrishnan and S. Floyd. A Proposal to Add Explicit Congestion Notification (ECN) to IP. RFC 2481, IETF, January 1999.
http://www.rfc-editor.org/rfc/rfc2481.txt.

[RGR97] Aviel Rubin, Daniel Geer, and Marcus Ranum. *Web Security Sourcebook*. John Wiley, 1997. ISBN 047118148X.

[Riv92] Ronald Rivest. The MD5 Message-Digest Algorithm. RFC 1321, IETF, April 1992. **http://www.rfc-editor.org/rfc/rfc1321.txt**.

[RL93] Yakov Rekhter and Tony Li. An Architecture for IP Address Allocation with CIDR. RFC 1518, IETF, September 1993.
http://www.rfc-editor.org/rfc/rfc1518.txt.

[RL95] Y. Rekhter and T. Li. A Border Gateway Prorocol 4 (BGP-4). RFC 1771, IETF, March 1995. **http://www.rfc-editor.org/rfc/rfc1771.txt**.

[Roc75] M. Rochkind. The Source Code Control System (SCCS). *IEEE Transactions on Software Engineering*, 1(4):364-370, December 1975.

[RSA] RSA Security. **http://www.rsa.com**.

[Sal95] Peter H. Salus. *Casting the Net: From ARPANET to INTERNET and Beyond*. Addison-Wesley, March 1995. ISBN 0201876744.

[SCFJ96] H. Schuzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 1889, IETF, January 1996. **http://www.rfc-editor.org/rfc/rfc1889.txt**.

[Sch96] H. Schuzrinne. RTP Profile for Audio and Video Conferences with Minimal Control. RFC 1890, IETF, January 1996.
http://www.rfc-editor.org/rfc/rfc1890.txt.

[Sec95] IESG Secretary. WG Action: HyperText Transfer Protocol (http), January 27, 1995.
http://www.isc.uci.edu/pub/ietf/http/hypermail/1995q1/0050.html.

[Sha86] M. Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Proc. 6th International Conference on Distributed Computer Systems*, pages 198-204, 1986.

[Sin95] Erik Sink. HTTP/1.2 stuff: try it out!, August 1995. HTTP-WG Mailing List Archives.
http://www.ics.uci.edu/pub/ietf/http/hypermail/1995q3/0405.html.

[SJ00] Mike Spreitzer and Bill Janssen. HTTP Next Generation. In *Proc. World Wide Web Conference*, May 2000.
http://www9.org/w9cdrom/60/60.html.

[SM94] K. Sollins and L. Masinter. Functional Requirements for Uniform Resource Names. RFC 1737, IETF, December 1994. **http://www.rfc-editor.org/rfc/rfc1737.txt.**

[Smi] World Wide Web Consortium, Synchronized Multimedia. **http://www.w3.org/AudioVideo/.**

[Sol] Solidspeed. **http://www.solidspeed.com.**

[Spea] SPECweb99 Benchmark. **http://www.spec.org/osg/web99/.**

[Speb] Speedera. **http://www.speedera.com.**

[Spi] The search engine watch SpiderSpotting chart. **http://www.searchingenginewatch.com/webmasters/spiderchart.htm.**

[SR99] Henning Schulzrinne and Jonothan Rosenberg. The IETF Internet Telephony Architecture and Protocols. *IEEE Network Magazine*, 13(3):18-23, May/June 1999.

[SR00] M.-T. Sun and A.R. Reibman, editors. *Compressed Video over Networks.* Marcel Dekker, September 2000. ISBN 0824794230.

[SRL98] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). RFC 2326, IETF, April 1998. **http://www.rfc-editor.org/rfc/rfc2326.txt.**

[SRT99] Subhabrata Sen, Jennifer Rexford and Don Towsley. Proxy Prefix Caching for Multimedia Streams. In *Proc. IEEE INFOCOM*, pages 1310-1319, April 1999. **http://www.ieee-infocom.org/1999/papers/09d-04.pdf.**

[SSL] Netscape Secure Sockets Layer (SSL) Version 3.0. **http://home.netscape.com/eng/ssl3/.**

[SSL95] Netscape Secure Sockets Layer (SSL) Documentation, 1995. **http://home.netscape.com/security/techbriefs/ssl.html.**

[SSR99] H. Schulzrinne, E. Schooler, and J. Rosenberg. SIP: Session Initiation Protocol. RFC 2543, IETF, March 1999. **http://www.rfc-editor.org/rfc/rfc2543.txt.**

[SSV97] Peter Scheuermann, Junho Shim, and Radek Vingralek. A Case for Delay-Conscious Caching of Web Documents. In *Proc. World Wide Web Conference*, April 1997. **http://www.bell-labs.com/user/rvingral/www97.html.**

[St.93] M. St. Johns. Identification Protocol. RFC 1413, IETF, February 1993. **http://www.rfc-editor.org/rfc/rfc1413.txt.**

[Ste94] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley, January 1994, ISBN 0201633469.

[Ste98a] Lincoln D. Stein. Web Security: *A Step-by-Step Reference Guide.* Addison-Wesley, October 1998. ISBN 0-201-63489-9.

[Ste98b] W. Richard Stevens. *UNIX Network Programming, Volume 1: Networking APIs — Sockets and XTI.* Prentice Hall, 1998. ISBN 013490012X.

[Ste99] John W. Stewart, *BGP4: Inter-Domain Routing in the Internet.* Addison-Wesley, January 1999. ISBN 0201379511.

[Syn98] Synchronized Multimedia Working Group. Synchronized Multimedia Integration Language (SMIL) 1.0. Technical Report Recommendation REC-smil-19980615, World Wide Web Consortium. July 1998. **http://www.w3.org/TR/1998/REC-smil/.**

[Tcp] Tcpdump public repository. **http://www.tcpdump.org.**

[Tha96] Robert Thau. Design Considerations for Apache Server API. In *Proc. World Wide Web Conference*, May 1996. **http://www5conf.inria.fr/fich_html/papers/P20/Overview.html.**

[Tho96] Stephen A. Thomas. RTP for Real Time Applications. *In IPng and the TCP/IP Protocols*, chapter 11, pages 351-374. John Wiley, January 1996. ISBN 0471130885.

[Tic85] Walter F. Tichy. RCS — A System for version control. *Software: Practice and Experience*, 15(7): 637-654, July 1985.

[Tou97] J. Touch. TCP Control Block Interdependence. RFC 2140, IETF, April 1997. **http://www.rfc-editor.org/rfc/rfc2140.txt.**

[TS95] G. Trent and M. Sake. WebStone: The First Generation in HTTP Server Benchmarking, February 1995. **http://www.mindcraft.com/webstone/paper.html.**

[Uni] Unitech Network Ltd — InelliDNS. **http://www.initechnetworks.com.**

[vCCS00] J. van der Merwe, R. Caceres, Y. Chu, and C. J. Sreenan. mm-dump : A Tool for Monitoring Internet Multimedia Traffic. *ACM Computer Communication Review*, 30(5):48-59, October 2000. **http://www.acm.org/sigcomm/ccr/archive/2000/oct00/ccr_200010-merwe.html.**

[WAS+96] Stephen Williams, Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, and Edward A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Proc. ACM SIGCOMM*, pages 293-305, August 1996. **http://www.acm.org/sigcomm/sigcomm96/papers/williams.html.**

[Wbe] Webbench. **http://www.zdnet.com/etestinglabs/stories/benchmarks/0,8829,2326243,00.html.**

[WC97a] D. Wessels and K. Claffy. Application of Internet Cache Protocol (ICP), Version 2, RFC 2187, IETF, September 1997. **http://www.rfc-editor.org/rfc/rfc2187.txt.**

[WC97b] D. Wessels and K. Claffy. Internet Cache Protocol (ICP), Version 2, RFC 2186, IETF, September 1997. **http://www.rfc-editor.org/rfc/rfc2186.txt.**

[WC98] Duane Wessels and K. Claffy. ICP and the Squid Web Cache. *IEEE Journal on Selected Areas in Communications*, 16(3):345-357, April 1998. **http://www.caida.org/outreach/papers/pdf/icp-sq.pdf.**

[WCA] Web Characterization Repository. **http://www.purl.org/net/repository/.**

[WCC99] Web Cache Coordination Protocol, June 1999. **http://www.cisco.com/univercd/cc/td/doc/product/software/ios111/ca111/wccp.htm.**

[Weba] Welcome to WebDAV Resources. **http://www.webdav.org/.**

[Webb] DAV Frequently Asked Questions. **http://www.webdav.org/other/faq.html.**

[Webc] Web History. **http://www.webhistory.org.**

[WG-99] HTTP-WG Mailing list archives, 1994-1999. **http://www.ics.uci.edu/pub/ietf/http/hypermail/.**

[WM99] Craig E. Wills and Mikhail Mikhailkov. Towards a Better Understanding of Web Resources and Server Responses for Improved Caching. In *Proc. World Wide Web Conference*, May 1999. **http://www.cs.wpi.edu/~cew/papers/www8.ps.gz.**

[WMB99] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes.* Morgan Kaufmann, second edition, May 1999. ISBN 1-55860-570-3.

[WTSW97] Walter Willinger, Murad S. Taqqu, Robert Sherman, and Daniel W. Wilson. Self-Similarity Through High Variability: Statistical Analysis of Ethernet LAN Traffic at Source Level. *IEEE/ACM Transactions on Networking*, 5(1), February 1997. **http://www.acm.org/pubs/articles/journals/ton/ 1997-5-1/p71-willinger/ p71-willinger.pdf.**

[You94] N. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525-541, June 1994.

[Zip49] G. K. Zipf. *Human Behavior and the Principle of Least-Effort.* Addison-Wesley, 1949.

Предметный указатель

A

B, C

D

E, F

G

H

I

J, K, L

M

N

O, P

Q, R

S

T

U

V, W, Y

А

Б

В

Г, Д

Ж, З

И

К

Л, М

Н

О

П

P

C

Т

У

Ф–Щ

Э, Я