**Транспортные службы, предоставляемые Интернетом**

До этого момента мы рассматривали транспортные службы, кото­рые *могли* бы обеспечить компьютерные сети вообще. Теперь давайте обратимся к основным типам транспортных служб, предоставляемых Интернетом. Интернет (и TCP/IP сети вообще) предоставляет прило­жениям два транспортных протокола — UDP и TCP. Каждый из этих протоколов предлагает разнообразный набор служб исполняющимся приложениям. На рис. 2.4 представлены требования к этим службам для некоторых выбранных приложений.

**Службы протокола TCP**

Модель обслуживания протокола TCP включает службу установления логического соединения, а также службу надежной передачи данных. Когда приложение использует протокол TCP, то обе эти службы предоставляются приложению.

Прикладной уровень

|  |  |  |  |
| --- | --- | --- | --- |
| **Приложение** | **Потери данных** | **Пропускная способность** | **Чувствительность к потере данных** |
| **Передача файлов/ загрузка** | Не допускаются | Эластичное | Нет |
| **Электронная почта** | Не допускаются | Эластичное | Нет |
| **Веб-документы** | Не допускаются | Эластичное (несколько Кбит/с) | Нет |
| **IР-телефония/ Видео-конфе­ренции** | Допускаются | Аудио: От нескольких Кбит/с до 1 Мбит/с  Видео: От 10 Кбит/с до 5 Мбит/с | Да, сотни миллисекунд |
| **Потоковый аудио и видеоконтент** | Допускаются | См. предыдущее | Да, несколько секунд |
| **Интерактивные игры** | Допускаются | От нескольких до 10 Кбит/с | Да, сотни миллисекунд |
| **Мгновенные сообщения** | Не допускаются | Эластичное | Да и нет |

**Рис.** 2.4. Требования к службам, предъявляемые определенными

сетевыми приложениями

• *Передача с установлением соединения.* Перед тем как начинают пере­даваться сообщения прикладного уровня, протокол TCP обеспечи­вает обмен управляющей информацией между клиентом и сервером на транспортном уровне. Эта так называемая процедура рукопожа­тия предупреждает клиентскую и серверную сторону, позволяя им подготовиться к началу обмена пакетами. После этапа рукопожатия говорят, что между сокетами клиентского и серверного процессов установлено **TCP-соединение.** Соединение является дуплексным — это означает, что оба процесса могут передавать сообщения друг другу в одно и то же время. Когда приложение завершит передачу сообщения, соединение должно быть разорвано.

**О БЕЗОПАСНОСТИ**

**Безопасность протокола TCP**

Ни один из протоколов TCP и UDP не предоставляет службу шифро­вания — данные, которые передающий процесс отправляет в свой сокет, идентичны данным, которые проходят через сеть к принима­ющему процессу. Так, например, если процесс-источник отправляет пароль открытым текстом (то есть незашифрованным) в свой сокет, то этот открытый текстовый пароль будет проходить все линии связи на пути между отправителем и получателем, в любой из них являясь потенциальной мишенью для перехвата. С некоторых пор сообра­жения конфиденциальности и безопасности информации при пере­даче по сетям стали более актуальными, и Интернет-сообщество разработало криптографический протокол **уровень защищенных сокетов** (Secure Sockets Layer, **SSL)** как доработку протокола TCP. Такой вариант TCP, расширенный протоколом SSL, предлагает не только традиционные службы TCP, но также обеспечивает безопас­ность передачи данных от процесса к процессу, включая шифрова­ние, контроль их целостности, а также конечную аутентификацию. Подчеркнем, что SSL — это не третий транспортный протокол в до­бавление к TCP и UDP, а лишь дополнение к TCP, с доработками, реа­лизованными на прикладном уровне. В частности, если приложению необходимо использовать службы протокола SSL, то нужно вклю­чить поддержку SSL (задействовать оптимизированные библиоте­ки и классы) в приложении, как на клиентской, так и на серверной стороне. Протокол SSL имеет свой собственный API сокета, похо­жий на API сокета протокола TCP. При использовании протокола SSL передающий процесс отправляет обычные текстовые данные в сокет; затем SSL на передающем хосте шифрует данные и передает их в сокет TCP. Зашифрованные данные отправляются по сети в сокет TCP к принимающему процессу. Сокет получателя передает зашиф­рованные данные в SSL, затем расшифровывает их. Наконец, SSL передает обычные текстовые данные через свой сокет в принимаю­щий процесс.

• *Надежная передача данных.* Взаимодействующие процессы могут полагаться на протокол TCP, который доставит все отправленные данные без ошибок и в строго определенном порядке. Когда одна сторона приложения отправляет поток байтов в сокет, она может быть уверена, что TCP доставит этот самый поток в принимающий сокет без потерь или дублирования.

Протокол TCP кроме всего прочего предоставляет механизм кон­троля перегрузки, который, правда, выгоден больше не взаимодейству­ющим процессам, а самой коммуникационной сети Интернет. Когда участок сети между отправителем и получателем перегружен, данный механизм заставляет передаваемый процесс (либо клиентский, либо серверный) снижать нагрузку на сеть.

**Службы протокола UDP**

UDP представляет собой простой протокол транспортного уровня, предлагающий минимальный набор служб. UDP является протоколом без установления соединения, то есть процедуры рукопожатия перед тем, как два процесса начинают взаимодействовать, не происходит. UDP обеспечивает ненадежную передачу данных — другими словами, когда процесс отправляет сообщение в сокет UDP, *нет* никакой гаран­тии, что сообщение будет получено принимающим процессом. Более того, сообщения могут поступать в принимающий процесс в произволь­ном порядке.

Механизма контроля перегрузок в протоколе UDP тоже нет. По­этому отправляющая сторона может передавать данные в нижележа­щий уровень (сетевой) с любой скоростью. (Заметим, что реальная сквозная пропускная способность может быть меньше этой скорости по причине ограничения скорости передачи либо перегрузки линии связи.)

**Службы, не предоставляемые транспортными протоколами Интернета**

Мы с вами отметили четыре критерия для служб транспортного про­токола: надежная передача данных, пропускная способность, время до­ставки и безопасность. Что из этого могут предложить протоколы TCP и UDP? Мы уже отметили, что TCP гарантирует надежную доставку данных. Мы также уже знаем, что добавление протокола SSL позволяет обеспечить безопасность данных при отправке через TCP. Но в нашем кратком описании протоколов TCP и UDP отсутствуют упоминания о гарантированном времени доставки и о пропускной способности — транспортные протоколы сегодняшнего Интернета это обеспечить *не* могут. Следует ли из этого, что приложения, чувствительные к скорости передачи, такие как IP-телефония, не могут работать в Интернете? Ко­нечно же, нет — такие приложения уже многие годы с успехом исполь­зуют Интернет, поскольку были они разработаны таким образом, чтобы максимально учесть отсутствие гарантий временной доставки данных. Тем не менее возможности «умного» дизайна приложений ограничены, например, когда задержки сети становятся очень большими либо пропускная способность каналов связи достаточно мала. Одним словом, можно сделать вывод, что сегодняшний Интернет, как правило, обеспечивает удовлетворительное обслуживание приложений, чувствительных ко времени доставки, но не может гарантировать их нормальную работу в исключительных условиях.

|  |  |  |
| --- | --- | --- |
| **Приложение** | **Протокол прикладного уровня** | **Базовый транспортный протокол** |
| Электронная почта | SMTP | TCP |
| Удаленный терминальный доступ | Telnet | **TCP** |
| Всемирная паутина | HTTP | TCP |
| Передача файлов | FTP | TCP |
| Потоковый мультимедий­ный контент | HTTP (например, YouTube) | TCP |
| IP-телефония | SIP, RTP или проприетарное (например, Skype) | **UDP** или TCP |

Рис. 2.5. Популярные Интернет-приложения и используемые ими протоколы прикладного и транспортного уровней

На рис. 2.5 показаны транспортные протоколы, используемые некоторыми популярными Интернет-приложениями. Мы видим, что электронная почта, удаленный терминальный доступ, Всемирная пау­тина и приложения передачи файлов — все они используют протокол TCP. Это связано с тем, что он обеспечивает надежную передачу дан­ных, гарантируя, что все данные в конечном итоге доберутся до места назначения. Поскольку приложения IP-телефонии (например, Skype) допускают некоторые потери данных, но требуют для эффективной сво­ей работы определенное значение минимальной скорости передачи, раз­работчики этих приложений обычно предпочитают запускать их через протокол UDP, избегая тем самым механизма управления перегрузкой и дополнительных затрат на гарантированную доставку пакетов, при­сущих протоколу TCP Но так как многие брандмауэры (большинство их типов) настроены на блокирование UDP-трафика, то приложения IP-телефонии очень часто разрабатываются с возможностью использо­вания протокола TCP в качестве резервного для случаев, когда соеди­ниться по протоколу UDP не удается.

**Протоколы прикладного уровня**

Мы только что узнали, что сетевые процессы взаимодействуют друг с другом, отправляя сообщения в сокеты. Но как построены эти сообще­ния? Что означают различные поля в сообщениях? Когда именно процессы отправляют эти сообщения? Все эти вопросы ведут нас в область протоколов прикладного уровня, которые определяют, как прикладные процессы, запущенные на различных конечных системах, передают друг другу сообщения. В частности, протокол прикладного уровня определяет:

* Типы сообщений, которыми обмениваются процессы, например, сообщения запроса или сообщения ответа
* Синтаксис различных типов сообщений, то есть состав полей сооб­щения и их порядок
* Семантика полей, то есть значение информации, содержащейся в каждом из них
* Набор правил для определения времени и порядка передачи сообще­ний и получения ответов

Некоторые из протоколов прикладного уровня описаны в докумен­тах RFC, поэтому являются общедоступными. Например, вы можете найти в RFC описание протокола прикладного уровня HTTP (HyperText Transfer Protocol, протокол передачи гипертекста). Если разработчик, браузера, например, следует правилам документов RFC относительно протокола HTTP, то, соответственно, браузер способен загружать веб­страницы с любого веб-сервера, который также соответствует правилам RFC для HTTP. Многие протоколы являются проприетарными (част­ными) и поэтому недоступны для общего пользования. Например Skype использует проприетарный протокол прикладного уровня.

Очень важно понимать различие между сетевыми приложениями и протоколами прикладного уровня. Протокол прикладного уровня — это только одна из частей сетевого приложения (хотя и очень важная часть!). Рассмотрим пару примеров. Всемирная паутина — это клиент-серверное приложение, позволяющее пользователям получать документы с веб-сервера по запросу. Веб-приложение состоит из множества компонентов, включающих стандарты для форматов документа (то есть, HTML), веб-браузеров (например, Firefox или Microsoft Internet Explorer), веб-серверов (например, Apache и Microsoft-серверов) и протокола прикладного уровня. Протокол прикладного уровня для Все­мирной паутины — HTTP — определяет формат и последовательность сообщений, которыми обмениваются браузер и веб-сервер. Таким образом, HTTP — это только одна часть (хотя и важная) веб-приложения. То же самое можно сказать про приложение электронной почты, кото­рое также включает множество компонентов: почтовые серверы, содержащие почтовые ящики пользователей; почтовые клиенты (такие, как Microsoft Outlook), которые позволяют пользователям читать и создавать сообщения; набор стандартов для определения структуры сообщения электронной почты; наконец, протоколы прикладного уровня, определяющие порядок прохождения сообщений между серверами и почтовыми клиентами и порядок интерпретации для содержимого заголовков этих сообщений. Основной протокол прикладного уровня для электронной почты — это SMTP (Simple Mail Transfer Protocol). Таким образом, SMTP, являющийся основным протоколом электронной почты, также является хотя и важным, но одним из приложений электронной почты.

**Всемирная паутина и HTTP**

До начала 90-х годов Интернет преимущественно использовался учеными, исследователями и студентами университетов для обеспечения удаленного доступа, передачи файлов с локальных хостов на удаленные и обратно, получения и отправки новостей, а также для работы с электронной почтой. Несмотря на то, что эти приложения были весьма полезны (какими и остаются по сей день), об Интернете мало кто знал за рамками исследовательских и ученых сообществ. В начале 1990-х на сцене появилось главное приложение Интернета — это Всемирная паутина (веб). Всемирная паутина стала первым Интернет-приложением, на которое обратила внимание общественность. Оно серьезно повлияло и продолжает влиять на то, как люди взаимодействуют между собой внутри своего рабочего окружения и за его пределами, и позволило поднять Интернет на новый уровень — в результате он стал, в сущности, единой крупной сетью данных.

Одна из возможных причин, почему Всемирная паутина привлекла пользователей — это то, что принцип ее функционирования — работа *по запросу.* Пользователи получают то, что они хотят и когда им это требуется, в отличие от традиционных радио- и телевещания, которые предоставляют информацию тогда, когда поставщик делает ее доступной для аудитории. Вдобавок к этому Всемирная паутина привлекательна для пользователей многими другими своими свойствами. На сегодняшний день сделать информацию доступной по сети — невероятно легкая вещь, доступная каждому: каждый пользователь может стать издателем, не применяя особых усилий. Гиперссылки и поисковые системы помогают нам не потеряться в океане веб-сайтов. Веб-трафика, формы, Java-апплеты и другие «навороты» позволяют нам взаимодействовать с веб-страницами и сайтами, а веб-серверы используются в качестве платформы для построения многих популярнейших приложений, включая YouTube, Gmail и Facebook.

**Обзор протокола HTTP**

**HTTP (протокол передачи гипертекста)** — это протокол прикладного уровня для Всемирной паутины, составляющий ее самое сердце. Он определен в документах RFC 1945 и RFC 261б. HTTP реализуется в двух частях приложений: клиентской и серверной. Клиентская и серверная части программ, исполняемые на различных конечных системах, общаются друг с другом, обмениваясь сообщениями HTTP. Протокол HTTP определяет структуру этих сообщений и порядок обмена между клиентом и сервером. Перед тем как детально объяснить HTTP, сделаем краткий обзор терминологии Всемирной паутины.

**Веб-страница** (также называемая документом) состоит из объектов. **Объект** — это простой файл, имеющий уникальный URL-адрес, например файл формата HTML, изображение в формате JPEG, Java-апплет или видеоклип. Большинство веб-страниц состоит из базового HTML-файла, который содержит ссылки на несколько объектов. Например, если веб-страница содержит HTML-текст и пять рисунков в формате JPEG, это значит, что веб-страница содержит шесть объектов: базовый файл плюс пять изображений. Базовый файл содержит в себе ссылки на другие объекты, в виде их URL-адресов, находящихся в тексте базового файла. Каждый URL-адрес состоит из двух частей: имени сервера, содержащего объект и пути до этого объекта. Например, URL-адрес

**http://www.someSchool.edu/someDepartment/picture.gif**

содержит адрес хоста [**www.someSchool.edu**](http://www.someSchool.edu)и имя пути **/someDepartment/picture**.**gif.** Мы будем использовать слова *браузер* и *клиент* как взаимозаменяемые, так как веб-браузеры (такие как Internet Explorer или Firefox) реализуют клиентскую сторону протокола HTTP. Веб-серверы, реализующие серверную сторону HTTP, содержат веб-объекты, на которые указывают ссылки. Популярными сегодня веб-серверами являются Apache и Microsoft Internet Information Server.

Протокол HTTP определяет порядок того, как веб-клиенты запрашивают веб-страницы с веб-сервера и как сервер передает эти страницы клиентам. Подробнее взаимодействие между клиентом и сервером мы рассмотрим позже, но основная идея продемонстрирована на рис. 2.6. Когда пользователь запрашивает веб-страницу (например, щелкает по гиперссылке), браузер отправляет HTTP-запрос объектов этой страницы серверу. Сервер получает запрос и отвечает сообщением HTTP, которое содержит объекты.

HTTP использует TCP в качестве базового транспортного протокола. Сначала HTTP-клиент инициирует TCP-соединение с сервером. Когда соединение установлено, процессы браузера и сервера получают доступ к TCP через свои сокеты. На клиентской стороне, как мы уже знаем, сокет — это дверь между клиентским процессом и ТСР-соединением; со стороны сервера — между серверным процессом и ТСР-соединением. Клиент отправляет HTTP-запрос, получая HTTP-ответ через свой сокет. Аналогично HTTP-сервер принимает запрос и отсылает ответное сообщение, используя свой сокет. Как только клиент отправил сообщение в сокет, оно попадает в руки протокола TCP. Нам известно, что TCP обеспечивает службу надежной доставки. Это означает, что каждый HTTP-запрос, посланный клиентским процессом, обязательно прибудет к серверу, и наоборот, каждый HTTP-ответ, посланный серверным процессом, обязательно будет получен клиентом. Тут мы видим основное преимущество иерархической (многоуровневой) архитектуры — протоколу HTTP не нужно заботиться о потерянных данных или о том, как восстанавливаются эти данные, переданные по сети. Это все работа протокола TCP и протоколов нижележащих уровней стека.

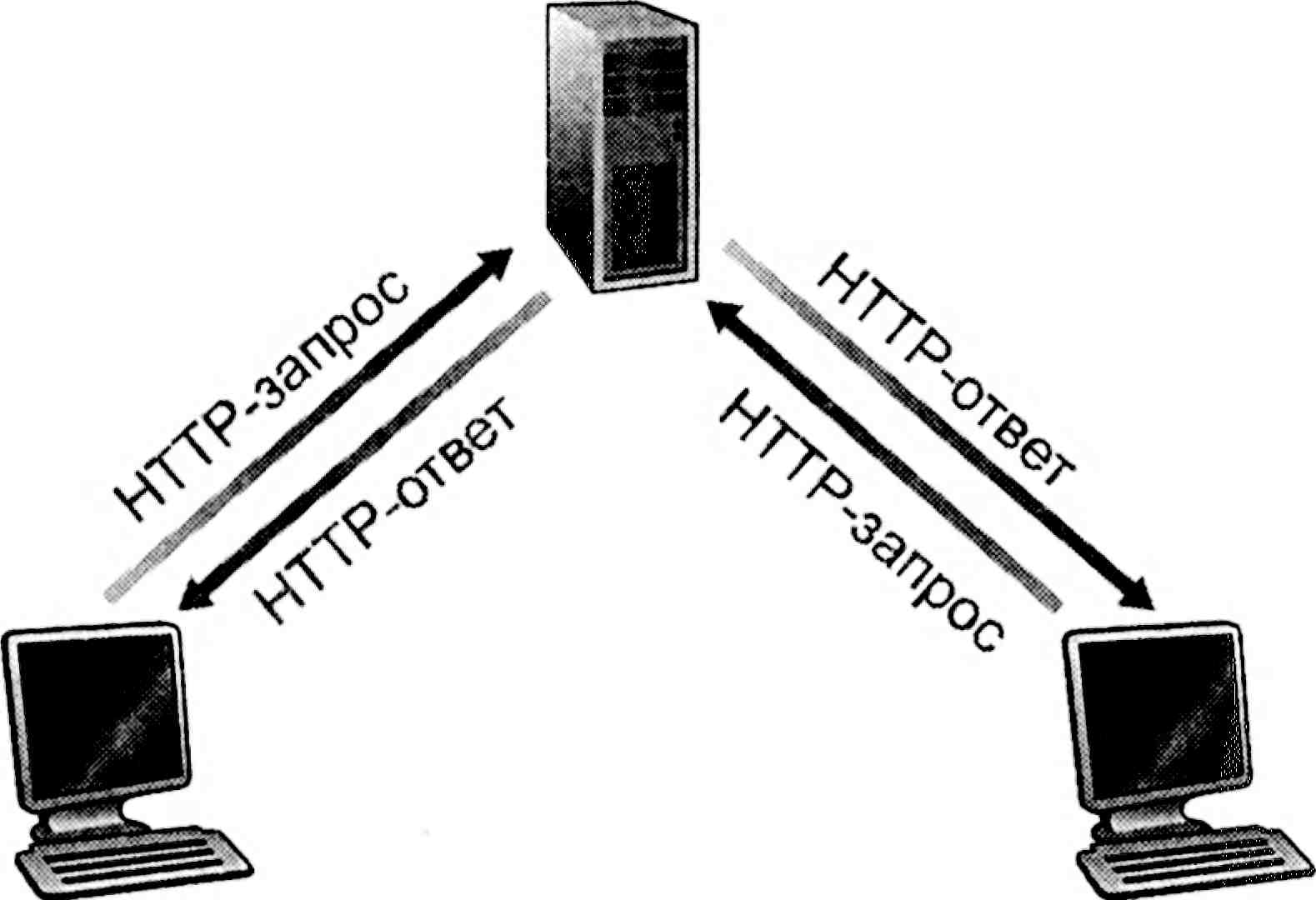


Рис. 2.6. Процедура запросов и ответов HTTP

Отметим важный момент, что сервер отправляет запрошенные файлы клиенту без сохранения какой-либо информации о нем. Если конкретный клиент запрашивает один и тот же объект дважды за определенный промежуток времени, сервер не сообщает, что такой объект только что был запрошен. Вместо этого он пересылает объект, как будто бы он уже забыл, что делал это только что. Так как HTTP-сервер не обрабатывает информацию о клиенте, говорят, что HTTP — это **протокол без сохранения состояния.** Отметим также, что Всемирная паутина использует клиент-серверную архитектуру. Веб-сервер всегда находится в режиме онлайн, имеет фиксированный IP-адрес и обслуживает запросы от потенциальных миллионов различных браузеров.

**Непостоянные и постоянные соединения**

В большинстве Интернет-приложений клиент и сервер взаимодействуют друг с другом продолжительный период времени, в течение которого клиент делает серию запросов, а сервер отвечает на каждый из них. Запросы в серии, в зависимости от типа приложения и от того, как оно используется, могут происходить один за другим или с некоторыми интервалами, как периодическими, так и произвольными. Когда взаимодействие клиента с сервером происходит через TCP, разработчик приложения должен принять важное решение — отправлять каждую пару запрос-ответ через *отдельное* либо через *одно и то же* TCP-соединение? В первом случае говорят, что используются **непостоянные соединения;** во втором — **постоянные соединения.** Чтобы это лучше понять, давайте рассмотрим преимущества и недостатки постоянных соединений в контексте определенного приложения, а именно с применением протокола HTTP, который способен работать как с разными типами соединения. Несмотря на то, что HTTP использует по умолчанию постоянное соединение, HTTP-клиенты и серверы могут быть сконфигурированы таким образом, чтобы переключить свою работу в режим непостоянных соединений.

**Непостоянные HTTP-соединения**

Давайте рассмотрим этапы передачи веб-страницы от сервера к клиенту в случае непостоянного соединения. Предположим, что страница состоит из базового HTML-файла и десяти изображений в формате JPEG и все эти 11 объектов находятся на одном сервере. Предположим, что URL-адрес для базового HTML-файла имеет вид

[**http://www.someSchool.edu/someDepartment/home.index**](http://www.someSchool.edu/someDepartment/home.index)

Вот что происходит:

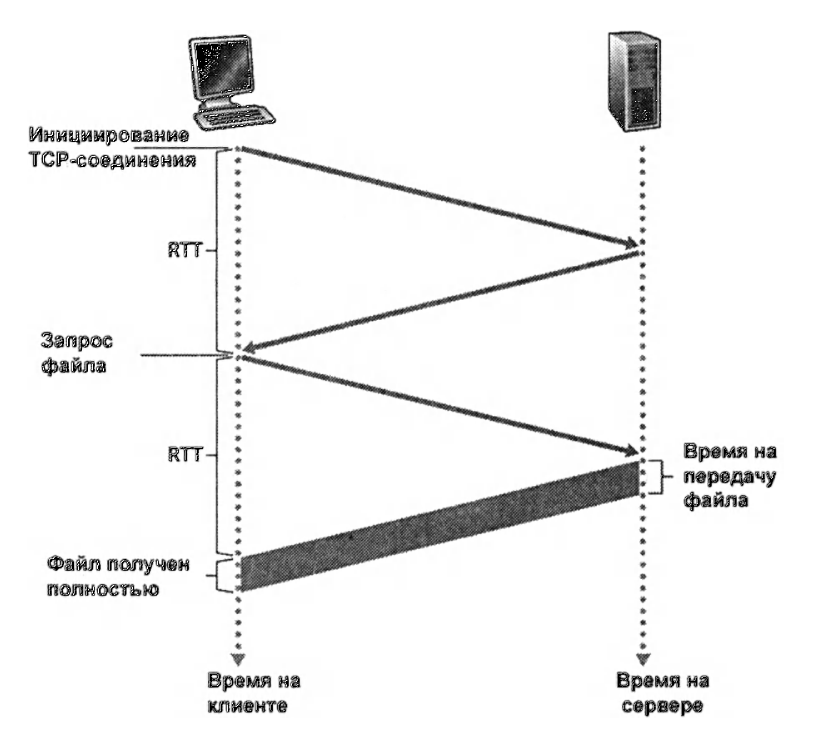
1. HTTP-клиент инициирует TCP-соединение с сервером www. someSchool.eduпо порту 80, являющемуся портом по умолчанию для протокола HTTP. Этому TCP-соединению выделяются сокеты на клиентской и северной стороне.
2. HTTP-клиент отправляет запрос серверу через свой сокет. Запрос включает путь к базовому файлу /someDepartment/home.index.(HTTP-сообщения более детально рассмотрим ниже)
3. Процесс HTTP-сервера получает запрос через свой сокет, извлекает объект /someDepartment/home.index из своего места хранения (оперативной памяти или диска), помещает объект в ответное HTTP-сообщение и отправляет клиенту через свой сокет.
4. Процесс HTTP-сервера дает команду протоколу TCP закрыть соединение (на самом деле, TCP-соединение не разрывается до тех пор, пока сервер не получит информацию об успешном получении ответа клиентом).
5. HTTP-клиент получает ответ от сервера, и TCP-соединение разрывается. Сообщение указывает, что полученный объект — это HTML-файл. Клиент извлекает файл из сообщения, обрабатывает его и находит ссылки на 10 объектов (файлов в формате JPEG).
6. Шаги с первого по четвертый повторяются для каждого из десяти JPEG-объектов.

Когда браузер получает веб-страницу, он отображает ее на экране. Два различных браузера, кстати, могут интерпретировать веб-страницу по-разному. Спецификации протокола HTTP (RFC 1945 и RFC 261 б) определяют только протокол взаимодействия между программой клиента и программой сервера, но ничего не говорят о том, как веб-страница должна интерпретироваться клиентом.

Описанные выше шаги демонстрируют использование непостоянных соединений HTTP, при которых TCP-соединение закрывается после того, как сервер получает объект. Таким образом, каждое ТСР-соединение передает ровно одно сообщение-запрос и одно сообщение – ответ. В нашем примере, когда пользователь запрашивает веб-страницу, устанавливаются 11 ТСР-соединений.

Здесь мы намеренно не уточняем, получил ли клиент эти 10 JPEG-файлов через 10 последовательно создаваемых ТСР-соединений, либо некоторые приняты через параллельное TCP-соединение. На самом деле большинство современных браузеров в режиме по умолчанию открывают от пяти до десяти параллельных ТСР-соединений, и каждое из них обрабатывает одну транзакцию из запроса и ответа, а степень этого параллелизма может быть сконфигурирована пользователем. Если тот пожелает, число параллельных соединений можно установить равным единице, и в этом случае 10 соединений будут устанавливаться последовательно. Использование параллельных соединений сокращает время ответа сервера.

Перед тем как продолжить, давайте произведем некоторые вычисления для оценки интервала времени, проходящего с отправки клиентом запроса базового HTML-файла до момента, когда весь файл получен клиентом. Для этого определим время оборота (round-trip time, RTT), которое есть не что иное, как время, требуемое пакету малого размера для передачи от клиента к серверу и обратно (еще его называют време­нем двусторонней задержки). Время оборота включает в себя задержку распространения, задержку ожидания в промежуточных маршрутизаторах или коммутаторах и задержку на обработку пакета. Рассмотрим, что происходит, когда пользователь нажимает на гиперссылку. Как показано на рис. 2.7, это приводит к тому, что браузер инициирует TCP-соединение с веб-сервером; оно включает «тройное рукопожатие» — клиент отправляет серверу небольшой TCP-сегмент, сервер подтверждает и отправляет в ответ также небольшой TCP-сегмент, и, наконец, клиент еще раз отправляет сегмент с подтверждением серверу.

****

**Рис.** 2.7. Расчет времени, необходимого на запрос и получение

HTML-файла

Первые две части тройного рукопожатия занимают одно время оборота или 1 RTT. После этого вместе с третьей частью клиент отправляет HTTP-запрос в TCP-соединение. Как только запрос прибывает на сервер, тот отправляет в ответ запрошенный HTML-файл. Данная пара запрос — ответ занимает еще одно время оборота (1 RTT). Таким образом, общее время ответа приблизительно равно удвоенному времени оборота плюс время на передачу HTML-файла.

**Постоянное HTTP-соединение**

Непостоянные соединения имеют некоторые недостатки. Во-первых, для *каждого запрашиваемого объекта* должно устанавливаться и обслуживаться новое соединение.

Для каждого из этих соединений протокол TCP должен выделить буфер, а также сохранить несколько переменных как на клиенте, так и на сервере. Это бывает затруднительно для веб-сервера, учитывая, что он может обслуживать сотни различных клиентов одновременно. Во-вторых, как мы уже сказали, передача каждого объекта вызывает задержку, равную двум значениям времени оборота: 1 RTT для установления TCP-соединения и еще 1 RTT — для отправки запроса и получения ответа, а это все дополнительное время задержки.

В случае с постоянным соединением сервер после отправки ответа клиенту оставляет ТСР-соединение открытым. Через одно и то же соединение можно отправить последовательность запросов и ответов между одним и тем же клиентом и сервером. В частности, одно постоянное ТСР-соединение позволяет передать всю веб-страницу (в примере выше это базовый HTML-файл и десять изображений).

Более того, через одно постоянное соединение можно отправить одному и тому же клиенту много веб-страниц, размещенных на том же сервере. Эти запросы объектов могут быть сделаны один за другим, без ожидания ответов на обрабатываемый запрос (так называемая *конвейеризация).* Обычно HTTP-сервер закрывает соединение, когда оно не используется в течение определенного времени (настраиваемый интервал тайм-аута). Когда сервер получает последовательные запросы, он отправляет объекты также один за другим. По умолчанию HTTP использует постоянное соединение с конвейеризацией.

**Формат HTTP-сообщения**

В документах по спецификации протокола HTTP (RFC 1945и RFC 261 б) содержится описание форматов HTTP-сообщений. В протоколе HTTP существуют два типа сообщений: сообщение-запрос и сообщение-ответ. Их мы обсудим ниже.

**Сообщение-запрос протокола HTTP**

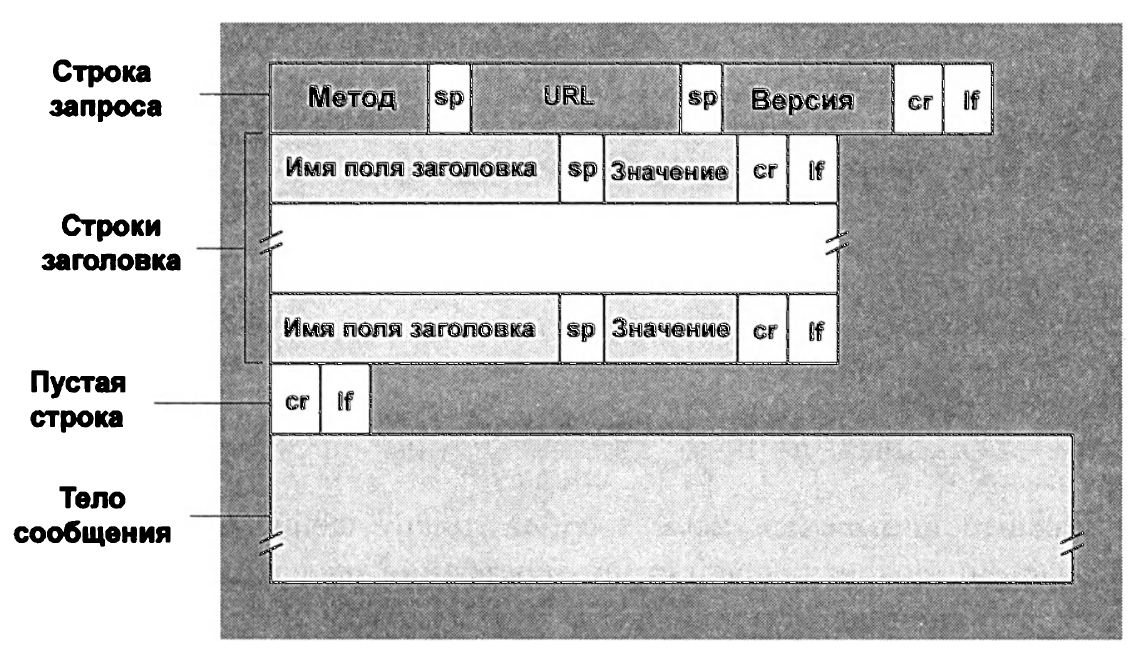
Ниже представлено типичное сообщение-запрос протокола HTTP:

GET /somedir/page.html HTTP/1.1  
Host: [www.someschool.edu](http://www.someschool.edu)Connection: close   
User-agent: Mozilla/5.0   
Accept-language: fr

Если внимательно посмотреть на это сообщение, можно многое из него почерпнуть. Во-первых, мы видим, что оно представлено в обычном текстовом формате ASCII, так что любой человек может его прочитать. Во-вторых, мы видим, что сообщение состоит из пяти строк, каждая из которых заканчивается символами возврата каретки и перевода строки, а последняя — двумя парами этих символов. Сообщение-запрос в общем случае может содержать количество строк от одной и более, но необязательно пять, как в нашем примере. Первая строка HTTP-сообщения называется **строкой запроса;** следующие строки называются **строками заголовка.** В строке запроса содержатся три поля: поле метода, поле URL и поле версии протокола HTTP. Поле метода может принимать несколько различных значений, включая GET, POST, HEAD, PUT и DELETE. Большинство сообщений-запросов протокола HTTP используют метод GET. Он применяется, когда браузер запрашивает объект, идентифицирующийся полем URL. В нашем случае браузер запрашивает объект /somedir/page.html. Поле версии протокола говорит само за себя; в нашем примере версия HTTP 1.1.

Теперь давайте посмотрим на строки заголовка в нашем примере. Строка заголовка Host: www.someschool.edu указывает адрес хоста, на котором размещается объект. Вы, возможно, подумаете, что данная строка не является необходимой, так как соединение с данным хостом по этому адресу уже установлено, но, как мы увидим далее, что эта информация требуется для кэширующего прокси-сервера. Включение строки заголовка Connection: close означает: браузер сообщает серверу, что он не собирается работать с постоянным соединением, и его нужно разорвать после отправки запрашиваемого объекта. Строка User-agent: указывает на агент пользователя, то есть, на тип браузера, который совершает запрос к серверу. В нашем случае агент пользователя Mozilla/5.0 — это браузер Firefox. Данная информация очень полезна, так как сервер на самом деле может отсылать различным типам браузеров разные версии одного и того же объекта (причем обе версии задаются одним URL-адресом). Наконец, строка заголовка Accept-language: fr указывает, что пользователь предпочитает получить версию объекта на французском языке, если, конечно, такая версия существует на сервере; в противном случае сервер будет отсылать версию по умолчанию. Заголовок Accept-language: является одним из многих заголовков согласования, доступных в протоколе HTTP.

После нашего примера давайте рассмотрим общий формат сообщения-запроса, представленный на рис. 2.8. Как мы видим, общий формат соответствует нашему примеру выше. Однако вы можете заметить, что после строк заголовка (при дополнительных символах возврата каретки и перевода строки) присутствует «тело сообщения». Данное тело сообщения является пустым при использовании метода GET и содержит информацию при использовании метода POST. Чаще всего метод POST применяется, когда пользователь заполняет формы, например, вводит слова поиска в поисковой системе. Используя сообщение POST, пользователь запрашивает веб-страницу сервера, но ее содержимое зависит от того, что пользователь ввел в поля формы. Если значением поля метода является POST, то тело сообщения содержит информацию, которую пользователь ввел в поля формы.

****

**Рис. 2.8.** Общий формат сообщения-запроса HTTP

Заметим, что запросы, генерируемые с помощью форм, необязательно используют метод POST. Очень часто в HTML-формах применяется и метод GET, а данные, введенные в поля формы, подставляются в URL-адрес запрашиваемой страницы. Например, если форма использует метод GET и содержит два поля, в которые вводятся значения monkeys и bananas, то URL-адрес примет следующий вид: **www.somesite.com/animalsearch?monkeys&bananas.** В ваших ежедневных путешествиях во Всемирной паутине вы, очевидно, встречались с URL-адресами такого типа.

Метод HEAD аналогичен методу GET. Когда сервер получает запрос с помощью метода HEAD, он отправляет ответное HTTP-сообщение, но не пересылает в нем запрашиваемый объект. Разработчики приложений обычно используют этот метод для целей отладки. Метод PUT часто применяется совместно с инструментами веб-публикаций. Он позволяет пользователю загружать объект по указанному адресу на конкретный веб-сервер. Он также используется приложениями, которые требуют загрузки объектов на веб-серверы. Метод DELETE позволяет пользователю либо приложению удалять объект на веб-сервере.

**Сообщение-ответ протокола HTTP**

Ниже мы приводим типичное ответное сообщение, используемое протоколом HTTP. Это может быть, например, ответом на сообщение-запрос, которое мы изучили в предыдущем разделе.

НТТР/1.1 200 ОК

Connection: close

Date: Tue, 09 Aug 2011 15:44:04 GMT

Server: Apache/2.2.3 (CentOS)

Last-Modified: Tue, 09 Aug 2011 15:11:03 GMT

Content-Length: 6821

Content-Type: text/html

(данные данные данные данные данные...)

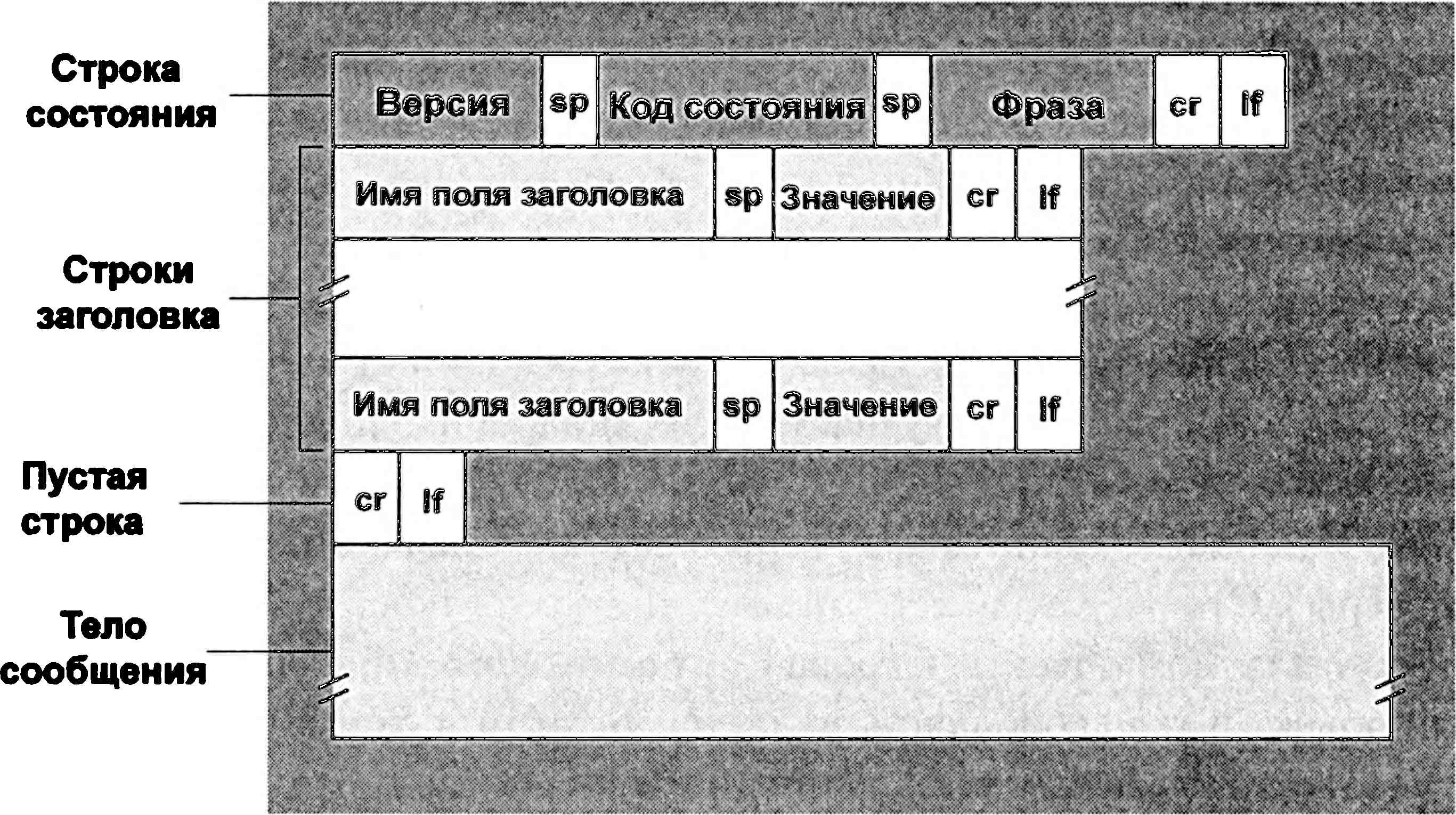
Давайте внимательно посмотрим на это сообщение. Оно состоит из трех частей: первоначальной **строки состояния, шести строк заголовка** и **тела сообщения.** Тело сообщения — это ключевая его часть, она содержит сам запрашиваемый объект (представлен здесь строкой данные данные данные данные данные. . .). Строка состояния содержит три поля: поле версии протокола, код состояния и соответствующее сообщение состояния (фраза состояния). В нашем примере строка состояния указывает, что сервер использует версию HTTP/1.1 и что все ОК (то есть сервер нашел запрашиваемый объект и начал его передачу).

Теперь посмотрим на строки заголовка. Сервер использует строку Connection: close, чтобы сообщить клиенту, что он собирается закрыть TCP-соединение после отправки сообщения. Строка Date: указывает время и дату создания ответа сервером. Обратите внимание, что это не время создания или изменения объекта, а время, когда сервер извлекает объект из своей файловой системы, вставляет его в ответное сообщение и отправляет клиенту. Строка заголовка Server: означает, что сообщение было генерировано веб-сервером Apache; она аналогична строке заголовка User-agent: сообщения-запроса. Время последнего изменения объекта указывает строка состояния Last-Modified:, которая, как мы вскоре узнаем более подробно, является наиболее критичной для кэширования объекта, как для локальных клиентов, так и в сетевых кэширующих серверах (также известных как прокси-серверы). Строка заголовка Content-Length: показывает число байт в пересылаемом объекте. Строка Content-Type: информирует о том, что объект в теле сообщения является текстом в формате HTML. (На тип объекта указывает не расширение файла, а именно строка заголовка Content-Type:.)

Теперь давайте посмотрим на показанный на рис. 2.9 общий формат сообщения-ответа, частным случаем которого является рассмотренный выше пример. Добавим несколько слов относительно кодов состояния и связанных с ними фраз, которые указывают на результат запроса. Наиболее используемые коды состояния включают:

* 200 ОК: запрос выполнен успешно, и информация возвращена в от­ветном сообщении.
* 301 Moved Permanently: запрошенный объект перемещен; новый URL-адрес указывается в заголовке ответного сообщения, содержа­щем строку Location:. Программное обеспечение клиента будет автоматически перенаправляться на этот новый URL-адрес.
* 400 Bad Request: это общепринятый код ошибки, указывающий, что запрос не может быть распознан сервером.
* 404 Not Found: запрошенный документ на сервере не существует.
* 505 HTTP Version Not Supported: запрошенная версия HTTP-протокола не поддерживается сервером.

Хотите посмотреть на реальное сообщение-ответ протокола HTTP? Очень рекомендуем, и это достаточно просто! Сначала нужно соединиться с помощью команды Telnet с вашим любимым веб-сервером, потом набрать однострочное сообщение с запросом какого-нибудь объекта, который размещен на этом сервере.



**Рис. 2.9. Общий формат сообщения-ответа протокола HTTP**

Например, наберите в командной строке следующее:

telnet [cis.poly.edu](http://cis.poly.edu) 80 GET /-ross/ HTTP/1.1 Host: cis.poly.edu

(Дважды нажмите клавишу **Enter** после того, как наберете последнюю строку.) В результате откроется TCP-соединение с хостом cis.poly.edu по порту 80 и ему отправится сообщение-запрос. Вы должны увидеть ответное сообщение, которое включает базовый файл формата HTML, расположенный на домашней странице профессора Росса. Если вы хотите посмотреть просто строки сообщения HTTP без получения самого объекта, замените GET на HEAD. Затем попробуйте заменить значение /~ross/ на /-banana/ и посмотрите, какое сообщение вы получите в результате.

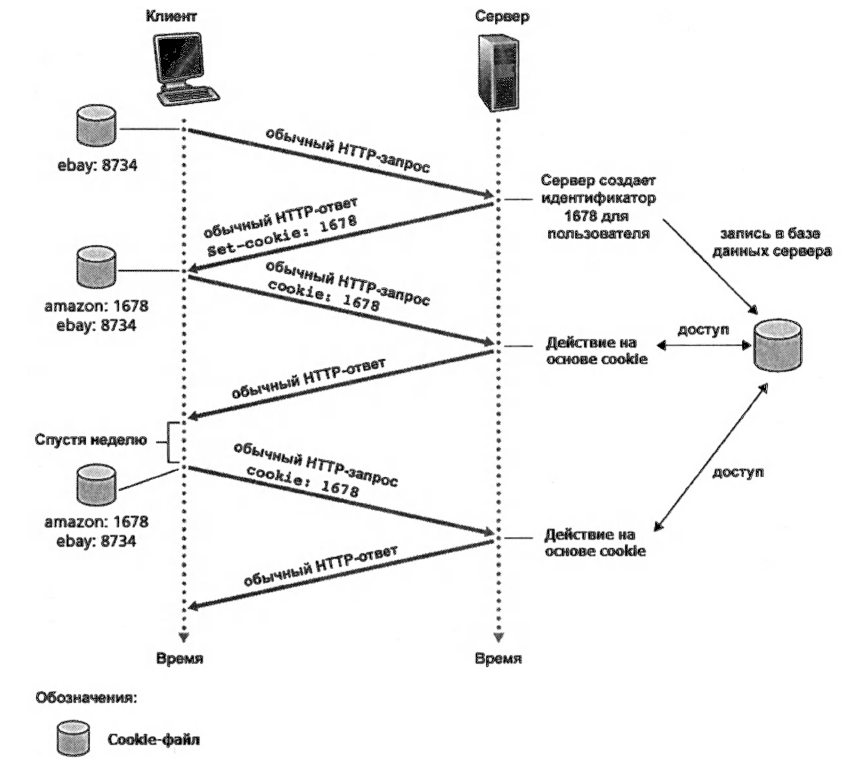
В данном разделе мы обсудили с вами несколько строк заголовка, которые могут применяться внутри запросов и ответов протокола HTTP. На самом деле в спецификации протокола определено очень много строк заголовков, используемых веб-серверами, браузерами и кэширующими серверами. Мы рассмотрели только небольшое количество из этой огромной массы. Еще несколько из них мы изучим, когда будем обсуждать кэширование.

Каким же образом решается, какие строки заголовка включать браузеру в сообщение-запрос, а веб-серверу — в ответное сообщение? На самом деле, строки заголовка, которые будет генерировать браузер, зависят от нескольких факторов: от типа браузера и его версии (например, браузер версии HTTP/1.0 не будет генерировать ни одной строки версии 1.1), от пользовательской конфигурации (например, используемого языка), а также от версии объекта — кэшированной, но, возможно, устаревшей, или последней, актуальной версии. Поведение веб-серверов аналогично: существуют различные продукты, версии и конфигурации, многообразие которых оказывает влияние на то, какие строки заголовка будут включаться в сообщение-ответ сервера.

**Взаимодействие пользователя и сервера: cookie-файлы**

Мы упомянули выше, что HTTP-сервер не сохраняет состояние соединения. Это упрощает разработку серверного программного обеспечения и позволяет создавать высокопроизводительные веб-серверы, способные обрабатывать тысячи одновременных TCP-соединений. Однако очень часто возможность идентифицировать пользователей очень даже желательна либо по причине того, что серверу нужно ограничить пользовательские права доступа, либо чтобы предоставлять набор услуг в зависимости от идентификации пользователя. Для этих целей в протоколе HTTP используются объекты *cookie («Куки»).* Механизм cookie, определенный в документе RFC 6265, позволяет веб-сайтам отслеживать состояние пользовательского соединения. В наши дни подавляющее большинство коммерческих веб-сайтов используют данный механизм.

Как мы видим на рис. 2.10, технология cookie включает четыре основных компонента: (1) строка заголовка в ответном HTTP-сообщении сервера; (2) строка заголовка в HTTP-запросе клиента; (3) cookie-файл, хранящийся на конечной системе пользователя и управляемый его браузером; (4) база данных на стороне веб-сервера. Давайте, используя рис. 2.10, посмотрим, как работает этот механизм. Предположим, Сьюзен, которая выходит в Интернет со своего домашнего компьютеpa, используя браузер Internet Explorer, в первый раз посещает сервер [**Amazon.com**](http://Amazon.com)**.** Предположим также, что она уже посещала ранее сайт еВау. Когда запрос приходит на веб-сервер Amazon, он создает уникальный идентификационный номер, а также запись в своей базе данных, которая индексируется этим идентификационным номером. Затем веб-сервер Amazon посылает ответ браузеру Сьюзен, включающий в HTTP-сообщение заголовок Set-cookie:, и в нем содержится соответствующий идентификационный номер. Например, строка заголовка может иметь вид: Set-cookie: 1678

****

**Рис. 2.10. Использование механизма cookie для сохранения состояния**

**пользовательского сеанса**

Когда браузер Сьюзен получает ответное HTTP-сообщение, он видит заголовок Set-cookie: и добавляет строку в специальный cookie-файл, которым управляет. Данная строка включает имя сервера и идентификационный номер из заголовка Set-cookie:. Заметим, что cookie-файл уже содержит запись для веб-сайта еВау, так как Сьюзен в прошлом уже посещала данный сайт. По мере того, как Сьюзен продолжает работать с сайтом сервера Amazon, каждый раз, когда она запрашивает веб-страницу, ее браузер обращается к своему cookie-файлу, извлекает идентификационный номер этого сайта и помещает строку cookie-заголовка, включающую идентификационный номер, в HTTP-запрос. В каждом таком HTTP-запросе к серверу Amazon содержится строка заголовка:

Cookie: 1678

С помощью такого инструмента сервер способен отслеживать активность Сьюзен на сайте Amazon. Хотя веб-сайту и не нужно знать имя Сьюзен, ему точно известно, какие страницы посетил пользователь 1678, в каком порядке и сколько раз! Amazon использует механизм cookie, чтобы организовать, например, карту покупок — он хранит информацию о товарах, выбранных Сьюзен в течение пользовательского сеанса браузера, и она может оплатить их все вместе в самом конце сеанса.

Если Сьюзен возвратится на сайт Amazon, скажем, через неделю, то ее браузер будет продолжать добавлять строку заголовка Cookie : 1678 в сообщение-запрос. Веб-сайт Amazon станет рекомендовать Сьюзен товары, основываясь на тех веб-страницах, которые она посещала в прошлом. Если вдобавок Сьюзен зарегистрируется — передав имя, адрес электронной почты, информацию кредитной карты — Amazon может включить эти сведения в свою базу данных, связав имя Сьюзен с ее идентификационным номером (и со всеми страницами, которые она посетила на сайте в прошлом!) Это дает возможность сайту Amazon и многим другим интернет-магазинам предлагать совершать покупки в один клик — когда Сьюзен выберет товар в следующий свой визит, ей не нужно будет заново вводить имя, номер карты или адрес.

Таким образом, мы с вами видим, что механизм cookie может быть использован, чтобы идентифицировать пользователя. При первом посещении сайта пользователь вводит некоторую информацию о своей идентификации (например, имя), а в последующих сеансах браузер передает cookie-заголовок на сервер, тем самым идентифицируя пользователя на сервере. Можно сказать, что механизм cookie создает как бы дополнительный сеансовый уровень поверх протокола HTTP, который не сохраняет информацию о состоянии соединения. Например, когда пользователь заходит в приложение электронной почты через веб-интерфейс (допустим, используя сервис Hotmail), браузер отсылает серверу информацию cookie, позволяющую ему идентифицировать пользователя в течение сеанса работы приложения.

Несмотря на то, что механизм cookie очень часто упрощает процесс Интернет-покупок для пользователей, с другой стороны, тут возникает проблема нарушения конфиденциальности информации. Как мы только что видели, использование комбинации объектов cookie и предоставленных пользователем учетных данных способно привести к тому, что веб-сайт может получить достаточно информации о пользователе и потенциально передать эту информацию третьей стороне.