Пример резервирования показан на рис. 5.33, *а*. Здесь хост 3 запросил канал к хосту 1. После создания канала поток пакетов от хоста 1 к хосту 3 может течь, не боясь попасть в затор. Рассмотрим, что произойдет, если теперь хост 3 зарезервирует канал к другому передатчику, хосту 2, чтобы пользователь мог одновременно смотреть две телевизионные программы. Зарезервированный второй канал показан на рис. 5.33, *б*. Обратите внимание: между хостом 3 и маршрутизатором *E* требуется наличие двух отдельных каналов, так как передаются два независимых потока.

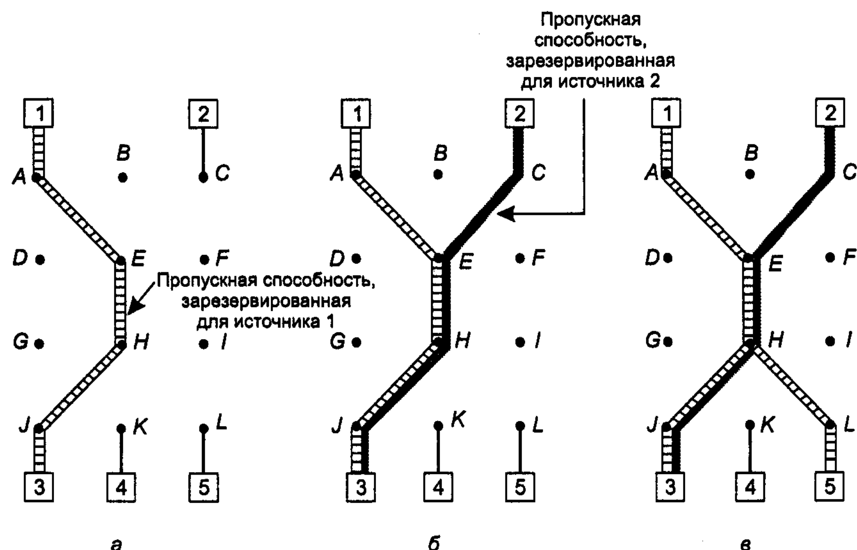


Рис. 5.33. Хост 3 запрашивает канал к хосту 1 (*а*); затем хост 3 запрашивает второй канал к хосту 2 (*б*); хост 5 запрашивает канал к хосту 1 (*в*)

Наконец, на рис. 5.33, *в* хост 5 решает посмотреть программу, передаваемую хостом 1, и также резервирует себе канал. Сначала требуемая пропускная способность резервируется до маршрутизатора *H*. Затем этот маршрутизатор замечает, что у него уже есть канал от хоста 1, поэтому дополнительное резервирование выше по дереву не требуется. Обратите внимание на то, что хосты 3 и 5 могут запросить различную пропускную способность (например, у хоста 3 черно-белый телевизор, поэтому ему не нужна информация о цвете), следовательно, маршрутизатор *H* должен зарезервировать пропускную способность, достаточную для удовлетворения самого жадного получателя.

При подаче запроса на резервирование получатель может (по желанию) указать один или несколько источников, от которых он хотел бы получать сигнал. Он также может указать, является ли выбор источников фиксированным в течение времени резервирования или он оставляет за собой право сменить источники. Данные сведения используются маршрутизаторами для оптимизации планирования пропускной способности. В частности, двум приемникам выделяется общий путь, только если они соглашаются не менять впоследствии свой источник.

В основе такой динамической стратегии лежит полная независимость зарезервированной пропускной способности от выбора источника. Получив зарезервированную пропускную способность, получатель может переключаться с одного источника на другой, сохраняя часть существующего пути, годящуюся для нового источника. Например, если хост 2 передает несколько видеопотоков, хост 3 может переключаться между ними по желанию, не меняя своих параметров резервирования: маршрутизаторам все равно, какую программу смотрит получатель.

Дифференцированное обслуживание

Потоковые алгоритмы способны обеспечивать хорошее качество обслуживания одного или нескольких потоков за счет резервирования любых необходимых ресурсов на протяжении всего маршрута. Однако есть у них и недостаток. Им требуется предварительная договоренность при установке канала для каждого потока. Это не позволяет в достаточной мере расширять систему, в которой применяются данные алгоритмы. Скажем, в системах с тысячами или миллионами потоков интегральное обслуживание применить не удастся. Кроме того, потоковые алгоритмы работают с внутренней информацией о каждом потоке, хранящейся в маршрутизаторах, что делает их уязвимыми к выходу из строя маршрутизаторов. Наконец, программные изменения, которые нужно производить в маршрутизаторах, довольно значительны и связаны со сложными процессами обмена между маршрутизаторами при установке потоков. В результате выжило крайне мало реализаций RSVP и чего-либо подобного.

По этим причинам IETF был создан упрощенный подход к повышению качества обслуживания. Его можно реализовать локально в каждом маршрутизаторе без предварительной настройки и без включения в процесс всех устройств вдоль маршрута. Подход известен как **ориентированное на классы** (в отличие от ориентированного на потоки) качество обслуживания. Проблемной группой IETF была стандартизована специальная архитектура под названием **дифференцированное обслуживание**, описываемая в документах RFC 2474, RFC 2475 и во многих других. Далее мы опишем ее.

Дифференцированное обслуживание (ДО) может предоставляться набором маршрутизаторов, образующих административный домен (например, интернет-провайдер или телефонную компанию). Администрация определяет множество классов обслуживания и соответствующие правила маршрутизации. Пакеты, приходящие от абонента, пользующегося ДО, содержат поле *Тип обслуживания*, и в зависимости от этого значения некоторым классам предоставляется улучшенный сервис по сравнению с остальными. К трафику класса могут предъявляться определенные требования, касающиеся его формы. Например, от него может потребоваться, чтобы он представлял собой «дырявое ведро» с определенной скоростью просачивания данных через «дырочку». Оператор, привыкший брать деньги за все, может взимать дополнительную плату за каждый пакет, обслуживаемый по высшему классу, либо может установить абонентскую плату за передачу *N* таких пакетов в месяц. Обратите внимания: здесь нет никакой предварительной

настройки, резервирования ресурсов и трудоемких согласований параметров для каждого потока, как при интегральном обслуживании. Это делает ДО относительно просто реализуемым.

Обслуживание, ориентированное на классы, возникает и в других областях. Например, службы доставки посылок могут предлагать на выбор несколько уровней обслуживания: доставка на следующий день, через день или через два дня. В самолетах обычно бывают первый класс, бизнесс-класс и второй класс. То же самое касается поездов дальнего следования. Даже в парижской подземке до недавних пор были вагоны двух классов. Что касается нашей тематики, то классы пакетов могут отличаться друг от друга задержками, флуктуациями времени доставки, вероятностью быть проигнорированными в случае коллизии, а также другими параметрами (коих, впрочем, не больше, чем у кадров Ethernet).

Чтобы разница между обслуживанием, ориентированным на классы, и обслуживанием, ориентированным на потоки, стала яснее, рассмотрим пример: интернет-телефонию. При потоковом алгоритме обслуживания каждому телефонному соединению предоставляются собственные ресурсы и гарантии. При классовом обслуживании все телефонные соединения совместно получают ресурсы, зарезервированные для телефонии данного класса. Эти ресурсы, с одной стороны, не может отнять никто извне (соединения других классов, потоки систем передачи файлов и т. п.), с другой стороны, ни одно телефонное соединение не может получить никакие ресурсы в частное пользование.

Срочная пересылка

Выбор классов обслуживания зависит от решения оператора, однако поскольку пакеты зачастую необходимо пересылать между подсетями, управляемыми разными операторами, проблемная группа IETF разрабатывает классы обслуживания, не зависящие от подсети. Простейший из них — класс **срочной пересылки**, с него и начнем. Он описывается документом RFC 3246.

Итак, идея, на которой построена срочная пересылка, очень проста. Существует два класса обслуживания: обычный и срочный. Ожидается, что подавляющая часть объема трафика будет использовать обычный класс обслуживания. Однако есть небольшая доля пакетов, которые необходимо передавать в срочном порядке. Их нужно пересылать между подсетями так, будто кроме них в сети больше нет вообще никаких пакетов. Графическое представление такой двухканальной системы показано на рис. 5.34. Имейте в виду, что физическая линия здесь только одна. Два логических пути — это своеобразный способ резервирования пропускной способности, а вовсе не протягивание второго провода для передачи данных рядом с основным.

Данную стратегию можно реализовать, запрограммировав маршрутизаторы таким образом, чтобы они формировали на выходе две очереди — для обычных и для срочных пакетов. Прибывший пакет ставится в очередь, соответствующую его классу обслуживания. Составление графика передачи пакетов должно опираться на алгоритм, подобный взвешенному справедливому обслуживанию. Например, если срочный трафик составляет 10 % общего объема, а обычный — 90 %, то 20 % пропускной способности можно выделить под срочные пакеты, а все остальное — под обычные. Если мы так сделаем, срочный трафик получит вдвое большую пропускную способность по сравнению со своими нуждами, за счет этого значительно уменьшатся задержки при передаче пакетов. Такое распределение может быть достигнуто, если на каждый срочный пакет приходится четыре обычных (при предположении, что средний размер пакетов обоих классов примерно одинаков). Таким образом, есть надежда, что срочный трафик будет думать, что подсеть пустынна и безжизненна, хотя на самом деле она может быть загружена чрезвычайно сильно.

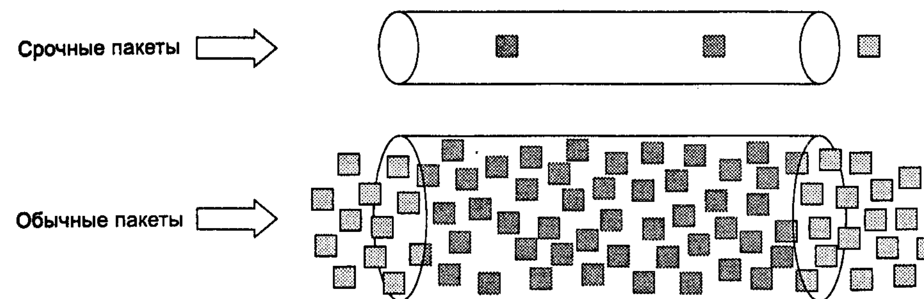


Рис. 5.34. Срочные пакеты движутся по свободной от трафика сети

Гарантированная пересылка

Более совершенная схема управления классами обслуживания называется схемой **гарантированной пересылки**. Эта стратегия описывается в документе RFC 2597. Подразумевается наличие четырех классов приоритетов, каждый из которых обладает своими ресурсами. Кроме того, определены три различных вероятности игнорирования пакетов, попавших в затор (низкая, средняя и высокая). Итого получается 12 сочетаний, то есть 12 классов обслуживания.

На рис. 5.35 показан один из способов обработки пакетов при гарантированной пересылке. На первом шаге пакеты разбиваются на четыре класса приоритетов. Эта процедура может выполняться на хосте-источнике (как показано на рисунке) или на первом маршрутизаторе. Преимуществом классификации пакетов на источнике является то, что в этой точке доступно больше информации о том, каким потокам принадлежат пакеты.

Вторым шагом является маркировка пакетов в соответствии с присвоенными им классами обслуживания. Для маркировки используется поле заголовка. К счастью, в заголовке IP-пакета, как мы вскоре увидим, имеется восьмибитное поле *Тип обслуживания*. Документ RFC 2597 говорит о том, что 6 из этих битов должны использоваться для обозначения класса обслуживания. Таким образом, есть возможность закодировать как все ныне существующие классы, так и те, которые могут появиться в будущем.

На третьем шаге пакеты прогоняются через фильтр, формирующий поток, который может задержать некоторые пакеты при организации четырех потоков. При этом могут использоваться, к примеру, алгоритмы дырявого или маркерного ведра. Если пакетов слишком много, некоторые из них могут вообще быть от-

вергнуты. Здесь большую роль играет категория игнорирования. Есть и более совершенные методы — с замерами и обратной связью.

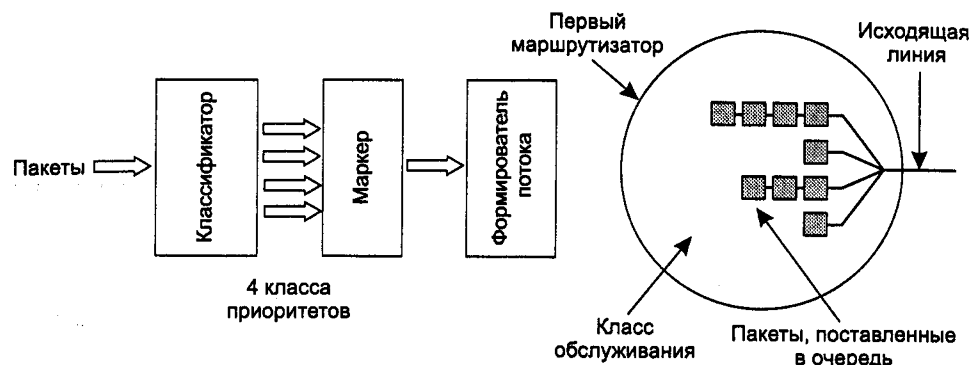


Рис. 5.35. Возможная реализация гарантированной пересылки потока данных

В приведенном примере все три шага выполняются на хосте-источнике, исходящий поток с которого направляется на первый маршрутизатор. Надо отметить, что все эти действия могут выполняться специальным сетевым программным обеспечением или даже операционной системой, что устраняет необходимость в замене существующих приложений.

Коммутация меток и MPLS

Пока проблемная группа IETF разрабатывала интегральные и дифференцированные виды обслуживания, производители маршрутизаторов стремились улучшить методы пересылки данных. Результатом их работы стали, в частности, появление метки в начале каждого пакета и маршрутизация, базирующаяся не на адресе назначения, а на этих метках. Если метку использовать в качестве индекса внутренней таблицы, то поиск подходящей исходящей линии сводится к просмотру таблицы. С использованием этого метода маршрутизация осуществляется очень быстро, и вдоль всего пути следования резервируются все необходимые ресурсы.

Конечно, метод расстановки меток рискует приблизиться слишком близко к виртуальным каналам. X.25, ATM, сети с ретрансляцией кадров, как и любые другие системы, в которых используются подсети с виртуальными каналами, также устанавливают метки (то есть идентификаторы виртуальных каналов) во все пакеты, затем осуществляют поиск по таблице и производят маршрутизацию на основе табличной записи. Несмотря на то, что очень многие представители Интернет-сообщества весьма презрительно относятся к сетям, ориентированным на соединение, похоже, что именно эта идея переживает второе рождение. На этот раз с ее помощью реализуются быстрая маршрутизация и высокое качество обслуживания. Тем не менее, есть существенные различия между тем, как в Интернете формируется маршрут и как это делается в сетях, ориентированных на соединение. Используется, конечно же, не архаичная коммутация пакетов, а нечто иное.

Это «иное» известно под самыми разными именами, включая **коммутацию меток** и **коммутацию тегов**. В конечном счете, проблемная группа IETF занялась стандартизацией своих идей, и это вылилось в появление стандарта **MPLS** (MultiProtocol Label Switching — мультипротокольная коммутация меток). Далее мы будем называть его MPLS. Стандарт описан в документе RFC 3031 и многих других.

Сделаем здесь небольшое отступление. Есть мнение, что *маршрутизация* и *коммутация* — это разные понятия. Маршрутизация — это процесс поиска адреса назначения по таблице и определения исходящей линии, на которую нужно послать данные, чтобы они дошли до адресата. Коммутация, напротив, использует метку, хранящуюся в пакете, в качестве индекса для таблицы пересылок. Впрочем, такие определения далеки от совершенства.

Первая проблема состоит вот в чем: куда поставить метку? Поскольку IP-пакеты не предназначены для виртуальных каналов, в их заголовке не предусмотрено место для номеров виртуальных каналов. Следовательно, нужно добавлять новый заголовок MPLS в начало IP-пакета. На линии между маршрутизаторами, использующей в качестве протокола кадрирования PPP, применяется формат, включающий в себя заголовки PPP, MPLS, IP и TCP, как показано на рис. 5.36. В каком-то смысле MPLS образует уровень с номером 2.5.

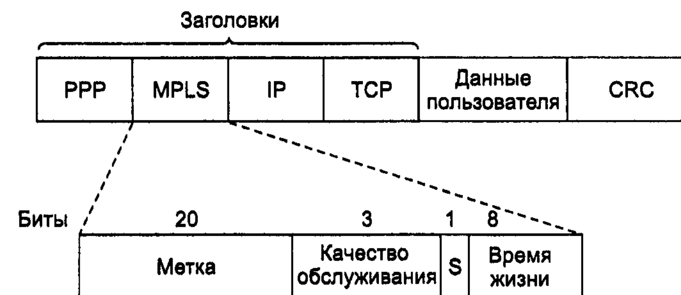


Рис. 5.36. Передача TCP-сегмента с использованием IP, MPLS и PPP

Обычно в заголовок MPLS входят четыре поля, наиболее важное из которых — поле *Метка*, значением которого является индекс. Поле *Качество обслуживания* указывает на применяемый класс обслуживания. Поле *S* связано со стеком меток в иерархических сетях (речь об этом пойдет далее). Если оно равно 0, пакет игнорируется. Благодаря этому исключаются бесконечные циклы в случае сбоя маршрутизации.

Заголовки MPLS не являются частью пакетов сетевого уровня, и к кадрам уровня передачи данных они отношения также не имеют. MPLS является методом, не зависящим от обоих этих уровней. Кроме всего прочего, это свойство означает, что можно создать такие коммутаторы MPLS, которые могут пересылать как IP-пакеты, так и ячейки ATM в зависимости от того, что необходимо в каждом конкретном случае. Именно отсюда следует «мультипротокольность» метода, отраженная в его названии.

Когда пакет (ячейка), расширенный за счет заголовка MPLS, прибывает на MPLS-совместимый маршрутизатор, извлеченная из него метка используется в качестве индекса таблицы, по которой определяются исходящая линия и значение новой метки. Смена меток используется во всех подсетях с виртуальными каналами, поскольку метки имеют только локальное значение, и два разных маршрутизатора могут снабдить независимые пакеты одной и той же меткой, если их нужно направить на одну и ту же линию третьего маршрутизатора. Поэтому, чтобы метки можно было различить на приемном конце, их приходится менять при каждом переходе. Мы видели этот механизм в действии — он был графически изображен на рис. 5.3. В MPLS используется такой же метод.

Еще одним отличием от традиционных виртуальных каналов является уровень агрегации. Конечно, можно каждому потоку, проходящему через подсеть, предоставить собственный набор меток. Однако более распространенным приемом является группировка потоков, заканчивающихся на данном маршрутизаторе или в данной ЛВС, и использование одной метки для всех таких потоков. О потоках, сгруппированных вместе и имеющих одинаковые метки, говорят, что они принадлежат одному **классу эквивалентности пересылок** (**FEC** — Forwarding Equivalence Class). В такой класс входят пакеты, не только идущие по одному и тому же маршруту, но и обслуживаемые по одному классу (в терминах дифференцированного обслуживания). Такие пакеты воспринимаются при пересылке одинаково.

При традиционной маршрутизации с использованием виртуальных каналов невозможно группировать разные пути с разными конечными пунктами в один виртуальный канал, потому что адресат не сможет их различить. В MPLS пакеты содержат не только метку, но и адрес назначения, поэтому в конце помеченного пути заголовок с меткой может быть удален, и дальнейшая маршрутизация может осуществляться традиционным способом — с использованием адреса назначения сетевого уровня.

Одним из основных отличий MPLS от обычных виртуальных каналов является способ построения таблицы маршрутизации. В традиционных сетях пользователь, желающий установить соединение, посылает установочный пакет для создания пути и соответствующей ему записи в таблице. В MPLS этого не происходит, потому что в этом методе вообще отсутствует установочная фаза для каждого соединения (в противном случае пришлось бы менять слишком большую часть существующего программного обеспечения Интернета).

Вместо этого существуют два альтернативных способа создания записей в таблице. При **методе, управляемом данными**, первый маршрутизатор, на который прибывает пакет, контактирует со следующим маршрутизатором на его пути и просит его создать метку для данного потока. Метод является рекурсивным. Можно сказать, что это своего рода создание виртуального канала по требованию.

Протоколы, обслуживающие этот метод, должны очень тщательно следить за предотвращением возникновения петель. Для этого часто используются так называемые **цветные потоки**. Обратное распространение FEC можно сравнить с передачей по подсети потока, окрашенного в уникальный цвет. Если маршрутизатор видит, что тот или иной цвет у него уже имеется, значит, возникла петля, которую необходимо ликвидировать. Метод, управляемый данными, шире всего применяется в сетях с ATM в качестве транспортного уровня (например, в большинстве телефонных систем).

Второй метод, использующийся в не-ATM-сетях, — это **метод с явным управлением**. Имеется несколько вариантов этого подхода. Один из них работает следующим образом. При загрузке маршрутизатора выявляется, для каких маршрутов он является пунктом назначения (например, какие хосты находятся в его ЛВС). Для них создается один или несколько FEC, каждому из них выделяется метка, значение которой сообщается соседям. Соседи, в свою очередь, заносят эти метки в свои таблицы пересылки и посылают новые метки своим соседям. Процесс продолжается до тех пор, пока все маршрутизаторы не получат представление о маршрутах. По мере формирования путей могут резервироваться ресурсы, что позволяет обеспечить надлежащее качество обслуживания.

MPLS может работать на нескольких уровнях одновременно. На высшем уровне оператор связи может рассматриваться в качестве метамаршрутизатора; подразумевается, что между метамаршрутизаторами существует путь от источника до приемника. Этот путь может использоваться MPLS. Однако внутри сети каждого оператора также может применяться MPLS (второй уровень использования мультипротокольной коммутации меток). На самом деле, в пакете может содержаться целый стек меток. Бит *S* (см. рис. 5.36) позволяет маршрутизатору, удаляющему метку, узнать, остались ли у пакета еще метки. Единичное значение бита говорит о том, что метка — последняя в стеке, а нулевое значение говорит об обратном. На практике эта возможность чаще всего используется при реализации частных виртуальных сетей и рекурсивных каналов.

Хотя основные идеи MPLS довольно просты, детали этого метода чрезвычайно сложны, причем существует множество вариаций и улучшений. Поэтому мы не будем уходить вглубь вопроса. Дополнительную информацию можно найти в книгах (Davie и Rekhter, 2000; Lin и др., 2002; Pepelnjak и Guichard, 2001; Wang, 2001).

Объединение сетей

До сих пор мы неявно предполагали наличие единой однородной сети, в которой каждая машина использует один и тот же протокол на всех уровнях. К сожалению, данное предположение слишком оптимистично. Существует множество различных сетей, включая локальные, региональные и глобальные. На каждом уровне широко применяются многочисленные и разнообразные протоколы. В следующих разделах особое внимание будет уделено вопросам, возникающим при объединении двух или более сетей, формирующих **интерсеть**.

По вопросу о том, является ли сегодняшнее изобилие разнообразных типов сетей временным явлением, которое скоро перестанет иметь место, как только все наконец поймут, как замечательна сеть [вставьте свою любимую сеть], или оно является неизбежной данностью окружающего нас мира, единого мнения

нет. Наличие различных сетей всегда приводит к возникновению различных протоколов.

Мы полагаем, что разнообразие сетей (а следовательно, и протоколов) будет оставаться всегда по следующим причинам. Прежде всего, установленная база существующих сетей уже достаточно велика и продолжает расти. Почти все персональные компьютеры используют протокол TCP/IP. Во многих больших компаниях еще остались мейнфреймы, использующие протоколы SNA фирмы IBM. Существенная доля телефонных сетей ориентирована на ATM. Локальные сети персональных компьютеров все еще пользуются протоколами Novell NCP/IPX или AppleTalk. Наконец, беспроводные сети — эта бурно развивающаяся сегодня область — внедряют свои протоколы. Такая тенденция будет сохраняться в ближайшие годы благодаря наличию большого количества существующих сетей и еще благодаря тому, что некоторые производители считают, что возможность клиента легко переходить в системы других производителей не в их интересах.

Во-вторых, по мере того как компьютеры и сети становятся все дешевле, уровень принятия решения о выборе той или иной технологии все опускается и опускается, и теперь уже этим занимаются организации, желающие установить у себя сеть. Многие компании придерживаются политики, что приобретения стоимостью более миллиона долларов должны быть одобрены высшим руководством, покупки дороже 100 000 долларов — руководством среднего звена, а покупки дешевле 100 000 долларов могут совершаться главами отделов безо всякого согласования с вышестоящими руководителями. Такая политика может легко привести к тому, что в техническом отделе будут установлены рабочие станции UNIX, ориентированные на протокол TCP/IP, а отдел маркетинга установит у себя машины Macintosh с AppleTalk.

В-третьих, различные сети (например, ATM и беспроводные сети) основаны на принципиально разных технологиях, поэтому вряд ли стоит удивляться, что с появлением нового оборудования появится и новое программное обеспечение для него. Например, средний дом сейчас напоминает средний офис, каким он был десять лет назад: он битком набит компьютерами, не соединенными друг с другом. В будущем, возможно, будет нормой объединять в единую сеть телефон, телевизор и другую бытовую технику, так чтобы ею можно было управлять дистанционно. Появление этой новой технологии, несомненно, повлечет создание новых протоколов.

Рассмотрим следующий пример взаимодействия нескольких различных сетей, показанный на рис. 5.37. На нем изображена корпоративная сеть, части которой находятся далеко друг от друга и соединены глобальной сетью ATM. В одной из частей для объединения Ethernet, беспроводной ЛВС 802.11 и мейнфреймовой сети SNA корпоративного центра данных используется оптическая магистраль FDDI.

Целью объединения этих сетей является предоставление пользователям возможности общаться с пользователями любой другой из этих сетей, а также получать доступ к данным всех частей сети. Для реализации этих возможностей требуется посылать пакеты из одной сети в другую. Поскольку сети зачастую различаются довольно сильно, это действие осуществить не так уж просто, как мы увидим далее.

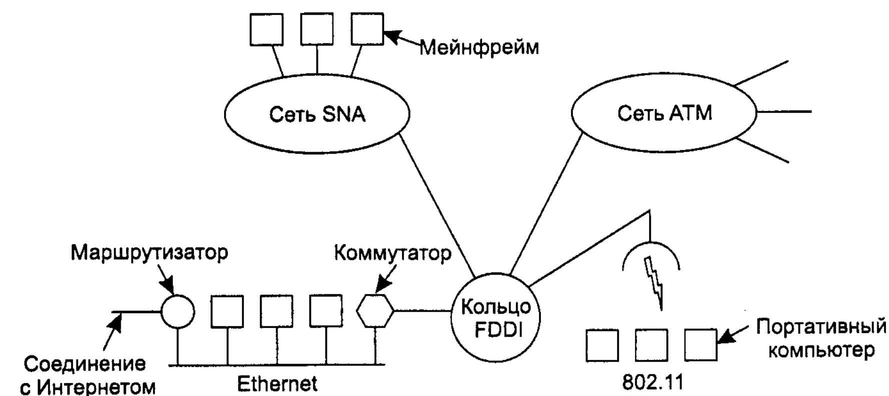


Рис. 5.37. Набор объединенных сетей

Различия сетей

Сети могут отличаться друг от друга довольно сильно и по разным параметрам. Некоторые из параметров, такие как методы модуляции или форматы кадров, нас сейчас не интересуют, поскольку они относятся к физическому уровню и уровню передачи данных. В табл. 5.5 приведен список некоторых параметров, которые могут встретиться на сетевом уровне. Именно сглаживание этих различий делает обеспечение работы объединенной сети значительно более сложным делом, чем обеспечение работы одной сети.

Когда пакетам приходится пересекать несколько сетей, отличных от исходной сети, может возникнуть много проблем, связанных с интерфейсами между сетями. Во-первых, когда пакеты из ориентированной на соединение сети должны пересечь не требующую соединений сеть, их порядок может нарушиться, причем для отправителя это может оказаться неожиданностью, а получатель может оказаться не подготовленным к такому событию. Часто будет требоваться преобразование протоколов, что может быть непросто, если необходимая функциональность не может быть выражена. Также понадобится преобразование форматов адресов, что может потребовать создания некой разновидности системы каталогов. Передача многоадресных пакетов через сеть, не поддерживающую многоадресную рассылку, потребует формирования отдельных пакетов для каждого адресата.

Различия в максимальном размере пакетов в разных сетях составляют главную головную боль. Как передать 8000-байтовый пакет по сети, в которой максимальный размер пакета равен 1500 байтам? При передаче пакета с обязательствами доставки в реальном масштабе времени по сети, не предоставляющей каких-либо гарантий работы в реальном времени, возникает проблема разницы в качестве обслуживания.

Методы обработки ошибок, управления потоком и борьбы с перегрузкой часто различаются в различных сетях. Если отправитель и получатель ожидают, что все пакеты будут доставлены без ошибок и с сохранением порядка, а сеть просто

игнорирует пакеты, когда ей угрожает перегрузка, или пакеты, направляясь различными путями, приходят к получателю совсем не в том порядке, в каком они были отправлены, то многие приложения просто не смогут работать в таких условиях. Различия в механизмах безопасности, установке параметров, правилах тарификации и даже в законах, охраняющих тайну переписки, могут послужить причиной многих проблем.

Таблица 5.5. Некоторые аспекты различия сетей

Аспект	Возможные значения
Предлагаемый сервис	Ориентированные на соединение или не требующие соединения
Протоколы	IP, IPX, SNA, ATM, MPLS, AppleTalk и др.
Адресация	Плоская (802) или иерархическая (IP)
Многоадресная рассылка	Присутствует или отсутствует (а также широковещание)
Размер пакета	У каждой сети есть свой максимум
Качество обслуживания	Может присутствовать и отсутствовать. Много разновидностей
Обработка ошибок	Надежная, упорядоченная и неупорядоченная доставка
Управление потоком	Скользящее окно, управление скоростью, другое или никакого
Борьба с перегрузкой	Дырявое ведро, маркерное ведро, сдерживающие пакеты, нерегулярное раннее обнаружение и др.
Безопасность	Правила секретности, шифрование и т. д.
Параметры	Различные тайм-ауты, спецификация потока и др.
Тарификация	По времени соединения, за пакет, побайтно или никак

Способы объединения сетей

Как уже было показано в главе 4, сети могут объединяться с помощью разных устройств. Давайте вкратце вспомним эту тему. На физическом уровне сети объединяются повторителями или концентраторами, которые просто переносят биты из одной сети в другую такую же сеть. Чаще всего это аналоговые устройства, ничего не смыслящие в цифровых протоколах (они просто-напросто регенерируют сигналы).

Если мы поднимемся на уровень выше, мы обнаружим мосты и коммутаторы, работающие на уровне передачи данных. Они могут принимать кадры, анализировать их МАС-адреса, направлять их в другие сети, осуществляя по ходу дела минимальные преобразования протоколов, например из Ethernet в FDDI или в 802.11.

На сетевом уровне у нас есть маршрутизаторы, соединяющие две сети. Если сетевые уровни у них разные, маршрутизатор может обеспечить перевод пакета из одного формата в другой, хотя такие преобразования сейчас выполняются все реже. Маршрутизатор может поддерживать несколько протоколов, тогда он называется **мультипротокольным маршрутизатором**.

На транспортном уровне существуют транспортные шлюзы, предоставляющие интерфейсы для соединений своего уровня. Транспортный шлюз позволяет, к примеру, передавать пакеты из сети TCP в сеть SNA (протоколы транспортного уровня у них различаются), склеивая одно соединение с другим.

Наконец, на прикладном уровне шлюзы занимаются преобразованием семантики сообщений. Например, шлюзы между электронной почтой Интернета (RFC 822) и электронной почтой X.400 должны анализировать содержимое сообщений и изменять различные поля электронного конверта.

В данной главе мы сосредоточим наше внимание на объединении сетей на сетевом уровне. Чтобы понять, в чем состоит его отличие от объединения на уровне передачи данных, рассмотрим рис. 5.38. На рис. рис. 5.38, *а* источник *S* пытается послать пакет приемнику *D*. Эти две машины работают в разных сетях Ethernet, соединенных коммутатором. Источник *S* вставляет пакет в кадр и отправляет его. Кадр прибывает на коммутатор, который по МАС-адресу определяет, что его надо переслать в ЛВС 2. Коммутатор просто снимает кадр с ЛВС 1 и передает его в ЛВС 2.

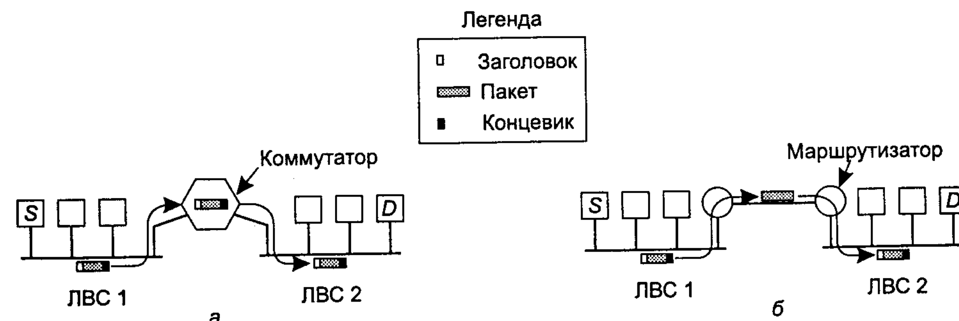


Рис. 5.38. Две сети Ethernet, объединенные коммутатором (*а*); две сети Ethernet, объединенные маршрутизаторами (*б*)

Теперь рассмотрим ту же ситуацию, но с применением другого оборудования. Допустим, две сети Ethernet объединены не коммутатором, а парой маршрутизаторов. Маршрутизаторы между собой соединены двухточечной линией, которая может представлять собой, например, выделенную линию длиной в тысячи километров. Что в данном случае будет происходить с кадром? Он принимается маршрутизатором, из его поля данных извлекается пакет. Далее маршрутизатор анализирует содержащийся в пакете адрес (например, IP-адрес). Этот адрес нужно отыскать в таблице маршрутизации. В соответствии с ним принимается решение об отправке пакета (возможно, упакованного в кадр нового вида — это зависит от протокола, используемого линией) на удаленный маршрутизатор. На противоположном конце пакет вставляется в поле данных кадра Ethernet и помещается в ЛВС 2.

В чем заключается основная разница между случаем коммутации (установки моста) и маршрутизации? Коммутатор (мост) пересылает весь пакет, обосновывая свое решение значением МАС-адреса. При применении маршрутизатора пакет извлекается из кадра, и для принятия решения используется адрес, содержащийся именно в пакете. Коммутаторы не обязаны вникать в подробности уст-

ройства протокола сетевого уровня, с помощью которого производится коммутация. А маршрутизаторы обязаны.

Сцепленные виртуальные каналы

Наиболее распространенными являются два стиля объединения сетей: ориентированное на соединение сцепление подсетей виртуальных каналов и дейтаграммный интерсетевой стиль. Мы рассмотрим их поочередно, однако необходимо предварить наше рассмотрение небольшим вступлением. В прошлом большинство сетей (общего пользования) были ориентированными на соединение (сети с ретрансляцией кадров, SNA, 802.16 и ATM по сей день являются таковыми). Со стремительным развитием Интернета все больше входили в моду дейтаграммы. Тем не менее, было бы ошибкой думать, что дейтаграммный способ будет существовать вечно. В этом деле единственное постоянство — это изменчивость. С ростом доли и важности мультимедийных данных в общем потоке растет вероятность того, что наступит эпоха возрождения для технологий, ориентированных на соединение. Причиной тому является тот простой факт, что при установлении соединения гораздо проще гарантировать определенный уровень обслуживания. Далее мы еще уделим некоторое место сетям, ориентированным на соединение.

В модели сцепленных виртуальных каналов, показанной на рис. 5.39, соединение с хостом в удаленной сети устанавливается способом, близким к тому, как устанавливаются обычные соединения. Подсеть видит, что адресат является удаленным, и создает виртуальный канал к ближайшему маршрутизатору из сети адресата. Затем строится виртуальный канал от этого маршрутизатора к внешнему **шлюзу** (многопротокольному маршрутизатору). Этот шлюз запоминает существование созданного виртуального канала в своих таблицах и строит новый виртуальный канал к маршрутизатору в следующей подсети. Процесс продолжается до тех пор, пока не будет достигнут хост-получатель.

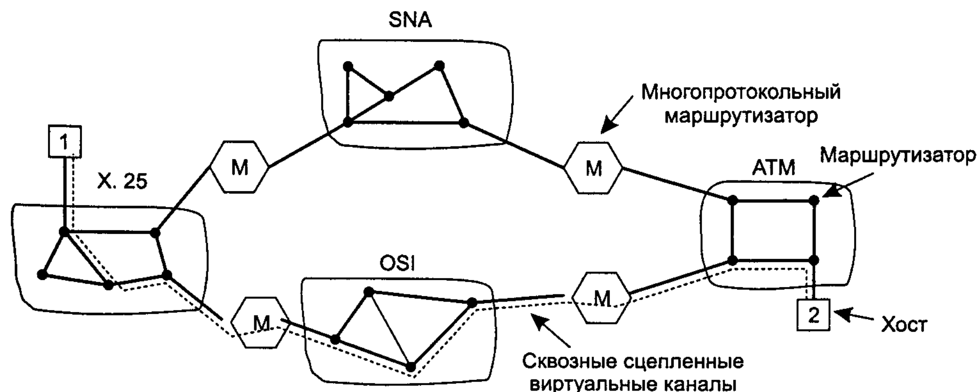


Рис. 5.39. Объединение сетей с помощью сцепленных виртуальных каналов

Когда по проложенному пути начинают идти пакеты данных, каждый шлюз переправляет их дальше, преобразуя формат пакетов и номера виртуальных каналов. Очевидно, что все информационные пакеты будут передаваться по одному и тому же пути и, таким образом, прибудут к пункту назначения с сохранением порядка отправления.

Существенной особенностью данного подхода является то, что последовательность виртуальных пакетов устанавливается от источника через один или более шлюзов к приемнику. Каждый шлюз хранит таблицы, содержащие информацию о проходящих через них виртуальных каналах, о том, как осуществлять маршрутизацию для них и каков номер нового виртуального канала.

Такая схема лучше всего работает, когда все сети обладают примерно одинаковыми свойствами. Например, если каждая из них гарантирует надежную доставку пакета сетевого уровня, то, исключив случай сбоя системы где-то на его пути, можно сказать, что и весь поток от источника до приемника будет надежным. С другой стороны, если машина-источник работает в сети, которая гарантирует надежную доставку, а какая-то промежуточная сеть может терять пакеты, то сцепление радикально изменит сущность сервиса.

Сцепленные виртуальные каналы часто применяются на транспортном уровне. В частности, можно построить битовый канал, используя, скажем, SNA, который заканчивается на шлюзе, и иметь при этом TCP-соединение между соседними шлюзами. Таким образом, можно построить сквозной виртуальный канал, охватывающий разные сети и протоколы.

Дейтаграммное объединение сетей

Альтернативной моделью объединения сетей является дейтаграммная модель, показанная на рис. 5.40. В данной модели единственный сервис, который сетевой уровень предоставляет транспортному уровню, состоит в возможности посылать в сеть дейтаграммы и надеяться на лучшее. На сетевом уровне нет никакого упоминания о виртуальных каналах, не говоря уже об их сцеплении. В этой модели пакеты не обязаны следовать по одному и тому же маршруту, даже если они принадлежат одному соединению. На рис. 5.40 показаны дейтаграммы, следующие от хоста 1 к хосту 2 и выбирающие различные маршруты по объединенной сети. Выбор маршрута производится независимо для каждого пакета. Он может зависеть от текущей загруженности сети. При такой стратегии могут использоваться различные маршруты, что позволяет достигать большей пропускной способности, чем при применении модели сцепленных виртуальных каналов. С другой стороны, не дается никакой гарантии того, что пакеты прибудут к получателю в нужном порядке, если они вообще прибудут.

Модель, изображенная на рис. 5.40, не так проста, как кажется. Во-первых, если каждая сеть обладает своим сетевым уровнем, то пакет из одной сети не сможет перейти в другую сеть. Можно представить себе многопротокольные маршрутизаторы, пытающиеся преобразовать пакет из одного формата в другой, но, если только эти форматы не являются близкими родственниками, такое преобразование всегда будет неполным и часто обречено на ошибку.

Еще более серьезные проблемы возникают с адресацией. Представьте себе простой случай: интернет-хост пытается послать IP-пакет хосту, находящемуся в соседней сети SNA. Адреса IP и SNA различаются. Значит, потребуется двухстороннее приведение одного формата адреса к другому. Более того, различается сама концепция адресации. В IP адресуемой единицей является хост (собственно, интерфейсная карта). В SNA адресуемыми могут быть не только хосты, но и другие сущности (например, физические устройства). В идеале необходимо поддерживать базу данных, отображающую одни адресуемые сущности на другие, но это задача исключительно трудоемкая, если не сказать невыполнимая.

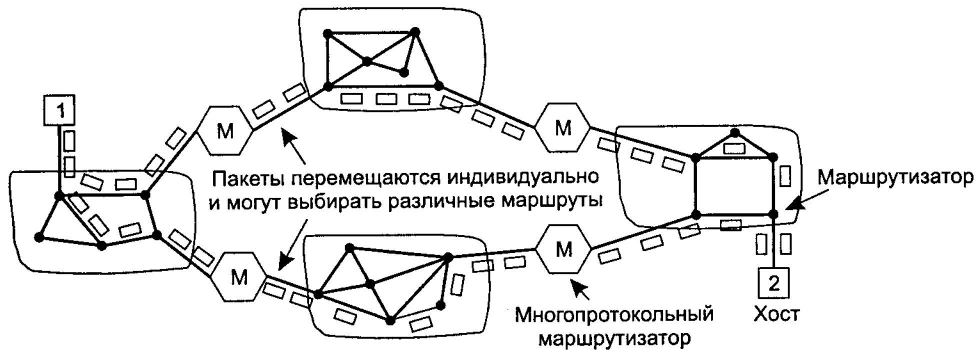


Рис. 5.40. Дейтаграммная объединенная сеть

Другая идея заключается в создании универсального межсетевого пакета, который распознавался бы всеми маршрутизаторами. Именно эта идея была взята за основу при разработке протокола IP: IP-пакеты созданы для передачи по разным сетям. Впрочем, может, конечно, оказаться так, что применяемый сегодня в Интернете протокол IPv4 вытеснит с рынка все остальные форматы, IPv6 (будущий протокол Интернета) не получит ожидаемого распространения и ничего нового вообще не будет изобретено, однако пока что исторически все складывается совсем не так. Заставить всех согласиться применять только один формат практически невозможно, особенно учитывая тот факт, что каждая компания считает самым большим своим достижением изобретение, внедрение и раскручивание собственного формата.

Подведем краткие итоги обсуждения двух способов объединения сетей. Модель сцепленных виртуальных каналов обладает теми же преимуществами, что и применение виртуальных каналов внутри единой подсети: буферы могут быть зарезервированы заранее, сохранение порядка пакетов гарантируется, могут использоваться короткие заголовки, кроме того, можно избежать неприятностей, вызываемых дубликатами пакетов.

Недостатки опять те же самые: маршрутизаторы должны хранить таблицы с записями для каждого открытого соединения, при возникновении затора обходные пути не используются, а выход маршрутизатора из строя обрывает все проходящие через него виртуальные каналы. Кроме того, очень сложно, может, даже невозможно реализовать систему виртуальных каналов, если в состав объединенной сети входит хотя бы одна ненадежная дейтаграммная сеть.

Свойства дейтаграммного подхода к объединению сетей те же самые, что и у дейтаграммных подсетей: риск возникновения перегрузки выше, но также больше возможностей для адаптации к ней, высокая надежность в случае отказов маршрутизаторов, однако требуются более длинные заголовки пакетов. В объединенной сети, как и в единой дейтаграммной сети, возможно применение различных адаптивных алгоритмов выбора маршрута.

Главное преимущество дейтаграммного подхода к объединению сетей заключается в том, что он может применяться в подсетях без виртуальных каналов. К дейтаграммным сетям относятся многие локальные сети, мобильные сети (в том числе применяемые в воздушном и морском транспорте) и даже некоторые глобальные сети. При включении одной из этих сетей в объединенную сеть стратегия объединения сетей на основе виртуальных каналов встречает серьезные трудности.

Туннелирование

Объединение сетей в общем случае является исключительно сложной задачей. Однако есть частный случай, реализация которого вполне осуществима. Это случай, при котором хост-источник и хост-приемник находятся в сетях одного типа, но между ними находится сеть другого типа. Например, представьте себе международный банк, у которого имеется одна TCP/IP-сеть на основе Ethernet в Париже и такая же сеть в Лондоне, а между ними находится какая-нибудь глобальная не-IP сеть (например, ATM), как показано на рис. 5.41.

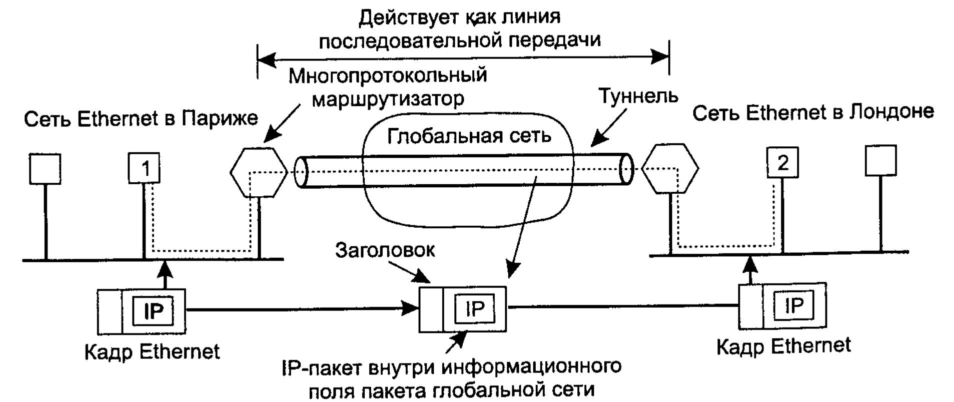


Рис. 5.41. Туннелирование пакета из Парижа в Лондон

Метод решения данной проблемы называется **туннелированием**. Чтобы послать IP-пакет хосту 2, хост 1 формирует пакет, содержащий IP-адрес хоста 2, помещает его в кадр Ethernet, адресованный парижскому многопротокольному маршрутизатору, и пересылает его по сети Ethernet. Получив кадр, многопротокольный маршрутизатор извлекает IP-пакет, помещает его в поле данных пакета сетевого уровня глобальной сети и пересылает его лондонскому многопрото-

кольному маршрутизатору. Когда пакет попадает туда, лондонский многопротокольный маршрутизатор извлекает IP-пакет и посылает его хосту 2 внутри кадра Ethernet.

Глобальную сеть при этом можно рассматривать как большой туннель, простирающийся от одного многопротокольного маршрутизатора до другого. IP-пакет, помещенный в красивую упаковку, просто перемещается от одного конца туннеля до другого. Ему не нужно беспокоиться о взаимодействии с глобальной сетью. Это же касается и хостов сетей Ethernet по обе стороны туннеля. Переупаковкой пакета и переадресацией занимаются многопротокольные маршрутизаторы. Для этого им нужно уметь разбираться в IP-адресах и обладать информацией о формате пакетов глобальной сети. В результате весь путь от середины одного многопротокольного маршрутизатора до середины другого работает, как линия последовательной передачи.

Чтобы сделать этот пример еще проще и понятнее, рассмотрим в качестве аналогии водителя автомобиля, направляющегося из Парижа в Лондон. По территории Франции автомобиль двигается по дороге сам. Но, достигнув Ла-Манша, он погружается в высокоскоростной поезд и транспортируется под проливом по туннелю (автомобилям не разрешается передвигаться по туннелю самим). Таким образом, автомобиль перевозится как груз (рис. 5.42). На другом конце туннеля автомобиль спускается с железнодорожной платформы на английское шоссе и снова продолжает путь своими силами. Точно такой же принцип применяется при туннелировании пакетов, проходящих через чужеродную сеть.

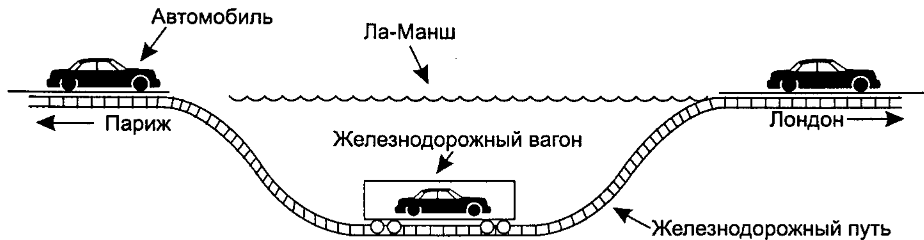


Рис. 5.42. Туннелирование автомобиля, едущего из Парижа в Лондон

Маршрутизация в объединенных сетях

Маршрутизация в объединенных сетях аналогична маршрутизации в единой подсети, но связана с некоторыми дополнительными сложностями. Рассмотрим, например, объединенную сеть, изображенную на рис. 5.43, *а*, в которой пять сетей соединены шестью (возможно, многопротокольными) маршрутизаторами. Построение графа для данной сети усложняется тем фактом, что каждый многопротокольный маршрутизатор может напрямую обращаться (то есть посылать пакеты) к любому другому многопротокольному маршрутизатору, соединенному с той же сетью, что и он. Например, многопротокольный маршрутизатор *B* на рис. 5.43, *а* может напрямую обращаться к многопротокольным маршрутизаторам *A* и *C* по сети 2, а также к многопротокольному маршрутизатору *B* по сети 3. В результате мы получим граф, показанный на рис. 5.43, *б*.

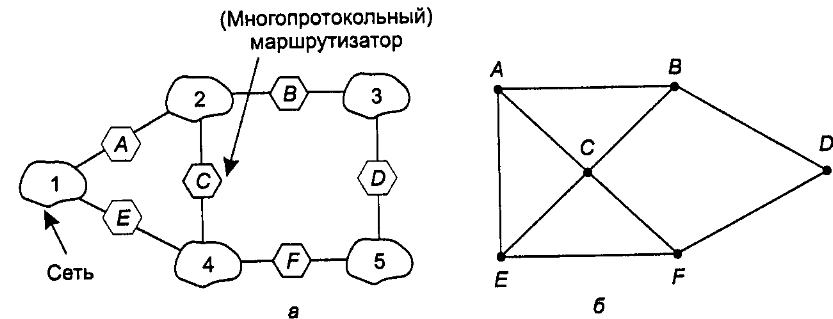


Рис. 5.43. Объединенная сеть (*а*); граф объединенной сети (*б*)

Когда граф построен, к множеству многопротокольных маршрутизаторов можно применить известные алгоритмы выбора маршрутов, такие как маршрутизация по вектору расстояний или алгоритм, учитывающий состояние линий. Таким образом, получается двухуровневый алгоритм маршрутизации: в пределах каждой сети используется **внутренний шлюзовый протокол**, но между сетями применяется **внешний шлюзовый протокол** (термин «шлюз» употреблялся раньше вместо термина «маршрутизатор»). Так как все сети независимы, в них могут применяться различные алгоритмы маршрутизации. Благодаря независимости сетей друг от друга, они часто называются **автономными системами (AS)** (АС).

Типичный интерсетевой пакет, отправляясь из своей локальной сети, адресуется своему локальному многопротокольному маршрутизатору (адрес прописывается в заголовке МАС-уровня). Когда он попадает туда, сетевой уровень с помощью своих таблиц решает, какому многопротокольному маршрутизатору переслать этот пакет. Если до соответствующего маршрутизатора можно добраться с помощью «родного» сетевого протокола пакета, он направляется туда напрямую. В противном случае он туннелируется туда, для чего используются пакет и протокол промежуточной сети. Этот процесс повторяется до тех пор, пока пакет не прибудет в сеть адресата.

Одно из различий внутрисетевой и межсетевой маршрутизаций состоит в том, что при межсетевой маршрутизации часто требуется пересечение международных границ. Неожиданно вступают в игру различные законы, как, например, строгое шведское законодательство, касающееся экспорта личных сведений о шведских гражданах за пределы Швеции. Другим примером является канадский закон, гласящий, что поток данных, начинающийся и заканчивающийся в Канаде, не может покидать пределы страны. Это означает, что трафик из Виндзора, штат Онтарио, в Ванкувер не может проходить через окрестности соседнего Детройта, даже если такой путь быстрее и дешевле.

Другим различием внутрисетевой и межсетевой маршрутизаций является их стоимость. В пределах одной сети обычно применяется единый алгоритм тарификации. Однако различные сети могут управляться различными организация-

ми, и один маршрут может оказаться дешевле другого. Аналогично, качество обслуживания, предлагаемое в разных сетях, также может различаться, что также может послужить причиной выбора того или иного маршрута.

Фрагментация

Все сети накладывают ограничения на размер своих пакетов. Эти пределы вызваны различными предпосылками, среди которых есть следующие:

1. Аппаратные (например, размер кадра Ethernet).
2. Операционная система (например, все буферы имеют размер 512 байт).
3. Протоколы (например, количество бит в поле длины пакета).
4. Соответствие какому-либо международному или национальному стандарту.
5. Желание снизить количество пакетов, пересылаемых повторно из-за ошибок передачи.
6. Желание предотвратить ситуацию, когда один пакет слишком долгое время занимает канал.

Результатом действия всех этих факторов является то, что разработчики не могут выбирать максимальный размер пакета по своему усмотрению. Максимальный размер поля полезной нагрузки варьируется от 48 байт (ATM-ячейки) до 65 515 байт (IP-пакеты), хотя на более высоких уровнях размер поля полезной нагрузки часто бывает больше.

Очевидно, возникает проблема, когда большой пакет хочет пройти по сети, в которой максимальный размер пакетов слишком мал. Одно из решений состоит в предотвращении возникновения самой проблемы. Другими словами, объединенная сеть должна использовать такой алгоритм маршрутизации, который не допускает пересылки пакетов по сетям, которые не могут их принять. Однако это решение вовсе не является решением. Что произойдет, если исходный пакет окажется слишком велик для сети адресата? Алгоритм маршрутизации будет в данном случае бессилен.

Следовательно, единственное решение проблемы заключается в разрешении шлюзам разбивать пакеты на **фрагменты** и посылать каждый фрагмент в виде отдельного межсетевого пакета. Однако, как вам скажет любой родитель маленького ребенка, преобразование объекта в небольшие фрагменты существенно проще, чем обратный процесс. (Физики даже дали этому эффекту ломания игрушек специальное название: второй закон термодинамики.) В сетях с коммутацией пакетов также существует проблема с восстановлением пакетов из фрагментов.

Для восстановления исходных пакетов из фрагментов применяются две противоположные стратегии. Первая стратегия заключается в том, чтобы фрагментация пакета, вызванная сетью с пакетами малых размеров, оставалась прозрачной для обоих хостов, обменивающихся пакетом. Этот вариант показан на рис. 5.44, *а*. «Мелкопакетная» сеть имеет шлюзы (скорее всего, это специализированные маршрутизаторы), предоставляющие интерфейсы другим сетям. Когда на такой шлюз приходит пакет слишком большого размера, он разбивается на фрагменты.

Каждый фрагмент адресуется одному и тому же выходному шлюзу, восстанавливающему из этих фрагментов исходный пакет. Таким образом, прохождение данных через мелкопакетную сеть оказывается прозрачным. Соседние сети даже не догадываются о том, что у них под боком пакеты страшным образом нарезаются, а потом снова склеиваются. В сетях ATM, например, есть даже специальная аппаратура для обеспечения прозрачной фрагментации пакетов (разбивания на ячейки) и обратной сборки ячеек в пакеты. В мире ATM фрагментацию называют сегментацией. Концепция та же самая, а отличия есть только в некоторых деталях.

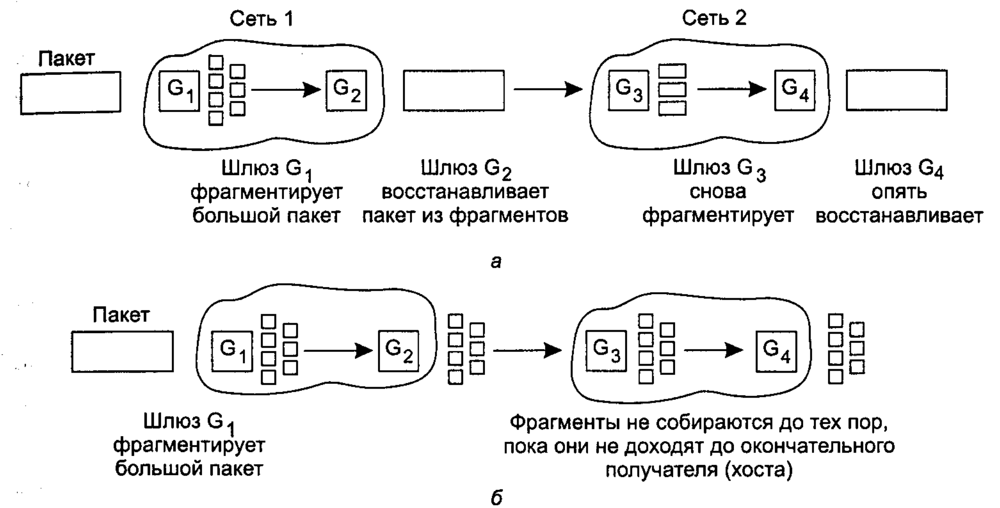


Рис. 5.44. Прозрачная фрагментация (*а*); непрозрачная фрагментация (*б*)

Прозрачная фрагментация проста, но, тем не менее, создает некоторые проблемы. Во-первых, выходной шлюз должен уметь определять момент получения последней части пакета, поэтому каждый фрагмент должен содержать либо поле счетчика, либо признак конца пакета. Во-вторых, все фрагменты должны выходить через один и тот же шлюз. Таким образом, налагается запрет на использование фрагментами различных путей к окончательному получателю, и в результате может оказаться потерянной часть производительности. Наконец, процессы фрагментации и последующей сборки пакетов при прохождении каждой сети с малым размером пакетов приводят к дополнительным накладным расходам. Сетям ATM требуется прозрачная фрагментация.

Другая стратегия фрагментации состоит в отказе от восстановления пакета из фрагментов на промежуточных маршрутизаторах. Как только пакет оказывается разбитым на отдельные фрагменты, с каждым фрагментом обращаются как с отдельным пакетом. Все фрагменты проходят через выходной шлюз (или несколько), как показано на рис. 5.44, *б*. Задача восстановления оригинального пакета возложена на получающий хост. Так работает IP.

С непрозрачной фрагментацией связаны свои проблемы. Например, она требует, чтобы *каждый* хост мог восстановить пакет из фрагментов. Кроме того, при фрагментации большого пакета возрастают суммарные накладные расходы, так

как каждый фрагмент должен иметь заголовок. В то время как в случае прозрачной фрагментации лишние заголовки при выходе из мелкопакетной сети исчезали, в данном методе накладные расходы сохраняются на протяжении всего пути. Однако преимущество непрозрачной фрагментации состоит в возможности использовать для передачи фрагментов несколько различных маршрутов, что повышает производительность. Естественно, при использовании модели сцепленных виртуальных каналов это преимущество оказывается бесполезным.

Фрагменты пакета должны нумероваться таким образом, чтобы можно было восстановить исходный поток данных. Один из способов нумерации фрагментов состоит в использовании дерева. Если пакет 0 должен быть расщеплен, фрагменты получают номера 0.0, 0.1, 0.2 и т. д. Если эти фрагменты в дальнейшем сами фрагментируются, получающиеся кусочки нумеруются так: 0.0.0, 0.0.1, 0.0.2, ..., 0.1.0, 0.1.1, 0.1.2 и т. д. Если в заголовках зарезервировано достаточно места для случая наиболее глубокого разбиения и при этом отсутствуют дубликаты, то такая схема гарантирует правильную сборку пакета получателем независимо от порядка, в котором будут получены отдельные фрагменты.

Однако если одна из сетей нечаянно потеряет или удалит один или несколько фрагментов, то понадобится повторная передача всего пакета с тяжелыми последствиями для системы нумерации. Представим, что передается пакет длиной 1024 байта. При первой передаче пакета он разбивается на четыре одинаковых фрагмента с номерами 0.0, 0.1, 0.2 и 0.3. Фрагмент 0.1 теряется по дороге, а остальные успешно добираются до получателя. Отправитель не получает подтверждения на переданный пакет и посылает его снова. Но на этот раз срабатывает закон бутерброда (или закон Мерфи): пакет пересылается по другому маршруту и проходит через сеть с 512-байтовым ограничением на размер пакетов, поэтому пакет разбивается всего на два фрагмента. Получив фрагмент с номером 0.1, получатель может подумать, что это — как раз недостающая деталь, и в результате соберет пакет неверно.

Совершенно иной и гораздо более совершенный подход к решению данной проблемы состоит в определении размера элементарного фрагмента, достаточно малого, чтобы он мог пройти по любой сети. То есть пакет разбивается на элементарные фрагменты одинакового размера плюс довесок, который может быть только короче всех остальных. Для пущей эффективности межсетевой пакет может содержать несколько фрагментов. Межсетевой заголовок должен содержать номер исходного пакета и номер (первого) элементарного фрагмента, содержащегося в нем. Как обычно, в заголовке должен содержаться признак конца исходного пакета.

Такой подход требует включения в заголовок межсетевого пакета двух полей: порядкового номера исходного пакета и порядкового номера фрагмента. Есть вполне очевидный компромисс между размером элементарного фрагмента и числом бит номера фрагмента. Поскольку размер элементарного пакета выбирается таким образом, что он может пройти по любой сети, дальнейшая фрагментация межсетевого пакета не составляет проблемы. Последним пределом является элементарный фрагмент размером с бит или байт, при этом номер фрагмента, содержащийся в заголовке пакета, превращается в смещение до этого бита или байта (рис. 5.45).

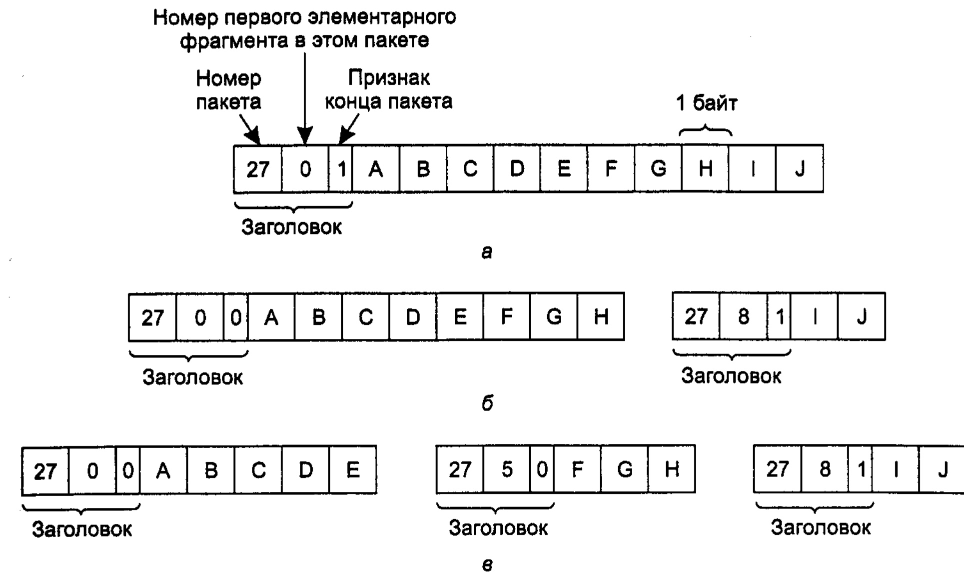


Рис. 5.45. Фрагментация при элементарном размере 1 байт: исходный пакет, содержащий 10 байт данных (*а*); фрагменты после прохождения через сеть с максимальным размером 8 байт (*б*); фрагменты после прохождения через шлюз размером 5 (*в*)

Некоторые межсетевые протоколы развивают этот метод дальше и рассматривают всю передачу по виртуальному каналу как один гигантский пакет, так что каждый фрагмент содержит абсолютный номер первого байта фрагмента.

Сетевой уровень в Интернете

Прежде чем начинать рассматривать особенности реализации сетевого уровня в интернет-технологиях, неплохо было бы вспомнить принципы, которые были основополагающими в прошлом, при его разработке, и которые обеспечили сегодняшний успех. В наше время люди все чаще пренебрегают ими. Между тем эти принципы пронумерованы и включены в документ RFC 1958, с которым полезно ознакомиться (а для разработчиков протоколов он должен быть просто обязательным для прочтения, с экзаменом в конце). Этот документ сильно перекликается с идеями, которые можно найти в книгах (Clark, 1988; Saltzer и др., 1984). Далее мы приведем 10 основных принципов, начиная с самых главных.

1. **Убедитесь в работоспособности.** Нельзя считать разработку (или стандарт) законченной, пока не проведен ряд успешных соединений между прототипами. Очень часто разработчики сначала пишут тысячестраничное описание стандарта, утверждают его, а потом обнаруживается, что он еще очень сырой или вообще неработоспособен. Тогда пишется версия 1.1 стандарта. Так бывает, но так быть не должно.

2. **Упрощайте.** Если есть сомнения, всегда самый простой выбор является самым лучшим. Уильям Оккам (William Occam) декларировал этот принцип еще в XIV веке («бритва Оккама»). Его можно выразить следующим образом: *«Борись с излишествами»*. Если какое-то свойство не является абсолютно необходимым, забудьте про него, особенно если такого же эффекта можно добиться комбинированием уже имеющихся свойств.
3. **Всегда делайте четкий выбор.** Если есть несколько способов реализации одного и того же, необходимо выбрать один из них. Увеличение количества способов — это порочный путь. В стандартах часто можно встретить несколько опций, режимов или параметров. Почему так получается? Лишь потому, что при разработке было несколько авторитетных мнений на тему того, что является наилучшим. Разработчики должны избегать этого и сопротивляться подобным тенденциям. Надо просто уметь говорить «Нет».
4. **Используйте модульный принцип.** Этот принцип напрямую приводит к идее стеков протоколов, каждый из которых работает на одном из независимых уровней. Таким образом, если обстоятельства складываются так, что необходимо изменить один из модулей или уровней, то это не приводит к необходимости других частей системы.
5. **Предполагайте разнородность.** В любой большой сети могут сосуществовать различные типы оборудования, средства передачи данных и различные приложения. Сетевая технология должна быть достаточно гибкой, простой и обобщенной, чтобы работать в таких условиях.
6. **Избегайте статичности свойств и параметров.** Если есть какие-то обязательные параметры (например, максимальный размер пакета), то лучше заставить отправителя и получателя договариваться об их конкретных значениях, чем жестко закреплять их.
7. **Проект должен быть хорошим, но он не может быть идеальным.** Очень часто разработчики создают хорошие проекты, но не могут предусмотреть какие-нибудь причудливые частные случаи. Не стоит портить то, что сделано хорошо и работает в большинстве случаев. Вместо этого имеет смысл переложить все бремя ответственности за «улучшения» проекта на тех, кто предъявляет свои странные требования.
8. **Тщательно продумывайте отправку данных, но будьте снисходительны при приеме данных.** Другими словами, посылайте пакеты, только убедившись в том, что они полностью соответствуют всем требованиям стандартов. При этом надо иметь в виду, что приходящие пакеты не всегда столь идеальны и их тоже нужно обрабатывать.
9. **Продумайте масштабируемость.** Если в системе работают миллионы хостов и миллиарды пользователей, о централизации можно забыть. Нагрузка должна быть распределена максимально равномерно между имеющимися ресурсами.
10. **Помните о производительности и цене.** Никого не заинтересует сеть низкопроизводительная или дорогостоящая.

Теперь перейдем от общих принципов к деталям построения сетевого уровня Интернета. На сетевом уровне Интернет можно рассматривать как набор подсетей или **автономных систем**, соединенных друг с другом. Структуры как таковой Интернет не имеет, но все же есть несколько магистралей. Они собраны из высокопроизводительных линий и быстрых маршрутизаторов. К магистралям присоединены региональные сети (сети среднего уровня), с которыми, в свою очередь, соединяются локальные сети многочисленных университетов, компаний и провайдеров. Схема этой квазииерархической структуры показана на рис. 5.46.

Вся эта конструкция «склеивается» благодаря протоколу сетевого уровня, **IP** (Internet Protocol — протокол сети Интернет). В отличие от большинства ранних протоколов сетевого уровня, IP с самого начала разрабатывался как протокол межсетевого обмена. Вот как можно описать данный протокол сетевого уровня: его работа заключается в приложении максимума усилий (тем не менее, без всяких гарантий) по транспортировке дейтаграмм от отправителя к получателю независимо от того, находятся эти машины в одной и той же сети или нет.

Соединение в сети Интернет представляет собой следующее. Транспортный уровень берет поток данных и разбивает его на дейтаграммы. Теоретически размер каждой дейтаграммы может достигать 64 Кбайт, однако на практике они обычно не более 1500 байт (укладываются в один кадр Ethernet). Каждая дейтаграмма пересылается по Интернету, возможно, разбиваясь при этом на более мелкие фрагменты, собираемые сетевым уровнем получателя в исходную дейтаграмму. Затем эта дейтаграмма передается транспортному уровню, вставляющему ее во входной поток получающего процесса. На рис. 5.46 видно, что пакет, посланный хостом 1, пересечет на своем пути шесть сетей, прежде чем доберется до хоста 2. На практике промежуточных сетей оказывается гораздо больше.

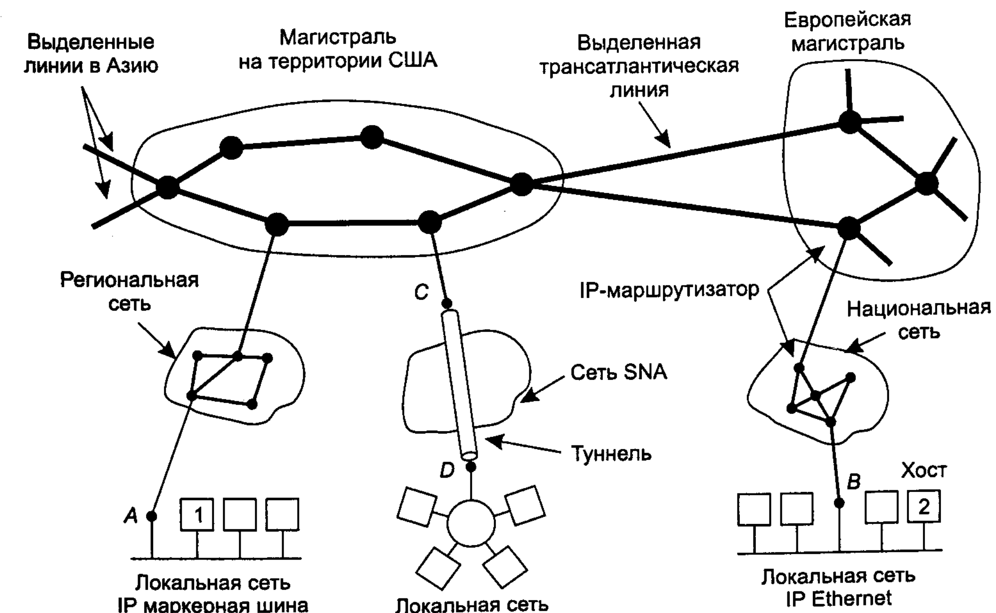


Рис. 5.46. Интернет представляет собой набор соединенных друг с другом сетей

Протокол IP

Начнем изучение сетевого уровня Интернета с формата IP-дейтаграмм. IP-дейтаграмма состоит из заголовка и текстовой части. Заголовок содержит обязательную 20-байтную часть, а также необязательную часть переменной длины. Формат заголовка показан на рис. 5.47. Он передается слева направо, то есть старший бит поля *Версия* передается первым. (В процессоре SPARC байты располагаются слева направо, в процессоре Pentium — наоборот, справа налево.) На машинах, у которых старший байт располагается после младшего, как, например, у семейства процессоров корпорации Intel, требуется программное преобразование как при передаче, так и при приеме.

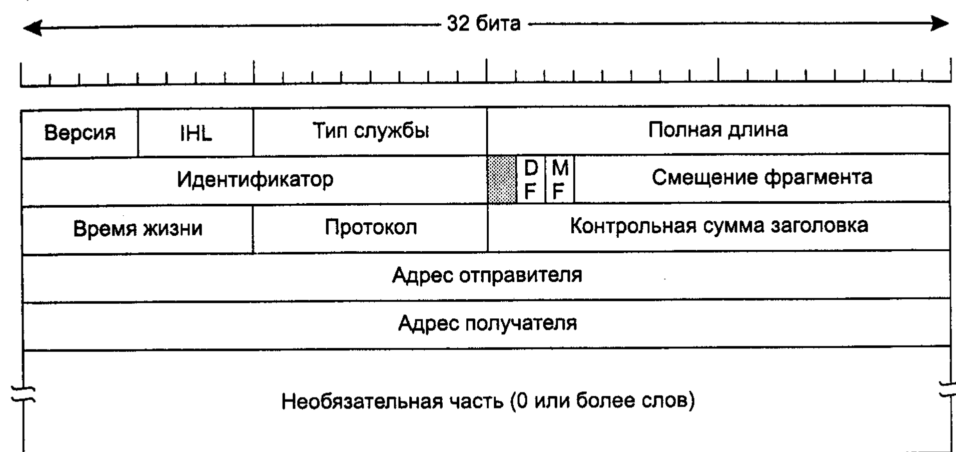


Рис. 5.47. Заголовок IP-дейтаграммы IPv4

Поле *Версия* содержит версию протокола, к которому принадлежит дейтаграмма. Включение версии в каждую дейтаграмму позволяет использовать разные версии протокола на разных машинах. Дело в том, что с годами протокол изменялся, и на одних машинах сейчас работают новые версии, тогда как на других продолжают использоваться старые. Сейчас происходит переход от версии IPv4 к версии IPv6. Он длится уже много лет, и не похоже, что скоро завершится (Durand, 2001; Wiljakka, 2002; Waddington и Chang, 2002). Некоторые даже считают, что это не произойдет никогда (Weiser, 2001). Что касается нумерации, то ничего странного в ней нет, просто в свое время существовал мало кому известный экспериментальный протокол реального масштаба времени IPv5.

Длина заголовка является переменной величиной, для хранения которой выделено поле *IHL* (информация в нем представлена в виде 32-разрядных слов). Минимальное значение длины (при отсутствии необязательного поля) равно 5. Максимальное значение этого 4-битового поля равно 15, что соответствует заголовку длиной 60 байт; таким образом, максимальный размер необязательного поля равен 40 байтам. Для некоторых приложений, например, для записи маршрута, по которому должен быть переслан пакет, 40 байт слишком мало. В данном случае дополнительное поле оказывается бесполезным.

Поле *Тип службы* — единственное поле, смысл которого с годами несколько изменился. Оно было (впрочем, и до сих пор) предназначено для различения классов обслуживания. Возможны разные комбинации надежности и скорости. Для оцифрованного голоса скорость доставки важнее точности. При передаче файла, наоборот, передача без ошибок важнее быстрой доставки.

Изначально 6-разрядное поле *Тип службы* состояло из трехразрядного поля *Precedence* и трех флагов — *D*, *T* и *R*. Поле *Precedence* указывало приоритет, от 0 (нормальный) до 7 (управляющий сетевой пакет). Три флаговых бита позволяли хосту указать, что беспокоит его сильнее всего, выбрав из набора {Delay, Throughput, Reliability} (Задержка, Пропускная способность, Надежность). Теоретически, эти поля позволяют маршрутизаторам выбрать, например, между спутниковой линией с высокой пропускной способностью и большой задержкой и выделенной линией с низкой пропускной способностью и небольшой задержкой. На практике сегодняшние маршрутизаторы часто вообще игнорируют поле *Тип службы*.

Поле *Полная длина* содержит длину всей дейтаграммы, включая как заголовок, так и данные. Максимальная длина дейтаграммы 65 535 байт. В настоящий момент этот верхний предел достаточен, однако с появлением гигабитных сетей могут понадобиться дейтаграммы большего размера.

Поле *Идентификатор* позволяет хосту-получателю определить, какой дейтаграмме принадлежат полученные им фрагменты. Все фрагменты одной дейтаграммы содержат одно и то же значение идентификатора.

Следом идет один неиспользуемый бит и два однобитных поля. Бит *DF* означает Don't Fragment (Не фрагментировать). Это команда маршрутизатору, запрещающая ему фрагментировать дейтаграмму, так как получатель не сможет восстановить ее из фрагментов. Например, при загрузке компьютера его ПЗУ может запросить образ памяти в виде единой дейтаграммы. Пометив дейтаграмму битом *DF*, отправитель гарантирует, что дейтаграмма дойдет единым блоком, даже если для ее доставки придется избегать сетей с маленьким размером пакетов. От всех машин требуется способность принимать фрагменты размером 576 байт и менее.

Бит *MF* означает More Fragments (Продолжение следует). Он устанавливается во всех фрагментах, кроме последнего. По этому биту получатель узнает о прибытии последнего фрагмента дейтаграммы.

Поле *Смещение фрагмента* указывает положение фрагмента в исходной дейтаграмме. Длина всех фрагментов в байтах, кроме длины последнего фрагмента, должна быть кратна 8. Так как на это поле выделено 13 бит, максимальное количество фрагментов в дейтаграмме равно 8192, что дает максимальную длину дейтаграммы 65 536 байт, на 1 байт больше, чем может содержаться в поле *Полная длина*.

Поле *Время жизни* представляет собой счетчик, ограничивающий время жизни пакета. Предполагалось, что он будет отсчитывать время в секундах, таким образом, допуская максимальное время жизни пакета в 255 с. На каждом маршрутизаторе это значение должно было уменьшаться как минимум на единицу плюс время стояния в очереди. Однако на практике этот счетчик просто считает

количество переходов через маршрутизаторы. Когда значение этого поля становится равным нулю, пакет отвергается, а отправителю отсылается пакет с предупреждением. Таким образом удается избежать вечного странствования пакетов, что может произойти, если таблицы маршрутизаторов по какой-либо причине испортятся.

Собрав дейтаграмму из фрагментов, сетевой уровень должен решить, что с ней делать. Поле *Протокол* сообщит ему, какому процессу транспортного уровня ее передать. Это может быть TCP, UDP или что-нибудь еще. Нумерация процессов глобально стандартизирована по всему Интернету. Номера протоколов вместе с некоторыми другими были сведены в RFC 1700, однако теперь доступна интернет-версия в виде базы данных, расположенной по адресу *www.iana.org.*

Поле *Контрольная сумма заголовка* защищает от ошибок только заголовок. Подобная контрольная сумма полезна для обнаружения ошибок, вызванных неисправными микросхемами памяти маршрутизаторов. Алгоритм вычисления суммы просто складывает все 16-разрядные полуслова в дополнительном коде, преобразуя результат также в дополнительный код. Таким образом, проверяемая получателем контрольная сумма заголовка (вместе с этим полем) должна быть равна нулю. Этот алгоритм надежнее, чем обычное суммирование. Обратите внимание на то, что значение *Контрольной суммы заголовка* должно подсчитываться заново на каждом транзитном участке, так как по крайней мере одно поле постоянно меняется (поле *Время жизни*). Для ускорения расчетов применяются некоторые хитрости.

Поля *Адрес отправителя* и *Адрес получателя* указывают номер сети и номер хоста. Интернет-адреса будут обсуждаться в следующем разделе. Поле *Необязательная часть* было создано для того, чтобы с появлением новых вариантов протокола не пришлось вносить в заголовок поля, отсутствующие в нынешнем формате. Оно же может служить пространством для различного рода экспериментов, испытания новых идей. Кроме того, оно позволяет не включать в стандартный заголовок редко используемую информацию. Размер поля *Необязательная часть* может варьироваться. В начале поля всегда располагается однобайтный идентификатор. Иногда за ним может располагаться также однобайтное поле длины, а затем один или несколько информационных байтов. В любом случае, размер поля *Необязательная часть* должен быть кратен 4 байтам. Изначально было определено пять разновидностей этого поля, перечисленных в табл. 5.6, однако с тех пор появилось несколько новых. Текущий полный список имеется в Интернете, его можно найти по адресу *www.iana.org/assignments/ip-parameters.*

Таблица 5.6. Некоторые типы необязательного поля IP-дейтаграммы

Тип	Описание
Безопасность	Указывает уровень секретности дейтаграммы
Свободная маршрутизация от источника	Задает список маршрутизаторов, которые нельзя миновать
Строгая маршрутизация от источника	Задает полный путь следования дейтаграммы
Запомнить маршрут	Требует от всех маршрутизаторов добавлять свой IP-адрес
Временной штамп	Требует от всех маршрутизаторов добавлять свой IP-адрес и текущее время

Параметр *Безопасность* указывает уровень секретности дейтаграммы. Теоретически, военный маршрутизатор может использовать это поле, чтобы запретить маршрутизацию дейтаграммы через территорию определенных государств. На практике все маршрутизаторы игнорируют этот параметр, так что его единственное практическое применение состоит в помощи шпионам в поиске ценной информации.

Параметр *Строгая маршрутизация от источника* задает полный путь следования дейтаграммы от отправителя до получателя в виде последовательности IP-адресов. Дейтаграмма обязана следовать именно по этому маршруту. Наибольшая польза этого параметра заключается в том, что с его помощью системный менеджер может послать экстренные пакеты, когда таблицы маршрутизатора повреждены, или замерить временные параметры сети.

Параметр *Свободная маршрутизация от источника* требует, чтобы пакет прошел через указанный список маршрутизаторов в указанном порядке, но при этом по пути он может проходить через любые другие маршрутизаторы. Обычно этот параметр указывает лишь небольшое количество маршрутизаторов. Например, чтобы заставить пакет, посылаемый из Лондона в Сидней, двигаться не на восток, а на запад, можно указать в этом параметре IP-адреса маршрутизаторов в Нью-Йорке, Лос-Анджелесе и Гонолулу. Этот параметр наиболее полезен, когда по политическим или экономическим соображениям следует избегать прохождения пакетов через определенные государства.

Параметр *Запомнить маршрут* требует от всех маршрутизаторов, встречающихся по пути следования пакета, добавлять свой IP-адрес к полю *Необязательная часть*. Этот параметр позволяет системным администраторам вылавливать ошибки в алгоритмах маршрутизации («Ну почему все пакеты, посылаемые из Хьюстона в Даллас, сначала попадают в Токио?»). Когда была создана сеть ARPANET, ни один пакет не проходил больше чем через девять маршрутизаторов, поэтому 40 байт для этого параметра было как раз достаточно. Как уже говорилось, сегодня размер поля *Необязательная часть* оказывается слишком мал.

Наконец, параметр *Временной штамп* действует полностью аналогично параметру *Запомнить маршрут*, но кроме 32-разрядного IP-адреса, каждый маршрутизатор записывает также 32-разрядную запись о текущем времени. Этот параметр также применяется в основном для отладки алгоритмов маршрутизации.

IP-адреса

У каждого хоста и маршрутизатора в Интернете есть IP-адрес, состоящий из номера сети и номера хоста. Эта комбинация уникальна: нет двух машин с одинаковыми IP-адресами. Все IP-адреса имеют длину 32 бита и используются в полях

Адрес отправителя и *Адрес получателя* IP-пакетов. Важно отметить, что IP-адрес, на самом деле, не имеет отношения к хосту. Он имеет отношение к сетевому интерфейсу, поэтому хост, соединенный с двумя сетями, должен иметь два IP-адреса. Однако на практике большинство хостов подключены к одной сети, следовательно, имеют один адрес.

В течение вот уже нескольких десятилетий IP-адреса делятся на пять классов, показанных на рис. 5.48. Такое распределение обычно называется **полноклассовой адресацией**. Сейчас такая адресация уже не используется, но ссылки на нее в литературе встречаются все еще довольно часто. Чуть позже мы обсудим то, что пришло ей на смену.

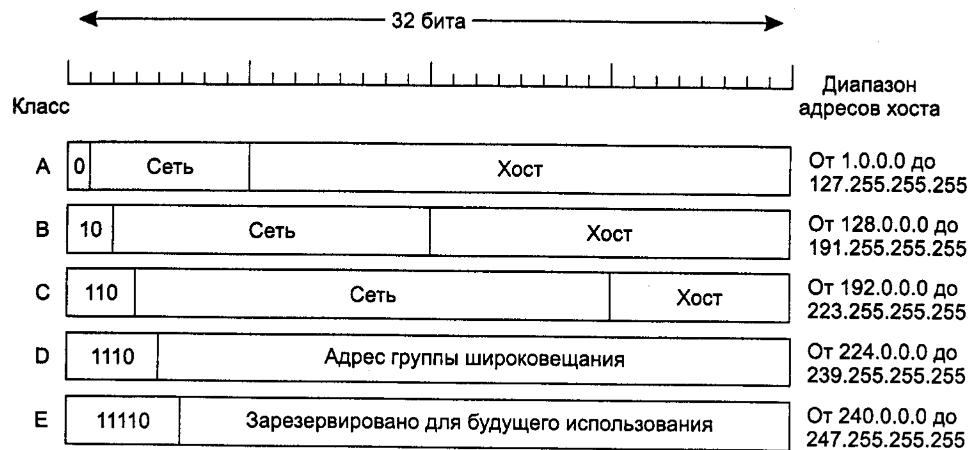


Рис. 5.48. Форматы IP-адреса

Форматы классов A, B, C и D позволяют задавать адреса до 128 сетей с 16 млн хостов в каждой, 16 384 сетей с 64 тысячами хостов или 2 миллионов сетей (например, ЛВС) с 256 хостами (хотя некоторые из них могут быть специализированными). Предусмотрен класс для многоадресной рассылки, при которой дейтаграммы рассылаются одновременно на несколько хостов. Адреса, начинающиеся с 1111, зарезервированы для будущего применения. В настоящее время к Интернету подсоединено более 500 000 сетей, и это число растет с каждым годом. Во избежание конфликтов, номера сетям назначаются некоммерческой **корпорацией по присвоению имен и номеров**, **ICANN** (Internet Corporation for Assigned Names and Numbers). В свою очередь, ICANN передала полномочия по присвоению некоторых частей адресного пространства региональным органам, занимающимся выделением IP-адресов провайдерам и другим компаниям.

Сетевые адреса, являющиеся 32-разрядными числами, обычно записываются в виде четырех десятичных чисел, которые соответствуют отдельным байтам, разделенных точками. Например, шестнадцатеричный адрес C0290614 записывается как 192.41.6.20. Наименьший IP-адрес равен 0.0.0.0, а наибольший — 255.255.255.255.

Как видно из рис. 5.49, числа 0 и –1 (единицы во всех разрядах) имеют особое назначение. Число 0 означает эту сеть или этот хост. Значение –1 используется для широковещания и означает все хосты указанной сети.

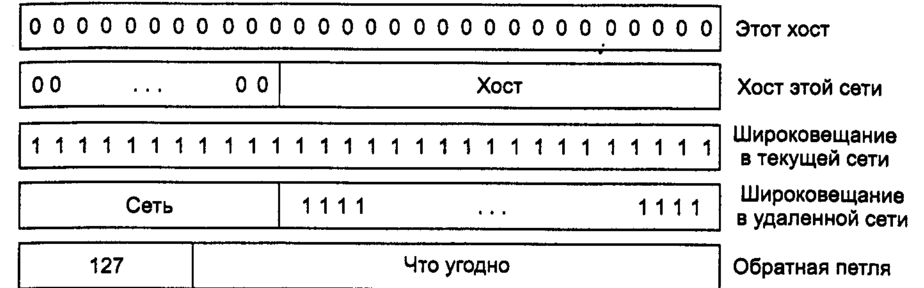


Рис. 5.49. Специальные IP-адреса

IP-адрес 0.0.0.0 используется хостом только при загрузке. IP-адреса с нулевым номером сети обозначают текущую сеть. Эти адреса позволяют машинам обращаться к хостам собственной сети, не зная ее номера (но они должны знать ее класс и количество используемых нулей). Адрес, состоящий только из единиц, обеспечивает широковещание в пределах текущей (обычно локальной) сети. Адреса, в которых указана сеть, но в поле номера хоста одни единицы, обеспечивают широковещание в пределах любой удаленной локальной сети, соединенной с Интернетом. Наконец, все адреса вида 127.xx.yy.zz зарезервированы для тестирования сетевого программного обеспечения методом обратной передачи. Отправляемые по этому адресу пакеты не попадают на линию, а обрабатываются локально как входные пакеты. Это позволяет пакетам перемещаться по локальной сети, когда отправитель не знает номера.

Подсети

Как было показано ранее, у всех хостов сети должен быть один и тот же номер сети. Это свойство IP-адресации может вызвать проблемы при росте сети. Например, представьте, что университет создал сеть класса B, используемую факультетом информатики в качестве Ethernet. Год спустя факультету электротехники понадобилось подключиться к Интернету, для чего был куплен повторитель для расширения сети факультета информатики и проложен кабель из его здания. Однако время шло, число компьютеров в сети росло, и четырех повторителей (максимальный предел для сети Ethernet) стало не хватать. Понадобилось создание новой архитектуры.

Получить второй сетевой адрес университету довольно сложно, потому что сетевые адреса — ресурс дефицитный, к тому же в одном сетевом адресе адресного пространства достаточно для подключения более 60 000 хостов. Проблема заключается в следующем: правилом установлено, что адрес одного класса (A, B или C) относится только к одной сети, а не к набору ЛВС. С этим столкнулось

множество организаций, в результате чего были произведены небольшие изменения в системе адресации.

Проблема решилась предоставлением сети возможности разделения на несколько частей с точки зрения внутренней организации. При этом с точки зрения внешнего представления сеть могла оставаться единой сущностью. Типичная сеть университетского городка в наши дни выглядит так, как показано на рис. 5.50. Здесь главный маршрутизатор соединен с провайдером или региональной сетью, а на каждом факультете может быть установлена своя локальная сеть Ethernet. Все сети Ethernet с помощью своих маршрутизаторов соединяются с главным маршрутизатором университетской сети (возможно, с помощью магистральной ЛВС, однако здесь важен сам принцип межмаршрутизаторной связи).

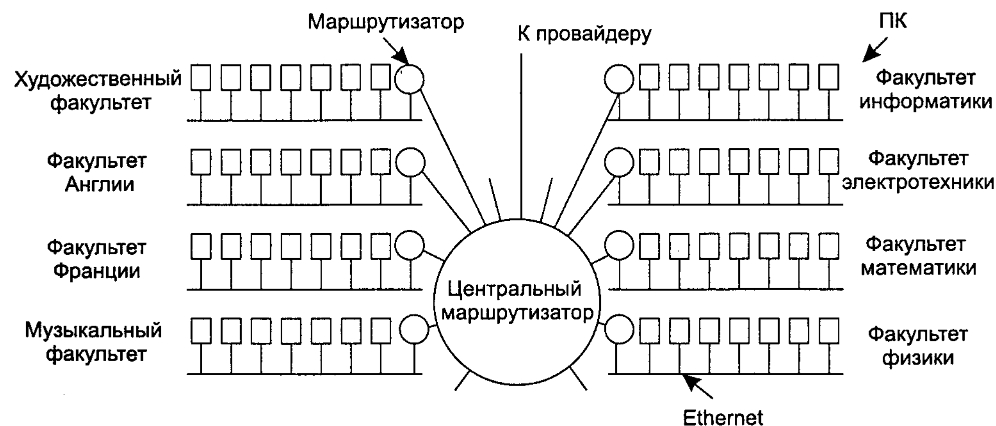


Рис. 5.50. Университетская сеть, состоящая из ЛВС разных факультетов

В литературе, посвященной интернет-технологиям, части сети называются **подсетями**. Как уже упоминалось в главе 1, подобное использование этого термина конфликтует со старым понятием «подсети», обозначающим множество всех маршрутизаторов и линий связи в сети. К счастью, по контексту обычно бывает ясно, какой смысл вкладывается в это слово. По крайней мере, в данном разделе «подсеть» будет употребляться только в новом значении.

Как центральный маршрутизатор узнает, в какую из подсетей (Ethernet) направить пришедший пакет? Одним из способов является поддержание маршрутизатором таблицы из 65 536 записей, говорящих о том, какой из маршрутизаторов использовать для доступа к каждому из хостов. Эта идея будет работать, но потребуется очень большая таблица и много операций по ее обслуживанию, выполняемых вручную, при добавлении, перемещении и удалении хостов.

Была изобретена альтернативная схема работы. Вместо одного адреса класса B с 14 битами для номера сети и 16 битами для номера хоста было предложено использовать несколько другой формат — формировать адрес подсети из нескольких битов. Например, если в университете существует 35 подразделений, то 6-битным номером можно кодировать подсети, а 10-битным — номера хостов. С помощью такой адресации можно организовать до 64 сетей Ethernet по 1022 хоста в каждой (адреса 0 и −1 не используются, как уже говорилось, поэтому не 1024 (2^{10}), а именно 1022 хоста). Такое разбиение может быть изменено, если окажется, что оно не очень подходит.

Все, что нужно маршрутизатору для реализации подсети, это наложить **маску подсети**, показывающую разбиение адреса на номер сети, подсети и хоста (рис. 5.51). Маски подсетей также записываются в виде десятичных чисел, разделенных точками, с добавлением косой черты, за которой следует число битов номера сети и подсети. Например, на рис. 5.51 маску подсети можно записать в виде 255.255.252.0. Альтернативная запись будет включать /22, показывая, что маска подсети занимает 22 бита.

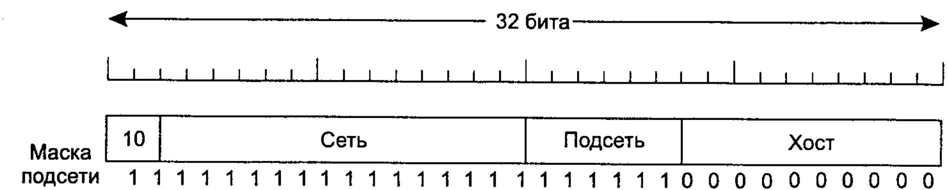


Рис. 5.51. Сеть класса B, разделенная на 64 подсети

За пределами сети разделение на подсети незаметно, поэтому нет нужды с появлением каждой подсети обращаться в ICANN или изменять какие-либо внешние базы данных. В данном примере первая подсеть может использовать IP-адреса, начиная с 130.50.4.1; вторая — начиная с 130.50.8.1; третья — 130.50.12.1, и т. д. Чтобы понять, почему на каждую подсеть уходит именно четыре единицы в адресе, вспомните двоичную запись этих адресов:

```
Подсеть 1: 10000010 00110010 000001|00 00000001
Подсеть 2: 10000010 00110010 000010|00 00000001
Подсеть 3: 10000010 00110010 000011|00 00000001
```

Здесь вертикальная черта (|) показывает границу номера подсети и хоста. Слева расположен 6-битный номер подсети, справа — 10-битный номер хоста.

Чтобы понять, как функционируют подсети, следует рассмотреть процесс обработки IP-пакетов маршрутизатором. У каждого маршрутизатора есть таблица, содержащая IP-адреса сетей (вида <*сеть*, 0>) и IP-адреса хостов (вида <*эта_сеть, хост*>). Адреса сетей позволяют получать доступ к удаленным сетям, а адреса хостов — обращаться к локальным хостам. С каждой таблицей связан сетевой интерфейс, применяющийся для получения доступа к пункту назначения, а также другая информация.

Когда IP-пакет прибывает на маршрутизатор, адрес получателя, указанный в пакете, ищется в таблице маршрутизации. Если пакет направляется в удаленную сеть, он пересылается следующему маршрутизатору по интерфейсу, указанному в таблице. Если пакет предназначен локальному хосту (например, в локальной сети маршрутизатора), он посылается напрямую адресату. Если номера сети, в которую посылается пакет, в таблице маршрутизатора нет, пакет пересылается маршрутизатору по умолчанию, с более подробными таблицами. Такой алгоритм

означает, что каждый маршрутизатор должен учитывать только другие сети и локальные хосты, а не пары <*сеть, хост*>, что значительно уменьшает размер таблиц маршрутизатора.

При разбиении сети на подсети таблицы маршрутизации меняются — добавляются записи вида <*эта_сеть, подсеть*, 0> и <*эта_сеть, эта_подсеть, хост*>. Таким образом, маршрутизатор подсети *k* знает, как получить доступ ко всем другим подсетям и как добраться до всех хостов своей подсети. Ему нет нужды знать детали адресации хостов в других подсетях. На самом деле, все, что для этого требуется от маршрутизатора, это выполнить двоичную операцию И над маской подсети, чтобы избавиться от номера хоста, а затем найти получившийся адрес в таблицах (после определения класса сети). Например, пакет, адресованный хосту с IP-адресом 130.50.15.6 и прибывающий на центральный маршрутизатор, после выполнения операции И с маской 255.255.252.0/22 получает адрес 130.50.12.0. Это значение ищется в таблицах маршрутизации, и с его помощью определяется выходная линия маршрутизатора к подсети 3. Итак, разбиение на подсети уменьшает объем таблиц маршрутизаторов путем создания трехуровневой иерархии, состоящей из сети, подсети и хоста.

CIDR — бесклассовая междоменная маршрутизация

В течение многих лет IP оставался чрезвычайно популярным протоколом. Он работал просто прекрасно, и основное подтверждение тому - экспоненциальный рост сети Интернет. К сожалению, протокол IP скоро стал жертвой собственной популярности: стало подходить к концу адресное пространство. Эта угроза вызвала бурю дискуссий и обсуждений в Интернет-сообществе. В этом разделе мы опишем как саму проблему, так и некоторые предложенные способы ее решения.

В теперь уже далеком 1987 году некоторые дальновидные люди предсказывали, что настанет день, когда в Интернете будет 100 000 сетей. Большинство экспертов посмеивались над этим и говорили, что если такое когда-нибудь и произойдет, то не раньше, чем через много десятков лет. Стотысячная сеть была подключена к Интернету в 1996 году. И, как уже было сказано, нависла угроза выхода за пределы пространства IP-адресов. В принципе, существует 2 миллиарда адресов, но на практике благодаря иерархической организации адресного пространства (см. рис. 5.48) это число сократилось на миллионы. В частности, одним из виновников этого является класс сетей В. Для большинства организаций класс А с 16 миллионами адресов — это слишком много, а класс С с 256 адресами — слишком мало. Класс В с 65 536 адресами — это то, что нужно. В интернет-фольклоре такая дилемма известна под названием **проблемы трех медведей** (вспомним *Машу и трех медведей*).

На самом деле и класс В слишком велик для большинства контор, которые устанавливают у себя сети. Исследования показали, что более чем в половине случаев сети класса В включают в себя менее 50 хостов. Безо всяких сомнений, всем этим организациям хватило бы и сетей класса С, однако почему-то все уверены, что в один прекрасный день маленькое предприятие вдруг разрастется настолько, что сеть выйдет за пределы 8-битного адресного пространства хостов. Сейчас, оглядываясь назад, кажется, что лучше было бы использовать в классе С

10-битную адресацию (до 1022 хостов в сети). Если бы это было так, то, возможно, большинство организаций приняло бы разумное решение устанавливать у себя сети класса С, а не В. Таких сетей могло бы быть полмиллиона, а не 16 384, как в случае сетей класса В.

Нельзя обвинять в создавшейся ситуации проектировщиков Интернета за то, что они не увеличили (или не уменьшили) адресное пространство сетей класса В. В то время, когда принималось решение о создании трех классов сетей, Интернет был инструментом научно-исследовательских организаций США (плюс несколько компаний и военных организаций, занимавшихся исследованиями с помощью сети). Никто тогда не предполагал, что Интернет станет коммерческой системой коммуникации общего пользования, соперничающей с телефонной сетью. Тогда кое-кто сказал, ничуть не сомневаясь в своей правоте: «В США около 2000 колледжей и университетов. Даже если все они подключатся к Интернету и к ним присоединятся университеты из других стран, мы никогда не превысим число 16 000, потому что высших учебных заведений по всему миру не так уж много. Зато мы будем кодировать номер хоста целым числом байт, что ускорит процесс обработки пакетов».

Даже если выделить 20 бит под адрес сети класса В, возникнет другая проблема: разрастание таблиц маршрутизации. С точки зрения маршрутизаторов, адресное пространство IP представляет собой двухуровневую иерархию, состоящую из номеров сетей и номеров хостов. Маршрутизаторы не обязаны знать номера вообще всех хостов, но им необходимо знать номера всех сетей. Если бы использовалось полмиллиона сетей класса С, каждому маршрутизатору пришлось бы хранить в таблице полмиллиона записей, по одной для каждой сети, в которых сообщалось бы о том, какую выходную линию использовать, чтобы добраться до той или иной сети, а также о чем-нибудь еще.

Физическое хранение полумиллиона строк таблицы, вероятно, выполнимо, хотя и дорого для маршрутизаторов, хранящих таблицы в статической памяти плат ввода-вывода. Более серьезная проблема состоит в том, что сложность обработки этих таблиц растет быстрее, чем сами таблицы, то есть зависимость между ними не линейная. Кроме того, большая часть имеющихся программных и программно-аппаратных средств маршрутизаторов разрабатывалась в те времена, когда Интернет объединял 1000 сетей, а 10 000 сетей казались отдаленным будущим. Методы реализации тех лет в настоящее время далеки от оптимальных.

Различные алгоритмы маршрутизации требуют от каждого маршрутизатора периодической рассылки своих таблиц (например, протоколов векторов расстояний). Чем больше будет размер этих таблиц, тем выше вероятность потери части информации по дороге, что будет приводить к неполноте данных и возможной нестабильности работы алгоритмов выбора маршрутов.

Проблема таблиц маршрутизаторов может быть решена при помощи увеличения числа уровней иерархии. Например, если бы каждый IP-адрес содержал поля страны, штата, города, сети и номера хоста. В таком случае каждому маршрутизатору нужно будет знать, как добраться до каждой страны, до каждого штата или провинции своей страны, каждого города своей провинции или штата и до

каждой сети своего города. К сожалению, такой подход потребует существенно более 32 бит для адреса, а адресное поле будет использоваться неэффективно (для княжества Лихтенштейн будет выделено столько же разрядов, сколько для Соединенных Штатов).

Таким образом, большая часть решений разрешает одну проблему, но взамен создает новую. Одним из решений, реализуемым в настоящий момент, является алгоритм маршрутизации **CIDR** (Classless InterDomain Routing — бесклассовая междоменная маршрутизация). Идея маршрутизации CIDR, описанной в RFC 1519, состоит в объединении оставшихся адресов в блоки переменного размера, независимо от класса. Если кому-нибудь требуется, скажем, 2000 адресов, ему выделяется блок из 2048 адресов на границе, кратной 2048 байтам.

Отказ от классов усложнил процесс маршрутизации. В старой системе, построенной на классах, маршрутизация происходила следующим образом. По прибытии пакета на маршрутизатор копия IP-адреса, извлеченного из пакета и сдвинутого вправо на 28 бит, давала 4-битный номер класса. С помощью 16-альтернативного ветвления пакеты рассортировывались на A, B, C и D (если этот класс поддерживался): восемь случаев было зарезервировано для A, четыре для B, два для C и по одному для D и E. Затем при помощи маскировки по коду каждого класса определялся 8-, 16- или 32-битный сетевой номер, который и записывался с выравниванием по правым разрядам в 32-битное слово. Сетевой номер отыскивался в таблице A, B или C, причем для A и B применялась индексация, а для C — хэш-функция. По найденной записи определялась выходная линия, по которой пакет и отправлялся в дальнейшее путешествие.

В CIDR этот простой алгоритм применить не удается. Вместо этого применяется расширение всех записей таблицы маршрутизации за счет добавления 32-битной маски. Таким образом, образуется единая таблица для всех сетей, состоящая из набора троек (IP-адрес, маска подсети, исходящая линия). Что происходит с приходящим пакетом при применении метода CIDR? Во-первых, извлекается IP-адрес назначения. Затем (концептуально) таблица маршрутизации сканируется запись за записью, адрес назначения маскируется и сравнивается со значениями записей. Может оказаться, что по значению подойдет несколько записей (с разными длинами масок подсети). В этом случае используется самая длинная маска. То есть если найдено совпадение для маски /20 и /24, то будет выбрана запись, соответствующая /24.

Был разработан сложный алгоритм для ускорения процесса поиска адреса в таблице (Ruiz-Sanchez и др., 2001). В маршрутизаторах, предполагающих коммерческое использование, применяются специальные чипы VLSI, в которые данные алгоритмы встроены аппаратно.

Чтобы лучше понять алгоритм маршрутизации, рассмотрим пример. Допустим, имеется набор из миллионов адресов, начиная с 194.24.0.0. Допустим также, что Кембриджскому университету требуется 2048 адресов, и ему выделяются адреса от 194.24.0.0 до 194.24.7.255, а также маска 255.255.248.0. Затем Оксфордский университет запрашивает 4096 адресов. Так как блок из 4096 адресов должен располагаться на границе, кратной 4096, то ему не могут быть выделены адреса начиная с 194.24.8.0. Вместо этого он получает адреса от 194.24.16.0 до

194.24.31.255 вместе с маской 255.255.240.0. Затем Эдинбургский университет просит выделить ему 1024 адреса и получает адреса от 194.24.8.0 до 194.24.11.255 и маску 255.255.252.0. Все эти присвоенные адреса и маски сведены в табл. 5.7.

Таблица 5.7. Набор присвоенных IP-адресов

Университет	Первый адрес	Последний адрес	Количество	Форма записи
Кембридж	194.24.0.0	194.24.7.255	2048	194.24.0.0/21
Эдинбург	194.24.8.0	94.24.11.255	1024	194.24.8.0/22
(Свободно)	194.24.12.0	94.24.15.255	1024	194.24.12.0/22
Оксфорд	194.24.16.0	94.24.16.255	4096	194.24.16.0/20

После этого таблицы маршрутизаторов по всему миру получают три новые строки, содержащие базовый адрес и маску. В двоичном виде эти записи выглядят так:

Адрес	Маска
К: 11000010 00011000 00000000 00000000	11111111 11111111 11111000 00000000
Э: 11000010 00011000 00001000 00000000	11111111 11111111 11111100 00000000
О: 11000010 00011000 00010000 00000000	11111111 11111111 11110000 00000000

Теперь посмотрим, что произойдет, когда пакет придет по адресу 194.24.17.4. В двоичном виде этот адрес представляет собой следующую 32-битную строку:

11000010 00011000 00010001 00000100

Сначала на него накладывается (выполняется логическое И) маска Кембриджа, в результате чего получается

11000010 00011000 00010000 00000000

Это значение не совпадает с базовым адресом Кембриджа, поэтому на оригинальный адрес накладывается маска Оксфорда, что дает в результате

11000010 00011000 00010000 00000000

Это значение совпадает с базовым адресом Оксфорда. Если далее по таблице совпадений нет, то пакет пересылается Оксфордскому маршрутизатору.

Теперь посмотрим на эту троицу университетов с точки зрения маршрутизатора в Омахе, штат Небраска, у которого есть всего четыре выходных линии: на Миннеаполис, Нью-Йорк, Даллас и Денвер. Получив три новых записи, маршрутизатор понимает, что он может скомпоновать их вместе и получить **одну агрегированную запись**, состоящую из адреса 194.24.0.0/19 и подмаски:

11000010 00000000 00000000 00000000 11111111 11111111 11100000 00000000

В соответствии с этой записью пакеты, предназначенные для любого из трех университетов, отправляются в Нью-Йорк. Объединив три записи, маршрутизатор в Омахе уменьшил размер своей таблицы на две строки.

В Нью-Йорке весь трафик Великобритании направляется по лондонской линии, поэтому там также используется агрегированная запись. Однако если имеются отдельные линии в Лондон и Эдинбург, тогда необходимы три отдельные записи. Агрегация часто используется в Интернете для уменьшения размеров таблиц маршрутизации.

И последнее замечание по данному примеру. Та же самая агрегированная запись маршрутизатора в Омахе используется для отправки пакетов по не присвоенным адресам в Нью-Йорк. До тех пор, пока адреса остаются не присвоенными, это не имеет значения, поскольку предполагается, что они вообще не встретятся. Однако если в какой-то момент этот диапазон адресов будет присвоен какой-нибудь калифорнийской компании, для его обработки понадобится новая запись: 194.24.12.0/22.

NAT — трансляция сетевого адреса

IP-адреса являются дефицитным ресурсом. У провайдера может быть /16-адрес (бывший класс B), дающий возможность подключить 65 534 хоста. Если клиентов становится больше, начинают возникать проблемы. Хостам, подключающимся к Интернету время от времени по обычной телефонной линии, можно выделять IP-адреса динамически, только на время соединения. Тогда один /16-адрес будет обслуживать до 65 534 *активных* пользователей, и этого, возможно, будет достаточно для провайдера, у которого несколько сотен тысяч клиентов. Когда сессия связи завершается, IP-адрес присваивается новому соединению. Такая стратегия может решить проблемы провайдеров, имеющих не очень большое количество частных клиентов, соединяющихся по телефонной линии, однако не поможет провайдерам, большую часть клиентуры которых составляют организации.

Дело в том, что корпоративные клиенты предпочитают иметь постоянное соединение с Интернетом, по крайней мере в течение рабочего дня. И в маленьких конторах, например туристических агенствах, состоящих из трех сотрудников, и в больших корпорациях имеются локальные сети, состоящие из некоторого числа компьютеров. Некоторые компьютеры являются рабочими станциями сотрудников, некоторые служат веб-серверами. В общем случае имеется маршрутизатор ЛВС, соединенный с провайдером по выделенной линии для обеспечения постоянного подключения. Такое решение означает, что с каждым компьютером целый день связан один IP-адрес. Вообще-то даже все вместе взятые компьютеры, имеющиеся у корпоративных клиентов, не могут перекрыть имеющиеся у провайдера IP-адреса. Для адреса длины /16 этот предел равен, как мы уже отмечали, 65 534. Однако если у поставщика услуг Интернета число корпоративных клиентов исчисляется десятками тысяч, то этот предел будет достигнут очень быстро.

Проблема усугубляется еще и тем, что все большее число частных пользователей желают иметь ADSL или кабельное соединение с Интернетом. Особенности этих способов заключаются в следующем: а) пользователи получают постоянный IP-адрес; б) отсутствует повременная оплата (взимается только ежемесячная абонентская плата). Пользователи такого рода услуг имеют постоянное подключение к Интернету. Развитие в данном направлении приводит к возрастанию дефицита IP-адресов. Присваивать IP-адреса «на лету», как это делается при телефонном подключении, бесполезно, потому что число активных адресов в каждый момент времени может быть во много раз больше, чем имеется у провайдера.

Часто ситуация еще больше усложняется за счет того, что многие пользователи ADSL и кабельного Интернета имеют дома два и более компьютера (например, по одному на каждого члена семьи) и хотят, чтобы все машины имели выход в Интернет. Что же делать — ведь есть только один IP-адрес, выданный провайдером! Решение таково: необходимо установить маршрутизатор и объединить все компьютеры в локальную сеть. С точки зрения провайдера, в этом случае семья будет выступать в качестве аналога маленькой фирмы с несколькими компьютерами. Добро пожаловать в корпорацию Васильевых!

Проблема дефицита IP-адресов отнюдь не теоретическая и отнюдь не относится к отдаленному будущему. Она уже актуальна, и бороться с ней приходится здесь и сейчас. Долговременный проект предполагает тотальный перевод всего Интернета на протокол IPv6 со 128-битной адресацией. Этот переход действительно постепенно происходит, но процесс идет настолько медленно, что затягивается на годы. Видя это, многие поняли, что нужно срочно найти какое-нибудь решение хотя бы на ближайшее время. Такое решение было найдено в виде метода **трансляции сетевого адреса, NAT** (Network Address Translation), описанного в RFC 3022. Суть его мы рассмотрим позже, а более подробную информацию можно найти в (Dutcher, 2001).

Основная идея трансляции сетевого адреса состоит в присвоении каждой фирме одного IP-адреса (или, по крайней мере, небольшого числа адресов) для интернет-трафика. *Внутри* фирмы каждый компьютер получает уникальный IP-адрес, используемый для маршрутизации внутреннего трафика. Однако как только пакет покидает пределы здания фирмы и направляется к провайдеру, выполняется трансляция адреса. Для реализации этой схемы было создано три диапазона так называемых частных IP-адресов. Они могут использоваться внутри компании по ее усмотрению. Единственное ограничение заключается в том, что пакеты с такими адресами ни в коем случае не должны появляться в самом Интернете. Вот эти три зарезервированных диапазона:

10.0.0.0	– 10.255.255.255/8	(16 777 216 хостов)
172.16.0.0	– 172.31.255.255/12	(1 048 576 хостов)
192.168.0.0	– 192.168.255.255/16	(65 536 хостов)

Итак, первый диапазон может обеспечить адресами 16 777 216 хостов (кроме 0 и –1, как обычно), и именно его обычно предпочитают компании, даже если им на самом деле столько внутренних адресов и не требуется.

Работа метода трансляции сетевых адресов показана на рис. 5.52. В пределах территории компании у каждой машины имеется собственный уникальный адрес вида 10.*x*.*y*.*z*. Тем не менее, когда пакет выходит за пределы владений компании, он проходит через **NAT-блок**, транслирующий внутренний IP-адрес источника (10.0.0.1 на рисунке) в реальный IP-адрес, полученный компанией от провайдера (198.60.42.12 для нашего примера). NAT-блок обычно представляет собой единое устройство с брандмауэром, обеспечивающим безопасность путем строго отслеживания входящего и исходящего трафика компании. Брандмауэры мы будем изучать отдельно в главе 8. NAT-блок может быть интегрирован и с маршрутизатором компании.

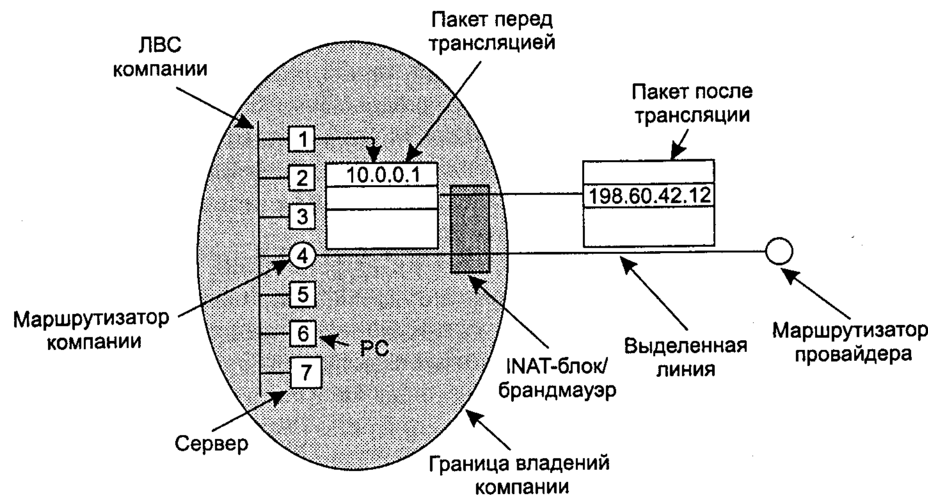


Рис. 5.52. Расположение и работа NAT-блока

Мы до сих пор обходили одну маленькую деталь: когда приходит ответ на запрос (например, от веб-сервера), он ведь адресуется 198.60.42.12. Как же NAT-блок узнает, каким внутренним адресом заменить общий адрес компании? Вот в этом и состоит главная проблема использования трансляции сетевых адресов. Если бы в заголовке IP-пакета было свободное поле, его можно было бы использовать для запоминания адреса того, кто посылал запрос. Но в заголовке остается неиспользованным всего один бит. В принципе, можно было бы создать такое поле для истинного адреса источника, но это потребовало бы изменения IP-кода на всех машинах по всему Интернету. Это не лучший выход, особенно если мы хотим найти быстрое решение проблемы нехватки IP-адресов.

На самом деле произошло вот что. Разработчики NAT подметили, что большая часть полезной нагрузки IP-пакетов – это либо TCP, либо UDP. Когда мы будем в главе 6 рассматривать TCP и UDP, мы увидим, что оба формата имеют заголовки, содержащие номера портов источника и приемника. Далее мы обсудим, что значит порт TCP, но надо иметь в виду, что с портами UDP связана точно такая же история. Номера портов представляют собой 16-разрядные целые числа, показывающие, где начинается и где заканчивается TCP-соединение. Место хранения номеров портов используется в качестве поля, необходимого для работы NAT.

Когда процесс желает установить TCP-соединение с удаленным процессом, он связывается со свободным TCP-портом на собственном компьютере. Этот порт становится **портом источника**, который сообщает TCP-коду информацию о том, куда направлять пакеты данного соединения. Процесс также определяет **порт назначения**. Посредством порта назначения сообщается, кому отдать пакет на удаленной стороне. Порты с 0 по 1023 зарезервированы для хорошо известных сервисов. Например, 80-й порт используется веб-серверами, соответственно, на них могут ориентироваться удаленные клиенты. Каждое исходящее сообщение TCP содержит информацию о порте источника и порте назначения. Вместе они служат для идентификации процессов на обоих концах, использующих соединение.

Проведем аналогию, которая несколько прояснит принцип использования портов. Допустим, у компании есть один общий телефонный номер. Когда люди набирают его, они слышат голос оператора, который спрашивает, с кем именно они хотели бы соединиться, и подключают их к соответствующему добавочному телефонному номеру. Основной телефонный номер является аналогией IP-адреса компании, а добавочные на обоих концах аналогичны портам. Для адресации портов используется 16-битное поле, которое идентифицирует процесс, получающий входящий пакет.

С помощью поля *Порт источника* мы можем решить проблему отображения адресов. Когда исходящий пакет приходит в NAT-блок, адрес источника вида 10.*x*.*y*.*z* заменяется настоящим IP-адресом. Кроме того, поле *Порт источника* TCP заменяется индексом таблицы перевода NAT-блока, содержащей 65 536 записей. Каждая запись содержит исходный IP-адрес и номер исходного порта. Наконец, пересчитываются и вставляются в пакет контрольные суммы заголовков TCP и IP. Необходимо заменять поле *Порт источника*, потому что машины с местными адресами 10.0.0.1 и 10.0.0.2 могут случайно пожелать воспользоваться одним и тем же портом (5000-м, например). Так что для однозначной идентификации процесса отправителя одного поля *Порт источника* оказывается недостаточно.

Когда пакет прибывает на NAT-блок со стороны провайдера, извлекается значение поля *Порт источника* заголовка TCP. Оно используется в качестве индекса таблицы отображения NAT-блока. По найденной в этой таблице записи определяются внутренний IP-адрес и настоящий *Порт источника* TCP. Эти два значения вставляются в пакет. Затем заново подсчитываются контрольные суммы TCP и IP. Пакет передается на главный маршрутизатор компании для нормальной доставки с адресом вида 10.*x*.*y*.*z*.

В случае применения ADSL или кабельного Интернета трансляция сетевых адресов может применяться для облегчения борьбы с нехваткой адресов. Присваиваемые пользователям адреса имеют вид 10.*x*.*y*.*z*. Как только пакет покидает пределы владений провайдера и уходит в Интернет, он попадает в NAT-блок, который преобразует внутренний адрес в реальный IP-адрес провайдера. На обратном пути выполняется обратная операция. В этом смысле для всего остального Интернета провайдер со своими клиентами, использующими ADSL и кабельное соединение, представляется в виде одной большой компании.

Хотя описанная выше схема частично решает проблему нехватки IP-адресов, многие приверженцы IP рассматривают NAT как некую заразу, распространяющуюся по Земле. И их можно понять.

Во-первых, сам принцип трансляции сетевых адресов никак не вписывается в архитектуру IP, которая подразумевает, что каждый IP-адрес уникальным образом идентифицирует только одну машину в мире. Вся программная структура Интернета построена на использовании этого факта. При трансляции сетевых адресов получается, что тысячи машин могут (и так происходит в действительности) иметь адрес 10.0.0.1.

Во-вторых, NAT превращает Интернет из сети без установления соединения в нечто подобное сети, ориентированной на соединение. Проблема в том, что NAT-блок должен поддерживать таблицу отображения для всех соединений, проходящих через него. Запоминать состояние соединения — дело сетей, ориентированных на соединение, но никак не сетей без установления соединений. Если NAT-блок ломается и теряются его таблицы отображения, то про все TCP-соединения, проходящие через него, можно забыть. При отсутствии трансляции сетевых адресов выход из строя маршрутизатора не оказывает никакого эффекта на деятельность TCP. Отправляющий процесс просто выжидает несколько секунд и посылает заново все неподтвержденные пакеты. При использовании NAT Интернет становится таким же восприимчивым к сбоям, как сеть с коммутацией каналов.

В-третьих, NAT нарушает одно из фундаментальных правил построения многоуровневых протоколов: уровень *k* не должен строить никаких предположений относительно того, что именно уровень *k* + 1 поместил в поле полезной нагрузки. Этот принцип определяет независимость уровней друг от друга. Если когда-нибудь на смену TCP придет TCP-2, у которого будет другой формат заголовка (например, 32-битная адресация портов), то трансляция сетевых адресов потерпит фиаско. Вся идея многоуровневых протоколов состоит в том, чтобы изменения в одном из уровней никак не могли повлиять на остальные уровни. NAT разрушает эту независимость.

В-четвертых, процессы в Интернете вовсе не обязаны использовать только TCP или UDP. Если пользователь машины *A* решит придумать новый протокол транспортного уровня для общения с пользователем машины *B* (это может быть сделано, например, для какого-нибудь мультимедийного приложения), то ему придется как-то бороться с тем, что NAT-блок не сможет корректно обработать поле *Порт источника* TCP.

В-пятых, некоторые приложения вставляют IP-адреса в текст сообщений. Получатель извлекает их оттуда и затем обрабатывает. Так как NAT не знает ничего про такой способ адресации, он не сможет корректно обработать пакеты, и любые попытки использования этих адресов удаленной стороной приведут к неудаче. **Протокол передачи файлов**, **FTP** (File Transfer Protocol), использует именно такой метод и может отказаться работать при трансляции сетевых адресов, если только не будут приняты специальные меры. Протокол интернет-телефонии H.323 (мы будем изучать его в главе 7) также обладает подобным свойством. Можно улучшить метод NAT и заставить его корректно работать с H.323, но невозможно же дорабатывать его всякий раз, когда появляется новое приложение.

В-шестых, поскольку поле *Порт источника* является 16-разрядным, то на один IP-адрес может быть отображено примерно 65 536 местных адресов машин. На самом деле это число несколько меньше: первые 4096 портов зарезервированы для служебных нужд. В общем, если есть несколько IP-адресов, то каждый из них может поддерживать до 61 440 местных адресов.

Эти и другие проблемы, связанные с трансляцией сетевых адресов, обсуждаются в RFC 2993. Обычно противники использования NAT говорят, что решение проблемы нехватки IP-адресов путем создания временной уродливой заплатки только мешает процессу настоящей эволюции, заключающемуся в переходе на IPv6. Поэтому NAT — это не добро, а зло для Интернета.

Управляющие протоколы Интернета

Помимо протокола IP, используемого для передачи данных, в Интернете есть несколько управляющих протоколов, применяемых на сетевом уровне, к которым относятся ICMP, ARP, RARP, BOOTP и DHCP. В данном разделе мы рассмотрим их все по очереди.

ICMP — протокол управляющих сообщений Интернета

За работой Интернета следят маршрутизаторы. Когда случается что-то неожиданное, о происшествии сообщается по протоколу **ICMP** (Internet Control Message Protocol — протокол управляющих сообщений Интернета), используемому также для тестирования Интернета. Протоколом ICMP определено около дюжины типов сообщений. Наиболее важные из них приведены в табл. 5.8. Каждое ICMP-сообщение вкладывается в IP-пакет.

Таблица 5.8. Основные типы ICMP-сообщений

Тип сообщения	Описание
Адресат недоступен	Пакет не может быть доставлен
Время истекло	Время жизни пакета упало до нуля
Проблема с параметром	Неверное поле заголовка
Гашение источника	Сдерживающий пакет
Переадресовать	Научить маршрутизатор географии
Запрос отклика	Спросить машину, жива ли она
Отклик	Да, я жива
Запрос временного штампа	То же, что и Запрос отклика, но с временным штампом
Отклик с временным штампом	То же, что и Отклик, но с временным штампом

Сообщение АДРЕСАТ НЕДОСТУПЕН используется, когда подсеть или маршрутизатор не могут обнаружить пункт назначения или когда пакет с битом *DF* (не фрагментировать) не может быть доставлен, так как путь ему преграждает сеть с маленьким размером пакетов.

Сообщение ВРЕМЯ ИСТЕКЛО посылается, когда пакет игнорируется, так как его счетчик уменьшился до нуля. Это событие является признаком того, что пакеты двигаются по замкнутым контурам, что имеется большая перегрузка или установлено слишком низкое значение таймера.

Сообщение ПРОБЛЕМА ПАРАМЕТРА указывает на то, что обнаружено неверное значение в поле заголовка. Это является признаком наличия ошибки в программном обеспечении хоста, отправившего этот пакет, или промежуточного маршрутизатора.

Сообщение ГАШЕНИЕ ИСТОЧНИКА ранее использовалось для усмирения хостов, которые отправляли слишком много пакетов. Хост, получивший такое сообщение,

должен был снизить обороты. В настоящее время подобное сообщение редко используется, так как при возникновении перегрузки подобные пакеты только подливают масла в огонь, еще больше загружая сеть. Теперь борьба с перегрузкой в Интернете осуществляется в основном на транспортном уровне. Это будет подробно обсуждаться в главе 6.

Сообщение ПЕРЕАДРЕСОВАТЬ посылается хосту, отправившему пакет, когда маршрутизатор замечает, что пакет адресован неверно.

Сообщения ЗАПРОС ОТКЛИКА и ОТКЛИК посылаются, чтобы определить, достижим ли и жив ли конкретный адресат. Получив сообщение ЗАПРОС ОТКЛИКА, хост должен отправить обратно сообщение ОТКЛИК. Сообщения ЗАПРОС ВРЕМЕННОГО ШТАМПА и ОТКЛИК С ВРЕМЕННЫМ ШТАМПОМ имеют то же назначение, но при этом в ответе проставляется время получения сообщения и время отправления ответа. Это сообщение используется для измерения производительности сети.

Кроме перечисленных сообщений, определены и другие. Их полный список хранится в Интернете по адресу www.iana.org/assignments/icmp-parameters.

ARP — протокол разрешения адресов

Хотя у каждой машины в Интернете есть один (или более) IP-адресов, они не могут использоваться для отправки пакетов, так как аппаратура уровня передачи данных не понимает интернет-адресов. В настоящее время большинство хостов соединены с локальными сетями с помощью интерфейсных карт, понимающих только адреса данной локальной сети. Например, каждая когда-либо выпущенная сетевая карта Ethernet имеет 48-разрядный Ethernet-адрес. Производители сетевых карт Ethernet запрашивают у центра блок адресов, что гарантирует уникальность Ethernet-адресов (это позволяет избежать конфликтов при наличии одинаковых сетевых карт в одной ЛВС). Сетевые карты отправляют и принимают кадры, основываясь на 48-разрядных Ethernet-адресах. О 32-разрядных IP-адресах им ничего не известно.

Таким образом, возникает вопрос: как устанавливается соответствие IP-адресов и адресов уровня передачи данных, таких как Ethernet-адреса? Чтобы понять, как это работает, рассмотрим показанный на рис. 5.53 пример, в котором изображен небольшой университет с несколькими сетями класса C (ныне называемыми сетями класса /24). На рисунке мы видим две сети Ethernet: одна на факультете кибернетики с IP-адресом 192.31.65.0, а другая — с IP-адресом 192.31.63.0 на электротехническом факультете. Они соединены кольцом FDDI с IP-адресом 192.31.60.0. У каждой машины сетей Ethernet есть уникальный Ethernet-адрес (на рисунке — от *E*1 до *E*6), а у каждой машины кольца FDDI есть FDDI-адрес (от *F*1 до *F*3).

Рассмотрим, как пользователь хоста 1 посылает пакет пользователю хоста 2. Допустим, отправителю известно имя получателя, например, mary@eagle.cs.uni.edu. Сначала надо найти IP-адрес для хоста 2, известного как eagle.cs.uni.edu. Этот поиск осуществляется службой имен доменов DNS (Domain Name System), которую мы рассмотрим в главе 7. На данный момент мы просто предположим, что служба DNS возвращает IP-адрес для хоста 2 (192.31.65.5).

Теперь программное обеспечение верхнего уровня хоста 1 создает пакет со значением 192.31.65.5 в поле *Адрес получателя* и передает его IP-программе для пересылки. Программное обеспечение протокола IP может посмотреть на адрес и увидеть, что адресат находится в его собственной сети, но ему нужно как-то определить Ethernet-адрес получателя. Одно из решений состоит в том, чтобы хранить в системе конфигурационный файл, в котором были бы перечислены соответствия всех локальных IP-адресов Ethernet-адресам. Такое решение, конечно, возможно, но в организациях с тысячами машин обновление этих файлов потребует много времени и подвержено ошибкам.

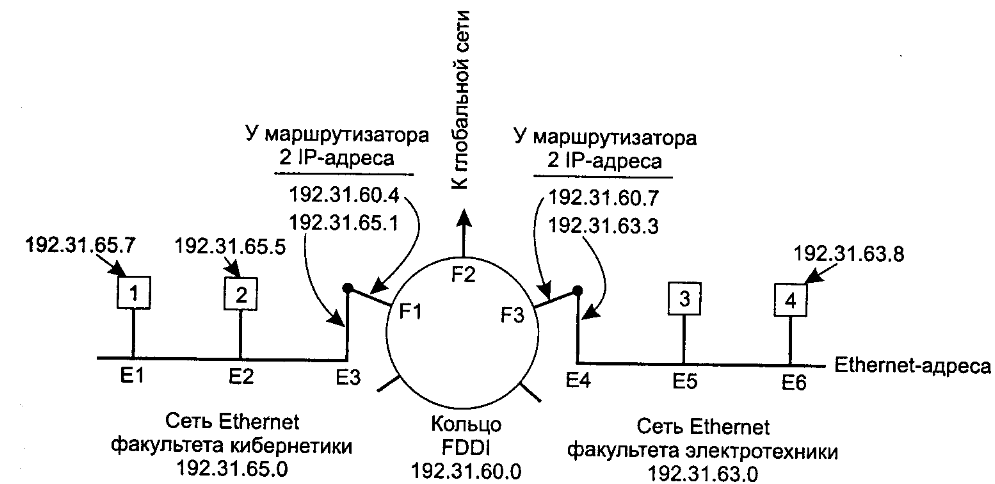


Рис. 5.53. Три объединенные сети класса /24: две сети Ethernet и кольцо FDDI

Более удачное решение заключается в рассылке хостом 1 по сети Ethernet широковещательного пакета с вопросом: «Кому принадлежит IP-адрес 192.31.65.5?» Этот пакет будет получен каждой машиной сети Ethernet 192.31.65.0, а хост 2 ответит на вопрос своим Ethernet-адресом *E*2. Таким образом, хост 1 узнает, что IP-адрес 192.31.65.5 принадлежит хосту с Ethernet-адресом *E*2. Протокол, который задает подобный вопрос и получает ответ на него, называется **ARP** (Address Resolution Protocol — протокол разрешения адресов) и описан в RFC 826. Он работает почти на каждой машине в Интернете.

Преимущество протокола ARP над файлами конфигурации заключается в его простоте. Системный администратор должен всего лишь назначить каждой машине IP-адрес и решить вопрос с маской подсети. Все остальное сделает протокол ARP.

Затем программное обеспечение протокола IP хоста 1 создает Ethernet-кадр для *E*2, помещает в его поле полезной нагрузки IP-пакет, адресованный 192.31.65.5, и посылает его по сети Ethernet. Сетевая карта Ethernet хоста 2 обнаруживает кадр, замечает, что он адресован ей, считывает его и вызывает прерывание. Ethernet-драйвер извлекает IP-пакет из поля полезной нагрузки и передает его IP-программе, которая, видя, что пакет адресован правильно, обрабатывает его.

Существуют различные методы повышения эффективности протокола ARP. Во-первых, машина, на которой работает протокол ARP, может запоминать результат преобразования адреса на случай, если ей придется снова связываться с той же машиной. В следующий раз она найдет нужный адрес в своем кэше, сэкономив, таким образом, на рассылке широковещательного пакета. Скорее всего, хосту 2 понадобится отослать ответ на пакет, что также потребует от него обращения к ARP для определения адреса отправителя. Этого обращения можно избежать, если отправитель включит в ARP-пакет свои IP- и Ethernet-адреса. Когда широковещательный ARP-пакет прибудет на хост 2, пара (192.31.65.7, *E*1) будет сохранена хостом 2 в ARP-кэше для будущего использования. Более того, эту пару адресов могут сохранить у себя все машины сети Ethernet.

Кроме того, каждая машина может рассылать свою пару адресов во время загрузки. Обычно эта широковещательная рассылка производится в виде ARP-пакета, запрашивающего свой собственный IP-адрес. Ответа на такой запрос быть не должно, но все машины могут запомнить эту пару адресов. Если ответ все же придет, это будет означать, что двум машинам назначен один и тот же IP-адрес. При этом вторая машина должна проинформировать системного администратора и прекратить загрузку.

Чтобы разрешить изменение соответствий адресов, например, при поломке и замене сетевой карты на новую (с новым Ethernet-адресом), записи в ARP-кэше должны устаревать за несколько минут.

Посмотрим снова на рис. 5.53. На этот раз хост 1 хочет послать пакет хосту 4 (192.31.63.8). Обращение к ARP не даст результата, так как хост 4 не увидит широковещательного пакета (маршрутизаторы не переправляют широковещательные пакеты Ethernet-уровня). Есть два решения данной проблемы. Во-первых, маршрутизатор факультета кибернетики можно настроить так, чтобы он отвечал на ARP-запросы к сети 192.31.63.0 (а также к другим местным сетям). В таком случае хост 1 добавит в свой ARP-кэш строку (192.31.63.8, E3) и будет счастлив посылать весь трафик для хоста 4 локальному маршрутизатору. Такой метод называется **ARP-прокси**. Второе решение состоит в том, чтобы хост 1 сразу выявлял нахождение адресата в удаленной сети и посылал весь внешний трафик по Ethernet-адресу, обрабатывающему все пакеты для удаленных адресатов, то есть по адресу маршрутизатора *E*3. В этом случае маршрутизатору факультета кибернетики не нужно знать, какую именно удаленную сеть он обслуживает.

В любом случае хост 1 помещает IP-пакет в поле полезной нагрузки Ethernet-кадра, адресованного маршрутизатору *E*3. Получив Ethernet-кадр, маршрутизатор факультета кибернетики извлекает из поля полезной нагрузки IP-пакет и ищет его IP-адрес в своих таблицах. Он обнаруживает, что пакеты, адресованные сети 192.31.63.0, должны пересылаться маршрутизатору 192.31.60.7. Если ему еще не известен FDDI-адрес маршрутизатора 192.31.60.7, то он посылает по кольцу широковещательный ARP-пакет и узнает, что нужный ему адрес *F*3. Затем он помещает IP-пакет в поле полезной нагрузки FDDI-кадра, адресованного маршрутизатору *F*3, и отправляет его по кольцу.

Когда кадр попадает на маршрутизатор факультета электротехники, FDDI-драйвер извлекает из поля полезной нагрузки IP-пакет и передает его IP-программе, которая понимает, что этот пакет следует переслать 192.31.63.8. Если такого IP-адреса еще нет в ARP-кэше, маршрутизатор посылает широковещательный ARP-запрос по сети Ethernet факультета электротехники и узнает, что нужный ему адрес принадлежит хосту *E*6, поэтому он создает Ethernet-кадр, адресованный хосту *E*6, помещает IP-пакет в поле полезной нагрузки и передает его по сети Ethernet. Получив Ethernet-кадр, хост 4 извлекает из поля полезной нагрузки IP-пакет и передает его IP-программе для обработки.

Пересылка хостом 1 пакетов в удаленную сеть по глобальной сети работает аналогично, с той разницей, что на этот раз маршрутизатор факультета кибернетики узнает из своих таблиц, что пакет следует переслать маршрутизатору глобальной сети, чей FDDI-адрес равен *F*2.

Протоколы RARP, BOOTP и DHCP

Протокол ARP решает проблему определения по заданному IP-адресу Ethernet-адреса хоста. Иногда бывает необходимо решить обратную задачу, то есть по заданному Ethernet-адресу определить IP-адрес. В частности, эта проблема возникает при загрузке бездисковой рабочей станции. Обычно такая машина получает двоичный образ своей операционной системы от удаленного файлового сервера. Но как ей узнать его IP-адрес?

Первым для решения проблемы был разработан протокол **RARP** (Reverse Address Resolution Protocol — протокол обратного определения адреса), описанный в RFC 903. Этот протокол позволяет только что загрузившейся рабочей станции разослать всем свой Ethernet-адрес и сказать: «Мой 48-разрядный Ethernet-адрес — 14.04.05.18.01.25. Знает ли кто-нибудь мой IP-адрес?» RARP-сервер видит этот запрос, ищет Ethernet-адрес в своих файлах конфигурации и посылает обратно соответствующий IP-адрес.

Использование протокола RARP лучше внедрения IP-адреса в образ загружаемой памяти, так как это позволяет использовать данный образ памяти для разных машин. Если бы IP-адреса хранились бы где-то в глубине образа памяти, каждой машине понадобился бы свой отдельный образ.

Недостаток протокола RARP заключается в том, что в нем для обращения к RARP-серверу используется адрес, состоящий из одних единиц (ограниченное широковещание). Однако эти широковещательные запросы не переправляются маршрутизаторами в другие сети, поэтому в каждой сети требуется свой RARP-сервер. Для решения данной проблемы был разработан альтернативный загрузочный протокол **BOOTP**. В отличие от RARP, он использует UDP-сообщения, пересылаемые маршрутизаторами в другие сети. Он также снабжает бездисковые рабочие станции дополнительной информацией, включающей IP-адрес файлового сервера, содержащего образ памяти, IP-адрес маршрутизатора по умолчанию, а также маску подсети. Протокол BOOTP описан в документах RFC 951, 1048 и 1084.

Серьезной проблемой, связанной с применением BOOTP, является то, что таблицы соответствия адресов приходится настраивать вручную. Когда к ЛВС подключается новый хост, протокол BOOTP невозможно использовать до тех пор, пока администратор сети не присвоит ему IP-адрес и не пропишет вручную в конфигурационных таблицах пару (Ethernet-адрес, IP-адрес). Для устранения

влияния этого фактора протокол BOOTP был изменен и получил новое имя: **DHCP** (Dynamic Host Configuration Protocol — протокол динамической настройки хостов). DHCP позволяет настраивать таблицы соответствия адресов как вручную, так и автоматически. Этот протокол описан в RFC 2131 и 2132. В большинстве систем он уже практически заменил RARP и BOOTP.

Подобно RARP и BOOTP, DHCP основан на идее специализированного сервера, присваивающего IP-адреса хостам, которые их запрашивают. Такой сервер не обязательно должен быть подключен к той же ЛВС, что и запрашивающий хост. Поскольку сервер DHCP может быть недоступен с помощью широковещательной рассылки, в каждой ЛВС должен присутствовать **агент ретрансляции**, как показано на рис. 5.54.

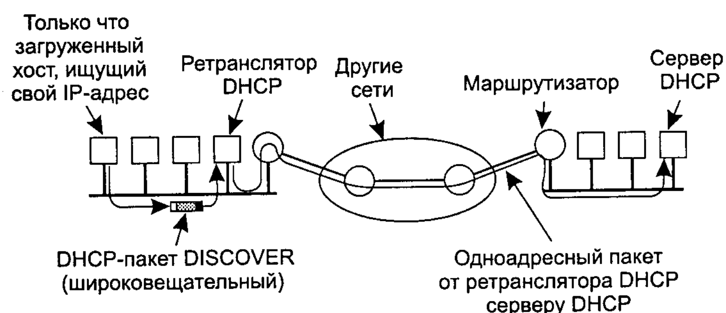


Рис. 5.54. Работа протокола DHCP

Для отыскания своего IP-адреса загружаемая машина широковещательным способом распространяет специальный пакет DISCOVER (Поиск). Агент ретрансляции DHCP перехватывает все широковещательные пакеты, относящиеся к протоколу DHCP. Обнаружив пакет DISCOVER, он превращает его из широковещательного в одноадресный и доставляет DHCP-серверу, который может находиться и в другой ЛВС. Агенту ретрансляции необходимо знать всего одну деталь: IP-адрес DHCP-сервера.

Встает вопрос: на какое время можно выдавать в автоматическом режиме IP-адреса из пула? Если хост покинет сеть и не освободит захваченный адрес, этот адрес будет навсегда утерян. С течением времени будет теряться все больше адресов. Для предотвращения этих неприятностей нужно выдавать IP-адреса не навсегда, а на определенное время. Такая технология называется **лизингом**. Перед окончанием срока действия лизинга хост должен послать на DHCP-сервер запрос о продлении срока пользования IP-адресом. Если такой запрос не был сделан или в просьбе было отказано, хост не имеет права продолжать использование выданного ранее адреса.

Протокол внутреннего шлюза OSPF

Итак, мы завершили изучение управляющих протоколов Интернета. Пришло время перейти к новой теме — маршрутизации в Интернете. Как уже упоминалось, Интернет состоит из большого количества автономных систем. Каждая автономная система управляется по-своему и может использовать внутри собственный алгоритм маршрутизации. Например, внутренние сети компаний X, Y и Z обычно рассматриваются как три автономные системы, если все они соединены с Интернетом. Они могут использовать различные внутренние алгоритмы маршрутизации. Тем не менее, наличие стандартов даже для внутренней маршрутизации упрощает реализацию на границах между автономными системами и позволяет повторно использовать программы. В данном разделе будет рассмотрена маршрутизация внутри автономной системы. В следующем разделе мы обсудим вопрос маршрутизации между автономными системами. Алгоритм маршрутизации внутри автономной системы называется **протоколом внутреннего шлюза**. Алгоритм маршрутизации между автономными системами называется **протоколом внешнего шлюза**.

Изначально в качестве протокола внутреннего шлюза в Интернете использовался протокол дистанционно-векторной маршрутизации RIP (Routing Information Protocol — протокол маршрутной информации), основанный на алгоритме Беллмана—Форда (Bellman—Ford) и унаследованный из ARPANET. Он хорошо работал в небольших системах, но по мере увеличения автономных систем стали проявляться его недостатки, такие как проблема счета до бесконечности и медленная сходимость, поэтому в мае 1979 года он был заменен протоколом состояния каналов. В 1988 году проблемная группа проектирования Интернета (IETF, Internet Engineering Task Force) начала работу над его преемником, которым в 1990 году стал алгоритм маршрутизации **OSPF** (Open Shortest Path First — открытый алгоритм предпочтительного выбора кратчайшего маршрута). В настоящее время он поддерживается многочисленными производителями маршрутизаторов и уже стал главным протоколом внутреннего шлюза. Далее будет дано краткое описание работы протокола OSPF. Более подробный рассказ о нем см. в RFC 2328.

Учитывая большой опыт работы с различными алгоритмами, группа разработчиков согласовывала свои действия с длинным списком требований, которые необходимо было удовлетворить.

Во-первых, этот алгоритм должен публиковаться в открытой литературе, отсюда буква «O» (Open — открытый) в OSPF. Из этого следовало, что патентованный алгоритм, принадлежащий одной компании, не годится.

Во-вторых, новый протокол должен был уметь учитывать широкий спектр различных параметров, включая физическое расстояние, задержку и т. д.

В-третьих, этот алгоритм должен был быть динамическим, а также автоматически и быстро адаптирующимся к изменениям топологии.

В-четвертых (это требование впервые было предъявлено именно к OSPF), он должен был поддерживать выбор маршрутов, основываясь на типе сервиса. Новый протокол должен был уметь по-разному выбирать маршрут для трафика реального времени и для других видов трафика. IP-пакет уже давно содержит поле *Тип службы*, но ни один из имевшихся протоколов маршрутизации не использовал его.

В-пятых, новый протокол должен был уметь распределять нагрузку на линии. Это связано с предыдущим пунктом. Большинство протоколов посылало все па-

кеты по одному лучшему маршруту. Следующий по оптимальности маршрут не использовался совсем. Между тем во многих случаях распределение нагрузки по нескольким линиям дает лучшую производительность.

В-шестых, необходима поддержка иерархических систем. К 1988 году Интернет вырос настолько, что ни один маршрутизатор не мог вместить сведения о его полной топологии. Таким образом, требовалась разработка нового протокола.

В-седьмых, требовался необходимый минимум безопасности, защищающий маршрутизаторы от обманывающих их студентов-шутников, присылающих неверную информацию о маршруте. Наконец, требовалась поддержка для маршрутизаторов, соединенных с Интернетом по туннелю. Предыдущие протоколы справлялись с этим неудовлетворительно.

Протокол OSPF поддерживает три следующих типа соединений и сетей:

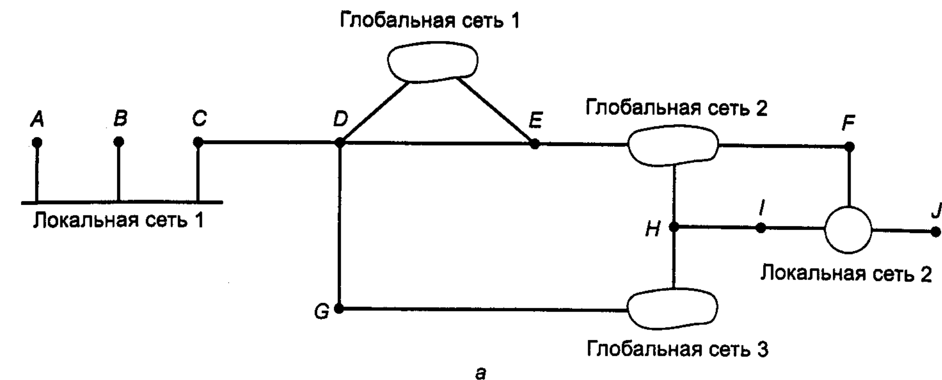
1. Двухточечные линии, соединяющие два маршрутизатора.
2. Сети множественного доступа с широковещанием (то есть большинство локальных сетей).
3. Сети множественного доступа без широковещания (то есть большинство глобальных сетей с коммутацией пакетов).

Сеть **множественного доступа** — это сеть, у которой может быть несколько маршрутизаторов, способных общаться друг с другом напрямую. Этим свойством обладают все локальные и глобальные сети. На рис. 5.55, *а* показана автономная система, содержащая все три типа сетей. Обратите внимание: хосты обычно не участвуют в отработке алгоритма OSPF.

В основе работы протокола OSPF лежит обобщенное представление о множестве сетей, маршрутизаторов и линий в виде направленного графа, в котором каждой дуге поставлена в соответствие ее цена (может выражаться в таких физических параметрах, как расстояние, задержка и т. д.). Затем, основываясь на весовых коэффициентах дуг, алгоритм вычисляет кратчайший путь. Последовательное соединение между двумя компьютерами представляется в виде пары дуг, по одной в каждом направлении. Их весовые коэффициенты могут быть различным. Сеть множественного доступа представляется в виде узла для самой сети, а также в виде узла для каждого маршрутизатора. Дуги, идущие от сетевого узла к узлам маршрутизатора, обладают нулевым весом и не включаются в граф.

Многие автономные системы в Интернете сами по себе довольно велики, и управлять ими непросто. Протокол OSPF позволяет делить их на пронумерованные **области**, то есть на сети или множества смежных сетей. Области не должны перекрываться, но не обязаны быть исчерпывающими, то есть некоторые маршрутизаторы могут не принадлежать ни одной области. Область является обобщением подсети. За пределами области ее топология и детали не видны.

У каждой автономной системы есть **магистральная область**, называемая областью 0. Все области соединены с магистралью, например, туннелями, так что по магистрали можно попасть из любой области автономной системы в ее любую другую область. Туннель представляется на графе в виде дуги и обладает определенной ценой. Каждый маршрутизатор, соединенный с двумя и более областями, является частью магистрали. Как и в случае других областей, топология магистрали за ее пределами не видна.



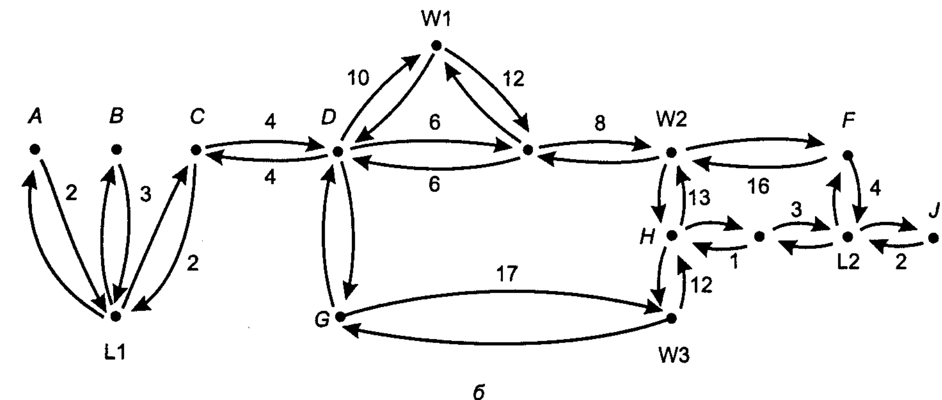


Рис. 5.55. Автономная система (*а*); представление *а* в виде графа (*б*)

У всех маршрутизаторов, принадлежащих к одной области, имеется одна и та же база данных состояния каналов и один алгоритм выбора кратчайшего пути. Работа маршрутизаторов заключается в расчете кратчайшего пути от себя до всех остальных маршрутизаторов этой области, включая маршрутизатор, соединенный с магистралью, который обязательно должен присутствовать в области, хотя бы один. Маршрутизатор, соединенный с двумя областями, должен иметь базы данных для каждой из них. Кратчайший путь для каждой области вычисляется отдельно.

При работе могут понадобиться три типа маршрутов: внутриобластные, межобластные и маршруты между автономными системами. Внутриобластные маршруты рассчитать легче всего, так как каждому маршрутизатору уже известен кратчайший путь до любого маршрутизатора своей области. Расчет межобластного маршрута состоит из трех этапов: от источника до магистрали, по магистрали до области назначения и от магистрали до адресата. Такой алгоритм приводит к конфигурации типа «звезда», в которой магистраль исполняет роль концентратора, а области являются лучами звезды. Пакеты направляются от отправителя к получателю в натуральном виде. Они не упаковываются в другие

пакеты и не туннелируются, кроме случаев, когда они направляются в области, с которыми магистраль соединена по туннелю. На рис. 5.56 показана часть Интернета с автономными системами и областями.

Протокол OSPF различает четыре класса маршрутизаторов:

1. Внутренние маршрутизаторы, расположенные целиком внутри области.
2. Маршрутизаторы границы области, соединяющие две и более областей.
3. Магистральные маршрутизаторы, находящиеся на магистрали.
4. Маршрутизаторы границы автономной системы, общающиеся с маршрутизаторами других автономных систем.

Эти классы могут перекрываться. Например, все пограничные маршрутизаторы автоматически являются магистральными. Кроме того, маршрутизатор, находящийся на магистрали, но не входящий ни в одну другую область, также является внутренним маршрутизатором. Примеры всех четырех классов показаны на рис. 5.56.

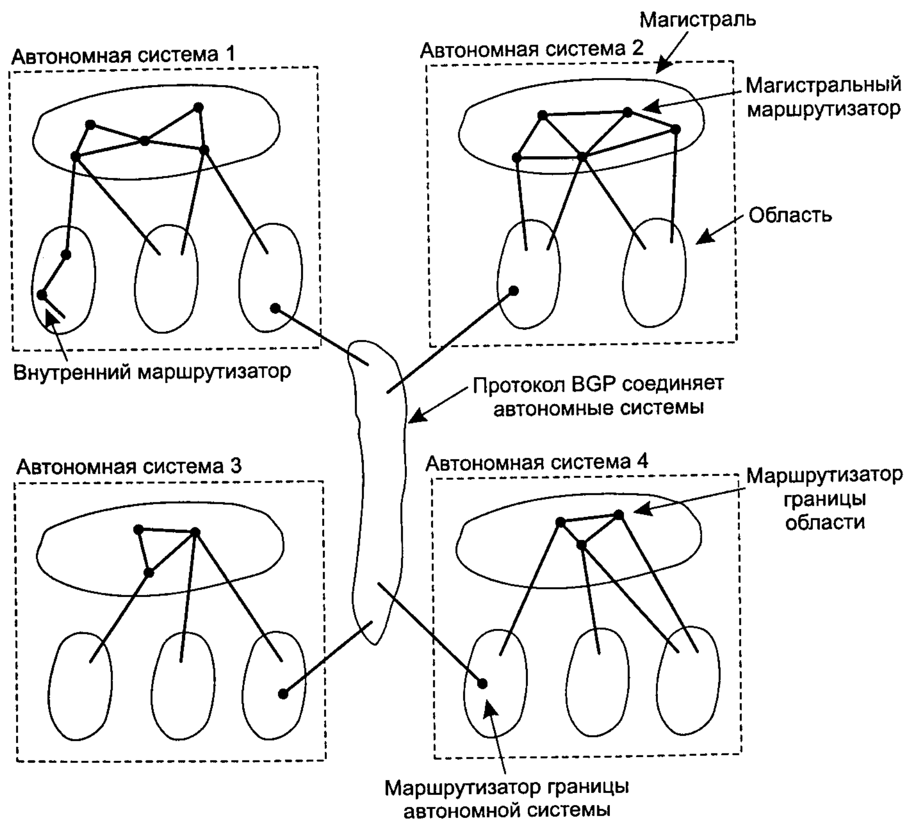


Рис. 5.56. Взаимосвязь между автономными системами, магистралями и областями в OSPF

При загрузке маршрутизатор рассылает сообщения HELLO по всем своим двухточечным линиям, производя многоадресную рассылку по локальным сетям для групп, состоящих из всех остальных маршрутизаторов. В глобальных сетях маршрутизатору требуется некая установочная информация, чтобы знать, с кем вступать в контакт. С помощью получаемых ответов каждый маршрутизатор знакомится со своими соседями.

Протокол OSPF работает при помощи обмена информацией между *смежными* маршрутизаторами, что не то же самое, что *соседние* маршрутизаторы. В частности, общение каждого маршрутизатора с каждым маршрутизатором локальной сети неэффективно. Поэтому один маршрутизатор выбирается **назначенным маршрутизатором**. Он считается **смежным** со всеми остальными маршрутизаторами и обменивается с ними информацией. Соседние маршрутизаторы, не являющиеся смежными, не обмениваются информацией друг с другом. На случай выхода из строя основного назначенного маршрутизатора всегда поддерживается в готовом состоянии запасной назначенный маршрутизатор.

При нормальной работе каждый маршрутизатор периодически рассылает методом заливки сообщение ОБНОВЛЕНИЕ СОСТОЯНИЯ КАНАЛОВ (LINK STATE UPDATE) всем своим смежным маршрутизаторам. Это сообщение содержит сведения о состоянии маршрутизатора и предоставляет информацию о цене, используемую в базах данных. В ответ на эти сообщения посылаются подтверждения, что повышает их надежность. Каждое сообщение получает последовательный номер, так что маршрутизатор может распознать, что новее: пришедшее сообщение или сообщение, хранимое им. Маршрутизаторы также рассылают эти сообщения, когда включается или выключается канал или изменяется его цена.

Сообщение ОПИСАНИЕ БАЗЫ ДАННЫХ (DATABASE DESCRIPTION) содержит порядковые номера всех записей о состоянии линий, которыми владеет отправитель. Сравнивая собственные значения со значениями отправителя, получатель может определить, у кого информация новее. Эти сообщения посылаются при восстановлении линии.

Каждый маршрутизатор может запросить информацию о состоянии линий у своего партнера с помощью сообщения ЗАПРОС О СОСТОЯНИИ КАНАЛА (LINK STATE REQUEST). В результате каждая пара смежных маршрутизаторов выясняет, чьи сведения являются более свежими, и, таким образом, по области распространяется наиболее новая информация. Все эти сообщения посылаются в виде IP-пакетов. Пять типов сообщений приведены в табл. 5.9.

Таблица 5.9. Пять типов сообщений протокола OSPF

Тип сообщения	Описание
Приветствие	Используется для знакомства с соседями
Обновление состояния каналов	Сообщает соседям информацию о каналах отправителя
Подтверждение состояния каналов	Подтверждает обновление состояния каналов
Описание базы данных	Сообщает о том, насколько свежей информацией располагает отправитель
Запрос состояния каналов	Запрашивает информацию у партнера

Подведем итоги. С помощью механизма заливки каждый маршрутизатор информирует все остальные маршрутизаторы своей области о своих соседях и цене каналов. Эта информация позволяет всем маршрутизаторам построить граф своей области и рассчитать кратчайшие пути. Маршрутизаторы магистральной области также занимаются этим. Кроме того, магистральные маршрутизаторы получают информацию от маршрутизаторов границ областей, с помощью которой они вычисляют оптимальные маршруты от каждого магистрального маршрутизатора до всех остальных маршрутизаторов. Эта информация рассылается обратно маршрутизаторам границ областей, которые распространяют ее в своих областях. С помощью этой информации маршрутизатор, собирающийся послать межобластной пакет, может выбрать оптимальный маршрутизатор, имеющий выход к магистрали.

Протокол внешнего шлюза BGP

В пределах одной автономной системы рекомендованным для применения в Интернете протоколом маршрутизации является OSPF (хотя он и не является единственным используемым протоколом). При выборе маршрута между различными автономными системами используется протокол **BGP** (Border Gateway Protocol — пограничный межсетевой протокол). Для выбора маршрута между различными автономными системами действительно требуется другой протокол, так как цели протоколов внутреннего и внешнего шлюзов различны. Задача протокола внутреннего шлюза ограничивается максимально эффективной передачей пакетов от отправителя к получателю. Политикой этот протокол не интересуется.

Протокол внешнего шлюза вынужден заниматься политикой (Metz, 2001). Например, корпоративной автономной системе может понадобиться возможность принимать и посылать пакеты на любой сайт Интернета. Однако прохождение через автономную систему пакета, отправитель и получатель которого находятся за пределами данного государства, может быть нежелательно, даже если кратчайший путь между отправителем и получателем пролегает через эту автономную систему («Это их заботы, а не наши»). С другой стороны, может оказаться желательным транзит трафика для других автономных систем, возможно, соседних, которые специально заплатили за эту услугу. Например, телефонные компании были бы рады оказывать подобные услуги, но только своим клиентам. Протоколы внешнего шлюза вообще и протокол BGP в частности разрабатывались для возможности учета различных стратегий при выборе маршрута между автономными системами.

Типичные стратегии выбора маршрутов учитывают политические, экономические факторы, а также соображения безопасности. К типичным примерам ограничений при выборе маршрутов относятся следующие:

1. Не пропускать трафик через определенные автономные системы.
2. Никогда не прокладывать через Ирак маршрут, начинающийся в Пентагоне.
3. Не использовать Соединенные Штаты при выборе маршрута из Британской Колумбии в штат Онтарио.

4. Прокладывать путь по Албании, только если нет альтернативных маршрутов.
5. Трафик, начинающийся или заканчивающийся в IBM®, не должен проходить через Microsoft®.

Стратегии настраиваются вручную на каждом BGP-маршрутизаторе (или представляют собой какой-нибудь скрипт). Они не являются частью протокола.

С точки зрения BGP-маршрутизатора, весь мир состоит из автономных систем и соединяющих их линий связи. Две автономные системы считаются соединенными, если есть общая линия между маршрутизаторами на их границах. Особая заинтересованность протокола BGP в транзитном трафике отразилась в разделении всех сетей на три категории. Первая категория представляет собой **тупиковые сети**, имеющие только одно соединение с BGP-графом. Они не могут использоваться для транзитного трафика, потому что на другой стороне ничего нет. Вторую категорию представляют **многосвязные сети**. Они могут применяться для транзитного трафика, если, только, конечно, согласятся на это. Наконец, имеются **транзитные сети** (например, магистрали), для которых транзитный трафик является желательным — возможно, с некоторыми ограничениями.

Пары BGP-маршрутизаторов общаются друг с другом, устанавливая TCP-соединения. Таким образом обеспечивается надежная связь и скрываются детали устройства сети, по которой проходит трафик.

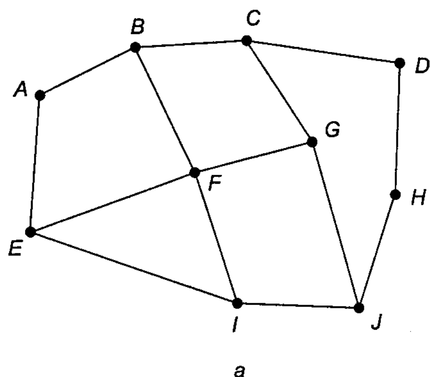
BGP по сути является протоколом маршрутизации по вектору расстояний, однако он значительно отличается от других подобных протоколов, например, протокола RIP (Routing Information Protocol — протокол маршрутной информации). Вместо того чтобы периодически сообщать всем своим соседям свои расчеты цены передачи до каждого возможного адресата, каждый BGP-маршрутизатор передает соседям точную информацию об используемых им маршрутах.

Для примера рассмотрим BGP-маршрутизаторы, показанные на рис. 5.57, *а*. В частности, рассмотрим таблицы маршрутизатора *F*. Предположим, что для доступа к маршрутизатору *D* он использует маршрут *FGCD*. Обмениваясь информацией о маршрутах, соседи сообщают друг другу полный используемый маршрут, как показано на рис. 5.57, *б* (для простоты показаны только пути к маршрутизатору *D*).

Получив все сведения о маршрутах от своих соседей, маршрутизатор *F* выбирает самый оптимальный из них. Он отбрасывает пути, используемые маршрутизаторами *I* и *E*, так как они проходят через маршрутизатор *F*. Таким образом, остается выбор между маршрутизаторами *B* и *G*. Каждый BGP-маршрутизатор содержит модуль, изучающий маршруты до каждого адресата и оценивающий расстояние до него. Каждый маршрут, нарушающий запрет политического характера, автоматически получает бесконечную оценку. Затем маршрутизатор принимает маршрут с кратчайшим расстоянием. Оценивающая функция не является частью протокола BGP, так что системный администратор может использовать любую оценивающую функцию.

Протокол BGP легко разрешает проблему счета до бесконечности, от которой страдают остальные алгоритмы дистанционно-векторной маршрутизации. Предположим, выходит из строя маршрутизатор *G* или линия *FG*. В этом случае маршрутизатор *F* получит информацию о маршрутах от своих трех оставшихся сосе-

дей. Этими маршрутами будут *BCD*, *IFGCD* и *EFGCD*. Второй и третий маршрут бесполезны для маршрутизатора *F*, так как они сами проходят через маршрутизатор *F*, поэтому в качестве нового пути к маршрутизатору *D* он выбирает маршрут *FBCD*. Другие алгоритмы дистанционно-векторной маршрутизации часто ошибаются, так как они не могут отличить соседей, обладающих независимыми маршрутами к адресату, от соседей, не обладающих такими маршрутами. Определение протокола BGP можно найти в RFC с 1771 по 1774.



Информация, которую маршрутизатор *F* получает от своих соседей о маршрутизаторе *D*

От *B*: «Я использую *BCD*»
От *G*: «Я использую *GCD*»
От *I*: «Я использую *IFGCD*»
От *E*: «Я использую *EFGCD*»

б

Рис. 5.57. Множество BGP-маршрутизаторов (*а*); информация, посланная маршрутизатору *F* (*б*)

Многоадресная рассылка в Интернете

Обычно IP-связь устанавливается между одним отправителем и одним получателем. Однако для некоторых приложений возможность послать сообщение одновременно большому количеству получателей является полезной. Такими приложениями могут быть, например, обновление реплицируемой распределенной базы данных, передача биржевых сводок брокерам и цифровые телеконференции (с участием нескольких собеседников).

Протокол IP поддерживает многоадресную рассылку при использовании адресов класса D. Каждый адрес класса D соответствует группе хостов. Для обозначения номера группы может быть использовано 28 бит, что делает возможным одновременное существование 250 миллионов групп. Когда процесс посылает пакет по адресу класса D, протокол прилагает максимальные усилия по его доставке всем членам группы, однако не дает гарантий доставки. Некоторые члены группы могут не получить пакета.

Поддерживаются два типа групповых адресов: постоянные и временные адреса. Постоянная группа не требует установки. У каждой постоянной группы есть постоянный адрес. Примерами постоянных групп являются:

- 224.0.0.1 — все системы локальной сети;
- 224.0.0.2 — все маршрутизаторы локальной сети;
- 224.0.0.5 — все OSPF-маршрутизаторы локальной сети;
- 224.0.0.6 — все назначенные OSPF-маршрутизаторы локальной сети.

Временные группы перед использованием следует создать. Процесс может попросить свой хост присоединиться к какой-либо группе. Когда последний процесс хоста покидает группу, группа более не присутствует на данном хосте. Каждый хост следит за тем, членами каких групп являются его процессы в текущий момент.

Многоадресная рассылка осуществляется специальными многоадресными маршрутизаторами, которые могут одновременно являться и стандартными маршрутизаторами. Примерно раз в минуту каждый многоадресный маршрутизатор совершает аппаратную (то есть на уровне передачи данных) многоадресную рассылку хостам на своей локальной сети (по адресу 224.0.0.1) с просьбой сообщить о группах, к которым в данный момент принадлежат их процессы. Каждый хост посылает обратно ответы для всех интересующих его адресов класса D.

Эти пакеты запросов и ответов используются протоколом **IGMP** (Internet Group Management Protocol — межсетевой протокол управления группами), являющимся грубым аналогом протокола ICMP (Internet Control Message Protocol — протокол контроля сообщений в сети Интернет). IGMP использует только два типа пакетов — запроса и ответа — фиксированного формата, содержащих управляющую информацию в первом слове поля полезной нагрузки и адрес класса D во втором. Этот формат описан в RFC 1112.

Многоадресная рассылка реализуется при помощи связующих деревьев. Каждый маршрутизатор многоадресной рассылки обменивается информацией со своими соседями с помощью модифицированного протокола дистанционно-векторной маршрутизации, что позволяет каждому построить для каждой группы связующее дерево, покрывающее всех членов группы. Для усечения дерева, чтобы исключить из него маршрутизаторы и сети, не являющиеся членами данной группы, применяются различные методы оптимизации. Чтобы миновать узлы сети, не являющиеся узлами связующего дерева, протокол использует туннелирование.

Мобильный IP

Многие пользователи Интернета обладают портативными компьютерами и заинтересованы в возможности оставаться в подключенном к сети состоянии, даже находясь в пути. К сожалению, система адресации IP такова, что реализовать это оказывается гораздо сложнее, чем кажется. В этом разделе мы познакомимся с этой проблемой и ее решением. Более подробное описание дано в (Perkins, 1998a).

Главным виновником проблемы является сама схема адресации. Каждый IP-адрес содержит номер сети и номер хоста. Например, рассмотрим машину с IP-адресом 160.80.40.20/16. Здесь 160.80 означает номер сети (8272 в десятичной системе счисления), 40.20 является номером хоста (10 260 в десятичной системе). Маршрутизаторы по всему миру содержат таблицы, в которых указывается, какую линию следует использовать, чтобы попасть в сеть 160.80. Когда приходит пакет с IP-адресом получателя вида 160.80.xxx.yyy, он отправляется по этой линии.

Если вдруг машина с этим адресом перевозится куда-то со своего места, адресованные ей пакеты будут продолжать направляться по ее домашнему адресу в ее локальную сеть (или ее маршрутизатору). До машины перестанет доходить электронная почта и т. п. Предоставление же машине нового адреса, соответствующего ее новому расположению, является нежелательным, так как об этом изменении придется информировать большое количество людей, программ и баз данных.

Другой подход состоит в использовании маршрутизаторами полного IP-адреса для определения маршрута, а не только полей класса и номера сети. Однако при такой стратегии каждый маршрутизатор должен будет содержать таблицы из миллионов записей, и стоимость поддержания Интернета в работоспособном состоянии составит астрономические суммы.

Когда потребность в мобильных хостах значительно возросла, проблемная группа проектирования Интернета (IETF, Internet Engineering Task Force) создала рабочую группу для поиска решения проблемы. Созданная рабочая группа быстро сформулировала набор целей, которых хотелось бы достичь, независимо от способа решения. Основными целями были признаны следующие:

1. Каждый мобильный хост должен иметь возможность использовать свой домашний IP-адрес где угодно.
2. Изменения программного обеспечения фиксированных хостов недопустимы.
3. Изменения программного обеспечения и таблиц маршрутизаторов недопустимы.
4. Большая часть пакетов, направляемых мобильным хостам, должны доставляться напрямую.
5. Не должно быть никаких дополнительных расходов, когда мобильный хост находится дома.

Рабочая группа выработала решение, описанное в разделе «Многоадресная рассылка». Суть его, напомним, заключалась в том, что везде, где требуется предоставить возможность перемещения в пространстве, следует создать внутреннего агента. Везде, где нужно принимать посетителей, следует создать внешнего агента. Когда мобильный хост прибывает на новое место, он связывается с местным внешним агентом и регистрируется. Затем внешний агент связывается с внутренним агентом пользователя и сообщает ему адрес для передачи сообщений прибывшему хосту. Обычно это IP-адрес внешнего агента.

Когда пакет прибывает в домашнюю локальную сеть пользователя, его получает маршрутизатор, соединенный с этой локальной сетью. При этом маршрутизатор пытается определить расположение хоста обычным способом, с помощью широковещательной рассылки ARP-пакета, спрашивая, например: «Каков Ethernet-адрес хоста 160.80.40.20?» Внутренний агент отвечает на этот запрос, выдавая свой собственный Ethernet-адрес. Маршрутизатор пересылает пакеты для 160.80.40.20 внутреннему агенту. Тот, в свою очередь, упаковывает их в поле данных IP-пакета, который туннелирует пакеты внешнему агенту. Внешний агент извлекает их и отсылает по адресу уровня передачи данных мобильного хоста. Внутренний агент также сообщает отправителю новый адрес мобильного хоста, так что последующие пакеты могут быть туннелированы напрямую внешнему агенту. Это решение удовлетворяет всем перечисленным выше требованиям.

Следует, пожалуй, отметить одну небольшую деталь. Когда мобильный хост перемещается, у маршрутизатора, скорее всего, остается в памяти его Ethernet-адрес (который скоро станет недействительным). Чтобы заменить этот адрес адресом внутреннего агента, применяется хитрость, называемая **добровольным ARP-сообщением**. Это особое сообщение, предоставляемое маршрутизатору по инициативе хоста, которое заставляет маршрутизатор заменить в своей таблице запись о хосте, собирающемся покинуть свое место. Когда позднее мобильный хост возвращается, то же сообщение используется для повторного изменения памяти маршрутизатора.

Ничто не мешает мобильному хосту быть собственным внешним агентом, но такой подход будет работать только в том случае, когда мобильный хост (в качестве внешнего агента) логически связан с Интернетом на своем месте. Также он должен получить (временный) IP-адрес в текущей сети.

Решение, предложенное проблемной группой IETF, разрешает ряд других, еще не упомянутых проблем с мобильными хостами. Например, как обнаружить агента? Для этого агент периодически рассылает широковещательным способом свой адрес и тип услуг, которые он предоставляет (то есть пишет о том, кто он: внутренний агент, внешний агент или и то, и другое). Прибыв на новое место, хост может просто подождать рассылки этих широковещательных пакетов, называемых **рекламными объявлениями**. В качестве альтернативы он может сам разослать методом широковещания пакет с объявлением о своем прибытии и надеяться, что местный внешний агент на него отзовется.

Еще одна проблема состоит в том, что делать с невежливыми мобильными хостами, которые уходят не попрощавшись. Для решения этой проблемы регистрация хоста считается действительной только в течение ограниченного интервала времени. Если она периодически не обновляется, то считается устаревшей, после чего внешний агент может удалить запись о прибывшем хосте из своих таблиц.

Еще одним вопросом является безопасность. Когда внутренний агент получает просьбу пересылать все пакеты, приходящие на имя Натальи, на некий IP-адрес, он не должен подчиняться, пока он не убедится, что источником этого запроса является Наталья, а не кто-то пытающийся выдать себя за Наталью. Для этого применяются протоколы криптографической аутентификации, которые будут рассматриваться в главе 8.

Наконец, поговорим еще об одном вопросе, связанном с уровнями мобильности. Представьте себе самолет с установленной на борту сетью Ethernet, используемой навигационными и авиационными компьютерами. В этой сети есть стандартный маршрутизатор, общающийся с обычным стационарным Интернетом на земле по радиосвязи. В один прекрасный день кому-нибудь приходит в голову идея установить Ethernet-разъемы во всех подлокотниках кресел, так чтобы пассажиры с мобильными компьютерами могли подключаться к сети.

Таким образом, мы получаем два уровня мобильности: компьютеры самолета, неподвижные относительно сети Ethernet, и компьютеры пассажиров, являю-

щиеся мобильными относительно нее. Кроме того, бортовой маршрутизатор является мобильным относительно наземных маршрутизаторов. Мобильность относительно системы, которая сама является мобильной, может поддерживаться при помощи рекурсивного туннелирования.

Протокол IPv6

Хотя системы адресации CIDR и NAT и могут помочь нынешнему IP продержаться еще несколько лет, всем понятно, что дни IPv4 сочтены. Помимо перечисленных ранее технических проблем, имеется еще одна, угрожающе вырастающая на горизонте. До недавних пор Интернетом пользовались в основном университеты, высокотехнологичные предприятия и правительственные организации (особенно Министерство обороны). С лавинообразным ростом интереса к Интернету, начавшимся в середине 90-х годов, в третьем тысячелетии, скорее всего, им будет пользоваться гораздо большее количество пользователей с принципиально разными требованиями. Во-первых, пользователи портативных компьютеров могут пользоваться Интернетом для доступа к домашним базам данных. Во-вторых, при неминуемом сближении компьютерной промышленности, средств связи и индустрии развлечений, возможно, очень скоро каждый телевизор планеты станет узлом Интернета, что в результате приведет к появлению миллиардов машин, используемых для видео по заказу. В таких обстоятельствах становится очевидно, что протокол IP должен эволюционировать и стать более гибким.

Предвидя появление этих проблем, проблемная группа проектирования Интернета IETF начала в 1990 году работу над новой версией протокола IP, в которой никогда не должна возникнуть проблема нехватки адресов, а также будут решены многие другие проблемы. Кроме того, новая версия протокола должна была быть более гибкой и эффективной. Были сформулированы следующие основные цели:

1. Поддержка миллиардов хостов даже при неэффективном использовании адресного пространства.
2. Уменьшение размера таблиц маршрутизации.
3. Упрощение протокола для ускорения обработки пакетов маршрутизаторами.
4. Более надежное обеспечение безопасности (аутентификации и конфиденциальности), чем в нынешнем варианте IP.
5. Необходимость обращать больше внимания на тип сервиса, в частности, при передаче данных реального времени.
6. Упрощение работы многоадресных рассылок с помощью указания областей рассылки.
7. Возможность изменения положения хоста без необходимости изменять его адрес.
8. Возможность дальнейшего развития протокола в будущем.
9. Возможность сосуществования старого и нового протоколов в течение нескольких лет.

Чтобы найти протокол, удовлетворяющий всем этим требованиям, IETF издал в RFC 1550 приглашение к дискуссиям и предложениям. Был получен двадцать один ответ. Далеко не все варианты содержали предложения, полностью удовлетворяющие этим требованиям. В декабре 1992 года были рассмотрены семь серьезных предложений. Их содержание варьировалось от небольших изменений в протоколе IP до полного отказа от него и замены совершенно другим протоколом.

Одно из предложений состояло в использовании вместо IP протокола CLNP, который с его 160-разрядным адресом обеспечивал бы достаточное адресное пространство на веки вечные. Кроме того, это решение объединило бы два основных сетевых протокола. Однако все же сочли, что при подобном выборе придется признать, что кое-что в мире OSI было сделано правильно, что было бы политически некорректным в интернет-кругах. Протокол CLNP, на самом деле, очень мало отличается от протокола IP. Окончательный выбор был сделан в пользу протокола, отличающегося от IP значительно сильнее, нежели CLNP. Еще одним аргументом против CLNP была его слабая поддержка типа сервиса, требовавшегося для эффективной передачи мультимедиа.

Три лучших предложения были опубликованы в журнале *IEEE Network Magazine* (Deering, 1993; Francis, 1993; Katz и Ford, 1993). После долгих обсуждений, переработок и борьбы за первое место была выбрана модифицированная комбинированная версия Диринга (Deering) и Фрэнсиса (Francis), называемая в настоящий момент протоколом **SIPP** (Simple Internet Protocol Plus — простой интернет-протокол Плюс). Новому протоколу было дано обозначение **IPv6** (протокол IPv5 уже использовался в качестве экспериментального протокола потоков реального времени).

Протокол IPv6 прекрасно справляется с поставленными задачами. Он обладает достоинствами протокола IP и лишен некоторых его недостатков (либо они проявляются в меньшей степени), к тому же наделен некоторыми новыми особенностями. В общем случае протокол IPv6 несовместим с протоколом IPv4, но зато совместим со всеми остальными протоколами Интернета, включая TCP, UDP, ICMP, IGMP, OSPF, BGP и DNS, для чего иногда требуются небольшие изменения (в основном чтобы работать с более длинными адресами). Основные особенности протокола IPv6 обсуждаются далее. Дополнительные сведения о нем можно найти в RFC с 2460 по 2466.

Прежде всего, у протокола IPv6 поля адресов длиннее, чем у IPv4. Они имеют длину 16 байт, что решает основную проблему, поставленную при разработке протокола, — обеспечить практически неограниченный запас интернет-адресов. Мы еще кратко упомянем об адресах чуть позднее.

Второе заметное улучшение протокола IPv6 по сравнению с IPv4 состоит в более простом заголовке пакета. Он состоит всего из 7 полей (вместо 13 у протокола IPv4). Таким образом, маршрутизаторы могут быстрее обрабатывать пакеты, что повышает производительность. Краткое описание заголовков будет приведено далее.

Третье усовершенствование заключается в улучшенной поддержке необязательных параметров. Подобное изменение действительно было существенным,

так как в новом заголовке требуемые прежде поля стали необязательными. Кроме того, изменился способ представления необязательных параметров, что упростило для маршрутизаторов пропуск не относящихся к ним параметров и ускорило обработку пакетов.

В-четвертых, протокол IPv6 демонстрирует большой шаг вперед в области безопасности. У проблемной группы проектирования Интернета IETF была полная папка вырезок из газет с сообщениями о том, как 12-летние мальчишки с помощью своего персонального компьютера по Интернету вломились в банк или военную базу. Было ясно, что надо как-то улучшить систему безопасности. Аутентификация и конфиденциальность являются ключевыми чертами нового IP-протокола.

Наконец, в новом протоколе было уделено больше внимания типу предоставляемых услуг. Для этой цели в заголовке пакета IPv4 было отведено 8-разрядное поле (на практике не используемое), но при ожидаемом росте мультимедийного трафика в будущем требовалось значительно больше разрядов.

Основной заголовок IPv6

Заголовок IPv6 показан на рис. 5.58. Поле *Версия* содержит число 6 для IPv6 (и 4 для IPv4). На период перехода с IPv4 на IPv6, который, вероятно, займет около десяти лет, маршрутизаторы по значению этого поля смогут различать пакеты нового и старого стандарта. Подобная проверка потребует нескольких тактов процессора, что может оказаться нежелательным в некоторых ситуациях, поэтому многие реализации, вероятно, попытаются избежать этих накладных расходов и будут отличать пакеты IPv4 от пакетов IPv6 с помощью некоторого поля в заголовке уровня передачи данных. При этом пакеты будут передаваться напрямую нужному сетевому уровню. Однако знакомство уровня передачи данных с типами пакетов сетевого уровня полностью нарушает принцип разделения протоколов на уровни, при котором каждый уровень не должен знать назначения битов из пакетов более высокого уровня. Дискуссия между лагерями, руководствующимися принципами «Делай правильно» и «Делай быстро», несомненно, будет долгой и энергичной.

Поле *Класс трафика* используется для того, чтобы различать пакеты с разными требованиями к доставке в реальном времени. Такое поле присутствовало в стандарте IP с самого начала, однако оно реально обрабатывалось маршрутизаторами лишь в единичных случаях. Сейчас проводятся эксперименты, направленные на то, чтобы определить, как лучше всего использовать это поле для передачи данных в реальном времени.

Поле *Метка потока* также пока является экспериментальным, но будет применяться для установки между отправителем и получателем псевдосоединения с определенными свойствами и требованиями. Например, поток пакетов между двумя процессами на разных хостах может обладать строгими требованиями к задержкам, что потребует резервирования пропускной способности. Поток устанавливается заранее и получает идентификатор. Когда прибывает пакет с отличным от нуля содержимым поля *Метка потока*, все маршрутизаторы смотрят в свои таблицы, чтобы определить, какого рода особая обработка ему требуется.

Таким образом, новый протокол пытается соединить достоинства подсетей различных типов: гибкость дейтаграмм и гарантии виртуальных каналов.

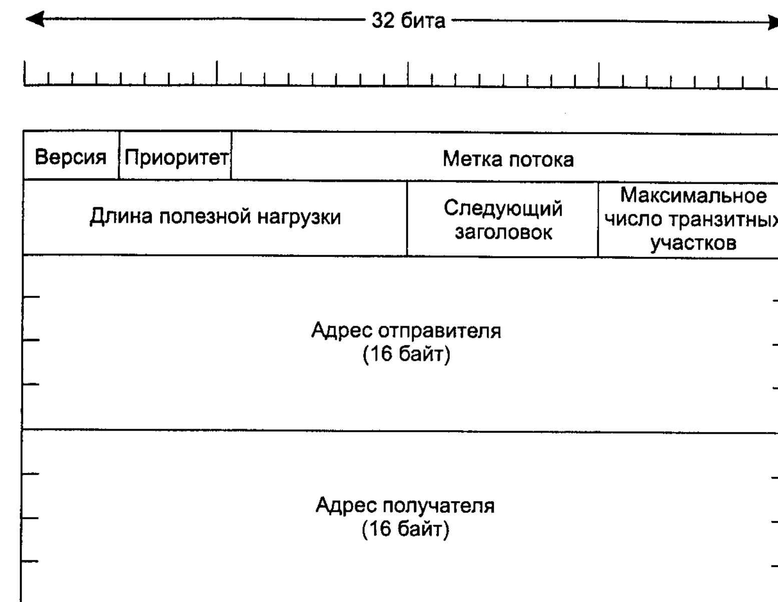


Рис. 5.58. Фиксированный заголовок IPv6 (обязательные поля)

Каждый поток описывается адресом источника, адресом назначения и номером потока, так что для каждой пары IP-адресов можно создать много активных потоков. Два потока с одинаковыми номерами, но различными адресами отправителя или получателя считаются разными потоками и различаются маршрутизаторами по адресам. Ожидается, что номера каналов будут выбираться случайным образом, а не назначаться подряд начиная с 1, что облегчит маршрутизаторам их распознавание.

Поле *Длина полезной нагрузки* сообщает, сколько байт следует за 40-байтовым заголовком, показанным на рис. 5.58. В заголовке IPv4 аналогичное поле называлось *Полная длина* и определяло весь размер пакета. В новом протоколе 40 байт заголовка учитываются отдельно.

Поле *Следующий заголовок* раскрывает секрет возможности использования упрощенного заголовка. Дело в том, что после обычного 40-байтового заголовка могут идти дополнительные (необязательные) расширенные заголовки. Это поле сообщает, какой из шести дополнительных заголовков (на текущий момент) следует за основным. В последнем IP-заголовке поле *Следующий заголовок* сообщает, какой обрабатывающей программе протокола транспортного уровня (то есть TCP или UDP) передать пакет.

Поле *Максимальное число транзитных участков* не дает пакетам вечно блуждать по сети. Оно имеет практически то же назначение, что и поле *Время жизни* в заголовке протокола IPv4. Это поле уменьшается на единицу на каждом тран-

зитном участке. Теоретически, в протоколе IPv4 это поле должно было содержать секунды времени жизни пакета, однако ни один маршрутизатор не использовал его подобным образом, поэтому имя поля было приведено в соответствие способу его применения.

Следом идут поля *Адрес отправителя* и *Адрес получателя*. В исходном предложении Диринга (протоколе SIPP) использовались 8-байтовые адреса, но при рассмотрении проекта было решено, что 8-байтовых адресов хватит лишь на несколько десятилетий, в то время как 16-байтовых адресов должно хватить навечно. Другие возражали, что 16 байтов для адресов слишком много, тогда как третьи настаивали на 20-байтных адресах для совместимости с дейтаграммным протоколом OSI. Еще одна фракция ратовала за адреса переменной длины. После продолжительных споров было решено, что наилучшим компромиссным решением являются 16-байтовые адреса фиксированной длины.

Для написания 16-байтовых адресов была выработана новая нотация. Адреса в IPv6 записываются в виде восьми групп по четыре шестнадцатеричных цифры, разделенных двоеточиями, например:

8000:0000:0000:0000:0123:4567:89AB:CDEF

Поскольку многие адреса будут содержать большое количество нулей, были разрешены три метода сокращенной записи адресов. Во-первых, могут быть опущены ведущие нули в каждой группе, например, 0123 можно записывать как 123. Во-вторых, одна или более групп, полностью состоящих из нулей, могут заменяться парой двоеточий. Таким образом, приведенный выше адрес принимает вид

8000::123:4567:89AB:CDEF

Наконец, адреса IPv4 могут записываться как пара двоеточий, после которой пишется адрес в старом десятичном формате, например:

::192.31.20.46

Возможно, нет необходимости говорить об этом столь подробно, но количество всех возможных 16-байтовых адресов очень велико — 2^{128}, что приблизительно равно $3 \cdot 10^{38}$. Если покрыть компьютерами всю планету, включая сушу и океаны, то протокол IPv6 позволит иметь около $7 \cdot 10^{23}$ IP-адресов на квадратный метр. Кто изучал химию, может заметить, что это число больше числа Авогадро. Хотя в планы разработчиков не входило предоставление собственного IP-адреса каждой молекуле на поверхности Земли, они оказались не так уж далеко от обеспечения такой услуги.

На практике не все адресное пространство используется эффективно, как, например, не используются абсолютно все комбинации телефонных номеров. Например, телефонные номера Манхэттена (код 212) почти полностью заняты, тогда как в штате Вайоминг (код 307) они почти не используются. В RFC 3194 Дьюранд (Durand) и Хуйтема (Huitema) приводят свои вычисления. Утверждается, что если ориентироваться на использование телефонных номеров, то даже при самом пессимистическом сценарии все равно получается более 1000 IP-адресов на квадратный метр поверхности Земли (включая как сушу, так и море).

При любом более вероятном сценарии обеспечиваются триллионы адресов на квадратный метр. Таким образом, маловероятно, что в обозримом будущем обнаружится нехватка адресов. Также следует отметить, что на сегодня только для 28 % адресного пространства придуманы применения. Остальные 72 % зарезервированы на будущее.

Полезно сравнить заголовок IPv4 (рис. 5.47) с заголовком IPv6 (рис. 5.58), чтобы увидеть, что осталось от старого стандарта. Поле *IHL* исчезло, так как заголовок IPv6 имеет фиксированную длину. Поле *Протокол* также было убрано, поскольку поле *Следующий заголовок* сообщает, что следует за последним IP-заголовком (то есть UDP- или TCP-сегмент).

Были удалены все поля, относящиеся к фрагментации, так как в протоколе IPv6 используется другой подход к фрагментации. Во-первых, все хосты, поддерживающие протокол IPv6, должны динамически определять нужный размер дейтаграммы. Это правило делает фрагментацию маловероятной. Во-вторых, минимальный размер пакета был увеличен с 576 до 1280, чтобы можно было передавать 1024 байт данных, плюс множество заголовков. Кроме того, когда хост посылает слишком большой IPv6-пакет вместо того, чтобы его фрагментировать, то маршрутизатор, не способный переслать пакет дальше, посылает обратно сообщение об ошибке. Получив это сообщение, хост должен прекратить всю передачу этому адресату. Гораздо правильнее будет научить все хосты посылать пакеты требуемого размера, нежели учить маршрутизаторы фрагментировать их на лету.

Наконец, поле *Контрольная сумма* было удалено, так как ее подсчет значительно снижает производительность. Поскольку в настоящее время все шире используются надежные линии связи, а на уровне передачи данных и на транспортном уровне подсчитываются свои контрольные суммы, наличие еще одной контрольной суммы не стоило бы тех затрат производительности, которых требовал бы ее подсчет. В результате перечисленных удалений получился простой, быстрый и в то же время гибкий протокол сетевого уровня с огромным адресным пространством.

Дополнительные заголовки

В опущенных полях заголовка иногда возникает необходимость, поэтому в протоколе IPv6 была представлена новая концепция (необязательного) **дополнительного заголовка**. На сегодня определены шесть типов дополнительных заголовков, которые перечислены в табл. 5.9. Все они являются необязательными, но в случае использования более чем одного дополнительного заголовка они должны располагаться сразу за фиксированным заголовком, желательно в указанном порядке.

Таблица 5.9. Дополнительные заголовки IPv6

Дополнительный заголовок	Описание
Параметры маршрутизации	Разнообразная информация для маршрутизаторов
Параметры получателя	Дополнительная информация для получателя
Маршрутизация	Частичный список транзитных маршрутизаторов на пути пакета

продолжение

Таблица 5.9 (*продолжение*)

Дополнительный заголовок	Описание
Фрагментация	Управление фрагментами дейтаграмм
Аутентификация	Проверка подлинности отправителя
Шифрованные данные	Информация о зашифрованном содержимом

У некоторых заголовков формат фиксированный, другие содержат переменное количество полей переменной длины. Для них каждый пункт кодируется в виде тройки (Тип, Длина, Значение). *Тип* представляет собой однобайтовое поле, содержащее код параметра. Первые два бита этого поля сообщают, что делать с пакетом, маршрутизаторам, не знающим, как обрабатывать данный параметр. Возможны четыре следующих варианта: пропустить параметр, игнорировать пакет, игнорировать пакет и отослать обратно ICMP-пакет, а также то же самое, что и предыдущий вариант, но не отсылать обратно ICMP-пакет в случае многоадресной рассылки (чтобы один неверный многоадресный пакет не породил миллионы ICMP-донесений).

Поле *Длина* также имеет размер 1 байт. Оно сообщает, насколько велико значение (от 0 до 255 байт). Поле *Значение* содержит необходимую информацию, размером до 255 байт.

Заголовок параметров маршрутизации содержит информацию, которую должны исследовать маршрутизаторы на протяжении всего пути следования пакета. Пока что был определен один вариант использования этого параметра: поддержка дейтаграмм, превышающих 64 Кбайт. Формат заголовка показан на рис. 5.59.

Следующий заголовок	0	194	0
Длина полезной нагрузки			

Рис. 5.59. Дополнительный заголовок для больших дейтаграмм

Как и все дополнительные заголовки, он начинается с байта, означающего тип следующего заголовка. Следующий байт содержит длину дополнительного заголовка в байтах, не считая первых 8 байт, являющихся обязательными. С этого начинаются все расширения.

Следующие два байта указывают, что данный параметр содержит размер дейтаграммы (код 194) в виде 4-байтового числа. Размеры меньше 65 536 не допускаются, так как могут привести к тому, что первый же маршрутизатор проигнорирует данный пакет и отошлет обратно ICMP-сообщение об ошибке. Дейтаграммы, использующие подобные расширения заголовка, называются **джамбограммами** (от слова «jumbo», означающего нечто большое и неуклюжее). Использование джамбограмм важно для суперкомпьютерных приложений, которым необходимо эффективно передавать по Интернету гигабайты данных.

Маршрутный заголовок содержит информацию об одном или нескольких маршрутизаторах, которые следует посетить по пути к получателю. Это очень

сильно напоминает свободную маршрутизацию стандарта IPv4 тем, что указанные в списке маршрутизаторы должны быть пройдены строго по порядку, тогда как не указанные проходятся между ними. Формат маршрутного заголовка показан на рис. 5.60.

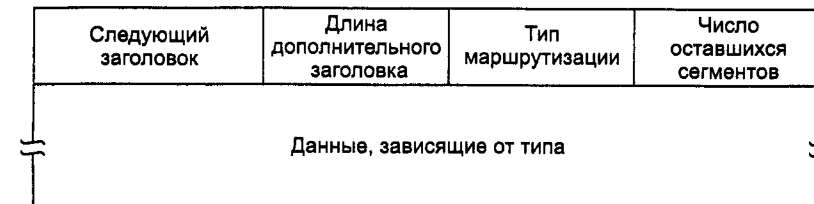


Рис. 5.60. Дополнительный заголовок для маршрутизации

Первые четыре байта дополнительного маршрутного заголовка содержат четыре однобайтовых целых числа. Поля *Следующий заголовок* и *Длина дополнительного заголовка* были описаны ранее. В поле *Тип маршрутизации* описывается формат оставшейся части заголовка. Если он равен 0, это означает, что далее следует зарезервированное 32-разрядное слово, а за ним — некоторое число адресов IPv6. В будущем, возможно, будут по мере необходимости изобретаться какие-то новые поля. Наконец, в поле *Число оставшихся сегментов* указывается, сколько адресов из списка еще осталось посетить. Его значение уменьшается при прохождении каждого адреса. Когда оно достигает нуля, пакет оставляется на произвол судьбы — никаких указаний относительно его дальнейшего маршрута не дается. Обычно в этот момент пакет уже находится достаточно близко к месту назначения, и оптимальный маршрут очевиден.

Заголовок фрагментации определяет фрагментацию способом, схожим с протоколом IPv4. Заголовок содержит идентификатор дейтаграммы, номер фрагмента и бит, информирующий о том, является ли этот фрагмент последним. В отличие от IPv4, в протоколе IPv6 фрагментировать пакет может только хост-источник. Маршрутизаторы фрагментировать пересылаемые пакеты не могут. Это порывающее с философией прошлого изменение в протоколе упрощает и ускоряет работу маршрутизаторов. Как уже было сказано, маршрутизатор отвергает слишком большие пакеты, посылая в ответ ICMP-пакет, указывающий хосту-источнику на необходимость заново передать пакет, выполнив его фрагментацию на меньшие части.

Заголовок аутентификации предоставляет механизм подтверждения подлинности отправителя пакета. Шифрование данных, содержащихся в поле полезной нагрузки, обеспечивает конфиденциальность: прочесть содержимое пакета сможет только тот, для кого предназначен пакет. Для выполнения этой задачи в заголовках используются криптографические методы.

Полемика

При той открытости, с которой происходил процесс разработки протокола IPv6, и при убежденности многочисленных разработчиков в собственной правоте не-

удивительно, что многие решения принимались в условиях весьма жарких дискуссий. О некоторых из них будет рассказано далее. Все кровавые подробности описаны в соответствующих RFC.

О спорах по поводу длины поля адреса уже упоминалось. В результате было принято компромиссное решение: 16-байтовые адреса фиксированной длины.

Другое сражение разгорелось из-за размера поля *Максимальное количество транзитных участков*. Один из противостоящих друг другу лагерей считал, что ограничение количества транзитных участков числом 255 (это явно следует из использования 8-битного поля) является большой ошибкой. В самом деле, маршруты из 32 транзитных участков уже стали обычными, а через 10 лет могут стать обычными более длинные маршруты. Сторонники этого лагеря заявляли, что использование полей адресов огромного размера было решением дальновидным, а применение крохотных счетчиков транзитных участков — недальновидным. Самый страшный грех, который, по их мнению, могут совершить специалисты по вычислительной технике, — это выделить для чего-нибудь недостаточное количество разрядов.

В ответ им было заявлено, что подобные аргументы можно привести для увеличения любого поля, что приведет к разбуханию заголовка. Кроме того, назначение поля *Максимальное количество транзитных участков* состоит в том, чтобы не допустить слишком долгого странствования пакетов, и 65 535 транзитных участков — это очень много. К тому же по мере роста Интернета будет создаваться все большее количество междугородных линий, что позволит передавать пакеты из любой страны в любую страну максимум за шесть транзитных пересылок. Если от получателя или отправителя до соответствующего международного шлюза окажется более 125 транзитных участков, то, видимо, что-то не в порядке с магистралями этого государства. В итоге эту битву выиграли сторонники 8-битового счетчика.

Еще одним предметом спора оказался максимальный размер пакета. Обладатели суперкомпьютеров настаивали на размере пакетов, превышающем 64 Кбайт. Когда суперкомпьютер начинает передачу, он занимается серьезным делом и не хочет, чтобы его прерывали через каждые 64 Кбайта. Аргумент против больших пакетов заключается в том, что если пакет размером в 1 Мбайт будет передаваться по линии T1 со скоростью 1,5 Мбит/с, то он займет линию на целых 5 секунд, что вызовет слишком большую задержку, заметную для интерактивных пользователей. В данном вопросе удалось достичь компромисса: нормальные пакеты ограничены размером 64 Кбайт, но с помощью дополнительного заголовка можно пересылать дейтаграммы огромного размера.

Еще одним спорным вопросом оказалось удаление контрольной суммы IPv4. Кое-кто сравнивал этот ход с удалением тормозов из автомобиля. При этом автомобиль становится легче и может двигаться быстрее, но если случится что-нибудь неожиданное, то могут быть проблемы.

Аргумент против контрольных сумм состоял в том, что каждое приложение, действительно заботящееся о целостности своих данных, все равно считает контрольную сумму на транспортном уровне, поэтому наличие еще одной на сетевом уровне является излишним (кроме того, контрольная сумма подсчитывается еще и на уровне передачи данных). Более того, эксперименты показали, что вычисление контрольной суммы составляло основные затраты протокола IPv4. Это сражение было выиграно лагерем противников контрольной суммы, поэтому в протоколе IPv6, как мы знаем, контрольной суммы нет.

Вокруг мобильных хостов также разгорелся спор. Если мобильный компьютер окажется на другом конце Земли, сможет ли он продолжать использовать прежний IPv6-адрес или должен будет использовать схему с внутренним и внешним агентами? Мобильные хосты также вносят асимметрию в систему маршрутизации. Вполне реальна ситуация, когда маленький мобильный компьютер хорошо слышит мощный сигнал большого стационарного маршрутизатора, но стационарный маршрутизатор не слышит слабого сигнала, передаваемого мобильным хостом. Поэтому появилось много желающих создать в протоколе IPv6 явную поддержку мобильных хостов. Эти попытки потерпели поражение, поскольку ни по одному конкретному предложению не удалось достичь консенсуса.

Вероятно, самые жаркие баталии разгорелись вокруг вопроса безопасности. Все были согласны, что это необходимо. Спорным было то, где и как следует реализовывать безопасность. Во-первых, где. Аргументом за размещение системы безопасности на сетевом уровне было то, что при этом она становится стандартной службой, которой могут пользоваться все приложения безо всякого предварительного планирования. Контраргумент заключался в том, что по-настоящему защищенным приложениям подходит лишь сквозное шифрование, когда шифрование осуществляется самим источником, а дешифровка — непосредственным получателем. Во всех остальных случаях пользователь оказывается в зависимости от, возможно, содержащей ошибки реализации сетевого уровня, над которой у него нет контроля. В ответ на этот аргумент можно сказать, что приложение может просто отказаться от использования встроенных в IP функций защиты и выполнять всю эту работу самостоятельно. Возражение на этот контраргумент состоит в том, что пользователи, не доверяющие сетям, не хотят платить за не используемую ими функцию, реализация которой утяжеляет и замедляет работу протокола, даже если сама функция отключена.

Другой аспект вопроса расположения системы безопасности касается того факта, что во многих (но не во всех) странах приняты строгие экспортные законы, касающиеся криптографии. В некоторых странах, особенно во Франции и Ираке, строго запрещено использование криптографии даже внутри страны, чтобы у населения не могло быть секретов от полиции. В результате любая реализация протокола IP, использующая достаточно мощную криптографическую систему, не может быть экспортирована за пределы Соединенных Штатов (и многих других стран). Таким образом, приходится поддерживать два набора программного обеспечения — один для внутреннего использования, а другой для экспорта, против чего решительно выступает большинство производителей компьютеров.

Единственный вопрос, по которому не было споров, состоял в том, что никто не ожидает, что IPv4-Интернет будет выключен в воскресенье, а в понедельник утром будет включен уже IPv6-Интернет. Вместо этого вначале появятся «островки» IPv6, которые будут общаться по туннелям. По мере роста острова IPv6

будут объединяться в более крупные острова. Наконец все острова объединятся, и Интернет окажется полностью трансформированным. Учитывая огромные средства, вложенные в работающие сегодня IPv4-маршрутизаторы, процесс трансформации, вероятно, займет около десяти лет. Поэтому были приложены гигантские усилия, чтобы гарантировать максимальную безболезненность этого перехода. Дополнительную информацию, касающуюся IPv6, можно найти в (Loshin, 1999).

Резюме

Сетевой уровень предоставляет услуги транспортному уровню. В его основе могут лежать либо виртуальные каналы, либо дейтаграммы. В обоих случаях его основная работа состоит в выборе маршрута пакетов от источника до адресата. В подсетях с виртуальными каналами решение о выборе маршрута осуществляется при установке виртуального канала. В дейтаграммных подсетях оно принимается для каждого пакета.

В компьютерных сетях применяется большое количество алгоритмов маршрутизации. К статическим алгоритмам относятся определение кратчайшего пути и заливка. Динамические алгоритмы включают в себя дистанционно-векторную маршрутизацию и маршрутизацию с учетом состояния каналов. В большинстве имеющихся сетей применяется один из этих алгоритмов. К другим важным методам маршрутизации относятся иерархическая маршрутизация, маршрутизация для мобильных хостов, широковещательная маршрутизация, многоадресная маршрутизация и маршрутизация в равноранговых сетях.

Подсети подвержены перегрузкам, увеличивающим задержки и снижающим пропускную способность. Разработчики сетей пытаются предотвратить перегрузку разнообразными способами, включающими формирование трафика, спецификации потока и резервирование пропускной способности. Если перегрузку предотвратить не удалось, с ней нужно бороться. Для этого могут применяться посылаемые отправителю сдерживающие пакеты, перераспределение нагрузки и другие методы.

Существующие на сегодня сети имеют много различий, поэтому при объединении могут возникнуть проблемы. Разработчики пытаются избавиться от них, создавая соответствующие решения. Среди используемых методов надо выделить стратегии повторной передачи, кэширования, управления потоком и т. д. Если перегрузка сети все же возникает, с ней приходится бороться. Для этого можно посылать сдерживающие пакеты, сбрасывать нагрузку или применять другие методы.

Следующим шагом после разрешения проблем с перегрузкой является попытка достижения гарантированного качества обслуживания. Методы, используемые для этого, включают в себя буферизацию на стороне клиента, формирование трафика, резервирование ресурсов, контроль доступа. Подходы, разработанные для обеспечения хорошего качества обслуживания, включают в себя интегрированное обслуживание (включая RSVP), дифференцированное обслуживание и MPLS.

Разные сети могут отличаться друг от друга весьма значительно, поэтому при попытке их объединения могут возникать определенные сложности. Иногда проблемы могут быть решены при помощи туннелирования пакета сквозь сильно отличающуюся сеть, но если отправитель и получатель находятся в сетях разных типов, этот подход не может быть применен. Если в сетях различаются ограничения на максимальный размер пакета, может быть применена фрагментация.

В Интернете существует большое разнообразие протоколов, относящихся к сетевому уровню. К ним относятся протокол транспортировки данных IP, управляющие протоколы ICMP, ARP и RARP, а также протоколы маршрутизации OSFP и BGP. Интернету перестает хватать IP-адресов, поэтому была разработана новая версия протокола IP, IPv6.

Вопросы

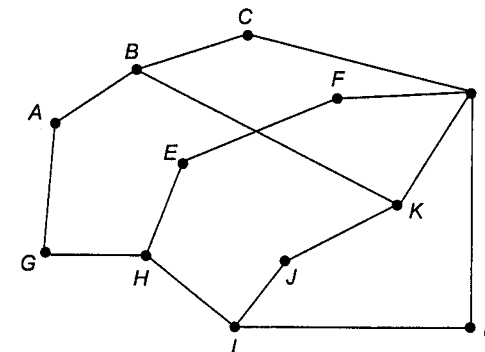
1. Приведите два примера приложений, для которых ориентированная на соединение служба является приемлемой. Затем приведите два примера приложений, для которых наилучшей будет служба без использования соединений.
2. Могут ли возникнуть обстоятельства, при которых служба виртуальных каналов доставлять пакеты без сохранения их порядка? Объясните.
3. Дейтаграммные подсети выбирают маршрут для каждого отдельного пакета независимо от других. В подсетях виртуальных каналов каждый пакет следует по заранее определенному маршруту. Означает ли это, что подсетям виртуальных каналов не требуется способность выбирать маршрут для отдельного пакета от произвольного источника произвольному адресату. Поясните свой ответ.
4. Приведите три примера параметров протокола, о значениях которых можно договориться при установке соединения.
5. Рассмотрите следующую проблему, касающуюся реализации службы виртуальных каналов. Если подсеть основана на виртуальных каналах, то каждый пакет данных должен иметь 3-байтовый заголовок, а каждому маршрутизатору требуется 8 байт для хранения информации о виртуальном канале. Если в подсети используется дейтаграммная служба, тогда необходимы 15-байтовые заголовки пакетов, но не требуется памяти для таблиц маршрутизаторов. Передача данных стоит 1 цент за 10^6 байт на транзитный участок. Очень быстрая память маршрутизаторов может быть приобретена по цене 1 цент за байт со сроком полной амортизации более двух лет (имеется в виду работа по 40 часов в неделю). Статистически усредненный сеанс продолжается 1000 с, за время которого передаются 200 пакетов. В среднем пакет пересылается через четыре транзитных участка. Какой способ реализации будет дешевле и насколько?

6. Предполагая, что все маршрутизаторы и хосты работают нормально и что их программное обеспечение не содержит ошибок, есть ли вероятность, хотя бы небольшая, что пакет будет доставлен неверному адресату?
7. Рассмотрите сеть на рис. 5.6, игнорируя весовые коэффициенты линий. Допустим, в качестве алгоритма маршрутизации применяется метод заливки. Пакет, посланный *A* для *D*, имеет ограничение на максимальное число переходов, равное 3. Перечислите все маршрутизаторы, через которые он пройдет. Сколько переходов относительно всей пропускной способности займет эта передача?
8. Укажите простой эвристический метод нахождения двух путей от данного источника к данному адресату, гарантирующий сохранение связи при обрыве любой линии (если такие два пути существуют). Маршрутизаторы считаются достаточно надежными, поэтому рассматривать возможность выхода их из строя не нужно.
9. Рассмотрите подсеть на рис. 5.11, *а*. Используется алгоритм дистанционно-векторной маршрутизации. На маршрутизатор *C* только что поступили следующие векторы: от *B* (5, 0, 8, 12, 6, 2); от *D* (16, 12, 6, 0, 9, 10); от *E* (7, 6, 3, 9, 0, 4). Измеренные задержки до *B*, *D* и *E* составляют 6, 3 и 5 соответственно. Какой будет новая таблица маршрутизатора *C*? Укажите используемые выходные линии и ожидаемое время задержки.
10. В сети, состоящей из 50 маршрутизаторов, значения времени задержки записываются как 8-битовые номера, а маршрутизаторы обмениваются векторами задержек дважды в секунду. Какая пропускная способность на каждой (дуплексной) линии съедается работой распределенного алгоритма маршрутизации? Предполагается, что каждый маршрутизатор соединен тремя линиями с другими маршрутизаторами.
11. На рис. 5.12 логическое ИЛИ двух наборов ACF-битов равно 111 для каждого ряда. Является ли это просто случайностью или же это сохраняется во всех подсетях при любых условиях?
12. Какие размеры регионов и кластеров следует выбрать для минимизации таблиц маршрутизации при трехуровневой иерархической маршрутизации, если количество маршрутизаторов равно 4800. Рекомендуется начать с гипотезы о том, что решение в виде k кластеров по k регионов из k маршрутизаторов близко к оптимальному. Это означает, что число k примерно равно корню кубическому из 4800 (около 16). Методом проб и ошибок подберите все три параметра так, чтобы они были близки к 16.
13. В тексте утверждалось, что когда мобильного хоста нет в его домашней сети, пакеты, посланные на адрес его домашней ЛВС, перехватываются внутренним агентом этой ЛВС. Как этот перехват осуществляет внутренний агент в IP-сети на основе локальной сети 802.3?
14. Сколько широковещательных пакетов формируется маршрутизатором *B* на рис. 5.5 с помощью:

    1) пересылки в обратном направлении;

    2) входного дерева?

15. Рассмотрите рис. 5.14, *а*. Допустим, добавляется одна новая линия между *F* и *G*, но входное дерево, показанное на рис. 5.14, *б*, остается без изменений. Какие изменения нужно внести в рис. 5.14, *в*?
16. Рассчитайте многоадресное связующее дерево для маршрутизатора *C* в подсети, показанной ниже, для группы, состоящей из маршрутизаторов *A*, *B*, *C*, *D*, *E*, *F*, *I* и *K*.



17. Рассмотрите рис. 5.18. При показанном поиске, начинающемся на узле *A*, будут ли когда-нибудь узлы *H* и *I* заниматься широковещанием?
18. Допустим, узел *B* на рис. 5.18 только что перезагрузился и не имеет никакой информации о маршрутизации в своих таблицах. Внезапно у него появляется необходимость в маршруте к узлу *H*. Он рассылает широковещательным способом наборы *TTL* на 1, 2, 3 и т. д. Сколько раундов потребуется на поиск пути?
19. В простейшем варианте алгоритма хорд при поиске в равноранговых сетях таблицы указателей не используются. Вместо этого производится линейный поиск по кругу в обоих направлениях. Может ли при этом узел предсказать, в каком направлении следует искать? Ответ аргументируйте.
20. Рассмотрите круг, используемый в алгоритме хорд и показанный на рис. 5.22. Допустим, узел 10 внезапно подключается к сети. Повлияет ли это на таблицу указателей узла 1, и если да, то как?
21. В качестве возможного механизма борьбы с перегрузкой в подсети, использующей виртуальные каналы, маршрутизатор может воздержаться от подтверждения полученного пакета в следующих случаях: 1) он знает, что его последняя передача по виртуальному каналу была получена успешно; 2) у него есть свободный буфер. Для простоты предположим, что маршрутизаторы используют протокол с ожиданием и что у каждого виртуального канала есть один буфер, выделенный ему для каждого направления трафика. Передача пакета (данных или подтверждения) занимает *T* секунд. Путь пакета проходит через *n* маршрутизаторов. С какой скоростью пакеты доставляются адресату? Предполагается, что ошибки очень редки, а связь между хостом и маршрутизатором почти не отнимает времени.

22. Дейтаграммная подсеть позволяет маршрутизаторам при необходимости выбрасывать пакеты. Вероятность того, что маршрутизатор отвергнет пакет, равна p. Рассмотрите маршрут, проходящий от хоста к хосту через два маршрутизатора. Если любой из маршрутизаторов отвергнет пакет, у хоста-отправителя в конце концов истечет интервал ожидания и он попытается переслать пакет еще раз. Если обе линии (хост—маршрутизатор и маршрутизатор—маршрутизатор) считать за транзитные участки, то чему равно среднее число:

    1) транзитных участков, преодолеваемых пакетом за одну передачу;

    2) передач для одного пакета;

    3) транзитных участков, необходимых для получения пакета?

23. В чем состоит основная разница между методом предупредительного бита и методом RED?
24. Почему алгоритм «дырявое ведро» должен позволять передачу лишь одного пакета за интервал времени, независимо от размеров пакета?
25. В некоторой системе используется вариант алгоритма «дырявое ведро» с подсчетом байтов. Правило гласит, что за один интервал времени может быть послан один 1024-байтовый пакет или два 512-байтовых пакета и т. д. В чем заключается не упомянутое в этом тексте серьезное ограничение такой системы?
26. Сеть ATM использует для формирования трафика схему маркерного ведра. Новый маркер помещается в ведро каждые 5 мкс. Чему равна максимальная скорость передачи данных в сети (не считая битов заголовка)?
27. Компьютер, подключенный к сети, скорость передачи в которой равна 6 Мбит/с, регулируется маркерным ведром. Маркерное ведро наполняется со скоростью 1 Мбит/с. Его начальная емкость составляет 8 Мбит. Как долго сможет передавать компьютер на полной скорости в 6 Мбит/с?
28. Представьте, что максимальный размер пакета в спецификации потока равен 1000 байт, скорость маркерного ведра равна 10 млн байт/с, объем маркерного ведра составляет 1 млн байт, а максимальная скорость передачи равна 50 млн байт/с. Как долго может продолжаться передача с максимальной скоростью?
29. Сеть на рис. 5.32 использует RSVP в деревьях групповой рассылки для хостов 1 и 2. Допустим, хост 3 запрашивает канал с пропускной способностью 2 Мбайт/с для потока от хоста 1 и еще один канал с пропускной способностью 1 Мбайт/с для потока от хоста 2. Одновременно хост 4 запрашивает 2-мегабайтный канал для потока от хоста 1, а хост 5 запрашивает 1-мегабайтный канал для потока от хоста 2. Какую суммарную пропускную способность необходимо зарезервировать для удовлетворения перечисленных запросов на маршрутизаторах *A*, *B*, *C*, *E*, *H*, *J*, *K* и *L*?
30. Центральный процессор маршрутизатора может обрабатывать 2 млн пакетов в секунду. Сколько времени уйдет на формирование очередей и обслуживание пакетов процессорами, если путь от источника до приемника содержит 10 маршрутизаторов?

31. Допустим, пользователь получает дифференцированный сервис со срочной пересылкой. Есть ли гарантия того, что срочные пакеты будут испытывать меньшую задержку, чем обычные? Ответ поясните.
32. Нужна ли фрагментация в интерсетях с объединенными виртуальными каналами или она необходима только в дейтаграммных системах?
33. Туннелирование сквозь подсеть сцепленных виртуальных каналов осуществляется следующим образом: многопротокольный маршрутизатор на одном конце устанавливает виртуальный канал с другим концом и посылает по нему пакеты. Можно ли применить туннелирование в дейтаграммных подсетях? Если да, как?
34. Допустим, хост *A* соединен с маршрутизатором *R*1. Тот, в свою очередь, соединен с другим маршрутизатором, *R*2, а *R*2 — с хостом *B*. Сообщение TCP, содержащее 900 байт данных и 20 байт TCP-заголовка, передается IP-программе, установленной на хосте *A*, для доставки его хосту *B*. Каковы будут значения полей *Общая длина*, *Идентификатор*, *DF*, *MF* и *Сдвиг фрагмента* IP-заголовка каждого пакета, передающегося по трем линиям. Предполагается, что на линии *A*—*R*1 максимальный размер кадра равен 1024 байта, включая 14-байтный заголовок кадра, на линии *R*1-*R*2 максимальный размер кадра составляет 512 байт, включая 8-байтный заголовок кадра, и на линии *R*2—*B* максимальный размер кадра составляет 512 байт, включая 12-байтный заголовок кадра
35. Маршрутизатор освобождает IP-пакеты, общая длина которых (включая данные и заголовок) равна 1024 байт. Предполагая, что пакеты живут в течение 10 с, сосчитайте максимальную скорость линии, с которой может работать маршрутизатор без опасности зацикливания в пространстве идентификационных номеров IP-дейтаграммы.
36. IP-дейтаграмма, использующая параметр *Строгая маршрутизация от источника*, должна быть фрагментирована. Копируется ли этот параметр в каждый фрагмент, или достаточно поместить его в первый фрагмент? Поясните свой ответ.
37. Допустим, вместо 16 бит в адресе класса В для обозначения номера сети отводилось бы 20 бит. Сколько было бы тогда сетей класса В?
38. Преобразуйте IP-адрес, шестнадцатеричное представление которого равно C22F 1582, в десятичный формат, разделенный точками.
39. Маска подсети сети Интернета равна 255.255.240.0. Чему равно максимальное число хостов в ней?
40. Существует множество адресов, начинающихся с IP-адреса 198.16.0.0. Допустим, организации *A*, *B*, *C* и *D* запрашивают, соответственно, 4000, 2000, 4000 и 8000 адресов. Для каждой из них укажите первый и последний выданные адреса, а также маску вида *w.x.y.z/s*.

41. Маршрутизатор только что получил информацию о следующих IP-адресах: 57.6.96.0/21, 57.6.112.0/21 и 57.6.120.0/21. Если для них используется одна и та же исходящая линия, можно ли их агрегировать? Если та, то во что? Если нет, то почему?
42. Набор IP-адресов с 29.18.0.0 по 19.18.128.255 агрегирован в 29.18.0.0/17. Тем не менее, остался пробел из 1024 не присвоенных адресов, с 29.18.60.0 по 29.18.63.255, которые внезапно оказались присвоены хосту, использующему другую исходящую линию. Необходимо ли теперь разделить агрегированный адрес на составляющие, добавить в таблицу новый блок, а потом посмотреть, можно ли что-нибудь агрегировать? Если нет, тогда что можно сделать?
43. Маршрутизатор содержит следующие записи (CIDR) в своей таблице маршрутизации:

Адрес/маска	Следующий переход
135.46.56.0/22	Интерфейс 0
135.46.60.0/22	Интерфейс 1
192.53.40.0/23	Маршрутизатор 1
По умолчанию	Маршрутизатор 2

44. Куда направит маршрутизатор пакеты со следующими IP-адресами?
    1) 135.46.63.10;
    2) 135.46.57.14;
    3) 135.46.52.2;
    4) 192.53.40.7;
    5) 192.53.56.7.
45. Многие компании придерживаются стратегии установки двух и более маршрутизаторов, соединяющих компанию с провайдером, что гарантирует некоторый запас прочности на случай, если один из маршрутизаторов выйдет из строя. Применима ли такая политика при использовании NAT? Ответ поясните.
46. Вы рассказали товарищу про протокол ARP. Когда вы закончили объяснения, он сказал: «Ясно. ARP предоставляет услуги сетевому уровню, таким образом, он является частью уровня передачи данных». Что вы ему ответите?
47. Протоколы ARP и RARP оба устанавливают соответствия адресов из разных адресных пространств. В этом смысле они похожи. Однако способы их реализации в корне различны. В чем их основное отличие?
48. Опишите способ сборки пакета из фрагментов в пункте назначения.
49. В большинстве алгоритмов сборки IP-дейтаграмм из фрагментов используется таймер, чтобы из-за потерянного фрагмента буфер, в котором производится повторная сборка, не оказался занят остальными фрагментами дейтаграммы. Предположим, дейтаграмма разбивается на четыре фрагмента. Первые три фрагмента прибывают к получателю, а четвертый задерживается в пути. У получателя истекает период ожидания, и три фрагмента, хранившиеся в его памяти, удаляются. Немного позднее наконец приползает последний фрагмент. Как следует с ним поступить?

50. Как в IP, так и в ATM контрольная сумма покрывает только заголовок, но не данные. Почему, как вы полагаете, была выбрана подобная схема?
51. Особа, живущая в Бостоне, едет в Миннеаполис и берет с собой свой персональный компьютер. К ее удивлению, локальная сеть в Миннеаполисе является беспроводной локальной сетью IP, поэтому ей нет необходимости подключать свой компьютер. Нужно ли, тем не менее, проходить процедуру с внутренним и внешним агентом, чтобы электронная почта и другой трафик прибывали правильно?
52. Протокол IPv6 использует 16-байтовые адреса. На какое время хватит этих адресов, если каждую пикосекунду назначать блок в 1 млн адресов?
53. Поле *Протокол*, используемое в заголовке IPv4, отсутствует в фиксированном заголовке IPv6. Почему?
54. Должен ли протокол ARP быть изменен при переходе на шестую версию протокола IP? Если да, то являются ли эти изменения концептуальными или техническими?
55. Напишите программу, моделирующую маршрутизацию методом заливки. Каждый пакет должен содержать счетчик, уменьшаемый на каждом маршрутизаторе. Когда счетчик уменьшается до нуля, пакет удаляется. Время дискретно, и каждая линия обрабатывает за один интервал времени один пакет. Создайте три версии этой программы: с заливкой по всем линиям, с заливкой по всем линиям, кроме входной линии, и с заливкой только k лучших линий (выбираемых статически). Сравните заливку с детерминированной маршрутизацией ($k = 1$) с точки зрения задержки и использования пропускной способности.
56. Напишите программу, моделирующую компьютерную сеть с дискретным временем. Первый пакет в очереди каждого маршрутизатора преодолевает по одному транзитному участку за интервал времени. Число буферов каждого маршрутизатора ограничено. Прибывший пакет, для которого нет свободного места, игнорируется и повторно не передается. Вместо этого используется сквозной протокол с тайм-аутами и пакетами подтверждения, который, в конце концов, вызывает повторную передачу пакета маршрутизатором-источником. Постройте график производительности сети как функции интервала сквозного времени ожидания при разных значениях частоты ошибок.
57. Напишите функцию, осуществляющую пересылку в IP-маршрутизаторе. У процедуры должен быть один параметр — IP-адрес. Имеется доступ к глобальной таблице, представляющей собой массив из троек значений. Каждая тройка содержит следующие целочисленные значения: IP-адрес, маску подсети и исходящую линию. Функция ищет IP-адрес в таблице, используя CIDR, и возвращает номер исходящей линии.

58. Используя программы *traceroute* (UNIX) или *tracert* (Windows), исследуйте маршрут от вашего компьютера до различных университетов мира. Составьте список трансокеанских линий. Вот некоторые адреса:

    www.berkeley.edu (Калифорния);
    www.mit.edu (Массачусетс);
    www.vu.nl (Амстердам);
    www.ucl.ac.uk (Лондон);
    www.usyd.edu.au (Сидней);
    www.u-tokyo.ac.jp (Токио);
    www.uct.ac.za (Кейптаун).

Глава 6 Транспортный уровень

- Транспортная служба
- Элементы транспортных протоколов
- Простой транспортный протокол
- Транспортные протоколы Интернета: UDP
- Транспортные протоколы Интернета: TCP
- Вопросы производительности
- Резюме
- Вопросы

Транспортный уровень — это не просто очередной уровень. Это сердцевина всей иерархии протоколов. Его задача состоит в предоставлении надежной и экономичной передачи данных от машины-источника машине-адресату вне зависимости от физических характеристик использующейся сети или сетей. Без транспортного уровня вся концепция многоуровневых протоколов потеряет смысл. В данной главе мы подробно рассмотрим транспортный уровень, включая его сервисы, устройство, протоколы и производительность.

Транспортная служба

В следующих разделах мы познакомимся с транспортной службой. Мы рассмотрим виды сервисов, предоставляемых прикладному уровню. Чтобы наш разговор не был слишком абстрактным, мы разберем два набора примитивов транспортного уровня. Сначала рассмотрим простой (но не применяемый на практике) набор, просто чтобы показать основные идеи, а затем — реально применяемый в Интернете интерфейс.

Услуги, предоставляемые верхним уровнем

Конечная цель транспортного уровня заключается в предоставлении эффективных, надежных и экономичных услуг (сервисов) своим пользователям, которыми обычно являются процессы прикладного уровня. Для достижения этой цели транспортный уровень пользуется услугами, предоставляемыми сетевым уровнем. Аппаратура и/или программа, выполняющая работу транспортного уровня, называется **транспортной сущностью** или **транспортным объектом**. Транспортный объект может располагаться в ядре операционной системы, в отдельном пользовательском процессе, в библиотечном модуле, загруженном сетевым приложением, или в сетевой интерфейсной плате. Логическая взаимосвязь сетевого, транспортного и прикладного уровней проиллюстрирована на рис. 6.1.

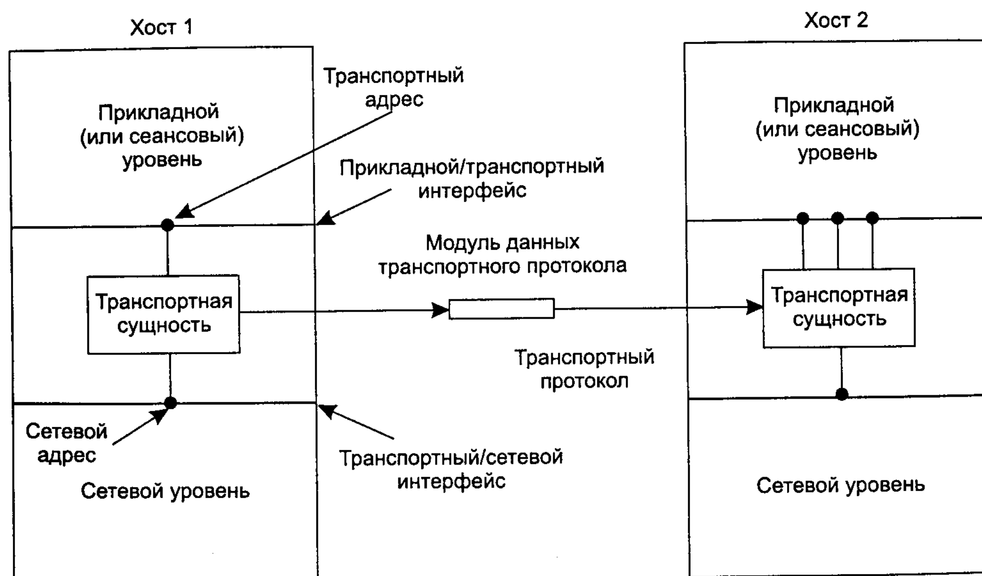


Рис. 6.1. Сетевой, транспортный и прикладной уровни

Сервисы транспортного уровня, как и сервисы сетевого уровня, могут быть ориентированными на соединение или не требующими соединений. Ориентированный на соединение транспортный сервис во многом похож на ориентированный на соединение сетевой сервис. В обоих случаях соединение проходит три этапа: установка, передача данных и разъединение. Адресация и управление потоком на разных уровнях также схожи. Более того, похожи друг на друга и не требующие соединений сервисы разных уровней.

Возникает закономерный вопрос: если сервис транспортного уровня так схож с сервисом сетевого уровня, то зачем нужны два различных уровня? Почему недостаточно одного уровня? Для ответа на этот важный вопрос следует вернуться к рис. 1.7. На рисунке мы видим, что транспортный код запускается целиком на пользовательских машинах, а сетевой уровень запускается в основном на маршрутизаторах, которые управляются оператором связи (по крайней мере, в глобальных сетях). Что произойдет, если сетевой уровень будет предоставлять ориентированный на соединение сервис, но этот сервис будет ненадежным? Например, если он часто будет терять пакеты? Можно себе представить, что случится, если маршрутизаторы будут время от времени выходить из строя.

В этом случае пользователи столкнутся с большими проблемами. У них нет контроля над сетевым уровнем, поэтому они не смогут улучшить качество обслуживания, используя хорошие маршрутизаторы или совершенствуя обработку ошибок уровня передачи данных. Единственная возможность заключается в использовании еще одного уровня, расположенного над сетевым, для улучшения качества обслуживания. Если транспортный объект узнает, что его сетевое соединение было внезапно прервано, но не получит каких-либо сведений о том, что случилось с передаваемыми в этот момент данными, он может установить новое соединение с удаленной транспортной сущностью. С помощью нового сетевого соединения он может послать запрос к равноранговому объекту и узнать, какие данные дошли до адресата, а какие нет, после чего продолжить передачу данных.

По сути, благодаря наличию транспортного уровня транспортный сервис может быть более надежным, чем лежащий ниже сетевой сервис. Транспортным уровнем могут быть обнаружены потерянные пакеты и искаженные данные, после чего потери могут быть компенсированы. Более того, примитивы транспортной службы могут быть разработаны таким образом, что они будут независимы от примитивов сетевой службы, которые могут значительно варьироваться от сети к сети (например, сервис локальной сети без соединений может значительно отличаться от сервиса ориентированной на соединение глобальной сети). Если спрятать службы сетевого уровня за набором примитивов транспортной службы, то для изменения сетевой службы потребуется просто заменить один набор библиотечных процедур другими, делающими то же самое, но с помощью других сервисов более низкого уровня.

Благодаря наличию транспортного уровня прикладные программы могут использовать стандартный набор примитивов и сохранять работоспособность в самых различных сетях. Им не придется учитывать имеющееся разнообразие интерфейсов подсетей и беспокоиться о ненадежной передаче данных. Если бы все реальные сети работали идеально и у всех сетей был один набор служебных примитивов, то транспортный уровень, вероятно, был бы не нужен. Однако в реальном мире он выполняет ключевую роль изолирования верхних уровней от деталей технологии, устройства и несовершенства подсети.

Именно по этой причине часто проводится разграничение между уровнями с первого по четвертый и уровнями выше четвертого. Нижние четыре уровня можно рассматривать как **поставщика транспортных услуг**, а верхние уровни — как **пользователя транспортных услуг**. Разделение на поставщика и пользователя оказывает серьезное влияние на устройство уровней и помещает транспортный уровень в ключевую позицию, поскольку он формирует основную границу между поставщиком и пользователем надежной службы передачи данных.

Примитивы транспортной службы

Чтобы пользователи могли получить доступ к транспортной службе, транспортный уровень должен совершать некоторые действия по отношению к прикладным программам, то есть предоставлять интерфейс транспортной службы. У всех транспортных служб есть свои интерфейсы. В этом разделе мы вначале рассмотрим простой (но гипотетический) пример транспортной службы и ее интерфейсов, просто чтобы узнать основные принципы и понятия. Следующий раздел будет посвящен реальному примеру.

Транспортная служба подобна сетевой, но имеет и некоторые существенные отличия. Главное отличие состоит в том, что сетевая служба предназначена для моделирования сервисов, предоставляемых реальными сетями, со всеми их особенностями. Реальные сети теряют пакеты, поэтому в общем случае сетевая служба ненадежна.

Ориентированная на соединение транспортная служба, напротив, является надежной. Конечно, реальные сети содержат ошибки, но именно транспортный уровень как раз и должен обеспечивать надежность сервисов ненадежных сетей.

В качестве примера рассмотрим два процесса, соединенных каналами в системе UNIX. Эти процессы предполагают, что соединение между ними идеально. Они не желают знать о подтверждениях, потерянных пакетах, заторах и т. п. Им требуется стопроцентно надежное соединение. Процесс А помещает данные в один конец канала, а процесс В извлекает их на другом. Именно для этого и предназначена ориентированная на соединение транспортная служба — скрывать несовершенство сетевого уровня, чтобы пользовательские процессы могли считать, что существует безошибочный поток битов.

Кстати, транспортный уровень может также предоставлять ненадежный (дейтаграммный) сервис, но о нем сказать почти нечего, поэтому мы в данной главе сконцентрируемся на транспортной службе, ориентированной на соединение. Тем не менее, некоторые приложения, например, клиент-серверные вычислительные системы и потоковое мультимедиа, даже выигрывают от дейтаграммных сервисов, поэтому далее мы еще упомянем их.

Второе различие между сетевой и транспортной службами состоит в том, для кого они предназначены. Сетевая служба используется только транспортными объектами. Мало кто пишет свои собственные транспортные объекты, и поэтому пользователи и программы почти не встречаются с голой сетевой службой. Транспортные примитивы, напротив, используются многими программами, а следовательно, и программистами. Поэтому транспортная служба должна быть удобной и простой в употреблении.

Чтобы получить представление о транспортной службе, рассмотрим пять примитивов, перечисленных в табл. 6.1. Этот транспортный интерфейс сильно упрощен, но он дает представление о назначении ориентированного на соединение транспортного интерфейса. Он позволяет прикладным программам устанавливать, использовать и освобождать соединения, чего вполне достаточно для многих приложений.

Таблица 6.1. Примитивы простой транспортной службы

Примитив	Посланный модуль данных транспортного протокола	Значение
LISTEN (ОЖИДАТЬ)	(нет)	Блокировать сервер, пока какой-либо процесс не попытается соединиться
CONNECT (СОЕДИНИТЬ)	ЗАПРОС СОЕДИНЕНИЯ	Активно пытаться установить соединение
SEND (ПОСЛАТЬ)	ДАННЫЕ	Послать информацию
RECEIVE (ПОЛУЧИТЬ)	(нет)	Блокировать сервер, пока не прибудут данные
DISCONNECT (РАЗЪЕДИНИТЬ)	ЗАПРОС РАЗЪЕДИНЕНИЯ	Прервать соединение

Чтобы понять, как могут быть использованы эти примитивы, рассмотрим приложение, состоящее из сервера и нескольких удаленных клиентов. Вначале сервер выполняет примитив LISTEN — обычно для этого вызывается библиотечная процедура, которая обращается к системе. В результате сервер блокируется, пока клиент не обратится к нему. Когда клиент хочет поговорить с сервером, он выполняет примитив CONNECT. Транспортный объект выполняет этот примитив, блокируя обратившегося к нему клиента и посылая пакет серверу. Поле данных пакета содержит сообщение транспортного уровня, адресованное транспортному объекту сервера.

Следует сказать пару слов о терминологии. За неимением лучшего термина, для сообщений, посылаемых одной транспортной сущностью другой транспортной сущности, нам придется использовать несколько неуклюжее сокращение **TPDU** (Transport Protocol Data Unit — модуль данных транспортного протокола). Модули данных, которыми обмениваются транспортные уровни, помещаются в пакеты (которыми обмениваются сетевые уровни). Эти пакеты, в свою очередь, содержатся в кадрах, которыми обмениваются уровни передачи данных. Получив кадр, уровень передачи данных обрабатывает заголовок кадра и передает содержимое поля полезной нагрузки кадра наверх, сетевой сущности. Сетевая сущность обрабатывает заголовок пакета и передает содержимое поля полезной нагрузки пакета наверх, транспортной сущности. Эта вложенность проиллюстрирована на рис. 6.2.

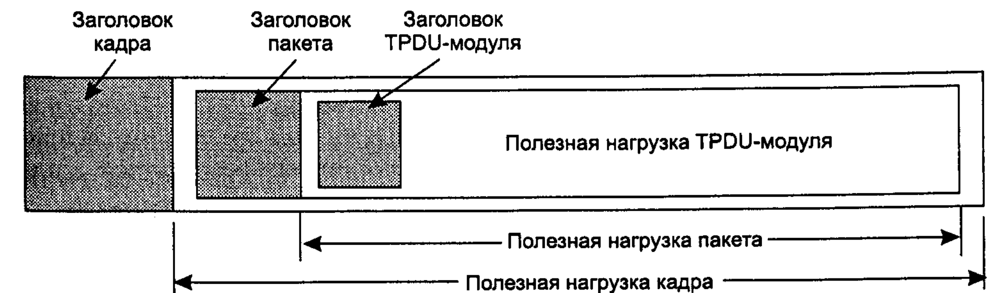


Рис. 6.2. Вложенность модулей данных транспортного протокола, пакетов и кадров

Итак, вернемся к нашему примеру общения клиента и сервера. В результате запроса клиента CONNECT серверу посылается модуль данных транспортного протокола, содержащий CONNECTION REQUEST (запрос соединения). Когда он прибывает, транспортная сущность проверяет, заблокирован ли сервер примитивом LISTEN (то есть заинтересован ли сервер в обработке запросов). Затем она разблокирует сервер и посылает обратно клиенту модуль данных CONNECTION ACCEPTED (соединение принято). Получив этот модуль, клиент разблокируется, после чего соединение считается установленным.

Теперь клиент и сервер могут обмениваться данными с помощью примитивов SEND и RECEIVE. В простейшем случае каждая из сторон может использовать блокирующий примитив RECEIVE для перехода в режим ожидания модуля данных, посылаемого противоположной стороной при помощи примитива SEND. Когда модуль данных прибывает, получатель разблокируется. Затем он может обработать полученный модуль и послать ответ. Такая схема прекрасно работает, пока обе стороны помнят, чей черед посылать, а чей — принимать.

Обратите внимание на то, что на сетевом уровне даже простая однонаправленная пересылка данных оказывается сложнее, чем на транспортном уровне. Каждый посланный пакет данных будет, в конце концов, подтвержден. Пакеты, содержащие управляющие модули данных, также подтверждаются, явно или неявно. Эти подтверждения управляются транспортными сущностями при помощи протокола сетевого уровня и не видны пользователям транспортного уровня. Аналогично транспортным сущностям нет необходимости беспокоиться о таймерах и повторных передачах. Все эти механизмы не видны пользователям транспортного уровня, для которых соединение представляется надежным битовым каналом. Один пользователь помещает в канал биты, которые волшебным образом появляются на другом конце канала. Эта способность скрывать сложность от пользователей свидетельствует о том, что многоуровневые протоколы являются довольно мощным инструментом.

Когда соединение больше не требуется, оно должно быть разорвано, чтобы можно было освободить место в таблицах двух транспортных сущностей. Разъединение существует в двух вариантах: симметричном и асимметричном. В асимметричном варианте любой пользователь транспортной службы может вызвать примитив DISCONNECT, в результате чего удаленной транспортной сущности будет послан управляющий модуль TPDU DISCONNECTION REQUEST (запрос разъединения). После получения модуля TPDU удаленной транспортной сущностью соединение разрывается.

В симметричном варианте каждое направление закрывается отдельно, независимо от другого. Когда одна сторона выполняет примитив DISCONNECT, это означает, что у нее больше нет данных для передачи, но что она все еще готова принимать данные от своего партнера. В этой схеме соединение разрывается, когда обе стороны выполняют примитив DISCONNECT.

Диаграмма состояний для установки и разрыва соединения показана на рис. 6.3. Каждый переход вызывается каким-то событием или примитивом, выполненным локальным пользователем транспортной службы или входящим пакетом. Для простоты мы будем считать, что каждый модуль TPDU подтверждается отдельно.

Мы также предполагаем, что используется модель симметричного разъединения, в которой клиент делает первый ход. Обратите внимание на простоту этой модели. Позднее мы рассмотрим более реалистичные модели.

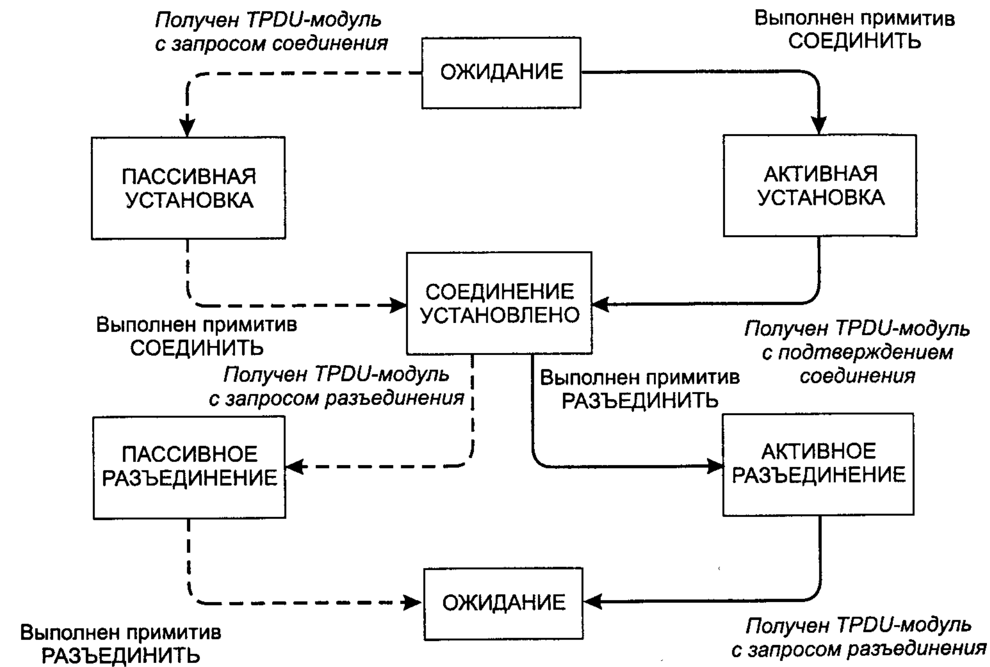


Рис. 6.3. Диаграмма состояний для простой схемы управления соединениями. Переходы, обозначенные курсивом, вызываются прибытием пакетов. Сплошными линиями показана последовательность состояний клиента. Пунктирными линиями показана последовательность состояний сервера

Сокеты Беркли

Теперь рассмотрим другой набор транспортных примитивов — примитивы сокетов (иногда называемых гнездами), используемые в операционной системе Berkeley UNIX для протокола TCP (Transmission Control Protocol — протокол управления передачей). Они приведены в табл. 6.2. Модель сокетов во многом подобна рассмотренной ранее модели транспортных примитивов, но обладает большей гибкостью и предоставляет больше возможностей. Модули TPDU, соответствующие этой модели, будут рассматриваться далее в этой главе, когда мы будем изучать TCP.

Первые четыре примитива списка выполняются серверами в таком же порядке. Примитив SOCKET создает новый сокет и выделяет для него место в таблице транспортной сущности. Параметры вызова указывают используемый формат адресов, тип требуемой услуги (например, надежный байтовый поток) и протокол. В случае успеха примитив SOCKET возвращает обычный описатель файла, ис-

пользуемого при вызове следующих примитивов, подобно тому, как возвращает описатель файла процедура OPEN.

Таблица 6.2. Примитивы сокетов для TCP

Примитив	Значение
SOCKET (СОКЕТ)	Создать новый сокет (гнездо связи)
BIND (СВЯЗАТЬ)	Связать локальный адрес с сокетом
LISTEN (ОЖИДАТЬ)	Объявить о желании принять соединение; указать размер очереди
ACCEPT (ПРИНЯТЬ)	Блокировать звонящего до получения попытки соединения
CONNECT (СОЕДИНИТЬ)	Активно пытаться установить соединение
SEND (ПОСЛАТЬ)	Посылать данные по соединению
RECEIVE (ПОЛУЧИТЬ)	Получать данные у соединения
CLOSE (ЗАКРЫТЬ)	Разрывать соединение

У только что созданного сокета нет сетевых адресов. Они назначаются с помощью примитива BIND. После того как сервер привязывает адрес к сокету, с ним могут связаться удаленные клиенты. Вызов SOCKET не создает адрес напрямую, так как некоторые процессы придают адресам большое значение (например, они использовали один и тот же адрес годами, и этот адрес всем известен), тогда как другим процессам это не важно.

Следом идет вызов LISTEN, который выделяет место для очереди входящих звонков на случай, если несколько клиентов попытаются соединиться одновременно. В отличие от примитива LISTEN в нашем первом примере, примитив LISTEN гнездовой модели не является блокирующим вызовом.

Чтобы заблокировать ожидание входящих соединений, сервер выполняет примитив ACCEPT. Получив TPDU-модуль с запросом соединения, транспортная сущность создает новый сокет с теми же свойствами, что и у исходного сокета, и возвращает описатель файла для него. При этом сервер может разветвить процесс или поток, чтобы обработать соединение для нового сокета и вернуться к ожиданию следующего соединения для оригинального сокета.

Теперь посмотрим на этот процесс со стороны клиента. В этом случае также сначала с помощью примитива SOCKET должен быть создан сокет, но примитив BIND здесь не требуется, так как используемый адрес не имеет значения для сервера. Примитив CONNECT блокирует вызывающего и инициирует активный процесс соединения. Когда этот процесс завершается (то есть когда соответствующий TPDU-модуль, посланный сервером, получен), процесс клиента разблокируется и соединение считается установленным. После этого обе стороны могут использовать примитивы SEND и RECV для передачи и получения данных по полнодуплексному соединению. Могут также применяться стандартные UNIX-вызовы READ и WRITE, если нет нужды в использовании специальных свойств SEND и RECV.

В модели сокетов используется симметричный разрыв соединения. Соединение разрывается, когда обе стороны выполняют примитив CLOSE.

Пример программирования сокета: файл-сервер для Интернета

В качестве примера использования вызовов сокета рассмотрим программу, демонстрирующую работу клиента и сервера, представленную в листинге 6.1. Имеется примитивный файл-сервер, работающий в Интернете и использующий его клиент. У программы много ограничений (о которых еще будет сказано), но, в принципе, данный код, описывающий сервер, может быть скомпилирован и запущен на любой UNIX-системе, подключенной к Интернету. Код, описывающий клиента, может быть запущен с определенными параметрами. Это позволит ему получить любой файл, к которому у сервера есть доступ. Файл отображается на стандартном устройстве вывода, но, разумеется, может быть перенаправлен на диск или какому-либо процессу.

Рассмотрим сперва ту часть программы, которая описывает сервер. Она начинается с включения некоторых стандартных заголовков, последние три из которых содержат основные структуры и определения, связанные с Интернетом. Затем SERVER_PORT определяется как 12 345. Значение выбрано случайным образом. Любое число от 1024 до 65 535 подойдет с не меньшим успехом, если только оно не используется каким-либо другим процессом. Понятно, что клиент и сервер должны обращаться к одному и тому же порту. Если сервер в один прекрасный день станет популярным во всем мире (что маловероятно, учитывая то, насколько он примитивен), ему будет присвоен постоянный порт с номером менее 1024, который появится на www.iana.org.

В последующих двух строках определяются две необходимые серверу константы. Первая из них задает размер участка данных для файловой передачи. Вторая определяет максимальное количество незавершенных соединений, после установки которых новые соединения будут отвергаться.

После объявления локальных переменных начинается сама программа сервера. Вначале она инициализирует структуру данных, которая будет содержать IP-адрес сервера. Эта структура будет связана с серверным сокетом. Вызов memset полностью обнуляет структуру данных. Последующие три присваивания заполняют три поля этой структуры. Последнее из них содержит порт сервера. Функции htonl и htons занимаются преобразованием значений в стандартный формат, что позволяет программе нормально выполняться на машинах с представлением числовых разрядов как в возрастающем порядке (например, SPARC), так и в убывающем (например, Pentium). Детали их семантики здесь роли не играют.

После этого сервером создается и проверяется на ошибки (определяется по $s < 0$) сокет. В конечной версии программы сообщение об ошибке может быть чуть более понятным. Вызов setsockopt нужен для того, чтобы порт мог использоваться несколько раз, а сервер — бесконечно, обрабатывая запрос за запросом. Теперь IP-адрес привязывается к сокету и выполняется проверка успешного завершения вызова bind. Конечным этапом инициализации является вызов listen, свидетельствующий о готовности сервера к приему входящих вызовов и сообщающий системе о том, что нужно ставить в очередь до QUEUE_SIZE вызовов, пока сервер обрабатывает текущий вызов. При заполнении очереди прибытие новых запросов спокойно игнорируется.

В этом месте начинается основной блок программы, который никогда не пропускается. Его можно остановить только извне. Вызов accept блокирует сервер на то время, пока клиент пытается установить соединение. Если вызов завершается успешно, accept возвращает дескриптор файла, который можно использовать для чтения и записи, аналогично тому, как файловые дескрипторы могут записываться и читаться в каналах. Однако, в отличие от однонаправленных каналов, сокеты двунаправлены, поэтому для чтения (и записи) данных из соединения надо использовать sa (адрес сокета).

После установки соединения сервер считывает имя файла. Если оно пока недоступно, сервер блокируется, ожидая его. Получив имя файла, сервер открывает файл и входит в цикл, который читает блоки данных из файла и записывает их в сокет. Это продолжается до тех пор, пока не будут скопированы все запрошенные данные. Затем файл закрывается, соединение разрывается, и начинается ожидание нового вызова. Данный цикл повторяется бесконечно.

Теперь рассмотрим часть кода, описывающую клиента. Чтобы понять, как работает программа, необходимо вначале разобраться, как она запускается. Если она называется client, ее типичный вызов будет выглядеть так:

```
client flits.cs.vu.nl /usr/tom/filename >f
```

Этот вызов сработает только в том случае, если сервер расположен по адресу flits.cs.vu.nl, файл usr/tom/filename существует и у сервера есть доступ для чтения этого файла. Если вызов произведен успешно, файл передается по Интернету и записывается на место f, после чего клиентская программа заканчивает свою работу. Поскольку серверная программа продолжает работать, клиент может запускать новые запросы на получение файлов.

Клиентская программа начинается с подключения файлов и объявлений. Работа начинается с проверки корректности числа аргументов (argc = 3 означает, что в строке запуска содержались имя программы и два аргумента). Обратите внимание на то, что argv[1] содержит имя сервера (например, flits.cs.vu.nl) и переводится в IP-адрес с помощью gethostbyname. Для поиска имени функция использует DNS. Мы будем изучать технологию DNS в главе 7. Затем создается и инициализируется сокет, после чего клиент пытается установить TCP-соединение с сервером посредством connect. Если сервер включен, работает на указанной машине, соединен с SERVER_PORT и либо простаивает, либо имеет достаточно места в очереди listen (очереди ожидания), то соединение с клиентом рано или поздно будет установлено. По данному соединению клиент передает имя файла, записывая его в сокет. Количество отправленных байтов на единицу превышает требуемое для передачи имени, поскольку нужен еще нулевой байт-ограничитель, с помощью которого сервер может понять, где кончается имя файла.

Теперь клиентская программа входит в цикл, читает файл блок за блоком из сокета и копирует на стандартное устройство вывода. По окончании этого процесса она просто завершается.

Процедура fatal выводит сообщение об ошибке и завершается. Серверу также требуется эта процедура, и она пропущена в листинге только из соображений экономии места. Поскольку программы клиента и сервера компилируются отдельно и в обычной ситуации запускаются на разных машинах, код процедуры fatal не может быть разделяемым.

Эти две программы (как и другие материалы, связанные с этой книгой) можно найти на веб-сайте книги по адресу http://www.prenhall.com/tanenbaum (ссылка рядом с изображением обложки). Их можно скачать и скомпилировать на любой UNIX-системе (например, Solaris, BSD, Linux). Делается это с помощью следующих командных строк:

```
cc -o client client.c -lsocket -lnsl
cc -o server server.c -lsocket -lnsl
```

Программу для сервера можно запустить, просто набрав

```
server
```

Клиентской программе нужны два аргумента, как описывалось ранее. На веб-сайте можно найти и Windows-версии программ.

Листинг 6.1. Программы использования сокетов для клиента и сервера

```
/* На этой странице содержится клиентская программа, запрашивающая файл у серверной
программы, расположенной на следующей странице. */
/* Сервер в ответ на запрос высылает файл.*/


#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 12345 /* По договоренности между клиентом и сервером */
#define BUF_SIZE 4096 /* Размер передаваемых блоков */

int main(int argc, char *argv)
{
  int c,s,bytes;
  char buf[BUF_SIZE];   /*буфер для входящего файла */
  struct hostent *h;    /*информация о сервере */
  struct sockaddr_in channel; /*хранит IP=адрес */

  if (argc!=3) fatal("Для запуска введите: клиент имя_сервера имя_файла");
  h = gethostbyname(argv[1]);     /* поиск IP-адреса хоста */
  if(!h) fatal("Ошибка выполнения gethostbyname")
  s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
  if (s<0) fatal("Сокет");
  memset(&channel, 0, sizeof(channel));
  channel.sin_family=AF_INET;
  memcpy(&channel.sin_addr.s_addr,h>h_addr,h>h_length);
  channel.sin_port=htons(SERVER_PORT);

  c = connect(s,(struct sockaddr *) &channel, sizeof(channel));
  if (c<0) fatal("Ошибка соединения");

  /* Соединение установлено. Посылается имя файла с нулевым байтом на конце */
  write*s, argv[2], strlen(argv[2])+1);

  /* Получить файл, записать на стандартное устройство вывода */
```

```
  while (1) {
        bytes = read(s, buf, BUF_SIZE);  /* Читать из сокета */
        if (bytes <= 0) exit(0);         /* Проверка конца файла */
        write(1, buf, bytes);            /* Записать на стандартное устройство вывода
*/
  }
}
fatal(char *string)
{
  printf("%s\n", string);
  exit(1);
}

/* Код программы для сервера */
#include <sys/types.h>
#include <sys/fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 12345  /* По договоренности между клиентом и сервером */
#define BUF_SIZE 4096      /* Размер передаваемых блоков */
#define QUEUE_SIZE 10

int main(int argc, char *argv[]);
{
  int s, b, l, fd, sa, bytes, on = 1;
  char buf[BUF_SIZE];              /* буфер для исходящего файла */
  struct sockaddr_in channel;      /* содержит IP-адрес */

  /* Создать структуру адреса для привязки к сокету */
  memset(&channel, 0, sizeof(channel));    /* Нулевой канал */
  channel.sin_family = AF_INET;
  channel.sin_addr.s_addr = htonl(INADDR_ANY);
  channel.sin_port = htons(SERVER_PORT);

  /* Пассивный режим. Ожидание соединения */
  s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP); /* создать сокет */
  if (s<0) fatal("ошибка сокета");
  setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *) &on, sizeof(on));
  b = bind(s, (struct sockaddr *) &channel, sizeof(channel));
  if (b<0) fatal("Ошибка связывания");

  l = listen(s, QUEUE_SIZE);    /* Определение размера очереди */
  if (l<0) fatal("Ошибка ожидания");

  /* Теперь сокет установлен и связан. Ожидание и обработка соединения */
  while (1) {
        sa = accept(s, 0, 0);    /* Блокировать обработку запроса */
        if (sa<0) fatal("Ошибка доступа");

        read(sa, buf, BUF_SIZE);        /* считать имя файла из сокета */

        /* Получить и вернуть файл */
        fd = open(buf, O_RDONLY);       /* Открыть файл для отсылки */
```



```
        if (fd < 0) fatal("Ошибка открытия файла");

        while (1) {
              bytes = read(fd, buf, BUF_SIZE); /* Читать из файла */
              if (bytes <= 0) break;           /* Проверка конца файла */
              write(sa, buf, bytes);           /* Записать байты в сокет */
        }
        close(fd);                             /* Закрыть файл */
        close(sa);                             /* Разорвать соединение */
  }
}
```

Кстати говоря, такой сервер построен далеко не по последнему слову техники. Осуществляемая проверка ошибок минимальна, а сообщения об ошибках реализованы весьма посредственно. Понятно, что ни о какой защите информации здесь говорить не приходится, а применение аскетичных системных вызовов UNIX — это не лучшее решение с точки зрения независимости от платформы. Делаются некоторые некорректные с технической точки зрения предположения, например, о том, что имя файла всегда поместится в буфер и будет передано без ошибок. Система будет обладать низкой производительностью, поскольку все запросы обрабатываются только последовательно (используется один поток запросов). Несмотря на эти недостатки, с помощью данной программы можно организовать полноценный работающий файл-сервер для Интернета. Более подробную информацию можно найти в (Stivens, 1997).

Элементы транспортных протоколов

Транспортная служба реализуется **транспортным протоколом**, используемым между двумя транспортными сущностями. В некоторых отношениях транспортные протоколы напоминают протоколы передачи данных, подробно изучавшиеся в главе 3. Все эти протоколы, наряду с другими вопросами, занимаются обработкой ошибок, управлением очередями и потоками.

Однако у протоколов разных уровней имеется и много различий, обусловленных различиями условий, в которых работают эти протоколы, как показано на рис. 6.4. На уровне передачи данных два маршрутизатора общаются напрямую по физическому каналу, тогда как на транспортном уровне физический канал заменен целой подсетью. Это отличие оказывает важное влияние на протоколы.

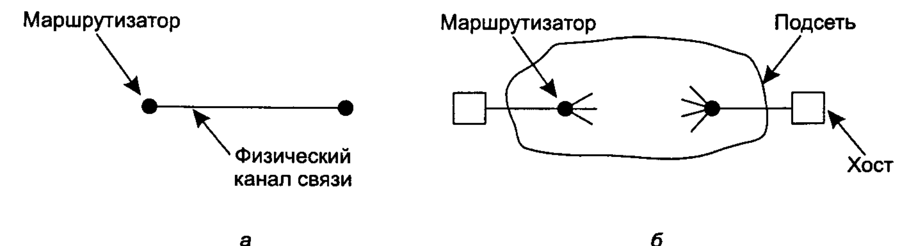


Рис. 6.4. Окружение уровня передачи данных (*а*); окружение транспортного уровня (*б*)

Во-первых, на уровне передачи данных маршрутизатору не требуется указывать, с каким маршрутизатором он хочет поговорить, — каждая выходная линия однозначно определяет маршрутизатор. На транспортном уровне требуется явно указывать адрес получателя.

Во-вторых, процесс установки соединения по проводу (рис. 6.4, *а*) прост: противоположная сторона всегда присутствует (если только она не вышла из строя). В любом случае, работы не очень много. На транспортном уровне начальная установка соединения, как будет показано далее, происходит более сложно.

Еще одно весьма досадное различие между уровнем передачи данных и транспортным уровнем состоит в том, что подсеть потенциально обладает возможностями хранения информации. Когда маршрутизатор посылает кадр, он может прибыть или потеряться, но кадр не может побродить где-то какое-то время, спрятаться в отдаленном уголке земного шара, а затем внезапно появиться в самый неподходящий момент 30 секунд спустя. Если подсеть использует дейтаграммы и адаптивную маршрутизацию, то всегда есть ненулевая вероятность того, что пакет будет храниться где-нибудь несколько секунд, а уже потом будет доставлен по назначению. Последствия способности подсети хранить пакеты иногда могут быть катастрофичными и требуют применения специальных протоколов.

Последнее различие между уровнем передачи данных и транспортным уровнем является скорее количественным, чем качественным. Буферизация и управление потоком необходимы на обоих уровнях, но наличие большого динамически изменяющегося количества соединений на транспортном уровне может потребовать принципиально другого подхода, нежели использовавшийся на уровне передачи данных. Некоторые протоколы, упоминавшиеся в главе 3, выделяют фиксированное количество буферов для каждой линии, так что, когда прибывает кадр, всегда имеется свободный буфер. На транспортном уровне из-за большого количества управляемых соединений идея выделения нескольких буферов каждому соединению выглядит не столь привлекательно. В следующих разделах мы изучим эти и другие важные вопросы.

Адресация

Когда один прикладной процесс желает установить соединение с другим прикладным процессом, он должен указать, с кем именно он хочет связаться. (У не требующей соединений транспортной службы проблемы такие же: кому следует посылать каждое сообщение?) Применяемый обычно метод состоит в определении транспортных адресов, к которым процессы могут посылать запросы на установку соединения. В Интернете такие конечные точки называются **портами**. В сетях ATM это точки доступа к службе **AAL-SAP** (Service Access Point). Мы будем пользоваться нейтральным термином **TSAP** (Transport Service Access Point — точка доступа к службам транспортного уровня). Аналогичные конечные точки сетевого уровня называются **NSAP** (Network Service Access Point — точка доступа к сетевому сервису). Примерами NSAP являются IP-адреса.

Рисунок 6.5 иллюстрирует взаимоотношения между NSAP, TSAP и транспортным соединением. Прикладные процессы как клиента, так и сервера могут

связываться с TSAP для установки соединения с удаленным TSAP. Такие соединения проходят через NSAP на каждом хосте, как показано на рисунке. TSAP нужны для того, чтобы различать конечные точки, совместно использующие NSAP, в сетях, где у каждого компьютера есть свой NSAP.

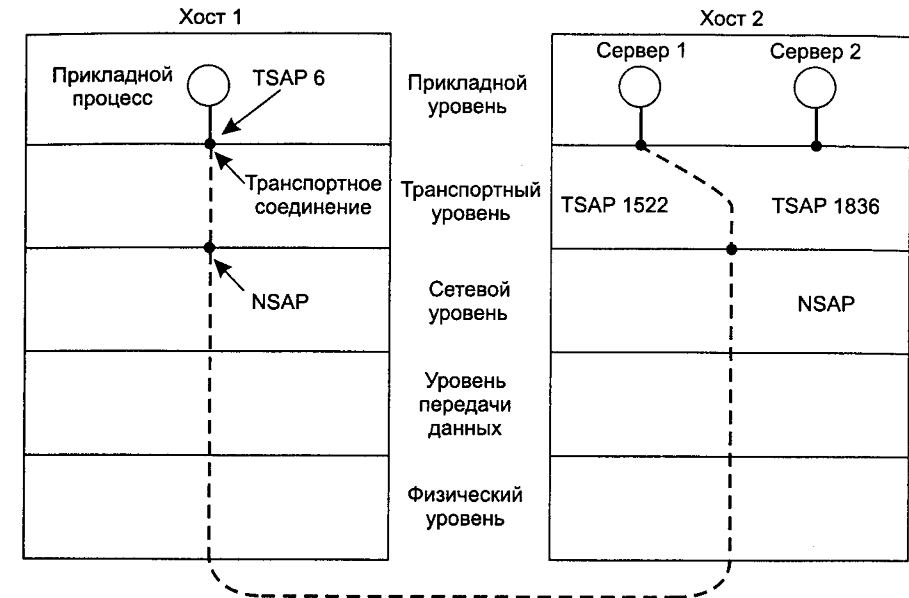


Рис. 6.5. Точки доступа к службам транспортного и сетевого уровня и транспортные соединения

Возможный сценарий для транспортного соединения выглядит следующим образом:

1. Серверный процесс хоста 2, сообщающий время суток, подсоединяется к точке доступа TSAP 1522 и ожидает входящего звонка. Вопрос о том, как процесс соединяется с TSAP, лежит за пределами сетевой модели и целиком зависит от локальной операционной системы. Например, может вызываться примитив, подобный `LISTEN`.
2. Прикладной процесс хоста 1 желает узнать, который час, поэтому он обращается к сети с запросом `CONNECT`, указывая TSAP 1208 в качестве адреса отправителя и TSAP 1522 в качестве адреса получателя. Это действие в результате приводит к установке транспортного соединения между прикладным процессом хоста 1 и сервером 1, расположенным на хосте 2.
3. Прикладной процесс отправляет запрос, надеясь выяснить, который час.
4. Сервер обрабатывает запрос и в качестве ответа посылает информацию о точном времени.
5. Транспортное соединение разрывается.

Обратите внимание на то, что на хосте 2 могут располагаться и другие серверы, соединенные со своими TSAP и ожидающие входящих запросов на соединение, приходящих с того же NSAP.

Нарисованная картинка всем хороша, но мы обошли стороной один маленький вопрос: как пользовательский процесс хоста 1 узнает, что сервер, сообщающий время, соединен с TSAP 1522? Возможно, сервер, сообщающий время, подключается к TSAP 1522 в течение долгих лет, и постепенно об этом узнают все пользователи сети. В этом случае службы имеют постоянные TSAP-адреса, хранящиеся в файлах, расположенных в известных местах, таких как etc/services в UNIX-системах. В файлах перечисляются серверы, за которыми жестко закреплены определенные порты.

Хотя постоянные TSAP-адреса могут хорошо подходить для небольшого количества никогда не меняющихся ключевых служб (например, таких как веб-сервер), в общем случае пользовательские процессы часто хотят пообщаться с другими пользовательскими процессами, существующими только в течение короткого времени и не обладающими постоянными TSAP-адресами, известным всем заранее. Кроме того, при наличии большого количества серверных процессов, большая часть которых редко используется, слишком расточительным делом оказывается поддержка всех их в активном состоянии с постоянными TSAP-адресами. То есть требуется другая модель.

Одна такая модель показана в упрощенном виде на рис. 6.6. Она называется **протоколом начального соединения**. Вместо того чтобы назначать всем возможным серверам хорошо известные TSAP-адреса, каждая машина, желающая предоставлять услуги удаленным пользователям, обзаводится специальным **обрабатывающим сервером**, действующим как прокси (посредник) для менее активно используемых серверов. Он прослушивает одновременно несколько портов, ожидая запроса на соединение. Потенциальные пользователи этой услуги начинают с того, что посылают запрос CONNECT, указывая TSAP-адрес нужной им службы. Если никакой сервер их не ждет, они получают соединение с обрабатывающим сервером, как показано на рис. 6.6, *а*.

Получив запрос, обрабатывающий сервер порождает подпроцесс на запрошенном сервере, позволяя ему унаследовать существующее соединение с пользователем. Новый сервер выполняет требуемую работу, в то время как обрабатывающий сервер возвращается к ожиданию новых запросов, как показано на рис. 6.6, *б*.

Хотя протокол начального соединения прекрасно работает с серверами, которые можно создавать по мере надобности, есть много ситуаций, в которых службы существуют независимо от обрабатывающего сервера. Например, файловый сервер должен работать на специальном оборудовании (машине с диском) и не может быть создан на ходу, когда кто-нибудь захочет к нему обратиться.

Чтобы справиться с этой ситуацией, часто используется другая схема. В этой модели используется специальный процесс, называющийся **сервером имен** или иногда **каталоговым сервером**. Чтобы найти TSAP-адрес, соответствующий данному имени службы, например «время суток», пользователь устанавливает соединение с сервером имен (TSAP-адрес которого всем известен). Затем пользователь посылает сообщение с указанием названия нужной ему услуги, и сервер имен сообщает ему TSAP-адрес этой службы. После этого пользователь разрывает соединение с сервером имен и устанавливает новое соединение с нужной ему службой.

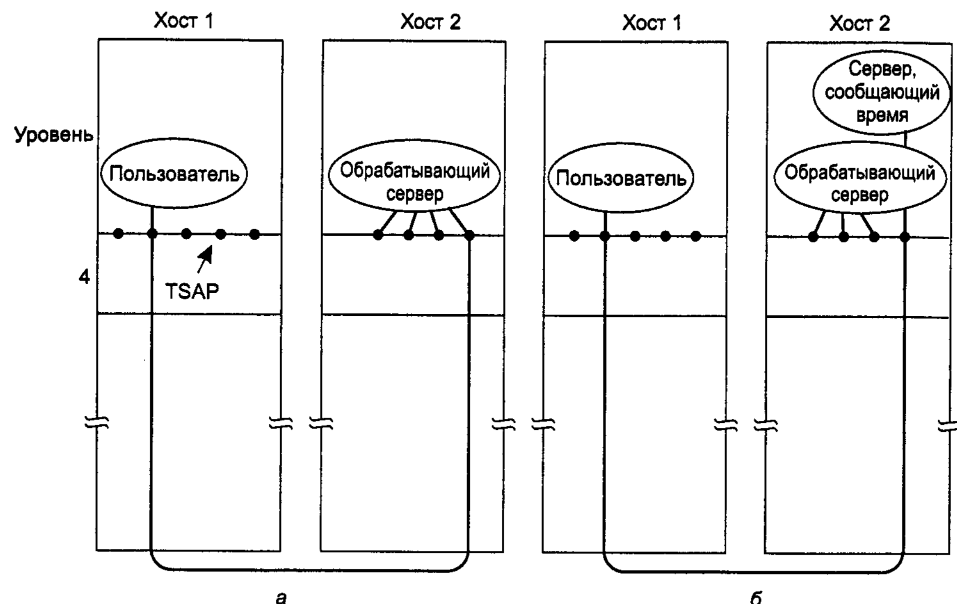


Рис. 6.6. Пользовательский процесс хоста 1 устанавливает соединение с сервером хоста 2

В этой модели, когда создается новая служба, она должна зарегистрироваться на сервере имен, сообщив ему название услуги (обычно строка ASCII) и TSAP-адрес. Сервер имен сохраняет полученную информацию в своей базе данных, чтобы иметь возможность отвечать на будущие запросы.

Функция сервера имен аналогична работе оператора телефонной справочной службы — он преобразует имена в номера. Как и в телефонной системе, важно, чтобы TSAP-адрес сервера имен (или обрабатывающего сервера в протоколе начального соединения) был действительно хорошо известен. Если вы не знаете номера телефонной справочной, вы не сможете позвонить оператору. Если вы полагаете, что номер справочной является очевидным, попытайтесь угадать его, находясь в другой стране.

Установка соединения

Установка соединения, хотя и просто звучит, неожиданно оказывается весьма непростым делом. На первый взгляд, должно быть достаточно одной транспортной сущности, для того чтобы послать адресату TPDU-модуль с запросом соединения CONNECTION REQUEST и услышать в ответ CONNECTION ACCEPTED (соединение принято). Неприятность заключается в том, что сеть может потерять, задержать или дублировать пакеты.

Представьте себе подсеть настолько перегруженную, что подтверждения практически никогда не доходят вовремя, каждый пакет опаздывает и пересылается повторно по два-три раза. Предположим, что подсеть основана на дейтаграммах и что каждый пакет следует по своему маршруту. Некоторые пакеты могут застрять в давке и прийти с большим опозданием.

Самый кошмарный сценарий выглядит следующим образом. Пользователь устанавливает соединение с банком и посылает сообщение с требованием банку перевести крупную сумму денег на счет не совсем надежного человека, после чего разрывает соединение. К несчастью, каждый пакет этого сценария дублируется и сохраняется в подсети. После разрыва соединения эти дубликаты пакетов наконец, добираются до адресата в нужном порядке. У банка нет способа определить, что это дубликаты. Он решает, что это вторая независимая транзакция, и еще раз переводит деньги. В оставшейся части этого раздела мы будем изучать проблему задержавшихся дубликатов, уделяя особое внимание алгоритмам, устанавливающим соединение надежным образом.

Основная проблема заключается в наличии задержавшихся дубликатов. Эту проблему можно попытаться решить несколькими способами, ни один из которых, на самом деле, не является удовлетворительным. Так, например, можно использовать одноразовые транспортные адреса. При таком подходе каждый раз, когда требуется транспортный адрес, генерируется новый адрес. Когда соединение разрывается, этот адрес уничтожается. Такая стратегия делает невозможной реализацию модели обрабатывающего сервера, изображенной на рис. 6.6.

Другая возможность состоит в том, что каждому соединению присваивается идентификатор соединения (то есть последовательный номер, который возрастает на единицу для каждого установленного соединения), выбираемый инициатором соединения и помещаемый в каждый TPDU-модуль, включая тот, который содержит запрос на соединение. После разрыва каждого соединения каждая транспортная сущность может обновить таблицу, в которой хранятся устаревшие соединения в виде пар (одноранговая транспортная сущность, идентификатор соединения). Для каждого приходящего запроса соединения может быть проверено, не хранится ли уже его идентификатор в таблице (он мог остаться там со времен разорванного ранее соединения).

К сожалению, у этой схемы есть существенный изъян: требуется, чтобы каждая транспортная сущность хранила неопределенно долго некоторое количество информации об истории соединений. Если машина выйдет из строя и потеряет свою память, она не сможет определить, какие соединения уже использовались, а какие нет.

Вместо этого можно применить другой подход. Следует разработать механизм, уничтожающий устаревшие заблудившиеся пакеты и не позволяющий им существовать в сети бесконечно долго. Если мы сможем гарантировать, что ни один пакет не сможет жить дольше определенного периода времени, проблема станет более управляемой.

Время жизни пакета может быть ограничено до известного максимума с помощью одного из следующих методов:

1. Проектирование подсети с ограничениями.

2. Помещение в каждый пакет счетчика транзитных участков.
3. Помещение в каждый пакет временного штампа.

К первому способу относятся все методы, предотвращающие зацикливание пакетов, в комбинации с ограничением задержки, вызванной перегрузкой по самому длинному возможному пути. Второй метод заключается в начальной установке счетчика на определенное значение и уменьшении на единицу этого значения на каждом маршрутизаторе. Сетевой протокол передачи данных просто игнорирует все пакеты, значение счетчика которых дошло до нуля. Третий метод состоит в том, что в каждый пакет помещается время его создания, а маршрутизаторы договариваются игнорировать все пакеты старше определенного времени. Для последнего метода требуется синхронизация часов маршрутизаторов, что само по себе является нетривиальной задачей. Впрочем, иногда синхронизацию удается производить с помощью внешнего источника, например GPS или радиостанции, передающей сигналы точного времени.

На практике нужно гарантировать не только то, что пакет мертв, но и что все его подтверждения также мертвы. Поэтому вводится некий интервал времени *T*, который в несколько раз превышает максимальное время жизни пакета. На какое число умножается максимальное время жизни пакета, зависит от протокола, и это влияет только на длительность интервала времени *T*. Если подождать в течение интервала времени *T* секунд момента отправки пакета, то можно быть уверенным, что все его следы уничтожены и что пакет не возникнет вдруг как гром среди ясного неба.

При ограниченном времени жизни пакетов можно разработать надежный способ безопасной установки соединений. Автором описанного далее метода является Томлинсон (Tomlinson) (1975). Этот метод решает проблему, но привносит некоторые собственные странности. Впоследствии (в 1978 году) этот метод был улучшен Саншайном (Sunshine) и Далалом (Dalal). Его варианты широко применяются на практике, и одним из примеров применения является TCP.

Чтобы обойти проблему потери машиной памяти предыдущих состояний (при выходе ее из строя), Томлинсон предложил снабдить каждый хост часами. Часы разных хостов нужно было синхронизировать. Предполагалось, что часы представляют собой двоичный счетчик, увеличивающийся через равные интервалы времени. Кроме того, число разрядов счетчика должно равняться числу битов в последовательных номерах (или превосходить его). Последнее и самое важное предположение состоит в том, что часы продолжают идти, даже если хост зависает.

Основная идея заключается в том, что два одинаково пронумерованных TPDU-модуля никогда не отправляются одновременно. При установке соединения младшие *k* битов часов используются в качестве начального порядкового номера (также *k* битов). Таким образом, в отличие от протоколов, описанных в главе 3, каждое соединение начинает нумерацию своих TPDU-модулей с разных чисел. Диапазон этих номеров должен быть достаточно большим, чтобы к тому моменту, когда порядковые номера сделают полный круг, старые TPDU-модули с такими же номерами уже давно исчезли. Линейная зависимость порядковых номеров от времени показана на рис. 6.7.

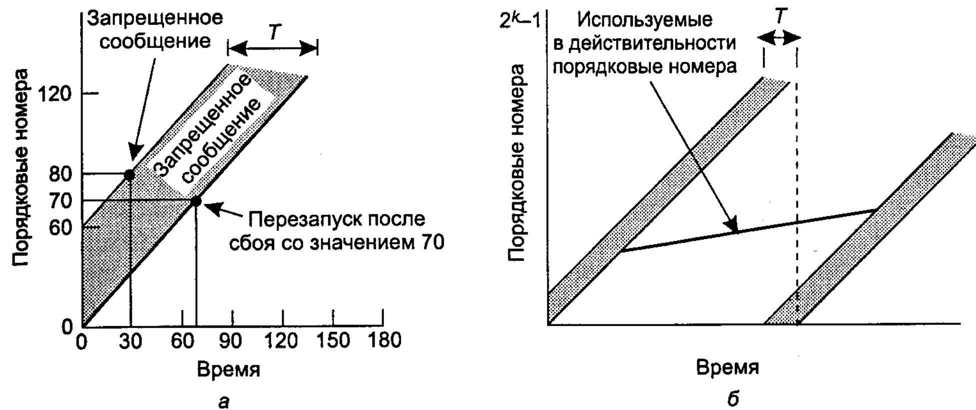


Рис. 6.7. TPDU-модули не могут заходить в запретную зону (*а*); проблема ресинхронизации (*б*)

Как только обе транспортные сущности договариваются о начальном порядковом номере, для управления потоком данных может применяться любой протокол скользящего окна. В действительности график порядковых номеров (показанный жирной линией) не прямой, а ступенчатый, так как показания часов увеличиваются дискретно. Впрочем, для простоты мы проигнорируем эту деталь.

Проблема возникает при выходе хоста из строя. Когда он снова включается, его транспортная сущность не помнит, где она находилась в пространстве порядковых номеров. Одно из решений заключается в том, что транспортная сущность должна подождать T секунд после восстановления, чтобы всех старых TPDU-модулей в сети не осталось. Однако в сложных сетях такая стратегия выглядит непривлекательно, так как значение T может оказаться довольно большим.

Чтобы не терять дополнительно T секунд после восстановления, необходимо ввести новое ограничение на использование порядковых номеров. Легче всего понять необходимость этого ограничения на примере. Пусть максимальное время жизни пакета $T = 60$ с, а часы тикают один раз в секунду. Как показано жирной линией на рис. 6.7, *а*, начальный порядковый номер для соединения, открываемого в момент времени x, будет равен x. Представим себе, что в момент времени $t = 30$ с по соединению номер 5 (открытому ранее) посылается обычный информационный TPDU-модуль, которому дается порядковый номер 80. Назовем его TPDU X. Немедленно после TPDU X хост сбрасывается и затем быстро перезагружается. В момент времени $t = 60$ с он начинает повторно открывать соединения с 0 по 4. В момент времени $t = 70$ с хост открывает повторно соединение 5, используя, как и требуется, начальный порядковый номер 70. В течение последующих 15 с он посылает TPDU-модули с порядковыми номерами от 70 до 80. Таким образом, в момент времени $t = 85$ с новый TPDU-модуль с порядковым номером 80 и номером соединения 5 попадает в подсеть. К несчастью, TPDU X еще жив. Если он прибудет к получателю ранее нового TPDU 80, TPDU X будет принят, а правильный TPDU 80 будет отвергнут как дубликат.

Чтобы избежать подобных ситуаций, необходимо запретить использование порядковых номеров на период времени T. Недопустимые комбинации времени и порядкового номера обозначены на рис. 6.7, *а* в виде **запретной зоны**. Прежде чем послать TPDU-модуль по любому соединению, транспортная сущность должна сначала прочитать показания часов и убедиться, что она не находится в запретной зоне.

Неприятности у протокола могут возникнуть по двум причинам. Если хост посылает слишком быстро и слишком много данных, кривая используемых в действительности порядковых номеров может оказаться круче линии зависимости начальных номеров от времени. Это означает, что скорость передачи данных в каждом открытом соединении должна быть ограничена одним TPDU-модулем за единицу времени. Кроме того, это означает, что транспортная сущность после восстановления, прежде чем открывать новое соединение, должна подождать, пока изменят свое состояние часы, чтобы один и тот же номер не использовался дважды. Следовательно, интервал изменения состояния часов должен быть коротким (несколько миллисекунд).

К сожалению, в запретную зону можно попасть не только снизу, передавая данные слишком быстро. Как видно из рис. 6.7, *б*, при любой скорости передачи данных, меньшей скорости часов, кривая используемых в действительности порядковых номеров попадет в запретную зону слева. Чем круче наклон этой кривой, тем дольше придется ждать этого события. Как уже упоминалось, непосредственно перед отправкой TPDU-модуля транспортная сущность должна проверить, не попадет ли она в ближайшее время в запретную зону, — и, если она находится близко от запретной зоны, отложить оправку TPDU-модуля на T секунд либо изменить синхронизацию порядковых номеров.

Использующий показания часов метод решает проблему опаздывающих дубликатов для информационных TPDU-модулей, но чтобы воспользоваться этим методом, соединение необходимо вначале установить. Так как управляющие TPDU-модули также могут задержаться в пути, возникает потенциальная проблема договоренности обеих сторон о начальном порядковом номере. Предположим, что для установления соединения хост 1 посылает TPDU-модуль с запросом соединения CONNECTION REQUEST, содержащим предлагаемый начальный порядковый номер и номер порта получателя, удаленному хосту 2. Получатель подтверждает получение этого запроса, посылая обратно TPDU-модуль CONNECTION ACCEPTED (соединение принято). Если оригинальный TPDU-модуль CONNECTION REQUEST будет потерян, а его дубликат неожиданно появится на хосте 2, соединение будет установлено некорректно.

Для разрешения этой проблемы Томлинсон (1975) предложил **«тройное рукопожатие»**. Этот протокол установки соединения не требует, чтобы обе стороны начинали передачу с одинаковыми порядковыми номерами, поэтому он может применяться вместе с методами синхронизации, отличными от метода глобальных часов. Нормальная процедура установки соединения показана на рис. 6.8, *а*. Хост 1 инициирует установку, выбирая порядковый номер x, и посылает TPDU-модуль CONNECTION REQUEST, содержащий этот начальный порядковый номер, хосту 2. Хост 2 отвечает TPDU-модулем ACK, подтверждая x и объявляя свой начальный порядковый номер y. Наконец, хост 1 подтверждает выбранный

хостом 2 начальный порядковый номер в первом посылаемом им информационном TPDU-модуле.

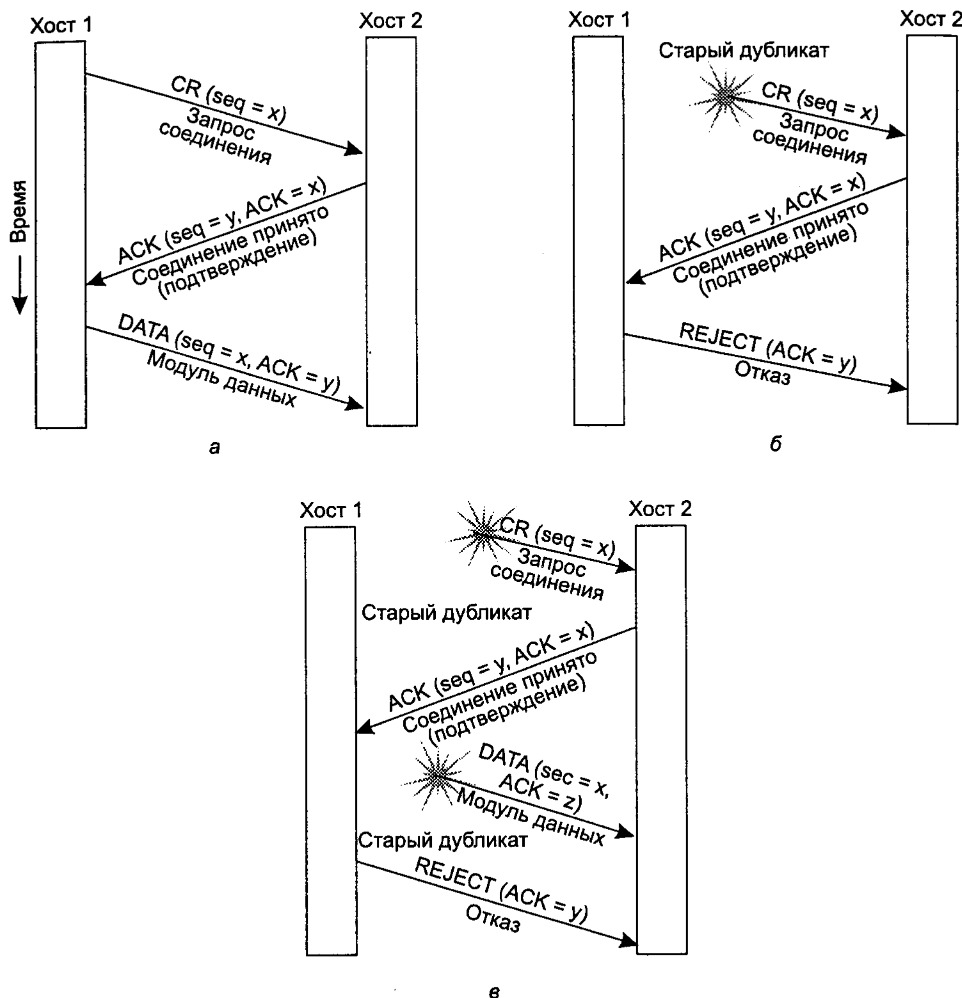


Рис. 6.8. Три сценария установки соединения с помощью «тройного рукопожатия». CR означает CONNECTION REQUEST. Нормальная работа (*а*); появление старого дубликата CONNECTION REQUEST (*б*); дубликат модуля CONNECTION REQUEST и дубликат модуля ACK (*в*)

Рассмотрим теперь работу «тройного рукопожатия» в присутствии задержавшегося дубликата управляющего TPDU-модуля. На рис. 6.8, *б* первый TPDU-модуль представляет собой задержавшийся дубликат модуля CONNECTION REQUEST от старого соединения. Этот TPDU-модуль прибывает на хост 2 тайком от хоста 1. Хост 2 реагирует на этот TPDU-модуль отправкой хосту 1 TPDU-модуля ACK, таким образом прося хост 1 подтвердить, что тот действительно пытался установить новое соединение. Когда хост 1 отказывается это сделать, хост 2 понимает, что он был обманут задержавшимся дубликатом, и прерывает соединение. Таким образом, задержавшийся дубликат не причиняет вреда.

При наихудшем сценарии оба TPDU-модуля — CONNECTION REQUEST и ACK — блуждают по подсети. Этот случай показан на рис. 6.8, *в*. Как и в предыдущем примере, хост 2 получает задержавшийся модуль CONNECTION REQUEST и отвечает на него. В этом месте следует обратить внимание на то, что хост 2 предложил использовать y в качестве начального порядкового номера для трафика от хоста 2 к хосту 1, хорошо зная, что TPDU-модулей, содержащих порядковый номер y, или их подтверждений в данный момент в сети нет. Когда хост 2 получает второй задержавшийся TPDU-модуль, он понимает, что это дубликат, так как в этом модуле подтверждается не y, а z. Здесь важно понять, что не существует такой комбинации TPDU-модулей, которая заставила бы протокол ошибиться и случайно установить соединение, когда оно никому не нужно.

Разрыв соединения

Разорвать соединение проще, чем установить. Тем не менее, здесь также имеются подводные камни. Как уже было сказано, существует два стиля разрыва соединения: асимметричный и симметричный. Асимметричный разрыв связи соответствует принципу работы телефонной системы: когда одна из сторон вешает трубку, связь прерывается. При симметричном разрыве соединение рассматривается в виде двух отдельных однонаправленных связей, и требуется раздельное завершение каждого соединения.

Асимметричный разрыв связи является внезапным и может привести к потере данных. Рассмотрим сценарий, показанный на рис. 6.9. После установки соединения хост 1 посылает TPDU-модуль, который успешно добирается до хоста 2. Затем хост 1 посылает другой TPDU-модуль. К несчастью, хост 2 посылает DISCONNECTION REQUEST (запрос разъединения) прежде, чем прибывает второй TPDU-модуль. В результате соединение разрывается, а данные теряются.

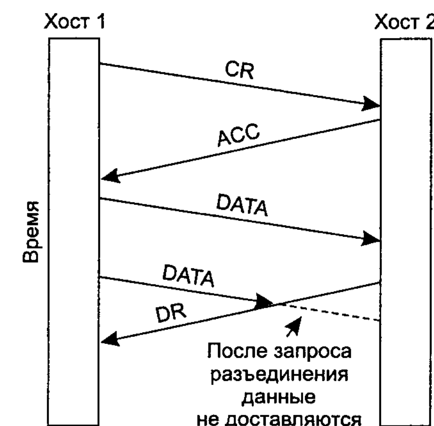


Рис. 6.9. Внезапное разъединение с потерей данных

Очевидно, требуется более сложный протокол, позволяющий избежать потери данных. Один из способов состоит в использовании симметричного разъединения, при котором каждое направление разъединяется независимо от другого. В этом случае хост может продолжать получать данные даже после того, как сам послал запрос разъединения.

Симметричное разъединение хорошо подходит для тех случаев, когда у каждой стороны есть фиксированное количество данных для передачи и каждая сторона точно знает, когда эти данные заканчиваются. В других случаях определить, что работа закончена и соединение может быть прервано, не так просто. Можно представить себе протокол, в котором хост 1 говорит: «Я закончил. Вы тоже закончили?» Если хост 2 отвечает: «Я тоже закончил. До свидания», соединение можно безо всякого риска разъединять.

К сожалению, этот протокол работает не всегда. Существует знаменитая проблема, называемая **проблемой двух армий**. Представьте, что армия белых расположилась в долине, как показано на рис. 6.10. На возвышенностях по обеим сторонам долины расположились две армии синих. Белая армия больше, чем любая из армий синих, но вместе синие превосходят белых. Если любая из армий синих атакует белых в одиночку, она потерпит поражение, но если синие сумеют атаковать белых одновременно, они могут победить.

Рис. 6.10. Проблема двух армий

Синие армии хотели бы синхронизировать свое выступление. Однако единственный способ связи заключается в отправке вестового пешком по долине, где он может быть схвачен, а донесение потеряно (то есть приходится пользоваться ненадежным каналом). Спрашивается: существует ли протокол, позволяющий армиям синих победить?

Предположим, командир 1-й армии синих посылает следующее сообщение: «Я предлагаю атаковать 29 марта, на рассвете. Сообщите ваше мнение». Теперь предположим, что сообщение успешно доставляется и что командир 2-й армии синих соглашается, а его ответ успешно доставляется обратно в 1-ю армию синих. Состоится ли атака? Вероятно, нет, так как командир 2-й армии не уверен, что его ответ получен. Если нет, то 1-я армия синих не будет атаковать, и было бы глупо с его стороны в одиночку ввязываться в сражение.

Теперь улучшим протокол с помощью «тройного рукопожатия». Инициатор оригинального предложения должен подтвердить ответ. Но и в этом случае останется неясным, было ли доставлено последнее сообщение. Протокол четырехкратного рукопожатия здесь также не поможет.

В действительности, можно доказать, что протокола, решающего данную проблему, не существует. Предположим, что такой протокол все же существует. В этом случае последнее сообщение протокола либо является важным, либо нет. Если оно не является важным, удалим его (а также все остальные несущественные сообщения), пока не останется протокол, в котором все сообщения являются существенными. Что произойдет, если последнее сообщение не дойдет до адресата? Мы только что сказали, что сообщение является важным, поэтому, если оно потеряется, атака не состоится. Поскольку отправитель последнего сообщения никогда не сможет быть уверен в его получении, он не станет рисковать. Другая синяя армия это знает и также воздержится от атаки.

Чтобы увидеть, какое отношение проблема двух армий имеет к разрыву соединения, просто замените слово «атаковать» на «разъединить». Если ни одна из сторон не готова разорвать соединение до тех пор, пока она не уверена, что другая сторона также готова к этому, то разъединение не произойдет никогда.

На практике к разрыву соединения обычно готовятся лучше, чем к атаке, поэтому ситуация не совсем безнадежна. На рис. 6.11 показаны четыре сценария разъединения, использующих «тройное рукопожатие». Хотя этот протокол и не безошибочен, обычно он работает успешно.

На рис. 6.11, *а* показан нормальный случай, в котором один из пользователей посылает запрос разъединения DR (DISCONNECTION REQUEST), чтобы инициировать разрыв соединения. Когда он прибывает, получатель посылает обратно также запрос разъединения DR и включает таймер на случай, если запрос потеряется. Когда запрос прибывает, первый отправитель посылает в ответ на него TPDU-модуль с подтверждением АСК и разрывает соединение. Наконец, когда прибывает АСК, получатель также разрывает соединение. Разрыв соединения означает, что транспортная сущность удаляет информацию об этом соединении из своей таблицы открытых соединений и сигнализирует о разрыве соединения владельцу соединения (пользователю транспортной службы). Эта процедура отличается от использования пользователем примитива DISCONNECT.

Если последний TPDU-модуль с подтверждением теряется (рис. 6.11, *б*), ситуацию спасает таймер. Когда время истекает, соединение разрывается в любом случае.

Теперь рассмотрим случай потери второго запроса разъединения DR. Пользователь, инициировавший разъединение, не получит ожидаемого ответа, у него истечет время ожидания, и он начнет все сначала. На рис. 6.11, *в* показано, как это происходит в случае, если все последующие запросы и подтверждения успешно доходят до адресатов.

Последний сценарий (рис. 6.11, *г*) аналогичен предыдущему — с той лишь разницей, что в этом случае предполагается, что все повторные попытки передать

запрос разъединения DR также терпят неудачу, поскольку все TPDU-модули теряются. После N повторных попыток отправитель наконец сдается и разрывает соединение. Тем временем у получателя также истекает время, и он тоже разрывает соединение.

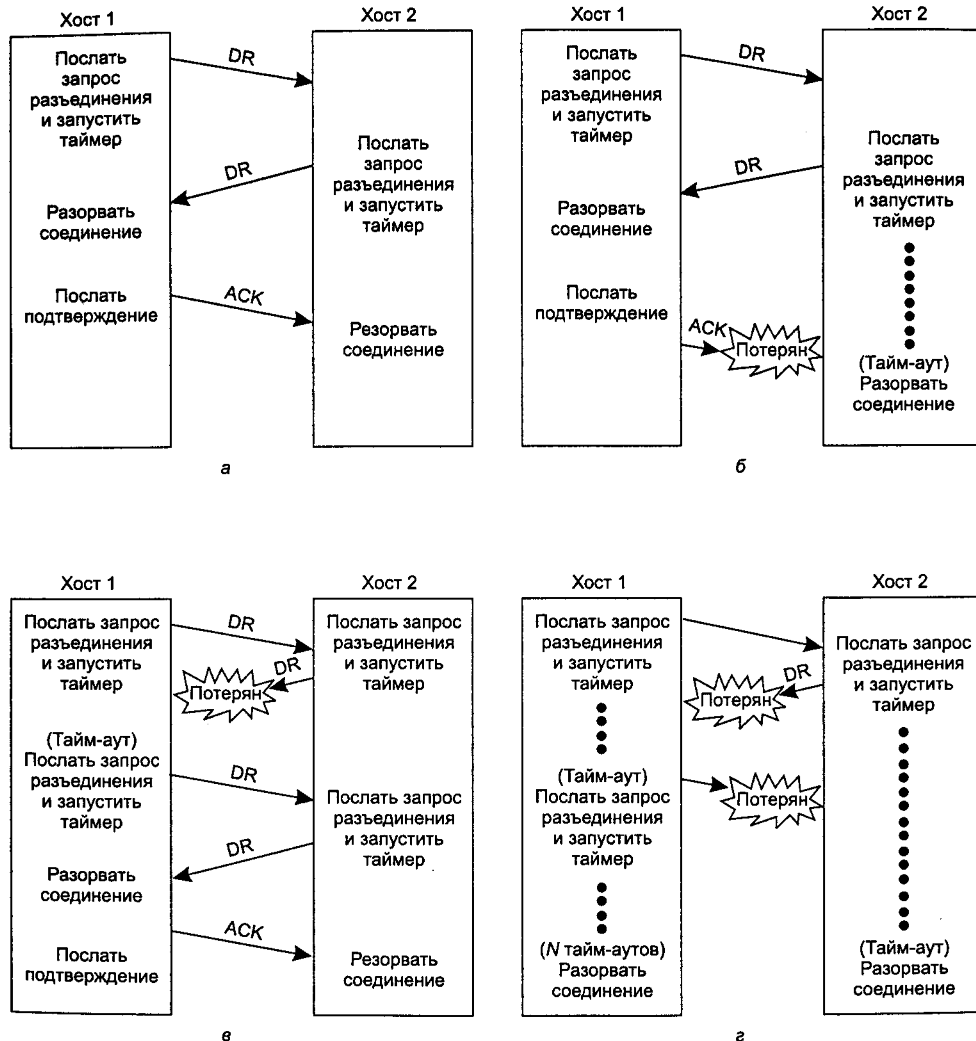


Рис. 6.11. Четыре сценария разрыва соединения: нормальный случай «тройного рукопожатия» (*а*); потеряно последнее подтверждение (*б*); потерян ответ (*в*); потерян ответ и последующие запросы (*г*)

Хотя такого протокола обычно бывает вполне достаточно, теоретически он может ошибиться, если потеряются начальный запрос разъединения DR и все N повторных попыток. Отправитель сдается и разрывает соединение, тогда как другая сторона ничего не знает о попытках разорвать связь и сохраняет активность. В результате получается полуоткрытое соединение.

Этой ситуации можно было бы избежать, если не позволять отправителю сдаваться после N повторных попыток, а заставить его продолжать попытки, пока не будет получен ответ. Однако если другой стороне будет разрешено разрывать связь по таймеру, тогда отправитель действительно будет вечно повторять попытки, так как ответа он не получит никогда. Если же получающей стороне также не разрешать разрывать соединение по таймеру, тогда протокол зависнет в ситуации, изображенной на рис. 6.11, *г*.

Чтобы удалять полуоткрытые соединения, можно применять правило, гласящее, что если по соединению в течение определенного времени не прибывает ни одного TPDU-модуля, соединение автоматически разрывается. Таким образом, если одна сторона отсоединится, другая обнаружит отсутствие активности и также отсоединится. Для реализации этого правила каждая сторона должна управлять таймером, перезапускаемым после передачи каждого TPDU-модуля. Если этот таймер срабатывает, посылается пустой TPDU-модуль — лишь для того, чтобы другая сторона не повесила трубку. С другой стороны, если применяется правило автоматического разъединения и теряется слишком большое количество TPDU-модулей подряд, то соединение автоматически разрывается сначала одной стороной, а затем и другой.

На этом мы заканчиваем обсуждение этого вопроса, но теперь должно быть ясно, что разорвать соединение без потери данных не так просто, как это кажется на первый взгляд.

Управление потоком и буферизация

Изучив процессы установки и разрыва соединения, рассмотрим, как происходит управление соединением во время его использования. Одним из ключевых вопросов, уже обсуждавшихся ранее, является управление потоком. В некоторых аспектах проблема управления потоком на транспортном уровне аналогична той же проблеме на уровне передачи данных, но в то же время они различаются по целому ряду аспектов. Основное сходство состоит в том, что на обоих уровнях для согласования скорости передатчика и приемника требуется некая схема, например протокол скользящего окна. Основным различием является то, что у маршрутизатора обычно относительно небольшое количество линий, тогда как хост может иметь большое число соединений. Из-за этого различия использование на транспортном уровне стратегии буферизации, применяемой на уровне передачи данных, является непрактичным.

В протоколах передачи данных, обсуждавшихся в главе 3, кадры буферировались как отправляющим, так и получающим маршрутизаторами. Например, в протоколе 6 и отправитель, и получатель должны были отвести по MAX_SEQ + 1 буферов для каждой линии — половину для входного потока, половину для выходного. Так, для хоста с максимальным количеством соединений, равным 64, и 4-битовым порядковым номером этот протокол потребует 1024 буфера.

На уровне передачи данных отправитель должен буферизировать выходные кадры, так как может понадобиться их повторная передача. Если подсеть предоставляет дейтаграммные услуги, отправляющая транспортная сущность должна также использовать буфер по той же самой причине. Если получатель знает, что отправитель хранит в буферах все TPDU-модули до тех пор, пока они не будут подтверждены, тогда получатель сможет использовать свои буферы по своему усмотрению. Например, получатель может содержать единый буферный накопитель, используемый всеми соединениями. Когда приходит TPDU-модуль, предпринимается попытка динамически выделить ему новый буфер. Если это удается, то TPDU-модуль принимается, в противном случае он отвергается. Поскольку отправитель готов к тому, чтобы передавать потерянные TPDU-модули повторно, игнорирование TPDU-модулей получателем не наносит вреда, хотя и расходует некоторые ресурсы. Отправитель просто повторяет попытки до тех пор, пока не получит подтверждения.

В итоге, если сетевая служба является ненадежной, отправитель должен буферизировать все посланные TPDU-модули, как и на уровне передачи данных. Однако при надежной сетевой службе возможны другие варианты. В частности, если отправитель знает, что у получателя всегда есть место в буфере, ему не нужно хранить копии посланных TPDU-модулей. Однако если получатель не может гарантировать, что каждый приходящий TPDU-модуль будет принят, получателю придется буферизировать посланные TPDU-модули. В последнем случае отправитель не может доверять подтверждениям сетевого уровня, так как они означают лишь то, что TPDU-модуль прибыл, но не означают, что он был принят. Позднее мы вернемся к этому важному пункту.

Даже если получатель соглашается буферизовать принимаемые TPDU-модули, остается вопрос о том, какого размера должен быть буфер. Если большинство TPDU-модулей имеют примерно одинаковые размеры, естественно организовать буферы в виде массива буферов равной величины, каждый из которых может вместить один TPDU-модуль, как показано на рис. 6.12, *а*. Однако если TPDU-модули сильно различаются по размеру — от нескольких символов, набранных на терминале, до нескольких тысяч символов при передаче файлов, — то массив из буферов фиксированного размера окажется неудобным. Если размер буфера выбирать равным наибольшему возможному TPDU-модулю, то при хранении небольших TPDU-модулей память будет расходоваться неэффективно. Если же сделать размер буфера меньшим, тогда для хранения большого TPDU-модуля потребуется несколько буферов с сопутствующими сложностями.

Другой метод решения указанной проблемы состоит в использовании буферов переменного размера, как показано на рис. 6.12, *б*. Преимущество этого метода заключается в более оптимальном использовании памяти, но платой за это является усложненное управление буферами. Третий вариант состоит в выделении соединению единого большого циклического буфера, как показано на рис. 6.12, *в*. Такая схема также довольно хорошо использует память, если все соединения сильно нагружены, однако при невысокой нагрузке некоторых соединений ее эффективность снижается.

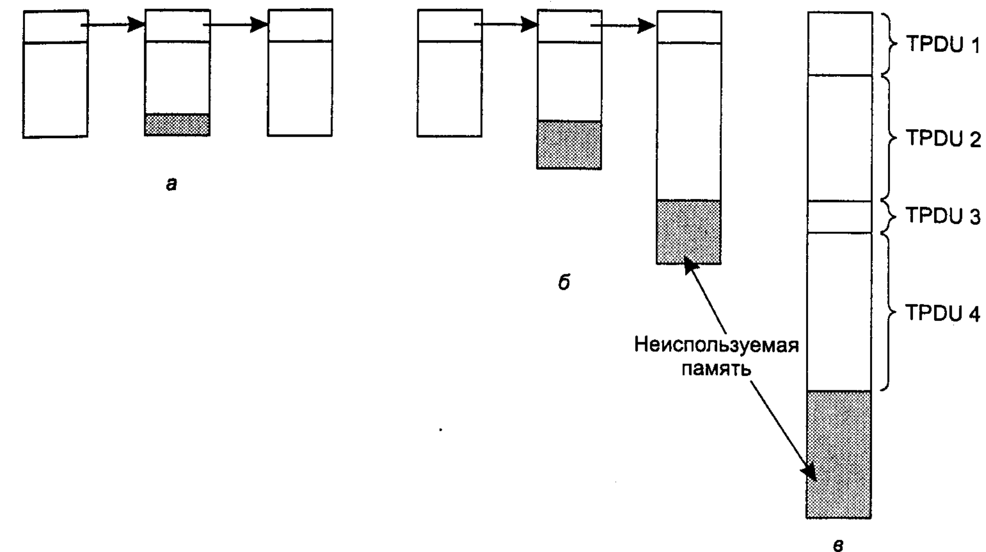


Рис. 6.12. Цепочка буферов фиксированного размера (*а*); цепочка буферов фиксированного размера (*б*); один большой циклический буфер для одного соединения (*в*)

Выбор компромиссного решения между буферизацией у отправителя и у получателя зависит от типа трафика соединения. Если трафик импульсный, небольшой мощности, как, например, трафик интерактивного терминала, лучше не выделять никаких буферов, а получать их динамически на обоих концах. Так как отправитель не уверен, что получатель сможет получить буфер, он должен будет сохранять копию TPDU-модуля, пока не получит относящееся к нему подтверждение. С другой стороны, при передаче файла будет лучше, если получатель выделит целое окно буферов, чтобы данные могли передаваться с максимальной скоростью. Таким образом, при пульсирующем трафике малой мощности буферизацию лучше производить у отправителя, а для трафика с постоянной большой скоростью — у получателя.

При открытии и закрытии соединений и при изменении формы трафика отправитель и получатель должны динамически изменять выделенные буферы. Следовательно, транспортный протокол должен позволять отправителю посылать запросы на выделение буфера на другом конце. Буферы могут выделяться для каждого соединения или коллективно на все соединения между двумя хостами. В качестве альтернативы запросам буферов получатель, зная состояние своих буферов, но не зная, какой трафик ему будет предложен, может сообщить отправителю, что он зарезервировал для него X буферов. При увеличении числа открытых соединений может потребоваться уменьшить количество или размеры выделенных буферов. Протокол должен предоставлять такую возможность.

В отличие от протоколов скользящего окна, описанных в главе 3, для реализации динамического выделения буферов следует отделить буферизацию от подтверждений. Динамическое выделение буферов означает, на самом деле, использование окна переменного размера. Вначале отправитель, основываясь на своих

потребностях, запрашивает определенное количество буферов. Получатель выделяет столько, сколько может. При отправлении каждого TPDU-модуля отправитель должен уменьшать на единицу число буферов, а когда это число достигнет нуля, он должен остановиться. Получатель отправляет обратно на попутных TPDU-модулях отдельно подтверждения и информацию об имеющихся у него свободных буферах.

На рис. 6.13 показан пример управления динамическим окном в дейтаграммной подсети с 4-битными порядковыми номерами. Предположим, что запрос на предоставление буферов пересылается в отдельных TPDU-модулях, а не добирается «автостопом» на попутных модулях. Вначале хост *A* запрашивает 8 буферов, но ему выделяется только 4. Затем он посылает три TPDU-модуля, из которых последний теряется. На шаге 6 хост *A* получает подтверждение получения посланных им TPDU-модулей 0 и 1, разрешает хосту *A* освободить буферы и послать еще три модуля (с порядковыми номерами 2, 3 и 4). Хост *A* знает, что TPDU-модуль номер 2 он уже посылал, поэтому он думает, что может послать модули 3 и 4, что он и делает. На этом шаге он блокируется, так как его счетчик буферов достиг нуля, и ждет предоставления новых буферов. На шаге 9 наступает тайм-аут хоста *A*, так как он до сих пор не получил подтверждения для TPDU-модуля 2. Этот модуль посылается еще раз. В строке 10 хост *B* подтверждает получение всех TPDU-модулей, включая 4-й, но отказывается предоставлять буферы хосту *A*. Такая ситуация невозможна в протоколах с фиксированным размером окна, описанных в главе 3. Следующий TPDU-модуль, посланный хостом *B*, разрешает хосту *A* передать еще один TPDU-модуль.

	A	Сообщение	B	Комментарии
1	→	< request 8 buffers>	→	*A* хочет 8 буферов
2	←	<ack = 15, buf = 4>	←	*B* позволяет переслать только сообщения 0–3
3	→	<seq = 0, data = m0>	→	У *A* теперь осталось 3 буфера
4	→	<seq = 1, data = m1>	→	У *A* теперь осталось 2 буфера
5	→	<seq = 2, data = m2>	•••	Сообщения потерялось, но *A* думает, что у него остался 1 буфер
6	←	<ack = 1, buf = 3>	←	*B* подтверждает получение модулей 0 и 1, разрешает передать со 2-го по 4-й
7	→	<seq = 3, data = m3>	→	У *A* остался буфер
8	→	<seq = 4, data = m4>	→	У *A* осталось 0 буферов, и он должен остановиться
9	→	<seq = 2, data = m2>	→	У *A* истекло время ожидания, и он передает еще раз
10	←	<ack = 4, buf = 0>	←	Все модули подтверждены, и он должен остановиться
11	←	<ack = 4, buf = 1>	←	Теперь *A* может послать модуль 5
12	←	<ack = 4, buf = 2>	←	*B* где-то нашел новый буфер
13	→	<seq = 5, data = m5>	→	У *A* остался 1 буфер
14	→	<seq = 6, data = m6>	→	*A* снова блокирован
15	←	<ack = 6, buf = 0>	←	*A* все еще блокирован
16	•••	<ack = 6, buf = 4>	←	Потенциальный тупик

Рис. 6.13. Динамическое выделение буферов. Стрелками показано направление передачи. Многоточие (...) означает потерянный TPDU-модуль

Потенциальные проблемы при такой схеме выделения буферов в дейтаграммных сетях могут возникнуть при потере управляющего TPDU-модуля. Взгляните на строку 16. Хост *B* выделил хосту *A* дополнительные буферы, но сообщение об этом было потеряно. Поскольку получение управляющих TPDU-модулей не подтверждается и, следовательно, управляющие TPDU-модули не посылаются повторно по тайм-ауту, хост *A* теперь оказался блокированным всерьез и надолго. Для предотвращения такой тупиковой ситуации каждый хост должен периодически посылать управляющий TPDU-модуль, содержащий подтверждение и состояние буферов для каждого соединения. Это позволит в конце концов выбраться из тупика.

До сих пор мы по умолчанию предполагали, что единственное ограничение, накладываемое на скорость передачи данных, состоит в количестве свободного буферного пространства у получателя. По мере все продолжающегося снижения цен на микросхемы памяти и винчестеры становится возможным оборудовать хосты таким количеством памяти, что проблема нехватки буферов будет возникать очень редко, если вообще будет возникать.

Если размер буферов перестанет ограничивать максимальный поток, возникнет другое узкое место: пропускная способность подсети. Если максимальная скорость обмена кадрами между соседними маршрутизаторами будет x кадров в секунду и между двумя хостами имеется k непересекающихся путей, то, сколько бы ни было буферов у обоих хостов, они не смогут пересылать друг другу больше чем kx TPDU-модулей в секунду. И если отправитель будет передавать с большей скоростью, то подсеть окажется перегружена.

Требуется механизм, основанный не столько на емкости буферов получателя, сколько на пропускной способности подсети. Очевидно, управление потоком должен проводить отправитель, в противном случае у него будет слишком много неподтвержденных TPDU-модулей. В 1975 году Белснес (Belsnes) предложил использовать схему управления потоком скользящего окна, в которой отправитель динамически приводит размер окна в соответствие с пропускной способностью сети. Если сеть может обработать c TPDU-модулей в секунду, а время цикла (включая передачу, распространение, ожидание в очередях, обработку получателем и возврат подтверждения) равно r, тогда размер окна отправителя должен быть равен cr. При таком размере окна отправитель работает, максимально используя канал. Любое уменьшение производительности сети приведет к его блокировке.

Для периодической настройки размера окна отправитель может отслеживать оба параметра и вычислять требуемое значение. Пропускная способность может быть определена простым подсчетом числа TPDU-модулей, получивших подтверждения за определенный период времени, и делением этого числа на тот же период времени. Во время измерения отправитель должен посылать данные с максимальной скоростью, чтобы удостовериться, что ограничивающим фактором является пропускная способность сети, а не низкая скорость передачи. Время, необходимое для получения подтверждения, может быть замерено аналогично. Так как пропускная способность сети зависит от количества трафика в ней, размер окна должен настраиваться довольно часто, чтобы можно было отслеживать

изменения пропускной способности. Как будет показано далее, в Интернете используется похожая схема.

Мультиплексирование

Объединение нескольких разговоров в одном соединении, виртуальном канале и по одной физической линии играет важную роль в нескольких уровнях сетевой архитектуры. Потребность в подобном уплотнении возникает в ряде случаев и на транспортном уровне. Например, если у хоста имеется только один сетевой адрес, он используется всеми соединениями транспортного уровня. Нужен какой-то способ, с помощью которого можно было бы различать, какому процессу следует передать входящий TPDU-модуль. Такая ситуация, называемая **восходящим мультиплексированием**, показана на рис. 6.14, *а*. На рисунке четыре различных соединения транспортного уровня используют одно сетевое соединение (например, один IP-адрес) с удаленным хостом.

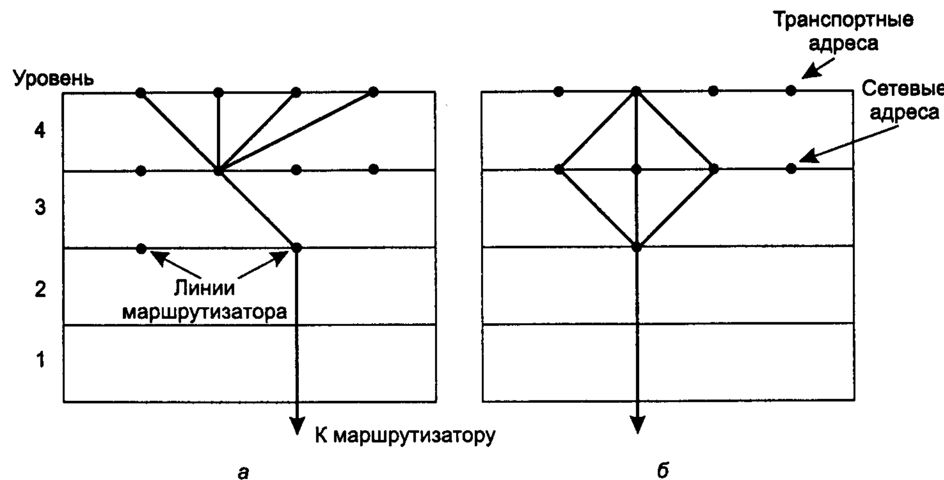


Рис. 6.14. Восходящее мультиплексирование (*а*); нисходящее мультиплексирование (*б*)

Уплотнение может играть важную роль на транспортном уровне и по другой причине. Предположим, например, что подсеть построена на основе виртуальных каналов и на каждом из них данные передаются с максимальной скоростью. Если пользователю требуется большая пропускная способность, нежели может предоставить один виртуальный канал, то можно попробовать решить эту проблему путем открытия нескольких сетевых соединений и распределения трафика между ними, используя виртуальные каналы поочередно, как показано на рис. 6.14, *б*. Такой метод называется **нисходящим мультиплексированием**. При k открытых сетевых соединениях эффективная пропускная способность увеличивается в k раз. В качестве примера можно привести нисходящее мультиплексирование, осуществляемое при работе частных пользователей, имеющих доступ к каналам ISDN. Такая линия обеспечивает установку двух отдельных соединений по 64 Кбит/с. Использование обоих соединений для доступа в Интернет и разделение трафика позволяют достигать эффективной пропускной способности 128 Кбит/с.

Восстановление после сбоев

Поскольку хосты и маршрутизаторы подвержены сбоям, следует рассмотреть вопрос восстановления после сбоев. Если транспортная сущность целиком помещается в хостах, восстановление после отказов сети и маршрутизаторов не вызывает затруднений. Если сетевой уровень поставляет дейтаграммные услуги, транспортные сущности постоянно ожидают потери TPDU-модулей и знают, как с этим бороться. Если сетевой уровень предоставляет услуги, ориентированные на соединение, тогда потеря виртуального канала обрабатывается при помощи установки нового виртуального канала с последующим запросом у удаленной транспортной сущности ее текущего состояния. При этом выясняется, какие TPDU-модули были получены, а какие нет. Потерянные TPDU-модули передаются повторно.

Более серьезную проблему представляет восстановление после сбоя хоста. В частности, для клиентов может быть желательной возможность продолжать работу после отказа и быстрой перезагрузки сервера. Чтобы пояснить, в чем тут сложность, предположим, что один хост — клиент — посылает длинный файл другому хосту — файловому серверу — при помощи простого протокола с ожиданием. Транспортный уровень сервера просто передает приходящие TPDU-модули один за другим пользователю транспортного уровня. Получив половину файла, сервер сбрасывается и перезагружается, после чего все его таблицы переинициализируются, поэтому он уже не помнит, что с ним было раньше.

Пытаясь восстановить предыдущее состояние, сервер может разослать широковещательный TPDU-модуль всем хостам, объявляя им, что он только что перезагрузился, и прося своих клиентов сообщить ему о состоянии всех открытых соединений. Каждый клиент может находиться в одном из двух состояний: один неподтвержденный TPDU-модуль (состояние *S1*) или ни одного неподтвержденного TPDU-модуля (состояние *S0*). Этой информации клиенту должно быть достаточно, чтобы решить, передавать ему повторно последний TPDU-модуль или нет.

На первый взгляд, здесь все очевидно: узнав о перезапуске сервера, клиент должен передать повторно последний неподтвержденный TPDU-модуль. То есть повторная передача требуется, если клиент находится в состоянии *S1*. Однако при более детальном рассмотрении оказывается, что все не так просто, как мы наивно предположили. Рассмотрим, например, ситуацию, в которой транспортная сущность сервера сначала посылает подтверждение, а уже затем передает пакет прикладному процессу. Запись TPDU-модуля в выходной поток и отправка подтверждения являются двумя различными неделимыми событиями, которые не могут быть выполнены одновременно. Если сбой произойдет после отправки подтверждения, но до того как выполнена запись, клиент получит подтверждение, а при получении объявления о перезапуске сервера окажется в состоянии *S0*.

Таким образом, клиент не станет передавать TPDU-модуль повторно, так как будет считать, что TPDU-модуль уже получен, что в конечном итоге приведет к отсутствию TPDU-модуля.

В этом месте вы, должно быть, подумаете: «А что если поменять местами последовательность действий, выполняемых транспортной сущностью сервера, чтобы сначала осуществлялась запись, а потом высылалось подтверждение?» Представим, что запись сделана, но сбой произошел до отправки подтверждения. В этом случае клиент окажется в состоянии *S1* и поэтому передаст TPDU-модуль повторно, и мы получим дубликат TPDU-модуля в выходном потоке.

Таким образом, независимо от того, как запрограммированы клиент и сервер, всегда могут быть ситуации, в которых протокол не сможет правильно восстановиться. Сервер можно запрограммировать двумя способами: так, чтобы он сначала передавал подтверждение, или так, чтобы сначала записывал TPDU-модуль. Клиент может быть запрограммирован одним из четырех способов: всегда передавать повторно последний TPDU-модуль, никогда не передавать повторно последний TPDU-модуль, передавать повторно TPDU-модуль только в состоянии *S0* и передавать повторно TPDU-модуль только в состоянии *S1*. Таким образом, получаем восемь комбинаций, но, как будет показано, для каждой комбинации имеется набор событий, заставляющий протокол ошибиться.

На сервере могут происходить три события: отправка подтверждения (*A*), запись TPDU-модуля в выходной процесс (*W*) и сбой (*C*). Они могут произойти в виде шести возможных последовательностей: *AC*(*W*), *AWC*, *C*(*AW*), *C*(*WA*), *WAC* и *WC*(*A*), где скобки означают, что после события *C* событие *A* или *B* может и не произойти (то есть уж сломался — так сломался). На рис. 6.15 показаны все восемь комбинаций стратегий сервера и клиента, каждая со своими последовательностями событий. Обратите внимание на то, что для каждой комбинации существует последовательность событий, приводящая к ошибке протокола. Например, если клиент всегда передает повторно неподтвержденный TPDU-модуль, событие *AWC* приведет к появлению неопознанного дубликата, хотя при двух других последовательностях событий протокол будет работать правильно.

Усложнение протокола не помогает. Даже если клиент и сервер обменяются несколькими TPDU-модулями, прежде чем сервер попытается записать полученный пакет, так что клиент будет точно знать, что происходит на сервере, у него нет возможности определить, когда произошел сбой на сервере: до или после записи. Отсюда следует неизбежный вывод: невозможно сделать отказ и восстановление хоста прозрачными для более высоких уровней.

В более общем виде это может быть сформулировано следующим образом: восстановление от сбоя уровня *N* может быть осуществлено только уровнем *N* + 1 и только при условии, что на более высоком уровне сохраняется достаточное количество информации о состоянии процесса. Как упоминалось ранее, транспортный уровень может обеспечить восстановление от сбоя на сетевом уровне, если каждая сторона соединения отслеживает свое текущее состояние.

Эта проблема подводит нас к вопросу о значении так называемого сквозного подтверждения. В принципе, транспортный протокол является сквозным, а не цепным, как более низкие уровни. Теперь рассмотрим случай обращения пользователя к удаленной базе данных. Предположим, что удаленная транспортная сущность запрограммирована сначала передавать TPDU-модуль вышестоящему уровню, а затем отправлять подтверждение. Даже в этом случае получение подтверждения машиной пользователя не означает, что удаленный хост успел обновить базу данных. Настоящее сквозное подтверждение, получение которого означает, что работа была сделана, и, соответственно, отсутствие которого означает обратное, вероятно, невозможно. Более подробно этот вопрос обсуждается в (Saltzer и др., 1984)

Стратегия, используемая получающим хостом

Стратегия, используемая передающим хостом	Сначала подтверждение, потом запись			Сначала запись, потом подтверждение		
	AC(*W*)	*AWC*	*C*(*AW*)	*C*(*WA*)	*WAC*	*WC*(*A*)
Всегда повторять передачу	OK	DUP	OK	OK	DUP	DUP
Никогда не повторять передачу	LOST	OK	LOST	LOST	OK	OK
Повторять передачу в *S0*	OK	DUP	LOST	LOST	DUP	OK
Повторять передачу в *S1*	LOST	OK	OK	OK	OK	DUP

OK = Протокол работает корректно
DUP = Протокол формирует дубликат сообщения
LOST = Протокол теряет сообщение

Рис. 6.15. Различные комбинации стратегий сервера и клиента

Простой транспортный протокол

Чтобы конкретизировать обсуждавшиеся ранее идеи, в данном разделе мы подробно изучим пример реализации транспортного уровня. В качестве абстрактных служебных примитивов будут использоваться ориентированные на соединение примитивы из табл. 6.1. Такие примитивы были выбраны, чтобы сделать пример похожим (с некоторыми упрощениями) на популярный протокол TCP.

Служебные примитивы примера транспортного протокола

Первая наша задача будет состоять в том, чтобы как можно более конкретно представить примитивы. С примитивом CONNECT (соединить) все довольно просто: у нас просто будет библиотечная процедура connect, которую можно вызывать с соответствующими параметрами для установки соединения. Параметрами этой процедуры являются локальный и удаленный TSAP-адреса. Программа, обра-

щающаяся к этой процедуре, блокируется (то есть приостанавливается) на время, пока транспортная сущность пытается установить соединение. Если установка соединения проходит успешно, программа разблокируется и может начинать передавать данные.

Когда процесс желает принимать входящие звонки, он обращается к процедуре listen (ожидать), указывая TSAP-адрес, соединение с которым ожидается. При этом процесс блокируется, пока какой-либо удаленный процесс не попытается установить соединение с его TSAP-адресом.

Обратите внимание: такая модель обладает сильной асимметрией. Пассивная сторона выполняет процедуру listen и ждет какого-либо события. Активная сторона инициирует соединение. Возникает интересный вопрос: что делать, если активная сторона начнет первой? Первая стратегия такова: при отсутствии ожидания на пассивной стороне попытка соединения считается неудачной. Другая стратегия заключается в блокировании инициатора (возможно, навсегда), пока на противоположном конце устанавливаемого соединения процесс не перейдет в режим ожидания.

Компромиссное решение, используемое в нашем примере, состоит в том, что на установку соединения процедуре connect отводится определенный интервал времени. Если процесс хоста, с которым пытаются установить связь, вызовет процедуру listen прежде, чем истечет интервал ожидания, соединение будет установлено. В противном случае звонящий получает отказ, разблокируется и получает сообщение об ошибке.

Для разрыва соединения мы будем применять процедуру disconnect. Соединение будет считаться разорванным, когда обе стороны вызовут эту процедуру. Другими словами, мы используем симметричную модель разъединения.

При передаче данных появляется та же проблема, что и при установлении соединения: передатчик активен, а получатель пассивен. Мы будем использовать при передаче данных то же решение, что и при установке соединения: активную процедуру send, передающую данные, и пассивную процедуру receive, блокирующую процесс до тех пор, пока не прибудет TPDU-модуль.

Таким образом, наша услуга определяется пятью примитивами: CONNECT, LISTEN, DISCONNECT, SEND и RECEIVE. Каждому примитиву соответствует библиотечная служба, выполняющая примитив. Параметры для служебных примитивов и библиотечных процедур следующие:

```
connum = LISTEN (local)
connum = CONNECT (local, remote)
status = SEND (connum, buffer, bytes)
status = RECEIVE (connum, buffer, bytes)
status = DISCONNECT (connum)
```

Примитив LISTEN объявляет о желании обращающейся к нему стороны принимать запросы соединения, обращенные к указанному TSAP-адресу. Пользователь примитива блокируется до тех пор, пока кто-либо не попытается с ним связаться. Понятия тайм-аута здесь нет.

Примитив CONNECT имеет на входе два параметра: локальный TSAP-адрес local и удаленный TSAP-адрес remote. Он пытается установить транспортное соединение между ними. Если это удается, он возвращает в качестве выходного параметра connum неотрицательное число, используемое для идентификации соединения при следующих вызовах процедур. Если же установить соединение не удалось, то причина неудачи помещается в connum в виде отрицательного числа. В нашей простой модели каждый TSAP-адрес может участвовать только в одном транспортном соединении, поэтому возможной причиной отказа может быть занятость одного из транспортных адресов. Среди других причин могут быть следующие: удаленный хост выключен, неверен локальный адрес или неверен удаленный адрес.

Примитив SEND передает содержимое буфера в виде сообщения по указанному транспортному соединению — может быть, в несколько приемов, если сообщение слишком велико. Возможные ошибки, возвращаемые в виде значения переменной status, таковы: нет соединения, неверный адрес буфера или отрицательное число байт.

Примитив RECEIVE означает готовность вызывающего его процесса принимать данные. Размер полученного сообщения помещается в переменную bytes. Если удаленный процесс разорвал соединение или адрес буфера указан неверно (например, за пределами программы пользователя), переменной status присваивается значение кода ошибки, указывающего на причину возникновения проблемы.

Примитив DISCONNECT разрывает транспортное соединение. Параметр connum сообщает номер соединения, которое следует разорвать. Могут возникать, например, такие ошибки: connum принадлежит другому процессу или connum является неверным идентификатором соединения. Переменной status присваивается 0 в случае успеха, в противном случае — код ошибки.

Транспортная сущность примера транспортного протокола

Прежде чем перейти к рассмотрению программы моделирования транспортной сущности, обратите внимание на то, что этот пример аналогичен примерам, приведенным в главе 3: они приводятся скорее в педагогических целях, нежели как серьезное предложение. Многие технические детали (как, например, исчерпывающая обработка ошибок), необходимые для действительно рабочей системы, ради простоты были здесь опущены.

Транспортный уровень использует примитивы сетевой службы для отправки и получения TPDU-модулей. Нам нужно будет выбрать примитивы сетевой службы, чтобы использовать их в этом примере. Одним из вариантов могла бы быть ненадежная дейтаграммная служба. Чтобы сохранить простоту примера, мы не стали останавливать свой выбор на этом варианте, так как в этом случае транспортная программа была бы большой и сложной и занималась бы в основном потерянными и опаздывающими пакетами. Кроме того, большая часть этих идей уже достаточно подробно обсуждалась в главе 3.

Вместо этого мы решили использовать ориентированную на соединение надежную сетевую службу. При этом мы сможем уделить максимум внимания

транспортным вопросам, не встречавшимся на более низких уровнях. Среди прочих, к ним относятся установка соединения, разрыв соединения и управление кредитованием. Подобным образом могла бы выглядеть простая транспортная служба, построенная на базе сети ATM.

В общем случае транспортная сущность может быть либо частью операционной системы, либо набором библиотечных процедур, работающих в адресном пространстве пользователя. В целях упрощения нашего примера мы будем полагать, что здесь используются библиотечные процедуры, однако с помощью небольших изменений можно добиться того, чтобы транспортная сущность стала частью операционной системы (эти изменения связаны в основном со способами доступа к буферам пользователя).

Следует отметить, тем не менее, что в этом примере «транспортная сущность» в действительности вообще не является отдельной сущностью, а представляет собой часть пользовательского процесса. В частности, когда пользователь выполняет некий блокирующий примитив, например LISTEN, вся транспортная сущность также блокируется. Такое решение является вполне удовлетворительным для однозадачного хоста, но на хосте с несколькими пользовательскими процессами более естественным было бы иметь транспортную сущность в виде отдельного процесса, отличного от всех пользовательских процессов.

Интерфейс с сетевым уровнем реализуется с помощью процедур to_net и from_net (не показаны). У каждой из них имеется по шесть параметров. Первый параметр означает идентификатор соединения, один в один соответствующий сетевому виртуальному каналу. Следом идут биты Q и M, которые, будучи установленными в 1, означают, соответственно, управляющее сообщение и то, что сообщение будет продолжено в следующем пакете. Следом за ними идет тип пакета, выбираемый из шести возможных, приведенных в табл. 6.3. Последние два параметра — это указатель на данные и целое число, указывающее на количество байтов данных.

Таблица 6.3. Пакеты сетевого уровня, используемые в примере

Сетевой пакет	Значение
CALL REQUEST	Запрос соединения. Посылается для установки соединения
CALL ACCEPTED	Вызов принят. Ответ на CALL REQUEST
CLEAR REQUEST	Запрос разъединения. Посылается для разрыва соединения
CLEAR CONFIRMATION	Подтверждение разъединения. Ответ на CLEAR REQUEST
DATA	Данные
CREDIT	Кредит. Служебный пакет для управления окном

При обращении к процедуре to_net транспортная сущность заполняет все значения параметров и передает их сетевому уровню. При вызове процедуры from_net сетевой уровень передает входящий пакет транспортной сущности. Благодаря передаче информации сетевому уровню в виде параметров процедуры вместо передачи самого входящего или исходящего пакета транспортная сущность оказывается защищенной от деталей протокола сетевого уровня. Если транспортная

сущность попытается послать пакет, когда в скользящем окне более низкого уровня нет свободных буферов, она приостановится в процедуре to_net до тех пор, пока в окне не появится место. Для транспортной сущности этот механизм прозрачен и управляется сетевым уровнем с помощью команд типа enable_transport_layer и disable_transport_layer, аналогичных описанным в протоколах главы 3. Управление окном также осуществляется сетевым уровнем.

Помимо этого прозрачного механизма приостановки есть также процедуры sleep и wakeup (не показаны), вызываемые транспортной сущностью. Процедура sleep вызывается, когда транспортная сущность логически блокирована ожиданием внешнего события, обычно прибытия пакета. После вызова процедуры sleep транспортная сущность (и пользовательский процесс, конечно) останавливаются.

Программа транспортной сущности показана в листинге 6.2. Каждое соединение может находиться в одном из следующих семи состояний:

1. IDLE — соединение еще не установлено.
2. WAITING — примитив CONNECT выполнен, пакет CALL REQUEST послан.
3. QUEUED — пакет CALL REQUEST прибыл. Примитив LISTEN еще не вызывался.
4. ESTABLISHED — соединение установлено.
5. SENDING — пользователь ожидает разрешения отправить пакет.
6. RECEIVING — примитив RECEIVE выполнен.
7. DISCONNECTING — примитив DISCONNECT выполнен локально.

Между состояниями могут происходить переходы при одном из следующих событий: выполняется примитив, прибывает пакет или истекает время ожидания.

Листинг 6.2. Пример транспортной сущности

```
#define MAX_CONN 32                    /* максимальное число одновременных
соединений */
#define MAX_MSG_SIZE 8192              /* максимальный размер сообщения в байтах */
#define MAX_PKT_SIZE 512               /* максимальный размер пакета в байтах */
#define TIMEOUT 20
#define CRED 1
#define OK 0

#define ERR_FULL -1
#define ERR_REJECT -2
#define ERR_CLOSED -3
#define LOW_ERR -3

typedef int transport_address;
typedef enum {CALL_REQ,CALL_ACC,CLEAR_REQ,CLEAR_CONF,DATA_PKT,CREDIT} pkt_type;
typedef enum {IDLE,WAITING,QUEUED,ESTABLISHED,SENDING,RECEIVING,DISCONN} cstate;
/* глобальные переменные */
transport_address listen_address;      /* локальный прослушиваемый адрес */
int listen_conn;                       /* идентификатор прослушиваемого соединения
*/
unsigned char data[MAX_PKT_SIZE];      /* временная область для пакетных данных */
```

```
struct conn {
  transport_address local_address, remote_address;
  cstate state;                       /* состояние этого соединения */
  unsigned char *user_buf_addr;       /* указатель на приемный буфер */
  int byte_count;                     /* счетчик передачи/приема */
  int clr_req_received;               /* устанавливается при получении пакета
CLEAR_REQ */
  int timer;                          /* используется для ожидания подтверждений
для пакетов CALL_REQ */
  int credits;                        /* число сообщений, которые разрешается
послать */
} conn[MAX_CONN+1];                   /* 0-й интервал не используется */

void sleep(void);                     /* прототипы */
void wakeup(void);
void to_net(int cid, int q, int m, pkt_type pt, unsigned char *p, int bytes);
void from_net(int *cid, int *q, int *m, pkt_type *pt, unsigned char *p, int *bytes);

int listen(transport_address t)
{/* Пользователь ждет соединения. Посмотреть, не прибыл ли уже пакет CALL_REQ. */
  int i = 1, found = 0;

  for (i = 1; i <= MAX_CONN; i++)     /* поиск CALL_REQ в таблице */
      if (conn[i].state == QUEUED && conn[i].local_address == t) {
          found = i;
          break;
      }

  if (found == 0) {
      /* Нет пакетов CALL_REQ, ждущих подтверждения. Можно спать, пока не прибудет
пакет или не сработает таймер. */
      listen_address = t; sleep(); i = listen_conn ;
  }
  conn[i].state = ESTABLISHED;        /* соединение установлено */
  conn[i].timer = 0;                  /* таймер не используется */
  listen_conn = 0;                    /* 0 считается неверным адресом */
  to_net(i, 0, 0, CALL_ACC, data, 0); /* велеть сетевому уровню принять сообщение
*/
  return(i);                          /* вернуть идентификатор соединения */
}

int connect(transport_address l, transport_address r)
{/* Пользователь желает установить соединение с удаленным процессом; послать пакет
CALL_REQ. */
  int i;
  struct conn *cptr;

  data[0] = r; data[1] = l;           /* это нужно для пакета CALL_REQ */
  i = MAX_CONN;                       /* поиск в таблице с конца */
  while (conn[i].state != IDLE && i > 1) i = i -1;
  if (conn[i].state == IDLE) {
      /* отметить в таблице, что CALL_REQ послан */
      cptr = &conn[i];
      cptr->local_address = l; cptr->remote_address = r;
```



```
      cptr->state = WAITING; cptr->clr_req_received = 0;
      cptr->credits = 0; cptr->timer = 0;
      to_net(i, 0, 0, CALL_REQ, data, 2);
      sleep();                        /* ждать CALL_ACC или CLEAR_REQ */
      if (cptr->state == ESTABLISHED) return(i);
      if (cptr->clr_req_received) {
          /* другая сторона отказалась от соединения */
          cptr->state = IDLE;         /* назад в состояние ожидания */
          to_net(i, 0, 0, CLEAR_CONF, data, 0);
          return(ERR_REJECT);
      }
  } else return(ERR_FULL);            /* отказаться от соединения: нет места в
таблице */
}

int send(int cid, unsigned char bufptr[], int bytes)
{/* Пользователь хочет послать сообщение. */
  int i, count, m;
  struct conn *cptr = &conn[cid];
  /* вход в состояние передачи */
  cptr->state = SENDING;
  cptr->byte_count = 0;               /* количество посланных байтов сообщения */
  if (cptr->clr_req_received == 0 && cptr->credits == 0) sleep();
  if (cptr->clr_req_received == 0) {
      /* кредит имеется; разбить сообщение на пакеты, если нужно */
      do {
          if (bytes - cptr->byte_count > MAX_PKT_SIZE) {/* многопакетное сообщение */
              count = MAX_PKT_SIZE; m = 1; /* остальные пакеты позже */
          } else { /* однопакетное сообщение */
              count = bytes .cptr->byte_count; m = 0;/* последний пакет сообщения */
          }
          for (i = 0; i < count; i++) data[i] = bufptr[cptr->byte_count + i];
          to_net(cid, 0, m, DATA_PKT, data, count);/* послать 1 пакет */
          cptr->byte_count = cptr->byte_count + count;/* увеличить число посланных
байтов */
      } while (cptr->byte_count < bytes); /* цикл, пока не будет послано все сообщение
*/
      cptr->credits--;                /* на каждое сообщение расходуется 1 кредит
*/
      cptr->state = ESTABLISHED;
      return(OK);
  } else {
      cptr->state = ESTABLISHED;
      return(ERR_CLOSED);             /* ошибка передачи: другая сторона хочет
разорвать соединение */
  }
}

int receive(int cid, unsigned char bufptr[], int *bytes)
{/* пользователь готов принять сообщение */
  struct conn *cptr = &conn[cid];

  if (cptr->clr_req_received == 0) {
      /* соединение установлено; попытка получения */
```

```
      cptr->state = RECEIVING;
      cptr->user_buf_addr = bufptr;
      cptr->byte_count = 0;
      data[0] = CRED;
      data[1] = 1;
      to_net(cid, 1, 0, CREDIT, data, 2); /* послать кредит */
      sleep(); /* ожидание данных */
      *bytes = cptr->byte_count;
  }
  cptr->state = ESTABLISHED;
  return(cptr->clr_req_received ? ERR_CLOSED : OK);
}

int disconnect(int cid)
{/* пользователь хочет разорвать соединение */
  struct conn *cptr = &conn[cid];

  if (cptr->clr_req_received) {          /* другая сторона инициировала окончание
связи */
      cptr->state = IDLE;                /* теперь соединение разорвано */
      to_net(cid, 0, 0, CLEAR_CONF, data, 0);
  } else {                               /* мы сами инициировали окончание связи */
      cptr->state = DISCONN;             /* соединение не разорвано, пока другая
сторона не согласится */
      to_net(cid, 0, 0, CLEAR_REQ, data, 0);
  }
  return(OK);
}

void packet_arrival(void)
{/* Прибыл пакет. Получить и обработать его. */
  int cid;                               /* соединение, по которому прибыл пакет */
  int count, i, q, m;
  pkt_type ptype;   /* CALL_REQ, CALL_ACC, CLEAR_REQ, CLEAR_CONF, DATA_PKT, CREDIT */
  unsigned char data[MAX_PKT_SIZE];      /* часть данных из пришедшего пакета */
  struct conn *cptr;

  from_net(&cid, &q, &m, &ptype, data, &count); /* получить пакет */
  cptr = &conn[cid];
  switch (ptype) {
      case CALL_REQ: /* удаленный пользователь хочет установить соединение */
          cptr->local_address = data[0]; cptr->remote_address = data[1];
          if (cptr->local_address == listen_address) {
              listen_conn = cid; cptr->state = ESTABLISHED; wakeup();
          } else {
              cptr->state = QUEUED; cptr->timer = TIMEOUT;
          }
          cptr->clr_req_received = 0; cptr->credits = 0;
          break;

      case CALL_ACC: /* удаленный пользователь принял наш CALL_REQ */
          cptr->state = ESTABLISHED;
          wakeup();
          break;
```



```
      case CLEAR_REQ: /* удаленный пользователь хочет разорвать соединение или
отвергнуть вызов */
          cptr->clr_req_received = 1;
          if (cptr->state == DISCONN) cptr->state = IDLE;/* clear collision */
          if (cptr->state == WAITING || cptr->state == RECEIVING || cptr->state ==
SENDING) wakeup();
          break;

      case CLEAR_CONF: /* удаленный пользователь согласен разорвать соединение */
          cptr->state = IDLE;
          break;

      case CREDIT: /* удаленный пользователь ожидает данные */
          cptr->credits += data[1];
          if (cptr->state == SENDING) wakeup();
          break;

      case DATA_PKT: /* удаленный пользователь послал данные */
          for (i = 0; i < count; i++) cptr->user_buf_addr[cptr->byte_count + i] =
data[i];
          cptr->byte_count += count;
          if (m == 0 ) wakeup();
  }
}

void clock(void)
{/* часы тикнули, проверить на тайм-ауты стоящие в очереди запросы на соединение */
  int i;
  struct conn *cptr;
  for (i = 1; i <= MAX_CONN; i++) {
      cptr = &conn[i];
      if (cptr->timer > 0) { /* таймер запущен */
          cptr->timer--;
          if (cptr->timer == 0) { /* теперь время истекло */
              cptr->state = IDLE;
              to_net(i, 0, 0, CLEAR_REQ, data, 0);
          }
      }
  }
}
```

В листинге 6.2 приведены процедуры двух типов. Большинство из них вызываются напрямую пользовательскими программами. Однако процедуры `packet_arrival` и `clock` отличаются от остальных. Они вызываются внешними событиями — прибытием пакета и срабатыванием таймера соответственно. Таким образом, они являются процедурами обработки прерываний. Мы будем предполагать, что они никогда не вызываются во время работы процедуры транспортной сущности, а вызываются только тогда, когда пользовательский процесс находится в режиме ожидания или управление находится за пределами транспортной сущности. Это их свойство является существенным для корректной работы транспортной сущности.

Наличие бита Q (Qualifier — спецификатор) в заголовке пакета позволяет избежать накладных расходов в заголовке транспортного уровня. Обычные информационные сообщения посылаются в виде пакетов данных с $Q = 0$. Управляющие сообщения транспортного протокола посылаются как информационные пакеты с $Q = 1$. В нашем примере такое сообщение только одно — CREDIT. Эти управляющие сообщения обнаруживаются и обрабатываются принимающей транспортной сущностью.

Основной структурой данных, используемой транспортной сущностью, является массив *conn*. Каждый элемент этого массива предназначается для одного потенциального соединения и содержит информацию о состоянии соединения, включая транспортные адреса обоих его концов, число посланных и полученных сообщений, текущее состояние, указатель на буфер пользователя, количество уже посланных и полученных байтов, бит, указывающий, что от удаленного пользователя получен запрос на разъединение, таймер и счетчик разрешений на передачу сообщений. Не все эти поля используются в нашем простом примере, но для полной реализации транспортной сущности потребовались бы все эти значения и, возможно, даже некоторые дополнительные. Предполагается, что изначально поле состояния соединения всех элементов массива *conn* инициализируется значением *IDLE*.

Когда пользователь обращается к примитиву CONNECT, сетевой уровень получает указание послать удаленной машине пакет CALL REQUEST, а пользователь переводится в режим ожидания. Когда этот пакет прибывает по указанному адресу, транспортная сущность удаленной машины прерывается на выполнение процедуры packet_arrival, проверяющей, ожидает ли локальный пользователь соединения с указанным адресом. Если да, то обратно отправляется пакет CALL ACCEPTED, а удаленный пользователь переводится в активное состояние. В противном случае запрос соединения ставится в очередь на период времени *TIMEOUT*. Если в течение этого интервала времени пользователь вызывает примитив LISTEN, соединение устанавливается, в противном случае время ожидания истекает, а просящий соединения получает отказ в виде пакета CLEAR REQUEST. Этот механизм необходим, чтобы инициатор соединения не оказался заблокированным навсегда, если удаленный процесс не желает устанавливать с ним соединение.

Хотя мы удалили заголовок транспортного протокола, нам, тем не менее, нужен метод, при помощи которого можно было бы отслеживать принадлежность пакетов тому или иному транспортному соединению, так как несколько соединений могут существовать одновременно. Проще всего в качестве номера соединения использовать номер виртуального канала сетевого уровня. Более того, номер виртуального канала может использоваться как индекс массива *conn*. Когда пакет приходит по виртуальному каналу k, он принадлежит транспортному соединению k, состояние которого хранится в $conn[k]$. Для соединений, инициированных на данном хосте, номер соединения выбирается инициирующей соединение транспортной сущностью.

Чтобы избежать необходимости предоставления буферов и управления ими в транспортной сущности, здесь используется механизм управления потоком, отличный от традиционного скользящего окна. Суть его в следующем: когда пользователь вызывает примитив RECEIVE, транспортной сущности посылающей машины отправляется специальное **кредитное сообщение**, содержащее разрешение на передачу определенного количества пакетов данных. Это число сохраняется в массиве *conn*. Когда вызывается примитив SEND, транспортная сущность проверяет, получен ли кредит указанным соединением. Если кредит не нулевой, сообщение посылается (при необходимости в нескольких пакетах), а значение кредита уменьшается, в противном случае транспортная сущность переходит в режим ожидания кредитов. Такой механизм гарантирует, что ни одно сообщение не будет послано, если другая сторона не вызвала примитив RECEIVE. В результате, когда сообщение прибывает, для него гарантированно имеется свободный буфер. Эту схему несложно усовершенствовать, позволив получателям предоставлять сразу несколько буферов и запрашивать несколько сообщений.

Необходимо помнить, что программа, приведенная в листинге 6.2, является сильно упрощенной. Настоящая транспортная сущность должна проверять правильность всех предоставляемых пользователем параметров, обеспечивать восстановление от сбоев сетевого уровня, обрабатывать столкновение вызовов и поддерживать более общие транспортные услуги, включающие такие возможности, как прерывания, дейтаграммы и неблокирующие версии примитивов SEND и RECEIVE.

Пример протокола как конечного автомата

Написание транспортной сущности является сложной и кропотливой работой, особенно для протоколов, применяющихся в действительности. Чтобы снизить вероятность ошибки, полезно представлять состояния протокола в виде конечного автомата.

Как мы уже видели, у соединений нашего протокола есть семь состояний. Можно выделить 12 событий, переводящих соединение из одного состояния в другое. Пять из этих событий являются служебными примитивами. Еще шесть соответствуют получению шести типов пакетов. Последнее событие — истечение времени ожидания. На рис. 6.16 в виде матрицы показаны основные действия протокола. Столбцы матрицы представляют собой состояния, а строки — 12 событий.

Каждая ячейка матрицы на рисунке (то есть модели конечного автомата) содержит до трех полей: предикат, действие и новое состояние. Предикат указывает, при каких условиях производилось действие. Например, в левом верхнем углу матрицы, если выполняется примитив LISTEN и нет свободного места в таблице (предикат *P1*), выполнение примитива LISTEN завершается неудачно, и состояние не изменяется. С другой стороны, если пакет CALL REQUEST для ожидаемого транспортного адреса уже прибыл (предикат *P2*), соединение устанавливается незамедлительно. Другая возможность состоит в том, что утверждение *P2* ложно, то есть пакет CALL REQUEST не прибыл. В этом случае соединение остается в состоянии *IDLE*, ожидая пакета CALL REQUEST.

		Состояние						
		Простой	Ожидание	В очереди	Установлено	Отправление	Получение	Разъединение
Примитивы	LISTEN	P1: ~/Простой P2: A1/Установлено $\overline{P2}$: A2/Простой		~/Установлено				
	CONNECT	P1: ~/Простой $\overline{P1}$: A3/Ожидание						
	DISCONNECT				P4: A5/Простой $\overline{P4}$: A6/Разъединение			
	SEND				P5: A7/Установлено $\overline{P5}$: A8/Отправление			
	RECEIVE				A9/Получение			
Входящие пакеты	Запрос соединения	P3: A1/Установлено $\overline{P3}$: A4/В очереди						
	Запрос принят		~/Установлено					
	Запрос разъединения		~/Простой		A10/Установлено	A10/Установлено	A10/Установлено	~/Простой
	Подтверждение разъединения							~/Простой
	Пакет данных						A12/Установлено	
	Кредит				A11/Установлено	A7/Установлено		
Часы	Тайм-аут			~/Простой				

Предикаты (утверждения)

P1: Таблица соединений полная
P2: Выполняется запрос соединения
P3: Выполняется примитив LISTEN
P4: Выполняется запрос разъединения
P5: Кредит есть

Действия

A1: Послать подтверждение соединения
A2: Ждать запрос соединения
A3: Послать подтверждение соединения
A4: Запустить таймер
A5: Послать подтверждения разъединения
A6: Послать запрос разъединения
A7: Послать сообщение
A8: Ждать кредита
A9: Послать кредит
A10: Установить флаг «Запрос разъединения получен»
A11: Записать кредит
A12: Принять сообщение

Рис. 6.16. Пример протокола как конечного автомата. У каждой ячейки матрицы может быть предикат, действие и новое состояние. Тильда означает, что основное действие не предпринималось. Черта над предикатом означает отрицание предиката. Пустые ячейки соответствуют невозможным или неверным событиям

Следует отметить, что выбор используемых состояний обусловлен не только самим протоколом. В нашем примере нет состояния *LISTENING*, которое вполне могло бы следовать после вызова примитива LISTEN. Состояния *LISTENING* нет потому, что оно связано с полем записи соединения, а примитив LISTEN не создает

записи соединения. Почему же? Потому что мы решили использовать в качестве идентификаторов соединений номера виртуальных каналов сетевого уровня, а для примитива LISTEN номер виртуального канала в конечном счете выбирается сетевым уровнем, когда прибывает пакет CALL REQUEST.

Действия с *A1* по *A12* представляют собой значительные действия, такие как отправление пакетов и запуск таймера. В таблице не перечислены менее значимые события, как, например, инициализация полей записи соединения. Если действие включает активизацию спящего процесса, то действие, следующее за активизацией, также учитывается. Например, если прибывает пакет CALL REQUEST и процесс находился в состоянии ожидания этого пакета, то передача пакета CALL ACCEPT, следующая за активизацией, рассматривается как часть реакции на прибытие пакета CALL REQUEST. После выполнения каждого действия соединение может переместиться в новое состояние, как показано на рис. 6.16.

Представление протокола в виде матрицы состояний обладает тремя преимуществами. Во-первых, такая форма значительно упрощает программисту проверку всех комбинаций состояний и событий при принятии решения о том, не требуется ли какое-либо действие со стороны протокола. В рабочих версиях протокола некоторые комбинации использовались бы для обработки ошибок. В матрице, показанной на рисунке, не проведено различия между невозможными и запрещенными состояниями. Например, если соединение находится в состоянии ожидания (waiting), то событие DISCONNECT является невозможным, так как пользователь блокирован и не может выполнять никакие примитивы. С другой стороны, в состоянии передачи (sending) получение пакета данных не ожидается, так как кредит на них не отпускался. Прибытие пакета данных в этом состоянии является ошибкой протокола.

Второе преимущество представления протокола в виде матрицы состоит в его реализации. Можно представить себе двумерный массив, в котором элемент $a[i][j]$ является указателем или индексом процедуры обработки события i в состоянии j. При этом можно написать транспортную сущность в виде короткого цикла ожидания события. Когда событие происходит, берется соответствующее соединение и извлекается его состояние. При известных значениях события и состояния транспортная сущность просто обращается к массиву a и вызывает соответствующую процедуру обработки. Результатом такого подхода будет более регулярное и систематическое строение программы, нежели в нашем примере транспортной сущности.

Третье преимущество использования конечного автомата заключается в способе описания протокола. В некоторых стандартных документах протоколы описываются как конечные автоматы того же вида, что и на рис. 6.17. Создать работающую транспортную сущность по такому описанию значительно легче, чем по многочисленным разрозненным документам.

Основной недостаток подхода на основе конечного автомата состоит в том, что он может оказаться более сложным для понимания, чем приведенный ранее в листинге пример программы. Однако частично эту проблему можно решить, изобразив конечный автомат в виде графа, как это сделано на рис. 6.17.

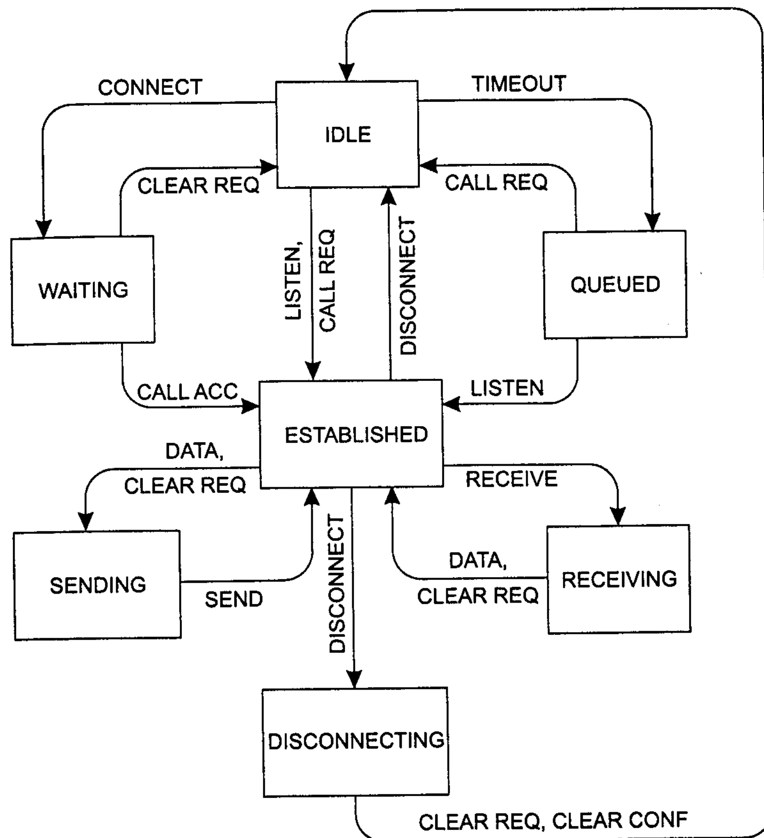


Рис. 6.17. Пример протокола в виде графа. Для простоты переходы, не изменяющие состояние соединения, были опущены

Транспортные протоколы Интернета: UDP

В Интернете нашли применение два основных протокола транспортного уровня, один из которых ориентирован на соединение, другой — нет. В следующих разделах мы изучим их. Протоколом без установления соединения является UDP. Протокол TCP, напротив, ориентирован на соединение. Так как UDP — это, на самом деле, просто IP с добавлением небольшого заголовка, мы изучим сперва его. Рассмотрим также два практических применения UDP.

Основы UDP

Среди набора протоколов Интернета есть транспортный протокол без установления соединения, **UDP** (User Datagram Protocol — пользовательский дейтаграммный протокол). UDP позволяет приложениям отправлять инкапсулированные IP-дейтаграммы без установления соединений. UDP описан в RFC 768.

С помощью протокола UDP передаются сегменты, состоящие из 8-байтного заголовка, за которым следует поле полезной нагрузки. Заголовок показан на рис. 6.18. Два номера портов служат для идентификации конечных точек внутри отправляющей и принимающей машин. Когда прибывает пакет UDP, содержимое его поля полезной нагрузки передается процессу, связанному с портом назначения. Это связывание происходит при выполнении примитива типа BIND. Это было продемонстрировано в листинге 6.1 применительно к TCP (в UDP процесс связывания происходит точно так же). В сущности, весь смысл использования UDP вместо обычного IP заключается как раз в указании портов источника и приемника. Без этих двух полей на транспортном уровне невозможно было бы определить действие, которое следует произвести с пакетом. В соответствии с полями портов производится корректная доставка сегментов.

Рис. 6.18. Заголовок UDP

Информация о порте источника требуется прежде всего при создании ответа, пересылаемого отправителю. Копируя значения поля *Порт источника* из входящего сегмента в поле *Порт назначения* исходящего сегмента, процесс, посылающий ответ, может указать, какому именно процессу на противоположной стороне он предназначается.

Поле *Длина UDP* содержит информацию о длине сегмента, включая заголовок и полезную нагрузку. *Контрольная сумма UDP* не является обязательной. Если она не подсчитывается, ее значение равно 0 (настоящая нулевая контрольная сумма кодируется всеми единицами).

Отключать функцию подсчета контрольной суммы глупо, за исключением одного случая — когда нужна высокая производительность (например, при передаче оцифрованной речи).

Наверное, стоит прямо сказать о том, чего UDP *не* делает. Итак, UDP не занимается контролем потока, контролем ошибок, повторной передачей после приема испорченного сегмента. Все это перекладывается на пользовательские процессы. Что же он делает? UDP предоставляет интерфейс для IP путем демультиплексирования нескольких процессов, использующих порты. Это все, что он делает. Для процессов, которым хочется управлять потоком, контролировать ошибки и временные интервалы, протокол UDP — это как раз то, что доктор прописал.

Одной из областей, где UDP применяется особенно широко, является область клиент-серверных приложений. Зачастую клиент посылает короткий запрос серверу и надеется получить короткий ответ. Если запрос или ответ теряется, клиент по прошествии определенного временного интервала может попытаться еще раз. Это позволяет не только упростить код, но и уменьшить требуемое количе-

ство собщений по сравнению с протоколами, которым требуется начальная настройка.

DNS (Domain Name System — служба имен доменов) — это приложение, которое использует UDP именно так, как описано ранее. Мы изучим его в главе 7. В двух словах, если программе нужно найти IP-адрес по имени хоста, например, www.cs.berkeley.edu, она может послать UDP-пакет с этим именем на сервер DNS. Сервер в ответ на запрос посылает UDP-пакет с IP-адресом хоста. Никакой предварительной настройки не требуется, как не требуется и разрыва соединения после завершения задачи. По сети просто передаются два сообщения.

Вызов удаленной процедуры

В определенном смысле процессы отправки сообщения на удаленный хост и получения ответа очень похожи на вызов функции в языке программирования. В обоих случаях необходимо передать один или несколько параметров, и вы получаете результат. Это соображение навело разработчиков на мысль о том, что можно попробовать организовать запросно-ответное взаимодействие по сети, выполняемое в форме вызовов процедур. Такое решение позволяет упростить и сделать более привычной разработку сетевых приложений. Например, представьте себе процедуру с именем `get_IP_address(имя_хоста)`, работающую посредством отправки UDP-пакетов на сервер DNS, ожидания ответа и отправки повторного запроса в случае наступления тайм-аута (если одна из сторон работает недостаточно быстро). Таким образом, все детали, связанные с особенностями сетевых технологий, скрыты от программиста.

Ключевая работа в этой области написана в 1984 году Бирреллом (Birrell) и Нельсоном (Nelson). По сути дела, было предложено разрешить программам вызывать процедуры, расположенные на удаленных хостах. Когда процесс на машине 1 вызывает процедуру, находящуюся на машине 2, вызывающий процесс машины 1 блокируется и выполняется вызванная процедура на машине 2. Информация от вызывающего процесса может передаваться в виде параметров и приходить обратно в виде результата процедуры. Передача сообщений по сети скрыта от программиста. Такая технология известна под названием **RPC** (Remote Procedure Call — удаленный вызов процедуры) и стала основой многих сетевых приложений. Традиционно вызывающая процедура считается клиентом, а вызываемая — сервером. Мы и здесь будем называть их так же.

Идея RPC состоит в том, чтобы сделать вызов удаленной процедуры максимально похожим на локальный вызов. В простейшем случае для вызова удаленной процедуры клиентская программа должна быть связана с маленькой библиотечной процедурой, называемой **клиентской заглушкой**, которая отображает серверную процедуру в пространство адресов клиента. Аналогично сервер должен быть связан с процедурой, называемой **серверной заглушкой**. Эти процедуры скрывают тот факт, что вызов клиентом серверной процедуры осуществляется не локально.

Реальные шаги, выполняемые при удаленном вызове процедуры, показаны на рис. 6.19. Шаг 1 заключается в вызове клиентом клиентской заглушки. Это локальный вызов процедуры, параметры которой самым обычным образом помещаются в стек. Шаг 2 состоит в упаковке параметров клиентской заглушки в сообщение и в осуществлении системного вызова для отправки этого сообщения. Упаковка параметров называется **маршалингом**. На шаге 3 ядро системы передает сообщение с клиентской машины на сервер. Шаг 4 заключается в том, что ядро передает входящий пакет серверной заглушке. Последняя на пятом шаге вызывает серверную процедуру с демаршализованными параметрами. При ответе выполняются те же самые шаги, но передача происходит в обратном направлении.

Важнее всего здесь то, что клиентская процедура, написанная пользователем, выполняет обычный (то есть локальный) вызов клиентской заглушки, имеющей то же имя, что и серверная процедура. Поскольку клиентская процедура и клиентская заглушка существуют в одном и том же адресном пространстве, параметры передаются обычным образом. Аналогично серверная процедура вызывается процедурой, находящейся в том же адресном пространстве, с ожидаемыми параметрами. С точки зрения серверной процедуры, не происходит ничего необычного. Таким образом, вместо ввода-вывода с помощью сокетов сетевая коммуникация осуществляется обычным вызовом процедуры.

Несмотря на элегантность концепции RPC, в ней есть определенные подводные камни. Речь идет прежде всего об использовании указателей в качестве параметров. В обычной ситуации передача указателя процедуре не представляет никаких сложностей. Вызываемая процедура может использовать указатель так же, как и вызывающая, поскольку они обе существуют в одном и том же виртуальном адресном пространстве. При удаленном вызове процедуры передача указателей невозможна, потому что адресные пространства клиента и сервера различаются.

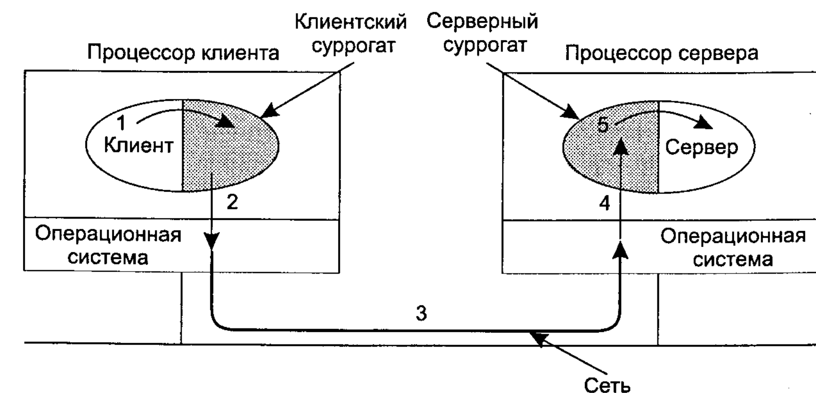


Рис. 6.19. Этапы выполнения удаленного вызова процедуры. Заглушки затенены

Иногда с помощью некоторых уловок все же удается передавать указатели. Допустим, первым параметром является указатель на целое число k. Клиентская заглушка может выполнить маршализацию k и передать его серверу. Серверная

заглушка создаст указатель на полученную переменную *k* и передаст его серверной процедуре. Именно этого та и ожидала. Когда серверная процедура возвращает управление серверной заглушке, последняя отправляет *k* обратно клиенту, где обновленное значение этой переменной записывается вместо старого (если оно было изменено сервером). В принципе, стандартная последовательность действий, выполняемая при вызове по ссылке, заменилась последовательностью копирования и восстановления. Увы, этот трюк не всегда удается применить — в частности, нельзя это сделать, если указатель ссылается на граф или иную сложную структуру данных. По этой причине на параметры удаленно вызываемых процедур должны быть наложены определенные ограничения.

Вторая проблема заключается в том, что в языках со слабой типизацией данных (например, в C) можно совершенно законно написать процедуру, которая подсчитывает скалярное произведение двух векторов (массивов), не указывая их размеры. Каждая из этих структур в качестве ограничителя имеет какое-то значение, известное только вызывающей и вызываемой процедурам. При этих обстоятельствах клиентская заглушка не способна маршализовать параметры: нет никакой возможности определить их размеры.

Третья проблема заключается в том, что не всегда можно распознать типы параметров по спецификации или по самому коду. В качестве примера можно привести процедуру `printf`, у которой может быть любое число параметров (не меньше одного), и они могут представлять собой смесь целочисленных, коротких вещественных, длинных вещественных, символьных, строковых, вещественных с плавающей запятой различной длины и других типов. Задача удаленного вызова процедуры `printf` может оказаться практически невыполнимой из-за такой своеобразной толерантности языка C. Тем не менее, нет правила, говорящего, что удаленный вызов процедур возможен только в том случае, если используется не C (C++), — это подорвало бы репутацию метода RPC.

Четвертая проблема связана с применением глобальных переменных. В нормальной ситуации вызывающая и вызываемая процедуры могут общаться друг с другом посредством глобальных переменных (кроме общения с помощью параметров). Если вызываемая процедура переедет на удаленную машину, программа, использующая глобальные переменные, не сможет работать, потому что глобальные переменные больше не смогут служить в качестве разделяемого ресурса.

Эти проблемы не означают, что метод удаленного вызова процедур безнадежен. На самом деле, он широко используется, просто нужны некоторые ограничения для его нормальной практической работы.

Конечно, RPC не нуждается в UDP-пакетах, однако они хорошо подходят друг для друга, и UDP обычно используется совместно с RPC. Тем не менее, когда параметры или результаты оказываются больше максимальных размеров UDP-пакетов или запрашиваемая операция не идемпотентна (то есть не может повторяться без риска сбоя — например, операция инкрементирования счетчика), может понадобиться установка TCP-соединения и отправки запроса по TCP, а не по UDP.

Транспортный протокол реального масштаба времени

Клиент-серверный удаленный вызов процедур — это та область, в которой UDP применяется очень широко. Еще одной такой областью являются мультимедийные приложения реального времени. В частности, с ростом популярности интернет-радио, интернет-телефонии, музыки и видео по заказу, видеоконференций и других мультимедийных приложений стало понятно, что все они пытаются заново изобрести велосипед, используя более или менее одинаковые транспортные протоколы реального времени. Вскоре пришли к мысли о том, что было бы здорово иметь один общий транспортный протокол для мультимедийных приложений. Так появился на свет протокол **RTP** (Real-Time Transport Protocol — транспортный протокол реального масштаба времени). Он описан в RFC 1889 и ныне широко используется.

RTP занимает довольно странное положение в стеке протоколов. Было решено, что RTP будет принадлежать пользовательскому пространству и работать (в обычной ситуации) поверх UDP. Делается это так. Мультимедийное приложение может состоять из нескольких аудио-, видео-, текстовых и некоторых других потоков. Они прописываются в библиотеке RTP, которая, как и само приложение, находится в пользовательском пространстве. Библиотека уплотняет потоки и помещает их в пакеты RTP, которые, в свою очередь, отправляются в сокет. На другом конце сокета (в ядре операционной системы) генерируются UDP-пакеты, которые внедряются в IP-пакеты. Теперь остается передать IP-пакеты по сети. Если компьютер подключен к локальной сети Ethernet, IP-пакеты для передачи разбиваются на кадры Ethernet. Стек протоколов для описанной ситуации показан на рис. 6.20, *а*. Система вложенных пакетов показана на рис. 6.20, *б*.

В результате оказывается непросто определить, к какому уровню относится RTP. Так как протокол работает в пользовательском пространстве и связан с прикладной программой, он, несомненно, выглядит, как прикладной протокол. С другой стороны, он является общим, независимым от приложения протоколом, который просто предоставляет транспортные услуги, не более. С этой точки зрения он может показаться транспортным протоколом. Наверное, лучше всего будет определить его таким образом: RTP — это транспортный протокол, реализованный на прикладном уровне.

Основной функцией RTP является уплотнение нескольких потоков реального масштаба времени в единый поток пакетов UDP. Поток UDP можно направлять либо по одному, либо сразу по нескольким адресам. Поскольку RTP использует обычный UDP, его пакеты не обрабатываются маршрутизаторами каким-либо особым образом, если только не включены свойства доставки с качеством обслуживания уровня IP. В частности, нет никаких гарантий касательно доставки, дрожания (неустойчивой синхронизации) и т. д.

Каждый пакет, посылаемый с потоком RTP, имеет номер, на единицу превышающий номер своего предшественника. Такой способ нумерации позволяет получателю определить пропажу пакетов. Если обнаруживается исчезновение какого-либо пакета, то лучшее, что может сделать получатель, — это путем ин-

терполяции аппроксимировать пропущенное значение. Повторная передача в данном случае не является хорошим решением, поскольку это займет много времени, и повторно переданный пакет окажется уже никому не нужным. Поэтому протокол RTP не осуществляет управление потоком, контроль ошибок, и в стандарте не предусмотрены никакие подтверждения и механизмы запроса повторной передачи.

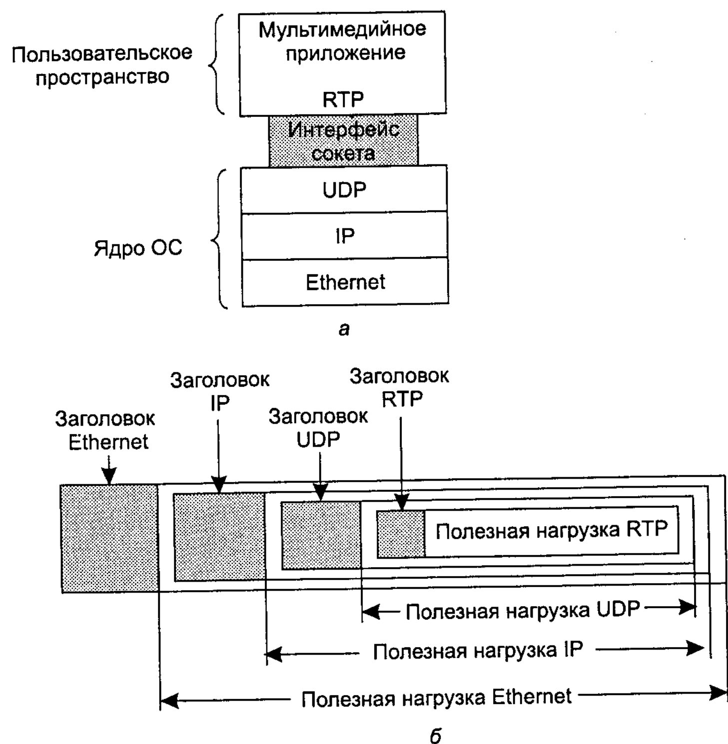


Рис. 6.20. Положение RTP в стеке протоколов (*а*); вложение пакетов (*б*)

В поле полезной нагрузки RTP может содержаться несколько символов данных, которые могут иметь формат, соответствующий отправившему их приложению. Межсетевое взаимодействие обеспечивается в протоколе RTP за счет определения нескольких профилей (например, отдельных аудиопотоков), каждому из которых может сопоставляться несколько форматов кодирования. Скажем, аудиопоток может кодироваться при помощи PCM (8-битными символами с полосой 8 кГц), дельта-кодирования, кодирования с предсказанием, GSM, MP3 и т. д. В RTP имеется специальное поле заголовка, в котором источник может указать метод кодирования, однако далее источник никак не влияет на процесс кодирования.

Еще одна функция, необходимая приложениям реального времени, — это отметки времени. Идея состоит в том, чтобы позволить источнику связать отметку времени с первым символом каждого пакета. Отметки времени ставятся относительно момента начала передачи потока, поэтому важны только интервалы *между* отметками. Абсолютные значения, по сути дела, никакой роли не играют. Такой механизм позволяет приемнику буферизировать небольшое количество данных и проигрывать каждый отрезок спустя правильное число миллисекунд после начала потока независимо от того, когда на самом деле приходит пакет, содержащий данный отрезок. За счет этого не только снижается эффект джиттера (флуктуаций, неустойчивости синхронизации), но и появляется возможность синхронизации между собой нескольких потоков. Например, в цифровом телевидении может быть видеопоток и два аудиопотока. Второй аудиопоток обычно нужен либо для обеспечения стереозвучания, либо для дублированной на иностранный язык звуковой дорожки фильма. У каждого потока свой физический источник, однако с помощью временных отметок, генерируемых единым таймером, эти потоки при воспроизведении можно синхронизовать даже в том случае, если они приходят не совсем одновременно.

Заголовок RTP показан на рис. 6.21. Он состоит из трех 32-разрядных слов и некоторых возможных расширений. Первое слово содержит поле *Версия*, которое в настоящий момент уже имеет значение 2. Будем надеяться, что текущая версия окажется окончательной или хотя бы предпоследней, поскольку в идентифицирующем ее двухбитном поле осталось место только для одного нового номера (впрочем, код 3 может обозначать, что настоящий номер версии содержится в поле расширения).

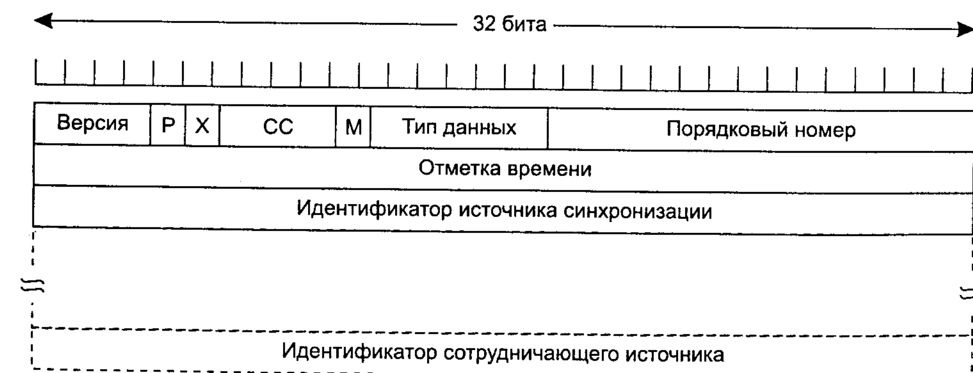


Рис. 6.21. Заголовок RTP

Бит *P* указывает на то, что размер пакета сделан кратным 4 байтам за счет байтов заполнения. При этом в последнем байте заполнения содержится общее число байтов заполнения. Бит *X* говорит о том, что присутствует расширенный заголовок. Формат и назначение расширенного заголовка не определяются. Обязательным для него является только то, что первое слово расширения должно содержать общую длину расширения. Это запасная возможность для разнообразных непредсказуемых будущих требований.

Поле *CC* говорит о том, сколько сотрудничающих источников формируют поток. Их число может колебаться от 0 до 15 (см. далее). Бит *M* — это маркер, свя-

занный с конкретным приложением. Он может использоваться для обозначения начала видеокадра, начала слова в аудиоканале или еще для чего-нибудь, важного и понятного для приложения. Поле *Тип данных* содержит информацию об используемом алгоритме кодирования (например, несжатое 8-битное аудио, MP3 и т. д.). Поскольку такое поле есть в каждом пакете, метод кодирования может изменяться прямо во время передачи потока. *Порядковый номер* — это просто счетчик, который инкрементируется в каждом пакете RTP. Он используется для определения потерявшихся пакетов.

Отметка времени генерируется источником потока и служит для записи момента создания первого слова пакета. Отметки времени помогают снизить эффект джиттера на приемнике за счет того, что момент воспроизведения делается независимым от времени прибытия пакета. *Идентификатор источника синхронизации* позволяет определить, какому потоку принадлежит пакет. Применяется метод уплотнения и распределения потоков данных, следующих в виде единого потока UDP-пакетов. Наконец, *Идентификаторы сотрудничающих источников*, если таковые имеются, используются, когда конечный поток формируется несколькими источниками. В этом случае микширующее устройство является источником синхронизации, а в полях идентификаторов источников перечисляются смешиваемые потоки.

У протокола RTP есть небольшой родственный протокол под названием **RTCP** (Real-Time Transport Control Protocol — управляющий транспортный протокол реального времени). Он занимается поддержкой обратной связи, синхронизацией, обеспечением пользовательского интерфейса, однако не занимается передачей каких-либо данных. Первая его функция может использоваться для обратной связи по задержкам, джиттеру, пропускной способности, перегрузке и другим свойствам сети, о которых сообщается источникам. Полученная информация может приниматься во внимание кодировщиком для увеличения скорости передачи данных (что приведет к улучшению качества), когда это позволяет делать состояние сети, или уменьшения скорости при возникновении в сети каких-либо проблем. Постоянная обратная связь обеспечивает динамическую настройку алгоритмов кодирования на обеспечение наилучшего качества при текущих обстоятельствах. Например, пропускная способность при передаче потока может как увеличиваться, так и уменьшаться, и в соответствии с этим могут изменяться методы кодирования — скажем, MP3 может заменяться 8-битным PCM или дельта-кодированием. Поле *Тип данных* сообщает приемнику о том, какой алгоритм кодирования применяется для данного пакета, что позволяет изменять их по требованию при передаче потока.

RTCP также обеспечивает межпотоковую синхронизацию. Проблема состоит в том, что разные потоки могут использовать разные таймеры с разной степенью разрешения и разными скоростями дрейфа. RTCP помогает решить эти проблемы и синхронизировать потоки с разными параметрами.

Наконец, RTCP позволяет именовать различные источники (например, с помощью обычного ASCII-текста). Эта информация может отображаться на приемнике, позволяя определить источник текущего потока.

Более подробную информацию о протоколе RTP можно найти в (Perkins, 2002).

Транспортные протоколы Интернета: TCP

UDP является простым протоколом и имеет определенную область применения. В первую очередь, это клиент-серверные взаимодействия и мультимедиа. Тем не менее, большинству интернет-приложений требуется надежная, последовательная передача. UDP не удовлетворяет этим требованиям, поэтому требуется иной протокол. Такой протокол называется TCP, и он является рабочей лошадкой Интернета. Позже мы рассмотрим его детально.

Основы TCP

Протокол **TCP** (Transmission Control Protocol — протокол управления передачей) был специально разработан для обеспечения надежного сквозного байтового потока по ненадежной интерсети. Объединенная сеть отличается от отдельной сети тем, что ее различные участки могут обладать сильно различающейся топологией, пропускной способностью, значениями времени задержки, размерами пакетов и другими параметрами. При разработке TCP основное внимание уделялось способности протокола адаптироваться к свойствам объединенной сети и отказоустойчивости при возникновении различных проблем.

Протокол TCP описан в RFC 793. Со временем были обнаружены различные ошибки и неточности, и по некоторым пунктам требования были изменены. Подробное описание этих уточнений и исправлений дается в RFC 1122. Расширения протокола приведены в RFC 1323.

Каждая машина, поддерживающая протокол TCP, обладает транспортной сущностью TCP, являющейся либо библиотечной процедурой, либо пользовательским процессом, либо частью ядра системы. В любом случае, транспортная сущность управляет TCP-потоками и интерфейсом с IP-уровнем. TCP-сущность принимает от локальных процессов пользовательские потоки данных, разбивает их на куски, не превосходящие 64 Кбайт (на практике это число обычно равно 1460 байтам данных, что позволяет поместить их в один кадр Ethernet с заголовками IP и TCP), и посылает их в виде отдельных IP-дейтаграмм. Когда IP-дейтаграммы с TCP-данными прибывают на машину, они передаются TCP-сущности, которая восстанавливает исходный байтовый поток. Для простоты мы иногда будем употреблять «TCP» для обозначения транспортной сущности TCP (части программного обеспечения) или протокола TCP (набора правил). Из контекста будет понятно, что имеется в виду. Например, в выражении «Пользователь передает данные TCP» подразумевается, естественно, транспортная сущность TCP.

Уровень IP не гарантирует правильной доставки дейтаграмм, поэтому именно TCP приходится следить за истекшими интервалами ожидания и в случае необходимости заниматься повторной передачей пакетов. Бывает, что дейтаграммы прибывают в неправильном порядке. Восстанавливать сообщения из таких дейтаграмм обязан также TCP. Таким образом, протокол TCP призван обеспечить надежность, о которой мечтают многие пользователи и которая не предоставляется протоколом IP.

Модель службы TCP

В основе службы TCP лежат так называемые сокеты (гнезда или конечные точки), создаваемые как отправителем, так и получателем. Они обсуждались в разделе «Сокеты Беркли». У каждого сокета есть номер (адрес), состоящий из IP-адреса хоста и 16-битного номера, локального по отношению к хосту, называемого **портом**. Портом в TCP называют TSAP-адрес. Для обращения к службе TCP между сокетом машины отправителя и сокетом машины получателя должно быть явно установлено соединение. Примитивы сокетов приведены в табл. 6.2

Один сокет может использоваться одновременно для нескольких соединений. Другими словами, два и более соединений могут оканчиваться одним сокетом. Соединения различаются по идентификаторам сокетов на обоих концах — (*socket1*, *socket2*). Номера виртуальных каналов или другие идентификаторы не используются.

Номера портов со значениями ниже 1024, называемые **популярными портами**, зарезервированы стандартными сервисами. Например, любой процесс, желающий установить соединение с хостом для передачи файла с помощью протокола FTP, может связаться с портом 21 хоста-адресата и обратиться, таким образом, к его FTP-демону. Список популярных портов приведен на сайте www.iana.org. Таких портов на данный момент более 300. Некоторые из них включены в табл. 6.4.

Можно было бы, конечно, связать FTP-демона с портом 21 еще во время загрузки, тогда же связать демона telnet с портом 23, и т. д. Однако если бы мы так сделали, мы бы только зря заняли память информацией о демонах, которые, на самом деле, большую часть времени простаивают. Вместо этого обычно пользуются услугами одного демона, называемого в UNIX **inetd**, который связывается с несколькими портами и ожидает первое входящее соединение. Когда оно происходит, inetd создает новый процесс, для которого вызывается подходящий демон, обрабатывающий запрос. Таким образом, постоянно активен только inetd, остальные вызываются только тогда, когда для них есть работа. Inetd имеет специальный конфигурационный файл, из которого он может узнать о назначении портов. Это значит, что системный администратор может настроить систему таким образом, чтобы с самыми загруженными портами (например, 80) были связаны постоянные демоны, а с остальными — inetd.

Таблица 6.4. Некоторые зарезервированные порты

Порт	Протокол	Использование
21	FTP	Передача файлов
23	Telnet	Дистанционный вход в систему
25	SMTP	Электронная почта
69	TFTP	Простейший протокол передачи файлов
79	Finger	Поиск информации о пользователе
80	HTTP	Мировая Паутина
110	POP-3	Удаленный доступ к электронной почте
119	NNTP	Группы новостей

Все TCP-соединения являются полнодуплексными и двухточечными. Полный дуплекс означает, что трафик может следовать одновременно в противоположные стороны. Двухточечное соединение подразумевает, что у него имеются ровно две конечные точки. Широковещание и многоадресная рассылка протоколом TCP не поддерживаются.

TCP-соединение представляет собой байтовый поток, а не поток сообщений. Границы между сообщениями не сохраняются. Например, если отправляющий процесс записывает в TCP-поток четыре 512-байтовых порции данных, эти данные могут быть доставлены получающему процессу в виде четырех 512-байтовых порций, двух 1024-байтовых порций, одной 2048-байтовой порции (см. рис. 6.22) или как-нибудь еще. Нет способа, которым получатель смог бы определить, каким образом записывались данные.

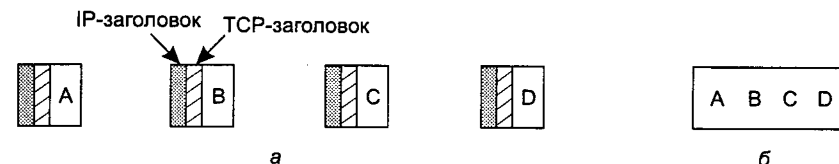


Рис. 6.22. Четыре 512-байтовых сегмента, посланные как отдельные IP-дейтаграммы (*а*); 2048 байт данных, доставленные приложению с помощью отдельного вызова процедуры READ (*б*)

Файлы в системе UNIX также обладают этим свойством. Программа, читающая файл, не может определить, как был записан этот файл: поблочно, побайтно или сразу целиком. Как и в случае с файлами системы UNIX, TCP-программы не имеют представления о назначении байтов и не интересуются этим. Байт для них — просто байт.

Получив данные от приложения, протокол TCP может послать их сразу или поместить в буфер, чтобы послать сразу большую порцию данных, по своему усмотрению. Однако иногда приложению бывает необходимо, чтобы данные были посланы немедленно. Допустим, например, что пользователь регистрируется на удаленной машине. После того как он ввел команду и нажал клавишу Enter, важно, чтобы введенная им строка была доставлена на удаленную машину сразу же, а не помещалась в буфер, пока не будет введена следующая строка. Чтобы вынудить передачу данных без промедления, приложение может установить флаг PUSH (протолкнуть).

Некоторые старые приложения использовали флаг PUSH как разделитель сообщений. Хотя этот трюк иногда срабатывает, не все реализации протокола TCP передают флаг PUSH принимающему приложению. Кроме того, если прежде чем первый пакет с установленным флагом PUSH будет передан в линию, TCP-сущность получит еще несколько таких пакетов (то есть выходная линия будет занята), TCP-сущность будет иметь право послать все эти данные в виде единой дейтаграммы, не разделяя их на отдельные порции.

Последней особенностью службы TCP, о которой следует упомянуть, являются **срочные данные**. Когда пользователь, взаимодействующий с программой

в интерактивном режиме, нажимает клавишу Delete или Ctrl-C, чтобы прервать начавшийся удаленный процесс, посылающее приложение помещает в выходной поток данных управляющую информацию и передает ее TCP-службе вместе с флагом URGENT (срочно). Этот флаг заставляет TCP-сущность прекратить накопление данных и без промедления передать в сеть все, что у нее есть для данного соединения.

Когда срочные данные прибывают по назначению, получающее приложение прерывается (то есть «получает сигнал», в терминологии UNIX), после чего оно может считать данные из входного потока и найти среди них срочные. Конец срочных данных маркируется, так что приложение может распознать, где они заканчиваются. Начало срочных данных не маркируется. Приложение должно само догадаться. Такая схема представляет собой грубый сигнальный механизм, оставляя все прочее приложению.

Протокол TCP

В данном разделе будет рассмотрен протокол TCP в общих чертах. В следующем разделе мы обсудим заголовок протокола, поле за полем.

Ключевым свойством TCP, определяющим всю структуру протокола, является то, что в TCP-соединении у каждого байта есть свой 32-разрядный порядковый номер. В первые годы существования Интернета базовая скорость передачи данных между маршрутизаторами по выделенным линиям составляла 56 Кбит/с. Хосту, постоянно выдающему данные с максимальной скоростью, потребовалось бы больше недели на то, чтобы порядковые номера совершили полный круг. При нынешних скоростях порядковые номера могут кончиться очень быстро, об этом еще будет сказано позже. Отдельные 32-разрядные порядковые номера используются для подтверждений и для механизма скользящего окна, о чем также будет сказано позже.

Отправляющая и принимающая TCP-сущности обмениваются данными в виде сегментов. **Сегмент** состоит из фиксированного 20-байтового заголовка (плюс необязательная часть), за которой могут следовать байты данных. Размер сегментов определяется программным обеспечением TCP. Оно может объединять в один сегмент данные, полученные в результате нескольких операций записи, или, наоборот, распределять результат одной записи между несколькими сегментами. Размер сегментов ограничен двумя пределами. Во-первых, каждый сегмент, включая TCP-заголовок, должен помещаться в 65 515-байтное поле полезной нагрузки IP-пакета. Во-вторых, в каждой сети есть **максимальная единица передачи** (**MTU**, Maximum Transfer Unit), и каждый сегмент должен помещаться в MTU. На практике размер максимальной единицы передачи составляет обычно 1500 байт (что соответствует размеру поля полезной нагрузки Ethernet), и таким образом определяется верхний предел размера сегмента.

Основным протоколом, используемым TCP-сущностями, является протокол скользящего окна. При передаче сегмента отправитель включает таймер. Когда сегмент прибывает в пункт назначения, принимающая TCP-сущность посылает обратно сегмент (с данными, если есть что посылать, или без данных) с номером подтверждения, равным порядковому номеру следующего ожидаемого сегмента. Если время ожидания подтверждения истекает, отправитель посылает сегмент еще раз.

Хотя этот протокол кажется простым, в нем имеется несколько деталей, которые следует рассмотреть подробнее. Сегменты могут приходить в неверном порядке. Так, например, возможна ситуация, в которой байты с 3072-го по 4095-й уже прибыли, но подтверждение для них не может быть выслано, так как байты с 2048-го по 3071-й еще не получены. К тому же сегменты могут задерживаться в сети так долго, что у отправителя истечет время ожидания и он передаст их снова. Переданный повторно сегмент может включать в себя уже другие диапазоны фрагментов, поэтому потребуется очень аккуратное администрирование для определения номеров байтов, которые уже были приняты корректно. Тем не менее, поскольку каждый байт в потоке единственным образом определяется по своему сдвигу, эта задача оказывается реальной.

Протокол TCP должен уметь справляться с этими проблемами и решать их эффективно. На оптимизацию производительности TCP-потоков было потрачено много сил. В следующем разделе мы обсудим несколько алгоритмов, используемых в различных реализациях протокола TCP.

Заголовок TCP-сегмента

На рис. 6.23 показана структура заголовка TCP-сегмента. Каждый сегмент начинается с 20-байтного заголовка фиксированного формата. За ним могут следовать дополнительные поля. После дополнительных полей может располагаться до 65 535 – 20 – 20 = 65 495 байт данных, где первые 20 байт — это IP-заголовок, а вторые — TCP-заголовок. Сегменты могут и не содержать данных. Такие сегменты часто применяются для передачи подтверждений и управляющих сообщений.

Рассмотрим TCP-заголовок поле за полем. Поля *Порт получателя* и *Порт отправителя* являются идентификаторами локальных конечных точек соединения. Популярные номера портов перечислены на www.iana.org, однако, что касается всех остальных портов, то каждый хост может сам решать, как их распределять. Номер порта вместе с IP-адресом хоста образуют уникальный 48-битный идентификатор конечной точки. Пара таких идентификаторов, относящихся к источнику и приемнику, однозначно определяет соединение.

Поля *Порядковый номер* и *Номер подтверждения* выполняют свою обычную функцию. Обратите внимание: поле *Номер подтверждения* относится к следующему ожидаемому байту, а не к последнему полученному. Оба они 32-разрядные, так как в TCP-потоке нумеруется каждый байт данных.

Поле *Длина TCP-заголовка* содержит размер TCP-заголовка, выраженный в 32-разрядных словах. Эта информация необходима, так как поле *Факультативные поля*, а вместе с ним и весь заголовок, может быть переменной длины. По сути, это поле указывает смещение от начала сегмента до поля данных, измеренное в 32-битных словах. Это то же самое, что длина заголовка.

Рис. 6.23. Заголовок TCP

Следом идет неиспользуемое 6-битное поле. Тот факт, что это поле выжило в течение четверти века, является свидетельством того, насколько хорошо продуман дизайн TCP.

Затем следуют шесть 1-битовых флагов. Бит *URG* устанавливается в 1 в случае использования поля *Указатель на срочные данные*, содержащего смещение в байтах от текущего порядкового номера байта до места расположения срочных данных. Таким образом в протоколе TCP реализуются прерывающие сообщения. Как уже упоминалось, протокол TCP лишь обеспечивает доставку сигнала пользователя до получателя, не интересуясь причиной прерывания.

Если бит *ACK* установлен в 1, значит, поле *Номер подтверждения* содержит осмысленные данные. В противном случае данный сегмент не содержит подтверждения, и поле *Номер подтверждения* просто игнорируется.

Бит *PSH* является, по сути, PUSH-флагом, с помощью которого отправитель просит получателя доставить данные приложению сразу по получении пакета, а не хранить их в буфере, пока тот не наполнится. (Получатель может заниматься буферизацией для достижения большей эффективности.)

Бит *RST* используется для сброса состояния соединения, которое из-за сбоя хоста или по другой причине попало в тупиковую ситуацию. Кроме того, он используется для отказа от неверного сегмента или от попытки создать соединение. Если вы получили сегмент с установленным битом *RST*, это означает наличие какой-то проблемы.

Бит *SYN* применяется для установки соединения. У запроса соединения бит $SYN = 1$, а бит $ACK = 0$, что означает, что поле подтверждения не используется. В ответе на этот запрос содержится подтверждение, поэтому значения этих битов в нем равны: $SYN = 1$, $ACK = 1$. Таким образом, бит *SYN* используется для обозначения сегментов CONNECTION REQUEST и CONNECTION ACCEPTED, а бит *ACK* — чтобы отличать их друг от друга.

Бит *FIN* используется для разрыва соединения. Он указывает на то, что у отправителя больше нет данных для передачи. Однако, даже закрыв соединение, процесс может продолжать получать данные в течение неопределенного времени. У сегментов с битами *FIN* и *SYN* есть порядковые номера, что гарантирует правильный порядок их выполнения.

Управление потоком в протоколе TCP осуществляется при помощи скользящего окна переменного размера. Поле *Размер окна* сообщает, сколько байт может быть послано после байта, получившего подтверждение. Значение поля *Размер окна* может быть равно нулю, что означает, что все байты вплоть до *Номер подтверждения*-1 получены, но у получателя в данный момент какие-то проблемы, и остальные байты он пока принять не может. Разрешение на дальнейшую передачу может быть получено путем отправки сегмента с таким же значением поля *Номер подтверждения* и ненулевым значением поля *Размер окна*.

В главе 3 мы обсуждали протоколы, в которых подтверждения приема кадров были связаны с разрешениями на продолжение передачи. Эта связь была следствием жестко закрепленного размера скользящего окна в этих протоколах. В TCP подтверждения отделены от разрешений на передачу данных. В сущности, приемник может сказать: «Я получил байты вплоть до *k*-го, но я сейчас не хочу продолжать прием данных». Такое разделение (выражающееся в скользящем окне *переменного размера*) придает протоколу дополнительную гибкость. Далее мы обсудим этот аспект более детально.

Поле *Контрольная сумма* служит для повышения надежности. Оно содержит контрольную сумму заголовка, данных и псевдозаголовка, показанного на рис. 6.24. При выполнении вычислений поле *Контрольная сумма* устанавливается равным нулю, а поле данных дополняется нулевым байтом, если его длина представляет собой нечетное число. Алгоритм вычисления контрольной суммы просто складывает все 16-разрядные слова в дополнительном коде, а затем вычисляет дополнение для всей суммы. В результате, когда получатель считает контрольную сумму всего сегмента, включая поле *Контрольная сумма*, результат должен быть равен 0.

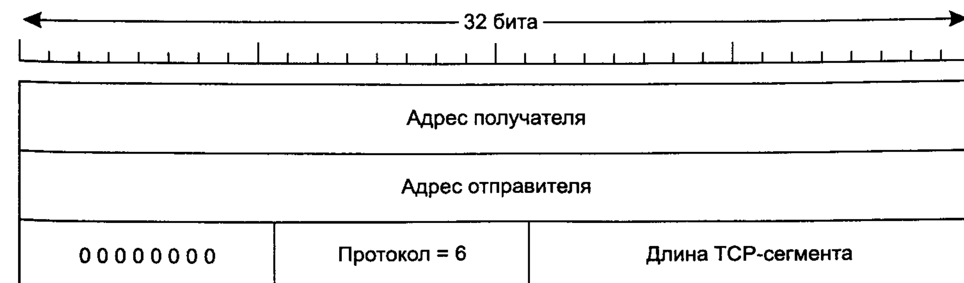


Рис. 6.24. Псевдозаголовок, включаемый в контрольную сумму TCP

Псевдозаголовок содержит 32-разрядные IP-адреса отправителя и получателя, номер протокола для TCP (6) и счетчик байтов для TCP-сегмента (включая заголовок). Включение псевдозаголовка в контрольную сумму TCP помогает обнаружить неверно доставленные пакеты, хотя это нарушает иерархию протоколов, так как IP-адреса в нем принадлежат IP-уровню, а не TCP-уровню. В UDP для контрольной суммы используется такой же псевдозаголовок.

Поле *Факультативные поля* предоставляет дополнительные возможности, не покрываемые стандартным заголовком. С помощью одного из таких полей каждый хост может указать максимальный размер поля полезной нагрузки, который он может принять. Чем больше размер используемых сегментов, тем выше эффективность, так как при этом снижается удельный вес накладных расходов в виде 20-байтных заголовков, однако не все хосты способны принимать очень большие сегменты. Хосты могут сообщить друг другу максимальный размер поля полезной нагрузки во время установки соединения. По умолчанию этот размер равен 536 байтам. Все хосты обязаны принимать TCP-сегменты размером 536 + 20 = 556 байт. Для каждого из направлений может быть установлен свой максимальный размер поля полезной нагрузки.

Для линий с большой скоростью передачи и/или большой задержкой окно размером в 64 Кбайт оказывается слишком маленьким. Так, для линии T3 (44,736 Мбит/с) полное окно может быть передано в линию всего за 12 мс. Если значение времени распространения сигнала в оба конца составляет 50 мс (что типично для трансконтинентального оптического кабеля), 3/4 времени отправитель будет заниматься ожиданием подтверждения. При связи через спутник ситуация будет еще хуже. Больший размер окна мог бы улучшить эффективность, но 16-битовое поле *Размер окна* не позволяет этого сделать. В RFC 1323 был предложен новый параметр *Масштаб окна*, о значении которого два хоста могли договориться при установке соединения. Это число позволяет сдвигать поле *Размер окна* до 14 разрядов влево, обеспечивая расширение размера окна до 2^{30} байт (1 Гбайт). В настоящее время большинство реализаций протокола TCP поддерживают эту возможность.

Еще одна возможность, предложенная в RFC 1106 и широко применяемая сейчас, состоит в использовании протокола выборочного повтора вместо возврата на *n*. Если адресат получает один плохой сегмент и следом за ним большое количество хороших, у нормального TCP-протокола в конце концов истечет время ожидания и он передаст повторно все неподтвержденные сегменты, включая те, что были получены правильно. В документе RFC 1106 было предложено использовать отрицательные подтверждения (NAK), позволяющие получателю запрашивать отдельный сегмент или несколько сегментов. Получив его, принимающая сторона может подтвердить все хранящиеся в буфере данные, уменьшая таким образом количество повторно передаваемых данных.

Установка TCP-соединения

В протоколе TCP-соединения устанавливаются с помощью «тройного рукопожатия», описанного в разделе «Установка соединения». Чтобы установить соединение, одна сторона (например, сервер) пассивно ожидает входящего соединения, выполняя примитивы LISTEN и ACCEPT, либо указывая конкретный источник, либо не указывая его.

Другая сторона (например, клиент) выполняет примитив CONNECT, указывая IP-адрес и порт, с которым он хочет установить соединение, максимальный размер TCP-сегмента и, по желанию, некоторые данные пользователя (например, пароль). Примитив CONNECT посылает TCP-сегмент с установленным битом *SYN* и сброшенным битом *ACK* и ждет ответа.

Когда этот сегмент прибывает в пункт назначения, TCP-сущность проверяет, выполнил ли какой-нибудь процесс примитив LISTEN, указав в качестве параметра тот же порт, который содержится в поле *Порт получателя*. Если такого процесса нет, она отвечает отправкой сегмента с установленным битом *RST* для отказа от соединения.

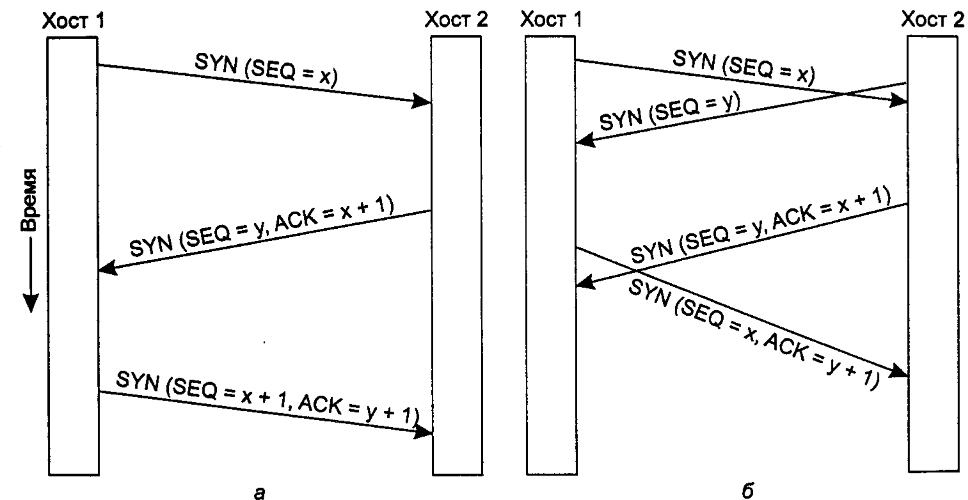


Рис. 6.25. Установка TCP-соединения в нормальном случае (*а*); столкновение вызовов (*б*)

Если какой-либо процесс прослушивает какой-либо порт, то входящий TCP-сегмент передается этому процессу. Последний может принять соединение или отказаться от него. Если процесс принимает соединение, он отсылает в ответ подтверждение. Последовательность TCP-сегментов, посылаемых в нормальном случае, показана на рис. 6.25, *а*. Обратите внимание на то, что сегмент с установленным битом *SYN* занимает 1 байт пространства порядковых номеров, что позволяет избежать неоднозначности в их подтверждениях.

Если два хоста одновременно попытаются установить соединение друг с другом, то последовательность происходящих при этом событий будет соответствовать рис. 6.25, *б*. В результате будет установлено только одно соединение, а не два, так как пара конечных точек однозначно определяет соединение. То есть если оба соединения пытаются идентифицировать себя с помощью пары (x, y), делается всего одна табличная запись для (x, y).

Начальное значение порядкового номера соединения не равно нулю по обсуждавшимся выше причинам. Используется схема, основанная на таймере, изменяющем свое состояние каждые 4 мкс. Для большей надежности хосту после сбоя запрещается перезагружаться ранее чем по прошествии максимального времени жизни пакета. Это позволяет гарантировать, что ни один пакет от прежних соединений не бродит где-нибудь в Интернете.

Разрыв соединения TCP

Хотя TCP-соединения являются полнодуплексными, чтобы понять, как происходит их разъединение, лучше считать их парами симплексных соединений. Каждое симплексное соединение разрывается независимо от своего напарника. Чтобы разорвать соединение, любая из сторон может послать TCP-сегмент с установленным в единицу битом *FIN*, что означает, что у него больше нет данных для передачи. Когда этот TCP-сегмент получает подтверждение, это направление передачи закрывается. Тем не менее, данные могут продолжать передаваться неопределенно долго в противоположном направлении. Соединение разрывается, когда оба направления закрываются. Обычно для разрыва соединения требуются четыре TCP-сегмента: по одному с битом *FIN* и по одному с битом *ACK* в каждом направлении. Первый бит *ACK* и второй бит *FIN* могут также содержаться в одном TCP-сегменте, что уменьшит количество сегментов до трех.

Как при телефонном разговоре, когда оба участника могут одновременно попрощаться и повесить трубки, оба конца TCP-соединения могут послать *FIN*-сегменты в одно и то же время. Они оба получают обычные подтверждения, и соединение закрывается. По сути, между одновременным и последовательным разъединениями нет никакой разницы.

Чтобы избежать проблемы двух армий, используются таймеры. Если ответ на посланный *FIN*-сегмент не приходит в течение двух максимальных интервалов времени жизни пакета, отправитель *FIN*-сегмента разрывает соединение. Другая сторона в конце концов заметит, что ей никто не отвечает, и также разорвет соединение. Хотя такое решение и не идеально, но, учитывая недостижимость идеала, приходится пользоваться тем, что есть. На практике проблемы возникают довольно редко.

Модель управления TCP-соединением

Этапы, необходимые для установки и разрыва соединения, могут быть представлены в виде модели конечного автомата, 11 состояний которого перечислены в табл. 6.6. В каждом из этих состояний могут происходить разрешенные и запрещенные события. В ответ на какое-либо разрешенное событие может осуществляться определенное действие. При возникновении запрещенных событий сообщается об ошибке.

Каждое соединение начинается в состоянии *CLOSED* (закрытое). Оно может выйти из этого состояния, предпринимая либо активную (CONNECT), либо пассивную (LISTEN) попытку открыть соединение. Если противоположная сторона осуществляет противоположные действия, соединение устанавливается и переходит в состояние *ESTABLISHED*. Инициатором разрыва соединения может выступить любая сторона. По завершении процесса разъединения соединение возвращается в состояние *CLOSED*.

Таблица 6.5. Состояния конечного автомата, управляющего TCP-соединением

Состояние	Описание
CLOSED	Закрыто. Соединение не является активным и не находится в процессе установки
LISTEN	Ожидание. Сервер ожидает входящего запроса
SYN RCVD	Прибыл запрос соединения. Ожидание подтверждения
SYN SENT	Запрос соединения послан. Приложение начало открывать соединение
ESTABLISHED	Установлено. Нормальное состояние передачи данных
FIN WAIT 1	Приложение сообщило, что ему больше нечего передавать
FIN WAIT 2	Другая сторона согласна разорвать соединение
TIMED WAIT	Ожидание, пока в сети не исчезнут все пакеты
CLOSING	Обе стороны попытались одновременно закрыть соединение
CLOSE WAIT	Другая сторона инициировала разъединение
LAST ACK	Ожидание, пока в сети не исчезнут все пакеты

Конечный автомат показан на рис. 6.26. Типичный случай клиента, активно соединяющегося с пассивным сервером, показан жирными линиями — сплошными для клиента и пунктирными для сервера. Тонкие линии обозначают необычные последовательности событий. Каждая линия на рис. 6.26 маркирована парой *событие/действие*. Событие может представлять собой либо обращение пользователя к системной процедуре (CONNECT, LISTEN, SEND или CLOSE), либо прибытие сегмента (*SYN*, *FIN*, *ACK* или *RST*), либо, в одном случае, окончание периода ожидания, равного двойному времени жизни пакетов. Действие может состоять в отправке управляющего сегмента (*SYN*, *FIN* или *RST*). Впрочем, может не предприниматься никакого действия, что обозначается прочерком. В скобках приводятся комментарии.

Диаграмму легче всего понять, если сначала проследовать по пути клиента (сплошная жирная линия), а затем — по пути сервера (жирный пунктир). Когда приложение на машине клиента вызывает примитив CONNECT, локальная TCP-сущность создает запись соединения, помечает его состояние как *SYN SENT* и посылает *SYN*-сегмент. Примечательно, что несколько приложений одновременно могут открыть несколько соединений, поэтому свое состояние, хранящееся в записи соединения, имеется у каждого отдельного соединения. Когда прибывает сегмент *SYN* + *ACK*, TCP-сущность посылает последний *ACK*-сегмент «тройного рукопожатия» и переключается в состояние *ESTABLISHED*. В этом состоянии можно пересылать и получать данные.

Когда у приложения заканчиваются данные для передачи, оно выполняет примитив CLOSE, заставляющий локальную TCP-сущность послать *FIN*-сегмент и ждать ответного *ACK*-сегмента (пунктирный прямоугольник с пометкой «активное

разъединение»). Когда прибывает подтверждение, происходит переход в состояние *FIN WAIT 2*, и одно направление соединения закрывается. Когда приходит встречный *FIN*-сегмент, в ответ на него также высылается подтверждение, и второе направление соединения также закрывается. Теперь обе стороны соединения закрыты, но TCP-сущность выжидает в течение времени, равного максимальному времени жизни пакета, чтобы можно было гарантировать, что все пакеты этого соединения больше не перемещаются по сети даже в том случае, если подтверждение было потеряно. Когда этот период ожидания истекает, TCP-сущность удаляет запись о соединении.

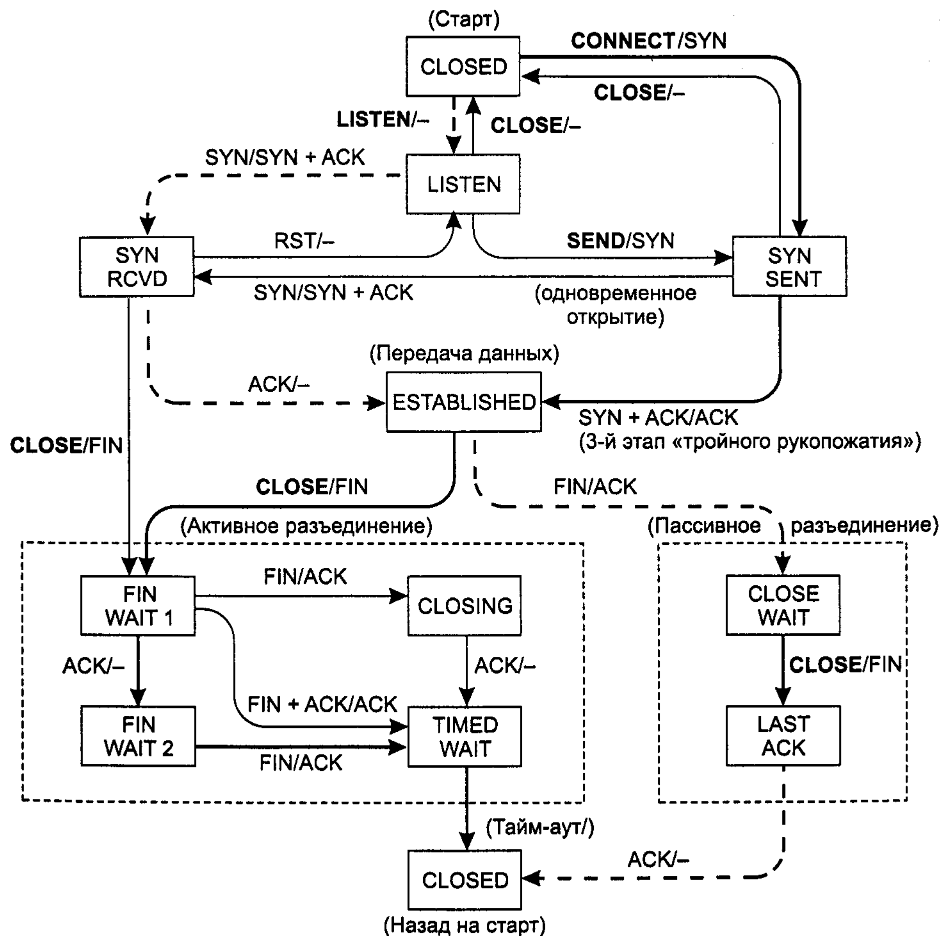


Рис. 6.26. Конечный автомат TCP-соединения. Жирная сплошная линия показывает нормальный путь клиента. Пунктиром показан нормальный путь сервера. Тонкими линиями обозначены необычные события

Рассмотрим теперь управление соединением с точки зрения сервера. Сервер выполняет примитив `LISTEN` и переходит в режим ожидания запросов соединения. Когда приходит *SYN*-сегмент, в ответ на него высылается подтверждение, после чего сервер переходит в состояние *SYN RCVD* (запрос соединения получен). Когда в ответ на *SYN*-подтверждение сервера от клиента приходит *ACK*-сегмент, процедура «тройного рукопожатия» завершается и сервер переходит в состояние *ESTABLISHED*. Теперь можно пересылать данные.

По окончании выполнения своей задачи клиент запускает примитив `CLOSE`, в результате чего на сервер прибывает *FIN*-сегмент (пунктирный прямоугольник, обозначенный как пассивное разъединение). Теперь сервер выполняет примитив `CLOSE`, а *FIN*-сегмент посылается клиенту. Когда от клиента прибывает подтверждение, сервер разрывает соединение и удаляет запись о нем.

Управление передачей в TCP

Как уже было сказано ранее, управление окном в TCP не привязано напрямую к подтверждениям, как это сделано в большинстве протоколов передачи данных. Например, предположим, что у получателя есть 4096-байтовый буфер, как показано на рис. 6.27. Если отправитель передает 2048-байтовый сегмент, который успешно принимается получателем, то получатель подтверждает его получение. Однако при этом у получателя остается всего лишь 2048 байт свободного буферного пространства (пока приложение не заберет сколько-нибудь данных из буфера), о чем он и сообщает отправителю, указывая соответствующий размер окна (2048) и номер следующего ожидаемого байта.

После этого отправитель посылает еще 2048 байт, получение которых подтверждается, но размер окна объявляется равным 0. Отправитель должен прекратить передачу до тех пор, пока получающий хост не освободит место в буфере и не увеличит размер окна.

При нулевом размере окна отправитель не может посылать сегменты, за исключением двух случаев. Во-первых, разрешается посылать срочные данные, например, чтобы пользователь мог уничтожить процесс, выполняющийся на удаленной машине. Во-вторых, отправитель может послать 1-байтовый сегмент, прося получателя повторить информацию о размере окна и ожидаемом следующем байте. Стандарт TCP явно предусматривает эту возможность для предотвращения тупиковых ситуаций в случае потери объявления о размере окна.

Отправители не обязаны передавать данные сразу, как только они приходят от приложения. Также никто не требует от получателей посылать подтверждения как можно скорее. Например, на рис. 6.27 TCP-сущность, получив от приложения первые 2 Кбайт данных и зная, что доступный размер окна равен 4 Кбайт, была бы совершенно права, если бы просто сохранила полученные данные в буфере до тех пор, пока не прибудут еще 2 Кбайт данных, чтобы передать сразу сегмент с 4 Кбайт полезной нагрузки. Эта свобода действий может использоваться для улучшения производительности.

Рассмотрим TELNET-соединение с интерактивным редактором, реагирующим на каждое нажатие клавиши. В худшем случае, когда символ прибывает к передающей TCP-сущности, она создает 21-байтовый TCP-сегмент и передает его IP-уровню, который, в свою очередь, посылает 41-байтовую IP-дейтаграмму.

На принимающей стороне TCP-сущность немедленно отвечает 40-байтовым подтверждением (20 байт TCP-заголовка и 20 байт IP-заголовка). Затем, когда редактор прочитает этот байт из буфера, TCP-сущность пошлет обновленную информацию о размере буфера, передвинув окно на 1 байт вправо. Размер этого пакета также составляет 40 байт. Наконец, когда редактор обработает этот символ, он отправляет обратно эхо, передаваемое 41-байтовым пакетом. Итого для каждого введенного с клавиатуры символа пересылается четыре пакета общим размером 162 байта. В условиях дефицита пропускной способности линий этот метод работы нежелателен.

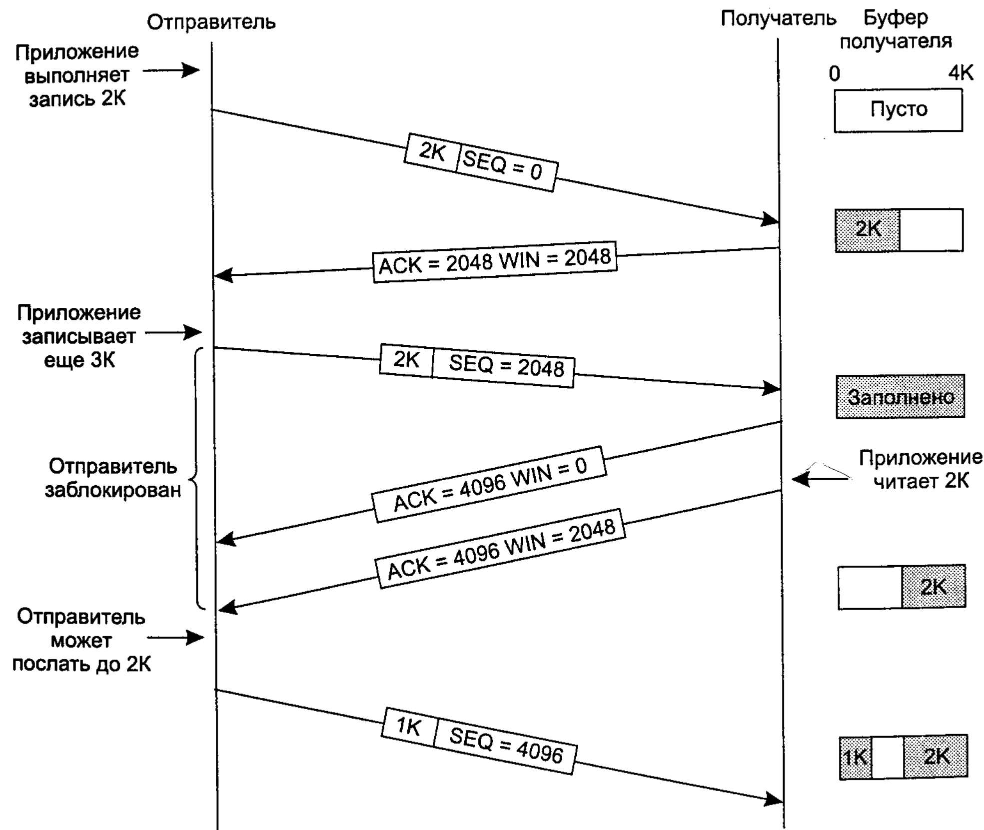


Рис. 6.27. Управление окном в TCP

Для улучшения ситуации многие реализации TCP используют задержку подтверждений и обновлений размера окна на 500 мс в надежде получить дополнительные данные, вместе с которыми можно будет отправить подтверждение одним пакетом. Если редактор успеет выдать эхо в течение 500 мс, удаленному пользователю нужно будет выслать только один 41-байтовый пакет, таким образом, нагрузка на сеть снизится вдвое.

Хотя такой метод задержки и снижает нагрузку на сеть, тем не менее, эффективность использования сети отправителем продолжает оставаться невысокой, так как каждый байт пересылается в отдельном 41-байтовом пакете. Метод, позволяющий повысить эффективность, известен как **алгоритм Нагля** (Nagle, 1984). Предложение Нагля звучит довольно просто: если данные поступают отправителю по одному байту, отправитель просто передает первый байт, а остальные помещает в буфер, пока не будет получено подтверждение приема первого байта. После этого можно переслать все накопленные в буфере символы в виде одного TCP-сегмента и снова начать буферизацию до получения подтверждения отосланных символов. Если пользователь вводит символы быстро, а сеть медленная, то в каждом сегменте будет передаваться значительное количество символов, таким образом, нагрузка на сеть будет существенно снижена. Кроме того, этот алгоритм позволяет посылать новый пакет, даже если число символов в буфере превышает половину размера окна или максимальный размер сегмента.

Алгоритм Нагля широко применяется различными реализациями протокола TCP, однако иногда бывают ситуации, в которых его лучше отключить. В частности, при работе приложения X-Windows в Интернете информация о перемещениях мыши пересылается на удаленный компьютер. (X-Window — это система управления окнами в большинстве ОС типа UNIX). Если буферизировать эти данные для пакетной пересылки, курсор будет перемещаться рывками с большими паузами, в результате чего пользоваться программой будет очень сложно, почти невозможно.

Еще одна проблема, способная значительно снизить производительность протокола TCP, известная под именем **синдрома глупого окна** (Clark, 1982). Суть проблемы состоит в том, что данные пересылаются TCP-сущностью крупными блоками, но принимающая сторона интерактивного приложения считывает их посимвольно. Чтобы ситуация стала понятнее, рассмотрим рис. 6.28. Начальное состояние таково: TCP-буфер приемной стороны полон, и отправителю это известно (то есть размер его окна равен 0). Затем интерактивное приложение читает один символ из TCP-потока. Принимающая TCP-сущность радостно сообщает отправителю, что размер окна увеличился, и что он теперь может послать 1 байт. Отправитель повинуется и посылает 1 байт. Буфер снова оказывается полон, о чем получатель и извещает, посылая подтверждение для 1-байтового сегмента с нулевым размером окна. И так может продолжаться вечно.

Дэвид Кларк (David Clark) предложил запретить принимающей стороне отправлять информацию об однобайтовом размере окна. Вместо этого получатель должен подождать, пока в буфере не накопится значительное количество свободного места. В частности, получатель не должен отправлять сведения о новом размере окна до тех пор, пока он не сможет принять сегмент максимального размера, который он объявлял при установке соединения, или его буфер не освободился хотя бы наполовину.

Кроме того, увеличению эффективности отправки может способствовать сам отправитель, отказываясь от отправки слишком маленьких сегментов. Вместо этого он должен подождать, пока размер окна не станет достаточно большим для

того, чтобы можно было послать полный сегмент или, по меньшей мере, равный половине размера буфера получателя. (Отправитель может оценить этот размер по последовательности сообщений о размере окна, полученных им ранее.)

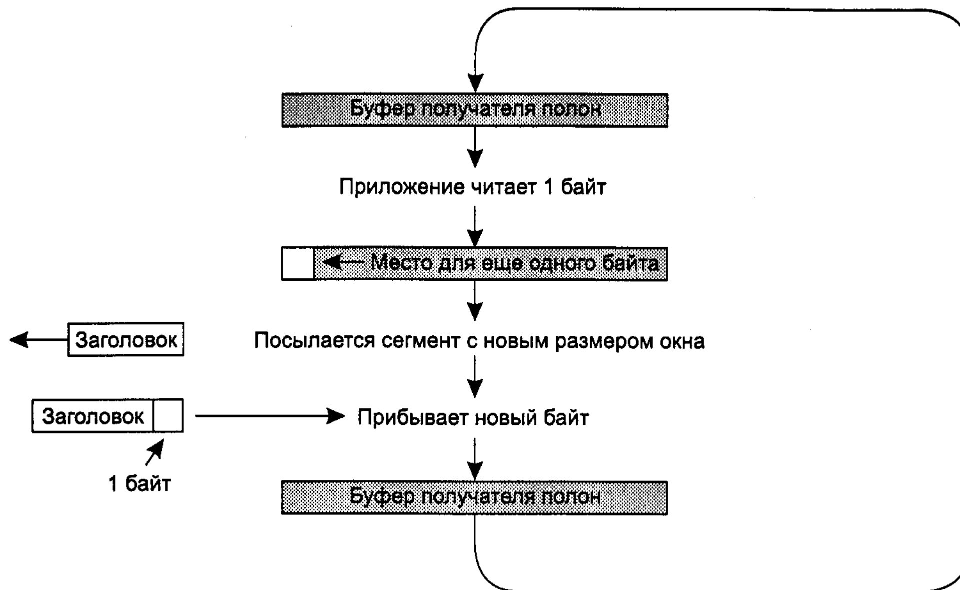


Рис. 6.28. Синдром глупого окна

В задаче избавления от синдрома глупого окна алгоритм Нагля и решение Кларка дополняют друг друга. Нагль пытался решить проблему приложения, предоставляющего данные TCP-сущности посимвольно. Кларк старался разрешить проблему приложения, посимвольно получающего данные у TCP. Оба решения хороши и могут работать одновременно. Суть их состоит в том, чтобы не посылать и не просить передавать данные слишком малыми порциями.

Принимающая TCP-сущность может пойти еще дальше в деле повышения производительности, просто обновляя информацию о размере окна большими порциями. Как и отправляющая TCP-сущность, она также может буферизировать данные и блокировать запрос на чтение READ, поступающий от приложения, до тех пор, пока у нее не накопится большого объема данных. Таким образом, снижается количество обращений к TCP-сущности и, следовательно, снижаются накладные расходы. Конечно, такой подход увеличивает время ожидания ответа, но для неинтерактивных приложений, например при передаче файла, сокращение времени, затраченного на всю операцию, значительно важнее увеличения времени ожидания ответа на отдельные запросы.

Еще одна проблема получателя состоит в сегментах, полученных в неправильном порядке. Они могут храниться или отвергаться по усмотрению получателя. Разумеется, подтверждение может быть выслано, только если все данные вплоть до подтверждаемого байта получены. Если до получателя доходят сегменты 0, 1, 2, 4, 5, 6 и 7, он может подтвердить получение данных вплоть до последнего байта сегмента 2. Когда у отправителя истечет время ожидания, он передаст сегмент 3 еще раз. Если к моменту прибытия сегмента 3 получатель сохранит в буфере сегменты с 4-го по 7-й, он сможет подтвердить получение всех байтов, вплоть до последнего байта сегмента 7.

Борьба с перегрузкой в TCP

Когда в какую-либо сеть поступает больше данных, чем она способна обработать, в сети образуются заторы. Интернет в этом смысле не является исключением. В данном разделе мы обсудим алгоритмы, разработанные (за последние двадцать пять лет) для борьбы с перегрузкой сети. Хотя сетевой уровень также пытается бороться с перегрузкой, основной вклад в решение этой проблемы, заключающееся в снижении скорости передачи данных, осуществляется протоколом TCP.

Теоретически, с перегрузкой можно бороться с помощью принципа, заимствованного из физики, — закона сохранения пакетов. Идея состоит в том, чтобы не передавать в сеть новые пакеты, пока ее не покинут (то есть не будут доставлены) старые. Протокол TCP пытается достичь этой цели с помощью динамического управления размером окна.

Первый шаг в борьбе с перегрузкой состоит в том, чтобы обнаружить ее. Пару десятилетий назад обнаружить перегрузку в сети было сложно. Трудно было понять, почему пакет не доставлен вовремя. Помимо возможности перегрузки сети имелась также большая вероятность потерять пакет вследствие высокого уровня помех на линии.

В настоящее время потери пакетов при передаче случаются относительно редко, так как большинство междугородных линий связи являются оптоволоконными (хотя в беспроводных сетях процент пакетов, теряемых из-за помех, довольно высок). Соответственно, большинство потерянных пакетов в Интернете вызвано заторами. Все TCP-алгоритмы Интернета предполагают, что потери пакетов вызываются перегрузкой сети, и следят за тайм-аутами как за предвестниками проблем, подобно шахтерам, наблюдающим за своими канарейками.

Прежде чем перейти к обсуждению того, как TCP реагирует на перегрузку, опишем сначала способы ее предотвращения, применяемые протоколом. При обнаружении перегрузки должен быть выбран подходящий размер окна. Получатель может указать размер окна, исходя из количества свободного места в буфере. Если отправитель будет иметь в виду размер отведенного ему окна, переполнение буфера у получателя не сможет стать причиной проблемы, однако она все равно может возникнуть из-за перегрузки на каком-либо участке сети между отправителем и получателем.

На рис. 6.29 эта проблема проиллюстрирована на примере водопровода. На рис. 6.29, *а* мы видим толстую трубу, ведущую к получателю с небольшой емкостью. До тех пор, пока отправитель не посылает воды больше, чем может поместиться в ведро, вода не будет проливаться. На рис. 6.29, *б* ограничительным фак-

тором является не емкость ведра, а пропускная способность сети. Если из крана в воронку вода будет литься слишком быстро, то уровень воды в воронке начнет подниматься и, в конце концов, часть воды может перелиться через край воронки.

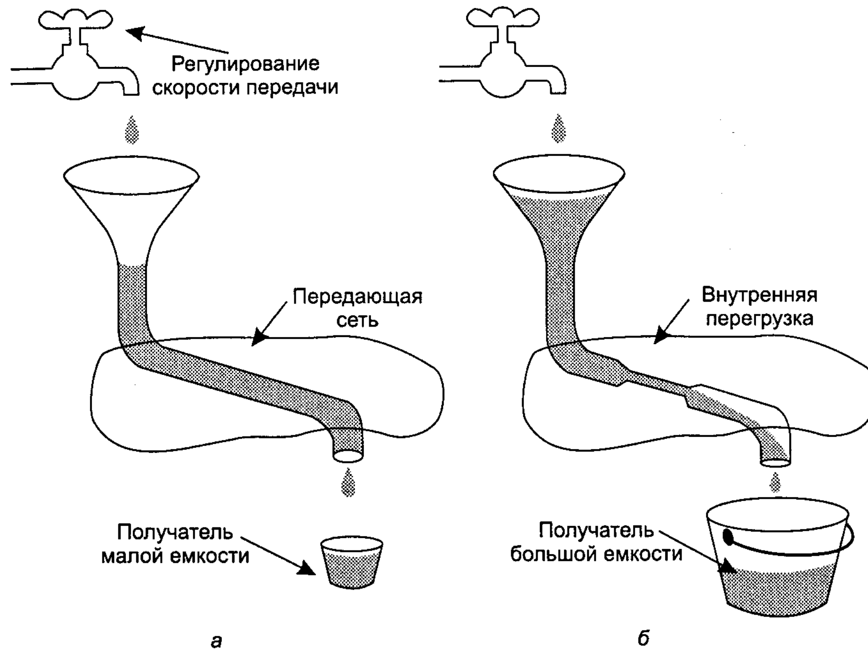


Рис. 6.29. Быстрая сеть и получатель малой емкости (*а*); медленная сеть и получатель большой емкости (*б*)

Решение, применяемое в Интернете, состоит в признании существования двух потенциальных проблем: низкой пропускной способности сети и низкой емкости получателя — и в раздельном решении обеих проблем. Для этого у каждого отправителя есть два окна: окно, предоставленное получателем, и **окно перегрузки**. Размер каждого из них соответствует количеству байтов, которое отправитель имеет право передать. Отправитель руководствуется минимальным из этих двух значений. Например, получатель говорит: «Посылайте 8 Кбайт», но отправитель знает, что если он пошлет более 4 Кбайт, то в сети образуется затор, поэтому он посылает все же 4 Кбайт. Если же отправитель знает, что сеть способна пропустить и большее количество данных, например 32 Кбайт, он передаст столько, сколько просит получатель (то есть 8 Кбайт).

При установке соединения отправитель устанавливает размер окна перегрузки равным размеру максимального используемого в данном соединении сегмента. Затем он передает один максимальный сегмент. Если подтверждение получения этого сегмента прибывает прежде, чем истекает период ожидания, к размеру окна добавляется размер сегмента, то есть размер окна перегрузки удваивается, и посылаются уже два сегмента. В ответ на подтверждение получения каждого из сегментов производится расширение окна перегрузки на величину одного максимального сегмента. Допустим, размер окна равен *n* сегментам. Если подтверждения для всех сегментов приходят вовремя, окно увеличивается на число байтов, соответствующее *n* сегментам. По сути, подтверждение каждой последовательности сегментов приводит к удвоению окна перегрузки.

Этот процесс экспоненциального роста продолжается до тех пор, пока не будет достигнут размер окна получателя или не будет выработан признак тайм-аута, сигнализирующий о перегрузке в сети. Например, если пакеты размером 1024, 2048 и 4096 байт дошли до получателя успешно, а в ответ на передачу пакета размером 8192 байта подтверждение не пришло в установленный срок, окно перегрузки устанавливается равным 4096 байтам. Пока размер окна перегрузки остается равным 4096 байтам, более длинные пакеты не посылаются, независимо от размера окна, предоставляемого получателем. Этот алгоритм называется **затяжным пуском**, или **медленным пуском**. Однако он не такой уж и медленный (Jacobson, 1988). Он экспоненциальный. Все реализации протокола TCP обязаны его поддерживать.

Рассмотрим теперь механизм борьбы с перегрузкой, применяемый в Интернете. Помимо окон получателя и перегрузки, в качестве третьего параметра в нем используется **пороговое значение**, которое изначально устанавливается равным 64 Кбайт. Когда возникает ситуация тайм-аута (подтверждение не возвращается в срок), новое значение порога устанавливается равным половине текущего размера окна перегрузки, а окно перегрузки уменьшается до размера одного максимального сегмента. Затем, так же как и в предыдущем случае, используется алгоритм затяжного пуска, позволяющий быстро обнаружить предел пропускной способности сети. Однако на этот раз экспоненциальный рост размера окна останавливается по достижении им порогового значения, после чего окно увеличивается линейно, на один сегмент для каждой следующей передачи. В сущности, предполагается, что можно спокойно урезать вдвое размер окна перегрузки, после чего постепенно наращивать его.

Иллюстрация работы данного алгоритма борьбы с перегрузкой приведена на рис. 6.30. Максимальный размер сегмента в данном примере равен 1024 байт. Сначала окно перегрузки было установлено равным 64 Кбайт, но затем произошел тайм-аут, и порог стал равным 32 Кбайт, а окно перегрузки — 1 Кбайт (передача 0). Затем размер окна перегрузки удваивается на каждом шаге, пока не достигает порога (32 Кбайт). Начиная с этого момента, размер окна увеличивается линейно.

Передача 13 оказывается несчастливой (как и положено), так как срабатывает тайм-аут. При этом пороговое значение устанавливается равным половине текущего размера окна (40 Кбайт пополам, то есть 20 Кбайт), и опять происходит затяжной пуск. После достижения порогового значения экспоненциальный рост размера окна сменяется линейным.

Если тайм-аутов больше не возникает, окно перегрузки может продолжать расти до размера окна получателя. Затем рост прекратится, и размер окна останется постоянным, пока не произойдет тайм-аут или не изменится размер окна получателя. Прибытие ICMP-пакета SOURCE QUENCH (гашение источника) обрабатывается таким же образом, как и тайм-аут. Альтернативный (и более современный) подход описан в RFC 3168.

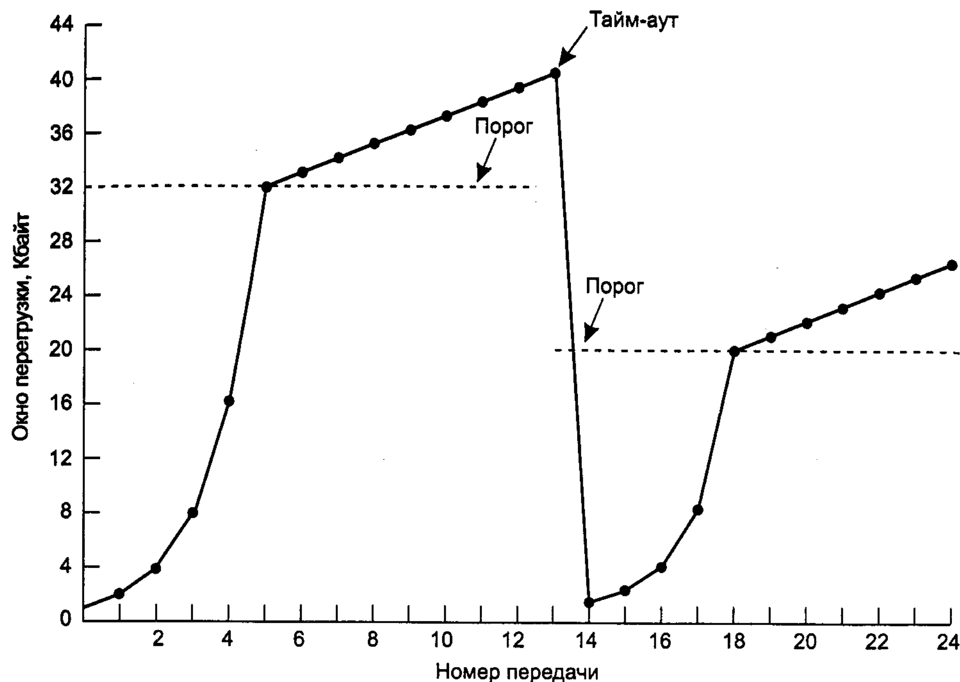


Рис. 6.30. Пример алгоритма борьбы с перегрузкой, применяемого в Интернете

Управление таймерами в TCP

В протоколе TCP используется много различных таймеров (по крайней мере, такова концепция). Наиболее важным из них является **таймер повторной передачи**. Когда посылается сегмент, запускается таймер повторной передачи. Если подтверждение получения сегмента прибывает раньше, чем истекает период таймера, таймер останавливается. Если же, наоборот, период ожидания истечет раньше, чем прибудет подтверждение, сегмент передается еще раз (а таймер запускается снова). Соответственно возникает вопрос: каким должен быть интервал времени ожидания?

На транспортном уровне Интернета эта проблема оказывается значительно сложнее, чем на уровне передачи данных, описанном в главе 3. На уровне передачи данных величину ожидаемой задержки довольно легко предсказать (ее разброс невелик), поэтому таймер можно установить на момент времени чуть позднее ожидаемого прибытия подтверждения (рис. 6.31, *а*). Поскольку на уровне передачи данных подтверждения редко запаздывают на больший срок (благодаря тому, что нет заторов), отсутствие подтверждения в течение установленного временного интервала с большой вероятностью означает потерю кадра или подтверждения.

Протокол TCP вынужден работать в совершенно иных условиях. Функция распределения плотности вероятности времени прибытия подтверждения на этом

уровне выглядит значительно более полого (рис. 6.31, *б*). Поэтому предсказать, сколько потребуется времени для прохождения данных от отправителя до получателя и обратно, весьма непросто. Если выбрать значение интервала ожидания слишком малым (например, T_1 на рис. 6.31, *б*), возникнут излишние повторные передачи, забивающие Интернет бесполезными пакетами. Если же установить значение этого интервала слишком большим (T_2), то из-за увеличения времени ожидания в случае потери пакета пострадает производительность. Более того, среднее значение и величина дисперсии времени прибытия подтверждений может изменяться всего за несколько секунд при возникновении и устранении перегрузки.

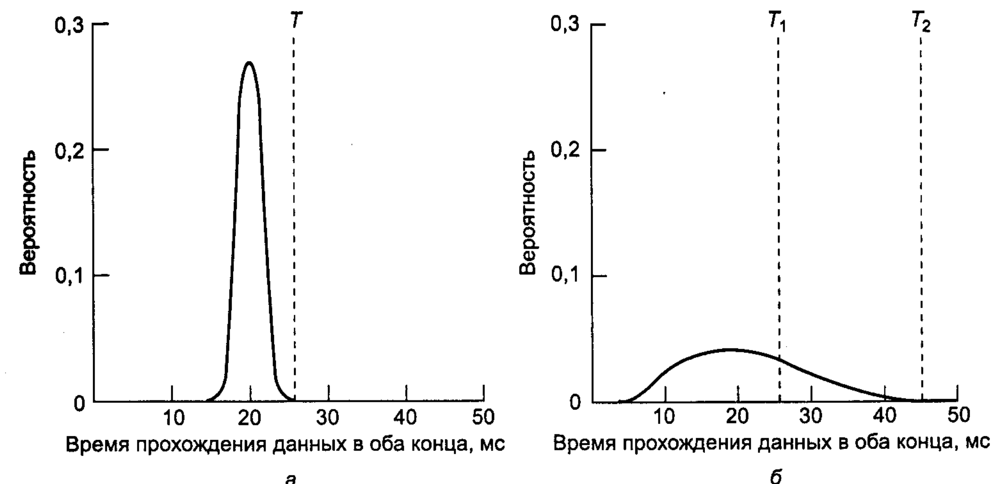


Рис. 6.31. Плотность вероятности времени прибытия подтверждения на уровне передачи данных (*а*); плотность вероятности времени прибытия подтверждения на транспортном уровне (*б*)

Решение заключается в использовании крайне динамичного алгоритма, постоянно изменяющего величину периода ожидания, основываясь на измерениях производительности сети. Алгоритм, применяемый в TCP, разработан Джекобсоном (Jacobson) в 1988 году и работает следующим образом. Для каждого соединения в протоколе TCP предусмотрена переменная *RTT* (Round-Trip Time — время перемещения в оба конца), в которой хранится наименьшее время получения подтверждения для данного соединения. При передаче сегмента запускается таймер, который измеряет время, требуемое для получения подтверждения, а также запускает повторную передачу, если подтверждение не приходит в срок. Если подтверждение успевает вернуться прежде чем истечет период ожидания, TCP-сущность измеряет время, потребовавшееся для его получения (*M*). Затем значение переменной *RTT* обновляется по следующей формуле:

$$RTT = \alpha RTT + (1 - \alpha)M,$$

где α — весовой коэффициент, обычно равный 7/8.

Даже при известном значении переменной *RTT* выбор периода ожидания подтверждения оказывается задачей нетривиальной. Обычно в протоколе TCP это значение вычисляется как βRTT. Остается только выбрать каким-нибудь хитрым образом соответствующее значение коэффициента β. В ранних реализациях протокола использовалось $\beta = 2$, однако экспериментально было показано, что постоянное значение β является негибким и плохо учитывает ситуации, при которых разброс значений времени прибытия подтверждения увеличивается.

В 1988 г. Джекобсон предложил использовать значение β, грубо пропорциональное среднеквадратичному отклонению (дисперсии) функции плотности вероятности времени прибытия подтверждения. Таким образом, при увеличении разброса значений времени прибытия увеличивалось бы и значение β, и наоборот. В частности, он предложил использовать *среднее линейное отклонение* в качестве легко вычисляемой оценки *среднеквадратичного отклонения*. В его версии алгоритма для каждого соединения вводилась еще одна сглаженная переменная *D*, отклонение. При получении каждого подтверждения вычислялась абсолютная величина разности между ожидаемым и измеренным значениями $|RTT - M|$. Сглаженное значение этой величины сохранялось в переменной *D*, вычисляемой по формуле

$$D = \alpha D + (1 - \alpha)\,|RTT - M|,$$

где β в общем случае могло выбираться отличным от значения, используемого в предыдущей формуле. Хотя значение переменной *D* и отличается от среднеквадратичного отклонения, оно достаточно хорошо подходит для данного алгоритма. Кроме того, Джекобсон показал, как можно вычислить значение этой переменной, используя только целочисленные сложение, вычитание и сдвиг, что явилось большим плюсом. В настоящее время этот алгоритм применяется в большинстве реализаций протокола TCP, а значение интервала ожидания устанавливается по формуле

$$\text{Время ожидания} = RTT + 4D.$$

Выбор множителя 4 является произвольным, однако это значение обладает двумя преимуществами. Во-первых, умножение целого числа на 4 может быть выполнено одной командой сдвига. Во-вторых, относительное количество излишних повторных передач при таком значении времени ожидания не превысит одного процента. (Вначале Джекобсон предлагал умножать *D* на 2, но последующие исследования показали, что 4 дает лучшую производительность.)

При динамической оценке величины *RTT* возникает вопрос, что делать со значением *RTT* при повторной передаче сегмента. Когда, наконец, прибывает подтверждение для такого сегмента, непонятно, относится это подтверждение к первой передаче пакета или же к последней. Неверная догадка может серьезно снизить точность оценки *RTT*. Эта проблема была обнаружена радиолюбителем Филом Карном (Phil Karn). Его интересовал вопрос передачи TCP/IP-пакетов с помощью коротковолновой любительской радиосвязи, известной своей ненадежностью (в лучшем случае до адресата доходит лишь половина пакетов). Предложение Ф. Карна было очень простым: не обновлять значение *RTT* для переданных повторно пакетов. Вместо этого при каждой повторной передаче время ожидания можно удваивать до тех пор, пока сегменты не пройдут с первой попытки. Это исправление получило название **алгоритма Карна**. Он применяется в большинстве реализаций протокола TCP.

В протоколе TCP используется не только таймер повторной передачи. Еще один применяемый в этом протоколе таймер называется **таймером настойчивости**. Он предназначен для предотвращения следующей тупиковой ситуации. Получатель посылает подтверждение, в котором указывает окно нулевого размера, давая тем самым отправителю команду подождать. Через некоторое время получатель посылает пакет с новым размером окна, но этот пакет теряется. Теперь обе стороны ожидают действий противоположной стороны. Когда срабатывает таймер настойчивости, отправитель посылает получателю пакет с вопросом, не изменилось ли текущее состояние. В ответ получатель сообщает текущий размер окна. Если он все еще равен нулю, таймер настойчивости запускается снова, и весь цикл повторяется. Если же окно увеличилось, отправитель может передавать данные.

В некоторых реализациях протокола используется третий таймер, называемый **дежурным таймером**. Когда соединение не используется в течение долгого времени, срабатывает дежурный таймер, заставляя одну сторону проверить, есть ли еще кто живой на том конце соединения. Если проверяющая сторона не получает ответа, соединение разрывается. Эта особенность протокола довольно противоречива, поскольку она приносит дополнительные накладные расходы и может разорвать вполне жизнеспособное соединение из-за кратковременной потери связи.

Последний таймер, используемый в каждом TCP-соединении, — это таймер, запускаемый в состоянии *TIMED WAIT* конечного автомата при закрытии соединения. Он отсчитывает двойное время жизни пакета, чтобы гарантировать, что после закрытия соединения в сети не останутся созданные им пакеты.

Беспроводные протоколы TCP и UDP

Теоретически, транспортные протоколы не должны зависеть от технологии, применяемой на расположенном ниже сетевом уровне. В частности, протоколу TCP должно быть все равно, передаются его сегменты по радио или по оптоволоконному кабелю. На практике, тем не менее, это имеет значение, так как основная часть реализаций протокола TCP подверглась детальной оптимизации, основанной на предположениях, справедливых для проводных сетей, но неверных для беспроводных. Игнорирование свойств беспроводной связи может привести к появлению реализации протокола TCP, которая будет логически корректной, но в то же время будет характеризоваться ужасающе низкой производительностью.

Основным вопросом является алгоритм борьбы с перегрузкой. Почти все нынешние реализации протокола TCP предполагают, что тайм-ауты вызываются заторами, а не потерей пакетов. Соответственно, когда время ожидания истекает, протокол TCP снижает скорость передачи (например, по алгоритму затяжного

пуска Джекобсона), чтобы ослабить нагрузку на сеть и тем самым способствовать устранению затора.

К сожалению, беспроводные линии передачи являются чрезвычайно ненадежными. Они постоянно теряют пакеты. Правильная реакция на потерю пакета должна состоять в его скорейшей повторной отправке. Снижение скорости может лишь ухудшить ситуацию. Если, к примеру, 20 % всех пакетов теряется, а отправитель передает по 100 пакетов в секунду, то до получателя доходит около 80 пакетов в секунду. Если отправитель снизит скорость до 50 пакетов в секунду, на выходе скорость упадет до 40 пакетов в секунду.

Таким образом, если пакет теряется в проводной сети, отправитель должен снизить скорость. Если же пакет теряется в беспроводной сети, отправитель должен не снижать скорость, а, возможно, даже увеличить ее (это напоминает разницу в способах вывода из заноса переднеприводных и заднеприводных автомобилей. — *Примеч. перев.*). Если отправитель не знает, в какой сети он находится, ему трудно принять верное решение.

Часто путь от отправителя до получателя оказывается неоднородным. Первые 1000 км могут проходить по проводной сети, но последний километр может оказаться беспроводным. В такой ситуации принять правильное решение в случае тайм-аута еще сложнее, так как его причины могут быть различными. Предложенное решение, получившее название **непрямого протокола TCP** (Bakne и Badrinath, 1995), состояло в разбиении TCP-соединения на два отдельных соединения, как показано на рис. 6.32. Первое соединение тянется от отправителя до базовой станции, а второе — от базовой станции до получателя. Базовая станция просто копирует пакеты из одного соединения в другое в обоих направлениях.

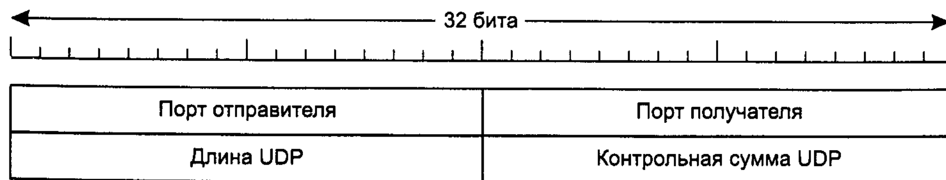


Рис. 6.32. Разбиение TCP-соединения на два

Преимущество этой схемы состоит в том, что два соединения, на которые разбивается TCP-соединение, оказываются однородными. В ответ на тайм-ауты в первом соединении отправитель может снизить скорость, а на тайм-ауты во втором — увеличить. Другие параметры также могут настраиваться независимо для каждого соединения. Недостаток заключается в том, что такое решение нарушает семантику протокола TCP. Поскольку каждая часть соединения представляет собой полноценное TCP-соединение, базовая станция сама подтверждает получение каждого сегмента обычным способом. Только теперь получение подтверждения отправителем означает не то, что сегмент успешно добрался до получателя, а только то, что его получила базовая станция.

Другое решение, разработанное Балакришнаном (Balakrishnan и др., 1995), не нарушает семантики протокола TCP. В его основе лежат небольшие изменения

в программе сетевого уровня, работающей на базовой станции. Одно из изменений состоит в добавлении специального следящего агента, просматривающего и кэширующего сегменты, направляемые мобильному хосту, и подтверждений, посылаемых им в ответ. Если следящий агент замечает, что в ответ на TCP-сегмент, пересылаемый им мобильному хосту, не поступает подтверждения, он просто передает этот сегмент еще раз, не информируя об этом отправителя. У следящего агента есть свой таймер для отслеживания подтверждений, устанавливаемый на относительно небольшой интервал времени. Он также повторно передает сегменты, когда получает от мобильного хоста дубликаты подтверждений, означающие, что хосту чего-то не хватает для счастья. Дубликаты подтверждений уничтожаются на месте, чтобы отправитель на другом конце кабельной сети не принял их за сигнал о перегрузке.

Недостаток такой прозрачности состоит в том, что при частых потерях сегментов на беспроводном участке у отправителя может истечь интервал ожидания, и он может запустить механизм борьбы с перегрузкой. В предыдущем варианте составного TCP-соединения, напротив, алгоритм борьбы с перегрузкой никогда бы не запустился, если только в сети действительно не образовался бы затор.

Метод Балакришнана также решает проблему потерянных сегментов, отправляемых мобильным хостом. Если базовая станция замечает разрыв в порядковых номерах получаемых сегментов, она просит повторить недостающие байты. Благодаря этим двум исправлениям участок беспроводной связи стал более надежным в обоих направлениях, причем для этого не потребовалось изменять семантику протокола TCP и даже информировать о возникающих проблемах отправителя.

Хотя протокол UDP и не страдает от тех же самых проблем, что и протокол TCP, беспроводная связь также представляет для него некоторые трудности. Основная сложность состоит в том, что программы, использующие протокол UDP, ожидают от него высокой надежности. Они знают, что гарантии не предоставляются, но, тем не менее, надеются, что протокол UDP почти совершенен. В условиях беспроводной связи до совершенства будет очень далеко. Программы, способные справляться с потерей UDP-сообщений (но довольно значимой ценой), попадая из окружения, в котором сообщения теоретически могут теряться, но теряются очень редко, в окружение, в котором они теряются постоянно, могут очень сильно потерять в производительности.

Беспроводная связь затрагивает также аспекты, отличные от производительности. Например, как мобильному хосту найти локальный принтер, чтобы не связываться со своим домашним принтером? В чем-то близкой является проблема получения веб-страницы для локальной соты, даже если ее имя неизвестно. Кроме того, веб-дизайнеры обычно рассчитывают на большую пропускную способность сети. Они размещают на каждой странице гигантский логотип, для передачи которого по беспроводному каналу потребуется 10 с при каждом обращении к этой странице, что чрезвычайно раздражает пользователей.

Беспроводные сети распространяются все шире, поэтому проблемы работы в них протокола TCP становятся все более острыми. Дополнительную информацию, касающуюся данных вопросов, можно найти в (Barakat и др., 2000; Ghani и Dixit, 1999; Huston, 2001; Xylomenos и др., 2001).

Транзакционный TCP

Мы уже рассматривали в этой главе, как осуществляется удаленный вызов процедуры в клиент-серверных системах. Если запрос и ответ достаточно малы, чтобы поместиться в один пакет, а операция идемпотентна, то можно смело использовать UDP. Однако если эти условия не выполняются, применение протокола UDP оказывается менее привлекательным. Например, если ответ может быть довольно объемным, то нужен механизм для последовательного выстраивания частей сообщения и повторной передачи потерянных пакетов. Получается, что речь идет о том, что приложению следует заново изобрести TCP.

Это, понятное дело, не очень привлекательная перспектива. Однако и TCP сам по себе в данном случае не кажется слишком привлекательным. Проблема заключается, прежде всего, в эффективности. На рис. 6.33, *а* показана обычная последовательность пакетов, необходимая для удаленного вызова процедуры. В лучшем случае потребуется обменяться девятью пакетами. Вот они:

1. Клиент посылает пакет *SYN* для установки соединения.
2. Сервер посылает пакет *ACK* для подтверждения приема *SYN*.
3. Клиент выполняет «тройное рукопожатие».
4. Клиент посылает, собственно, свой *запрос*.
5. Клиент посылает пакет *FIN*, сигнализирующий об окончании передачи.
6. Сервер подтверждает запрос и *FIN*.
7. Сервер посылает *ответ* клиенту.
8. Сервер посылает пакет *FIN*, сообщающий о том, что он также закончил передачу.
9. Клиент подтверждает *FIN* сервера.

И это в лучшем случае! Если же обстоятельства складываются не очень удачно, то запрос и *FIN* клиента подтверждаются раздельно. Раздельно могут подтверждаться и ответ и *FIN* сервера.

Само собой, возникает вопрос: нельзя ли объединить эффективность выполнения RPC с помощью UDP (всего два сообщения) с надежностью, которую гарантирует TCP? Ответ: отчасти. Можно попытаться применить экспериментальный вариант протокола TCP, называемый **транзакционным TCP (T/TCP**, Transactional TCP). Протокол T/TCP описан в RFC 1379 и 1644.

Центральная идея протокола состоит в том, чтобы немного изменить стандартную последовательность действий, выполняемых при установке соединения, и предоставить возможность передачи данных непосредственно во время установки соединения. Работа T/TCP показана на рис. 6.33, *б*. В первом пакете клиента содержится бит *SYN*, собственно запрос и *FIN*. В сущности, клиент говорит: «Я хочу установить соединение, вот данные для передачи, и на этом я считаю свою миссию выполненной».

Получив запрос, сервер ищет или вычисляет содержимое ответа и выбирает способ ответа. Если ответ укладывается в один пакет, он будет послан так, как показано на рис. 6.33, *б*, то есть сервер как бы говорит: «Подтверждаю получение

вашего *FIN*, вот мой ответ на запрос, и на этом я считаю свою миссию выполненной». Клиент подтверждает *FIN* сервера, и на этом работа протокола заканчивается. Обратите внимание: на весь процесс потребовалось всего три сообщения.

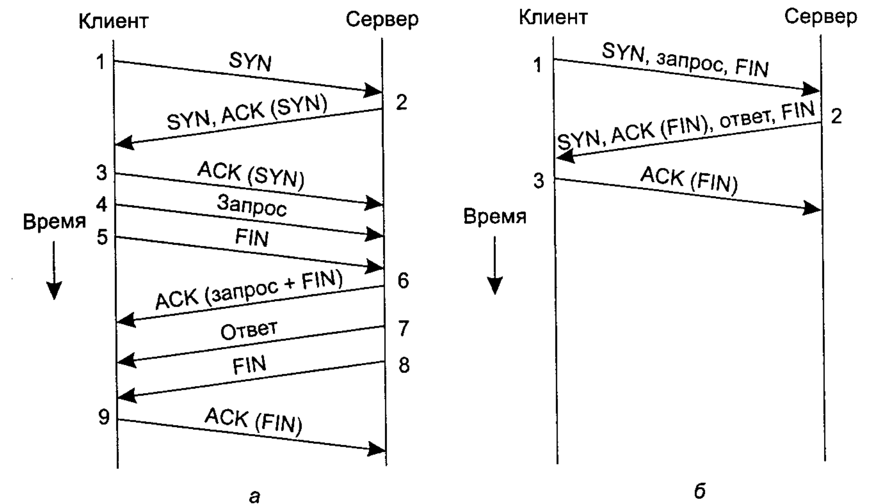


Рис. 6.33. Удаленный вызов процедуры с помощью обычного TCP (*а*); удаленный вызов процедуры с помощью T/TCP (*б*)

Однако если ответ не укладывается в один пакет, сервер может и не устанавливать бит *FIN* в первом же пакете. Он посылает столько пакетов, сколько нужно, прежде чем закрыть соединение в данном направлении.

Стоит отметить, что кроме T/TCP известны и другие вариации на тему TCP. Одним из таких протоколов является **протокол передачи с контролем потока (SCTP**, Stream Control Transmission Protocol). Среди его свойств можно выделить сохранение границ сообщений, несколько режимов доставки (например, неупорядоченная), множественную адресацию (дублирование адресатов), селективные подтверждения (Stewart и Metz, 2001). Тем не менее, когда кто-то предлагает улучшения того, что и так прекрасно работает в течение долгих лет, обычно возникают споры между сторонниками двух диаметрально противоположных принципов: «Все для расширения функциональности» и «Не трогать то, что пока еще не сломалось».

Вопросы производительности

Вопросы производительности играют важную роль в компьютерных сетях. Когда сотни или тысячи компьютеров соединены вместе, их взаимодействие становится очень сложным и может привести к непредсказуемым последствиям. Часто эта сложность приводит к низкой производительности, причины которой довольно трудно определить. В следующих разделах мы рассмотрим многие вопросы, свя-

занные с производительностью сетей, определим круг существующих проблем и обсудим методы их разрешения.

К сожалению, понимание производительности сетей — это скорее искусство, чем наука. Теоретическая база, допускающая хоть какое-то практическое применение, крайне скудна. Лучшее, что мы можем сделать, — это представить несколько практических методов, полученных в результате долгих экспериментов, а также привести несколько реально действующих примеров. Мы намеренно отложили эту дискуссию до того момента, когда будет изучен транспортный уровень в сетях TCP, чтобы иметь возможность иллюстрировать некоторые места примерами из TCP.

Вопросы производительности возникают не только на транспортном уровне. С некоторыми из них мы уже сталкивались в предыдущей главе. Тем не менее, сетевой уровень в основном занят вопросами маршрутизации и борьбы с перегрузкой. Более глобальные вопросы, касающиеся производительности всей системы в целом, оказываются прерогативой транспортного уровня, поэтому они будут рассматриваться именно в этой главе.

В следующих разделах мы рассмотрим следующие пять аспектов производительности сети.

1. Причины снижения производительности.
2. Измерение производительности сети.
3. Проектирование производительных систем.
4. Быстрая обработка TPDU-модулей.
5. Протоколы для будущих высокопроизводительных сетей.

Нам потребуется ввести название для единиц данных, которыми обмениваются транспортные сущности. Термин «сегмент», применяемый в протоколе TCP, в лучшем случае запутывает и никогда не используется в этом смысле за пределами мира TCP. Соответствующие ATM-термины — CS-PDU (протокольная единица обмена подуровня конвергенции), SAR-PDU и CPCS-PDU — применяются только в сетях ATM. Термин «пакет» относится к сетевому уровню, а сообщения применяются на прикладном уровне. За отсутствием стандартного термина мы будем продолжать называть единицы данных, которыми обмениваются транспортные сущности, TPDU-модулями. Когда будет иметься в виду TPDU-модуль вместе с пакетом, мы будем использовать в качестве обобщающего термина понятие «пакет», например: «Центральный процессор должен быть достаточно быстрым, чтобы успевать обрабатывать приходящие пакеты в режиме реального времени». При этом будет иметься в виду как пакет сетевого уровня, так и помещенный в него TPDU-модуль.

Причины снижения производительности компьютерных сетей

Некоторые виды снижения производительности вызваны временным отсутствием свободных ресурсов. Если на маршрутизатор вдруг прибывает трафик больше, чем он способен обработать, образуется затор, и производительность резко падает. Вопросы перегрузки подробно рассматривались в предыдущей главе.

Производительность также снижается, если возникает структурный дисбаланс ресурсов. Например, если гигабитная линия связи присоединена к компьютеру с низкой производительностью, то несчастный центральный процессор не сможет достаточно быстро обрабатывать приходящие пакеты, что приведет к потере некоторых пакетов. Эти пакеты рано или поздно будут переданы повторно, что приведет к увеличению задержек, непроизводительному использованию пропускной способности и снижению общей производительности.

Перегрузка может также возникать синхронно. Например, если TPDU-модуль содержит неверный параметр (например, номер порта назначения), во многих случаях получатель заботливо пошлет обратно сообщение об ошибке. Теперь рассмотрим, что случится, если неверный TPDU-модуль будет разослан широковещательным способом 10 000 машин. Каждая машина может послать обратно сообщение об ошибке. Образовавшийся в результате **широковещательный шторм** может надолго остановить нормальную работу сети. Протокол UDP страдал от подобной проблемы, пока не было решено, что хосты должны воздерживаться от отправки сообщений об ошибке в ответ на широковещательные TPDU-модули UDP.

Второй пример синхронной перегрузки может быть вызван временным отключением электроэнергии. Когда питание снова включается, все машины одновременно обращаются к своей постоянной памяти и начинают перезагружаться. Типичная последовательность загрузки может требовать обращения к какому-нибудь DHCP-серверу (сервер динамической конфигурации хоста), чтобы узнать свой истинный адрес, а затем к файловому серверу, чтобы получить копию операционной системы. Если сотни машин обратятся к серверу одновременно, он не сможет обслужить сразу всех.

Даже при отсутствии синхронной перегрузки и при достаточном количестве ресурсов производительность может снижаться из-за неверных системных настроек. Например, у машины может быть мощный процессор и много памяти, но недостаточно памяти выделено под буфер. В этом случае буфер будет переполняться, и часть TPDU-модулей потеряется. Аналогично, если процессу обработки поступающих пакетов дан недостаточно высокий приоритет, некоторые TPDU-модули могут быть потеряны.

Также на производительность могут повлиять неверно установленные значения таймеров ожидания. Когда посылается TPDU-модуль, обычно включается таймер — на случай, если этот модуль потеряется. Выбор слишком короткого интервала ожидания подтверждения приведет к излишним повторным передачам TPDU-модулей. Если же интервал ожидания сделать слишком большим, это приведет к увеличению задержки в случае потери TPDU-модуля. К настраиваемым параметрам также относятся интервал ожидания попутного модуля данных для отправки подтверждения и количество попыток повторной передачи в случае отсутствия подтверждений.

С появлением гигабитных сетей появились и новые проблемы. Рассмотрим, например, процесс отправки 64 Кбайт данных из Сан-Диего в Бостон при раз-

мере буфера получателя 64 Кбайт. Пусть скорость передачи данных в линии составляет 1 Гбит/с, а время прохождения сигнала в одну сторону, ограниченное скоростью света в стекле, равно 20 мс. Вначале ($t = 0$), как показано на рис. 6.34, *а*, канал пуст. Только 500 мкс спустя все TPDU-модули попадут в канал (рис. 6.34, *б*). Первый TPDU-модуль оказывается где-то в окрестностях Броли, все еще в Южной Калифорнии. Тем не менее, передатчик уже должен остановиться, пока он не получит в ответ новую информацию об окне.

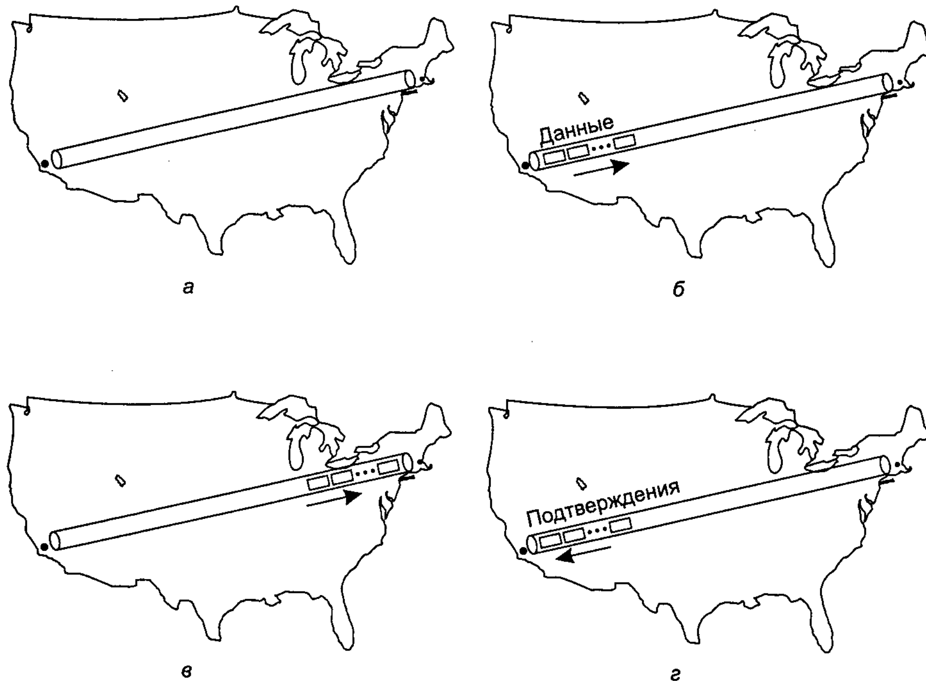


Рис. 6.34. Передача половины мегабита из Сан-Диего в Бостон: в момент времени $t = 0$ (*а*); через 500 мкс (*б*); через 20 мс (*в*); через 40 мс (*г*)

Через 20 мс первый TPDU-модуль, как показано на рис. 6.34, *в*, достигнет Бостона, и в ответ будет передано подтверждение. Наконец, через 40 мс после начала операции первое подтверждение возвращается к отправителю, после чего передается следующая порция данных. Поскольку линия передачи использовалась всего 0,5 мс из 40 мс, эффективность ее использования составит около 1,25 %. Такая ситуация является типичной для работы старых протоколов по гигабитным линиям.

При анализе производительности сетей полезно обращать внимание на **произведение пропускной способности и времени задержки**. Пропускная способность канала (в битах в секунду) умножается на время прохождения сигнала в оба конца (в секундах). В результате получается емкость канала в битах.

В примере на рис. 6.34 произведение пропускной способности и времени задержки равно 40 млн бит. Другими словами, отправитель к моменту получения ответа успеет переслать 40 млн бит, если будет передавать с максимальной скоростью. Столько бит потребуется, чтобы заполнить канал в обоих направлениях. Таким образом, порция данных в полмиллиона бит составляет всего 1,25 % емкости канала, что и выражается в 1,25-процентной эффективности использования канала.

Отсюда следует, что для эффективного использования канала размер окна получателя должен быть, по меньшей мере, равен произведению пропускной способности и времени задержки, а лучше — превосходить его, так как получатель может сразу и не ответить. Для трансконтинентальной гигабитной линии каждому соединению потребуется, по меньшей мере, по 5 Мбайт.

Если даже при пересылке одного мегабита эффективность использования канала оказывается столь ужасной, представьте, что происходит в случае передачи нескольких сот байт при вызове удаленной процедуры. Если не найти этой линии еще какого-либо применения, пока клиент ожидает ответа, гигабитная линия будет ничем не лучше мегабитной, только значительно более дорогой.

Еще одна проблема производительности связана с приложениями типа видео и аудио, для которых временные параметры являются критическими. Обеспечить малое среднее время передачи здесь недостаточно. Требуется также обеспечить небольшое значение его среднеквадратичного отклонения. Для достижения обеих целей требуется немало инженерных усилий.

Измерение производительности сети

Когда качество работы сети оказывается не слишком хорошим, ее пользователи часто жалуются сетевым операторам, требуя усовершенствований. Чтобы улучшить производительность сети, операторы должны сначала точно определить, в чем суть проблемы. Чтобы выяснить текущее состояние сети, операторы должны произвести измерения. В данном разделе мы рассмотрим вопрос измерения производительности сети. Приводимое ниже обсуждение основано на работе Могола (Mogul, 1993).

Основной цикл работ по совершенствованию производительности сети включает следующие этапы.

1. Измерение наиболее важных сетевых параметров и производительности сети.
2. Попытка понять, что происходит.
3. Изменение одного из параметров.

Эти шаги повторяются до тех пор, пока производительность не увеличится достаточно или пока не станет ясно, что этими методами производительность уже больше не увеличить.

Измерения могут быть произведены разными способами и во многих местах (как физически, так и в стеке протоколов). Наиболее распространенный тип измерений представляет собой включение таймера при начале какой-либо активности и измерение продолжительности этой активности. Например, одним из ключевых измерений является измерение времени, необходимого для получения отправителем подтверждения в ответ на отправку TPDU-модуля. Другие изме-

рения производятся при помощи счетчиков, в которых учитывается частота некоторых событий (например, количество потерянных TPDU-модулей). Наконец, часто измеряются такие количественные показатели как число байтов, обработанных за определенный временной интервал.

Процедура измерения производительности сети и других параметров содержит множество подводных камней. Далее мы перечислим некоторые из них. Следует тщательно избегать подобных ошибок при любых попытках измерить производительность сети.

Убедитесь, что выборка достаточно велика

Не следует ограничиваться единственным измерением какого-нибудь параметра — например, времени, необходимого для передачи одного TPDU-модуля. Повторите измерение, скажем, миллион раз и вычислите среднее значение. Чем больше будет выборка, тем выше окажется точность оценки среднего значения и его среднеквадратичного отклонения. Погрешность может быть вычислена при помощи стандартных формул статистики.

Убедитесь, что выборка является репрезентативной

Следует повторить всю последовательность миллиона измерений параметров в различное время суток и в разные дни недели, чтобы заметить влияние различной загруженности системы на измеряемые параметры. Так, измерение перегрузки вряд ли принесет пользу, если эти измерения производить, когда перегрузки нет. Иногда результаты могут показаться на первый взгляд странными, как, например, наличие серьезных заторов в сети в 10, 11, 13 и 14 часов, но их отсутствие в полдень (когда все пользователи обедают).

Используя часы с грубой шкалой, будьте внимательны

Компьютерные часы работают, добавляя единицу к некоему счетчику через равные интервалы времени. Например, миллисекундный таймер увеличивает на единицу значение счетчика раз в 1 мс. Применение такого таймера для измерения длительности события, занимающего менее 1 мс, возможно, но требует осторожности.

Например, чтобы измерить время, необходимое для передачи одного TPDU-модуля, следует считывать показания системных часов (скажем, в миллисекундах) при входе в программу транспортного уровня и выходе из нее. Если время, требуемое для передачи одного TPDU-модуля, равно 300 мкс, то измеряемая величина будет равна либо 0, либо 1 мс. Однако если повторить измерения миллион раз, сложить все значения и разделить на миллион, то полученное среднее значение будет отличаться от истинного значения менее чем на 1 мкс.

Убедитесь, что во время ваших тестов не происходит ничего неожиданного

Если проводить измерения в университетской системе в день сдачи главного лабораторного проекта, то полученные результаты могут сильно отличаться от результатов измерений, произведенных на следующий день. Аналогично, если в то

время, когда вы производите тестирование сети, какой-нибудь исследователь решит устроить в сети видеоконференцию, вы также можете получить искаженные результаты. Лучше всего запускать тесты на пустой системе, создавая всю нагрузку самому. Хотя и в этом случае можно ошибиться. Вы можете предполагать, что никто не пользуется сетью в 3 часа ночи, но может оказаться, что именно в это время программа автоматического резервного копирования начинает свою работу по архивации всех жестких дисков на магнитную ленту. Кроме того, именно в это время пользователи, находящиеся в другой временной зоне на другой стороне Земного шара, могут создавать довольно сильный трафик, просматривая ваши замечательные веб-страницы.

Кэширование может сильно исказить ваши измерения

Чтобы измерить время операции по передаче файла, самый очевидный путь состоит в том, чтобы открыть большой файл, прочитать его целиком и закрыть, измерив при этом время, затраченное на всю последовательность операций. Затем можно повторить измерение много раз, чтобы получить точное значение средней величины. Беда заключается в том, что система может, считав файл по сети всего один раз, запомнить его в локальном кэше. Таким образом, правильным будет только первое измерение, а при остальных операциях обращения к сети не будет. Результат такого многократного измерения будет бесполезен (если только вы не измеряете производительность кэша).

Часто можно обмануть алгоритм кэширования, просто заставляя переполняться кэш. Например, если размер кэша составляет 10 Мбайт, в тестовый цикл можно включить поочередное открытие, чтение и закрытие двух 10-мегабайтных файлов, пытаясь заставить систему каждый раз читать файлы по сети. Тем не менее, следует быть уверенным в том, что вы понимаете, как работает алгоритм кэширования.

Буферизация пакетов может производить аналогичный эффект. Одна популярная TCP/IP-программа измерения производительности славилась тем, что сообщала, что протокол UDP может достичь производительности, значительно превышающей максимально допустимую для данной физической линии. Как это происходило? Обращение к UDP обычно возвращает управление сразу, как только сообщение принимается ядром системы и добавляется в очередь на передачу. При достаточном размере буфера время выполнения 1000 обращений к UDP не означает, что за это время все данные были переданы в линию. Большая часть их все еще находится в буфере ядра.

Следует хорошо понимать, что измеряется

Когда вы измеряете время чтения удаленного файла, результаты ваших измерений зависят от сети, операционной системы клиента и сервера, аппаратного интерфейса сетевых карт, их драйверов и других факторов. Если все делать внимательно, то в конечном итоге вы получите значение времени передачи файла для данной конфигурации. Если вы ставите перед собой цель настроить именно эту конкретную конфигурацию, то беспокоиться не о чем.

Однако если вы проводите сходные измерения на трех различных системах, чтобы выбрать, какую сетевую карту купить, то ваши результаты могут оказаться никуда не годными из-за того, что какой-нибудь сетевой драйвер окажется неудачным и использующим лишь 10 % производительности сетевой карты.

Будьте осторожны с экстраполяцией результатов

Предположим, вы что-нибудь измеряете (например, время ответа удаленного хоста), моделируя нагрузку сети от 0 (простой) до 0,4 (40 % мощности), как показано жирной линией на рис. 6.35. Может оказаться соблазнительным линейно экстраполировать полученную кривую (пунктир). Однако в действительности многие параметры в теории массового обслуживания содержат в качестве сомножителя $1/(1 - \rho)$, где ρ — нагрузка, поэтому истинная кривая зависимости будет больше походить на гиперболу, показанную штриховой линией.

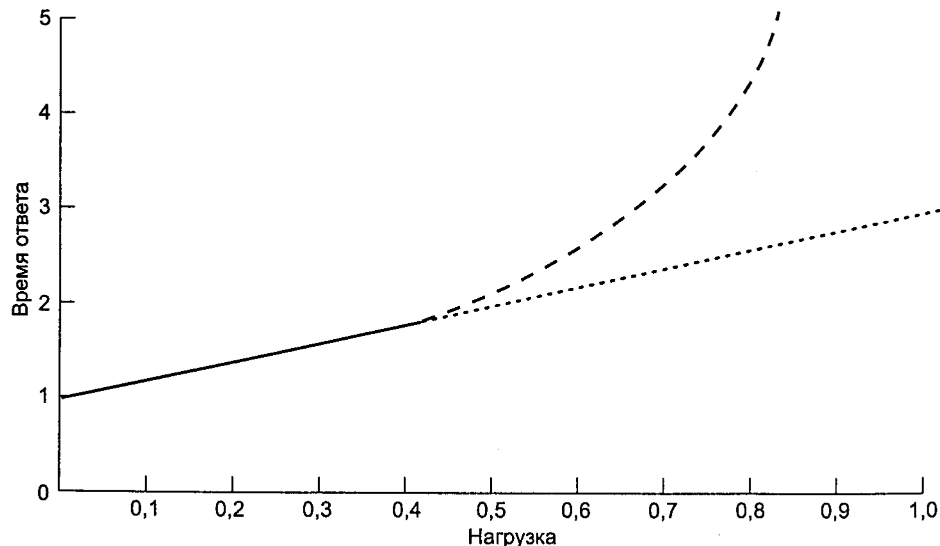


Рис. 6.35. Зависимость времени ответа от нагрузки

Проектирование производительных систем

Измерения и настройки часто позволяют значительно улучшить производительность сети, однако они никогда не заменят хорошо разработанного проекта. Плохо спроектированная сеть может быть усовершенствована только до определенного уровня. Для дальнейшего увеличения ее производительности ее потребуется переделать с нуля.

В данном разделе мы рассмотрим несколько эмпирических правил, основанных на опыте работы со многими сетями. Эти правила касаются не только устройства сети, но и системных аспектов, так как программное обеспечение и операционные системы часто оказываются важнее, чем маршрутизаторы и интерфейсные

карты. Большинство этих идей известны разработчикам сетей уже много лет и передаются из уст в уста от поколения к поколению. Впервые они были открыто записаны Моголом (Mogul, 1993). Наше повествование во многом пересекается с его книгой. Другим источником по этой же теме является (Metcalfe, 1993).

Правило 1: скорость центрального процессора важнее скорости сети

Длительные эксперименты показали, что почти во всех сетях накладные расходы операционной системы и протокола составляют основное время задержки сетевой операции. Например, теоретически минимальное время вызова удаленной процедуры (RPC, Remote Procedure Call) в сети Ethernet составляет 102 мкс, что соответствует минимальному запросу (64 байта), на который приходит минимальный (64-байтовый) ответ. На практике значительным достижением считается хотя бы какое-нибудь снижение накладных расходов, возникающих за счет программного обеспечения при вызове удаленной процедуры.

Аналогично, при работе с гигабитной линией основная задача заключается в достаточно быстрой передаче битов из буфера пользователя в линию, а также в том, чтобы получатель смог обработать их с той скоростью, с которой они приходят. Короче говоря, удвоение производительности процессора нередко может привести к почти удвоению пропускной способности канала. Удвоение же пропускной способности линии часто не дает никакого эффекта, поскольку узким местом обычно являются хосты.

Правило 2: уменьшайте число пакетов, чтобы уменьшить программные накладные расходы

Обработка каждого TPDU-модуля подразумевает определенное количество накладных расходов на каждый модуль (то есть на обработку его заголовка) и определенные затраты при обработке каждого байта (например, при подсчете контрольной суммы). При отправке 1 млн байт побайтовые затраты времени процессора на обработку не зависят от размера TPDU-модуля. Однако при использовании 128-байтовых TPDU-модулей затраты на обработку заголовков будут в 32 раза выше, чем для TPDU-модулей размером 4 Кбайт. И эти затраты увеличиваются очень быстро.

Помимо накладных расходов на обработку заголовков TPDU-модулей, следует рассмотреть накладные расходы нижних уровней. Прибытие каждого пакета вызывает прерывание. В современных RISC-процессорах каждое прерывание нарушает работу процессорного конвейера, снижает эффективность работы кэша, требует изменения контекста управления памятью и сохранения в стеке значительного числа регистров процессора. Таким образом, уменьшение на n числа посылаемых TPDU-модулей дает снижение числа прерываний и накладных расходов в целом в n раз.

Данное наблюдение служит аргументом в пользу сбора значительного количества данных перед их отправкой с целью снизить количество прерываний у по-

лучателя. Алгоритм Нагля и предложенный Кларком метод избавления от синдрома глупого окна действуют именно в этом направлении.

Правило 3: минимизируйте количество переключений контекста

Переключения контекста (например, из режима ядра в режим пользователя) обладают рядом неприятных свойств, в этом они сходны с прерываниями. Самое неприятное — потеря содержимого кэша. Количество переключений контекста может быть снижено при помощи библиотечной процедуры, посылающей данные во внутренний буфер до тех пор, пока их не наберется достаточное количество. Аналогично, на получающей стороне небольшие TPDU-модули следует собирать вместе и передавать пользователю за один раз, минимизируя количество переключений контекста.

В лучшем случае прибывший пакет вызывает одно переключение контекста из текущего пользовательского процесса в режим ядра, а затем еще одно переключение контекста при передаче управления принимающему процессу и предоставлении ему прибывших данных. К сожалению, во многих операционных системах происходит еще одно переключение контекста. Например, если сетевой менеджер работает в виде отдельного процесса в пространстве пользователя, поступление пакета вызывает передачу управления от процесса пользователя ядру, затем от ядра сетевому менеджеру, затем снова ядру и, наконец, от ядра получающему процессу. Эта последовательность переключений контекста показана на рис. 6.36. Все переключения контекста при получении каждого пакета сильно расходуют время центрального процессора и существенно снижают производительность сети.

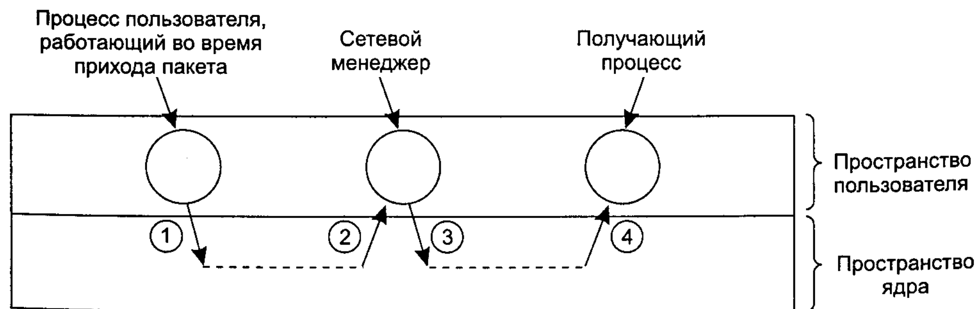


Рис. 6.36. Четыре переключения контекста для обработки одного пакета в сети, в которой сетевой менеджер находится в пространстве пользователя

Правило 4: минимизируйте число операций копирования

Еще больше времени процессора отнимается излишним копированием пакета. Часто полученный пакет копируется три или четыре раза, прежде чем содержащийся в нем TPDU-модуль доставляется по назначению. Сначала пакет принимается сетевым интерфейсом в специальный аппаратный буфер, расположенный на сетевой карте. Из аппаратного буфера пакет копируется в системный буфер ядра, откуда он копируется в буфер сетевого уровня, а затем — в буфер транспортного уровня и, наконец, доставляется получающему приложению.

Грамотно разработанные операционные системы копируют по одному машинному слову за такт процессора, но нередки случаи, когда копирование одного слова требует пять инструкций процессора (считывание, запись, увеличение на единицу индексного регистра, проверка на конец данных и условный переход на начало цикла). Если на одно копирование 32-битного слова требуется пять инструкций процессора, то при трех операциях копирования каждого пакета на каждый скопированный байт приходится 15/4, или почти 4 команды. На машине с производительностью в 500 млн инструкций в секунду (500 MIPS) каждая команда выполняется за 2 нс, следовательно, копирование каждого байта будет производиться процессором в течение 8 нс (около 1 нс на бит). Значит, максимальная скорость ограничивается 1 Гбит/с. С учетом накладных расходов на обработку заголовка и переключения контекстов, возможно, удастся достичь скорости в 500 Мбит/с, а ведь мы еще не учли обработку самих данных. Очевидно, что о поддержке 10-гигабитной линии не может быть и речи.

В действительности поддержка линии со скоростью в 500 Мбит/с также может оказаться абсолютно невозможной. В приведенных расчетах мы предполагали, что машина с производительностью 500 MIPS может выполнить 500 млн любых инструкций в секунду. На самом деле, машина может работать с такой скоростью, только если она не обращается к памяти. Операции с памятью часто оказываются в десятки раз медленнее, чем операции с использованием только внутренних регистров (на выполнение инструкции расходуется около 20 нс). Если 20 % инструкций связано с обращением к памяти (то есть имеются потери кэшируемых данных) — а это вполне вероятная цифра при обработке входящих пакетов, — среднее время выполнения инструкции будет равно 5,6 нс $(0{,}8 \cdot 2 + 0{,}2 \cdot 20)$. Предполагая, что на обработку байта требуются 4 инструкции, нам понадобится 22,4 нс/байт (или около 2,8 нс/бит), что в результате дает суммарную скорость около 357 Мбит/с. Допустим, половина этой производительности уйдет на обработку заголовков, тогда остается 178 Мбит/с. Обратите внимание на то, что аппаратные улучшения здесь не помогут. Проблема состоит в слишком большом числе операций копирования, выполняемых операционной системой.

Правило 5: можно купить более высокую пропускную способность, но не низкую задержку

Следующие три правила относятся не к протоколам, а к линиям связи. Первое правило утверждает, что если вам нужна более высокая пропускная способность, вы можете просто купить ее. Если проложить второй оптоволоконный кабель параллельно первому, пропускная способность удвоится, но время задержки от этого меньше не станет. Чтобы снизить задержку, следует улучшить программное обеспечение протоколов, операционную систему или сетевой интерфейс. И даже это не поможет, если загвоздка состоит во времени передачи по линии.

Правило 6: лучше избегать перегрузки, чем бороться с уже возникшей перегрузкой

Старая пословица, гласящая, что профилактика лучше лечения, справедлива и в деле борьбы с перегрузками в сетях. Когда в сети образуется затор, пакеты теряются, пропускная способность растрачивается впустую, увеличиваются задержки и т. п. Процесс восстановления после перегрузки требует времени и терпения. Гораздо более эффективной стратегией является предотвращение перегрузки, напоминающее прививку от болезни — это несколько неприятно, зато избавляет от возможных больших неприятностей.

Правило 7: избегайте тайм-аутов

Таймеры необходимы в сетях, но их следует применять умеренно, и следует минимизировать количество тайм-аутов. Когда срабатывает таймер, обычно повторяется какое-либо действие. Если повтор этого действия необходим, его следует повторить, однако повторение действия без особой необходимости является расточительным.

Чтобы избегать излишней работы, следует устанавливать период ожидания с небольшим запасом. Таймер, срабатывающий слишком поздно, несколько увеличивает задержку в (маловероятном) случае потери TPDU-модуля. Преждевременно срабатывающий таймер растрачивает попусту время процессора, пропускную способность и напрасно увеличивает нагрузку на, возможно, десятки маршрутизаторов.

Быстрая обработка TPDU-модулей

Мораль приведенной истории состоит в том, что основным препятствием на пути ускорения сетей является программное обеспечение протоколов. В данном разделе мы рассмотрим некоторые способы ускорения этих программ. Дополнительные сведения по этой теме см. в (Clark и др., 1989; Chase и др., 2001).

Накладные расходы по обработке TPDU-модулей состоят из двух компонентов: затрат по обработке заголовка и затрат по обработке каждого байта. Следует вести наступление сразу на обоих направлениях. Ключ к быстрой обработке TPDU-модулей лежит в выделении нормального случая (односторонней передачи данных) и отдельной обработке этого случая. Хотя для перехода в состояние *ESTABLISHED* требуется передача последовательности специальных TPDU-модулей, но как только это состояние достигнуто, обработка TPDU-модулей не вызывает затруднений, пока одна из сторон не начнет закрывать соединение.

Начнем с рассмотрения посылающей стороны, находящейся в состоянии *ESTABLISHED*, когда должны отправляться данные. Для простоты мы предположим, что транспортная сущность расположена в ядре, хотя те же самые идеи применимы и в случае, когда она представляет собой процесс, находящийся в пространстве пользователя, или набор библиотечных процедур в посылающем процессе. На рис. 6.37 отправляющий процесс эмулирует прерывание, выполняя примитив SEND, и передает управление ядру. Прежде всего, транспортная сущность проверяет, находится ли она в нормальном состоянии, то есть таком, когда установлено состояние *ESTABLISHED*, ни одна сторона не пытается закрыть соединение, посылается стандартный полный TPDU-модуль и у получателя достаточный размер окна. Если все эти условия выполнены, то никаких дополнительных проверок не требуется, и алгоритм транспортной сущности может выбрать быстрый путь. В большинстве случаев именно так и происходит.

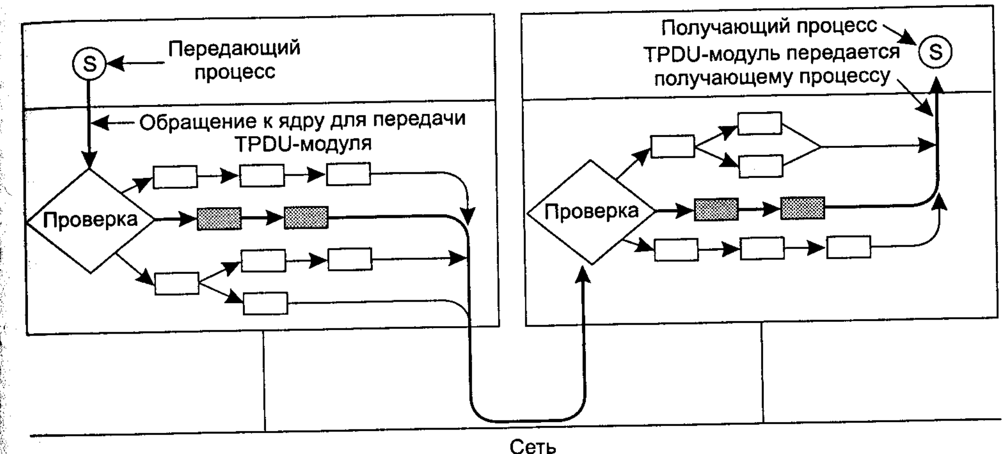


Рис. 6.37. Быстрый путь от отправителя до получателя показан жирной линией. Шаги обработки вдоль этого пути показаны затененными прямоугольниками

В нормальной ситуации заголовки соседних TPDU-модулей почти одинаковы. Чтобы использовать этот факт, транспортная сущность сохраняет в своем буфере прототип заголовка. В начале быстрого пути он как можно быстрее пословно копируется в буфер нового заголовка. Затем поверх перезаписываются все отличающиеся поля. Часто эти поля легко выводятся из переменных состояния — например, следующий порядковый номер TPDU-модуля. Затем указатель на полный TPDU-модуль и указатель на данные пользователя передаются сетевому уровню. Здесь может быть применена та же стратегия (на рис. 6.37 это не показано). Наконец, сетевой уровень передает полученный в результате пакет уровню передачи данных для отправки.

Чтобы увидеть, как работает этот принцип на практике, рассмотрим случай TCP/IP. На рис. 6.38, *а* изображен TCP-заголовок. Поля, одинаковые для заголовков последующих TPDU-модулей в однонаправленном потоке, затенены. Все, что нужно сделать передающей транспортной сущности, это скопировать пять слов заголовка-прототипа в выходной буфер, заполнить поле порядкового номера (скопировав одно слово), сосчитать контрольную сумму и увеличить на единицу значение переменной, хранящей текущий порядковый номер. Затем она может передать заголовок и данные специальной IP-процедуре, предназначенной для отправки стандартного максимального TPDU-модуля. Затем IP-процедура копирует свой заголовок-прототип из пяти слов (см. рис. 6.38, *б*) в буфер, заполняет поле *Идентификатор* и вычисляет контрольную сумму заголовка. Теперь пакет готов к передаче.

Рассмотрим теперь быстрый путь обработки пакета получающей стороной на рис. 6.37. Первый шаг состоит в нахождении для пришедшего TPDU-модуля записи соединения. В протоколе TCP запись соединения может храниться в хэш-таблице, ключом к которой является какая-нибудь простая функция двух IP-адресов и двух портов. После обнаружения записи соединения следует проверить адреса и номера портов, чтобы убедиться, что найдена верная запись.

Порт отправителя		Порт получателя
Порядковый номер		
Номер подтверждения		
Длина TCP-заголовка	Не используется	Размер окна
Контрольная сумма		Указатель на срочные данные

а

Версия	IHL	Тип службы	Полная длина
Идентификатор			Смещение фрагмента
TTL	Протокол		Контрольная сумма заголовка
Адрес отправителя			
Адрес получателя			

б

Рис. 6.38. TCP-заголовок (*а*); IP-заголовок (*б*). В обоих случаях затененные поля взяты у прототипа без изменений

Процесс поиска нужной записи можно дополнительно ускорить, если установить указатель на последнюю использовавшуюся запись и сначала проверить ее. Кларк с соавторами книги, вышедшей в 1989 году, исследовал этот вопрос и пришел к выводу, что в этом случае доля успешных обращений превысит 90 %. Другие эвристические методы поиска описаны в (McKenney и Dove, 1992).

Затем TPDU-модуль проверяется на адекватность: соединение в состоянии *ESTABLISHED*, ни одна сторона не пытается его разорвать, TPDU-модуль является полным, специальные флаги не установлены, и порядковый номер соответствует ожидающемуся. Программа, выполняющая всю эту проверку, состоит из довольно значительного количества инструкций. Если все эти условия удовлетворяются, вызывается специальная процедура быстрого пути TCP-уровня.

Процедура быстрого пути обновляет запись соединения и копирует данные пользователю. Во время копирования она одновременно подсчитывает контрольную сумму, что уменьшает количество циклов обработки данных. Если контрольная сумма верна, запись соединения обновляется и отправляется подтверждение. Метод, реализованный в виде отдельной процедуры, сначала производящей быструю проверку заголовка, чтобы убедиться, что заголовок именно такой, какой ожидается, называется **предсказанием заголовка**. Он применяется в большинстве реализаций протокола TCP. Использование этого метода оптимизации вместе с остальными, описанными в данном разделе, позволяет протоколу TCP достичь 90 % от скорости локального копирования из памяти в память при условии, что сама сеть достаточно быстрая.

Две другие области, в которых возможны основные улучшения производительности, — это управление буферами и управление таймерами. Как уже было сказано, при управлении буферами следует пытаться избегать излишнего копирования. При управлении таймерами следует учитывать, что они почти никогда не срабатывают. Они предназначены для обработки нестандартного случая потерь TPDU-модулей, но большинство TPDU-модулей и их подтверждений прибывают успешно, и поэтому истечение периода ожидания является редким событием.

В программе таймеры обычно реализуются в виде связанного списка таймеров, отсортированного по времени срабатывания. Заглавный элемент списка содержит счетчик, хранящий число минимальных интервалов времени, оставшихся до истечения периода ожидания. В каждом последующем элементе списка содержится счетчик, указывающий, через какой интервал времени относительно предыдущего элемента списка истечет время ожидания данного таймера. Например, если таймеры сработают через 3, 10 и 12 интервалов времени, счетчики списка будут содержать значения 3, 7 и 2 соответственно.

При каждом импульсе сигнала времени часов счетчик головного элемента списка уменьшается на единицу. Когда значение счетчика достигает нуля, обрабатывается связанное с этим таймером событие, а головным элементом списка становится следующий элемент. При такой схеме добавление и удаление таймера является операцией, требующей затрат ресурсов, при этом время выполнения операции пропорционально длине списка.

Если максимальное значение таймера ограничено и известно заранее, то может быть применен более эффективный метод. Здесь можно использовать массив, называющийся **колесом времени** (рис. 6.39). Каждое гнездо соответствует одному импульсу сигнала времени. Текущее время, показанное на рисунке, — $T = 4$. Таймеры должны сработать через 3, 10 и 12 импульсов сигнала времени. Если новый таймер должен сработать через 7 тиков, то все, что нужно сделать, — задать значение указателя в гнезде 11 на новый список таймеров. Аналогично, если таймер, установленный на время $T + 10$, должен быть отключен, нужно всего лишь обнулить запись в гнезде 14. Обратите внимание на то, что массив на рис. 6.39 не может поддерживать таймеры за пределами $T + 15$.

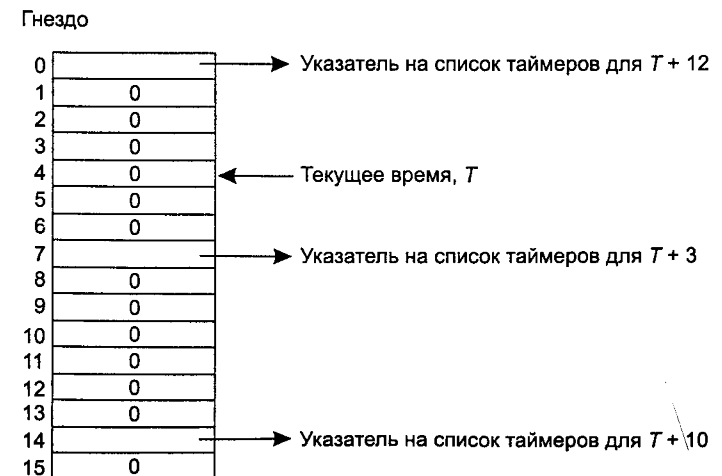


Рис. 6.39. Колесо времени

При каждом импульсе сигнала времени часов указатель текущего времени перемещается по колесу времени вперед (циклично) на одно гнездо. Если гнездо, на которое он указывает, не нулевое, обрабатываются все таймеры списка, на который указывает гнездо. Многочисленные варианты основной идеи обсуждаются в (Varghese и Lauck, 1989).

Протоколы для гигабитных сетей

Появление гигабитных сетей приходится на начало 90-х годов. Сначала к ним пытались применить старые протоколы, но при этом сразу же возникло множество проблем. Некоторые из этих проблем, а также методы их решения в протоколах будущих, даже более быстрых, сетей будут обсуждаться в данном разделе.

Первая проблема заключается в том, что многие протоколы используют 32-разрядные порядковые номера. В прежние годы, когда типичная скорость выделенных линий между маршрутизаторами равнялась 56 Кбит/с, хосту, который, бешено вращая глазами, постоянно выдавал бы данные в сеть, потребовалось бы больше недели на то, чтобы у него закончились порядковые номера. С точки зрения разработчиков TCP, 2^{32} считалось неплохим приближением к бесконечности, поскольку вероятность блуждания пакетов по сети в течение недели практически равна нулю. В Ethernet со скоростью 10 Мбит/с критическое время снизилось с одной недели до 57 минут. Это, конечно, гораздо меньше, но даже с таким интервалом еще можно иметь дело. Когда же Ethernet выдает в Интернет данные со скоростью 1 Гбит/с, порядковые номера закончатся примерно через 34 с. Это уже никуда не годится, поскольку максимальное время жизни пакета в Интернете равно 120 с. Внезапно оказалось, что 2^{32} совершенно не подходит в качестве значения, приближенного к бесконечности, поскольку стало очевидно, что отправителю, посылающему достаточно много пакетов, придется повторять их порядковые номера в то время, как старые пакеты все еще будут блуждать по сети. В RFC 1323 предлагается метод борьбы с этой ситуацией.

Проблема состоит в том, что многие разработчики протоколов просто предполагали, что время цикла пространства порядковых номеров значительно превосходит максимальное время жизни пакетов. Соответственно, в этих протоколах даже не рассматривалась сама возможность того, что старые пакеты еще могут находиться где-то в сети, когда порядковые номера опишут полный круг. В гигабитных сетях это предположение оказалось неверным.

Вторая проблема возникла, когда выяснилось, что скорости передачи данных увеличиваются значительно быстрее, чем скорости обработки данных. (Обращение к разработчикам компьютеров: идите и побейте разработчиков средств связи! Мы рассчитываем на вас.) В 70-е годы объединенная сеть ARPANET работала на скорости 56 Кбит/с и состояла из компьютеров с производительностью около 1 MIPS (1 млн инструкций в секунду). Использовались пакеты размером 1008 бит — таким образом, сеть ARPANET могла доставлять около 56 пакетов в секунду. Поскольку время передачи пакета составляло около 18 мс, хост мог затратить 18 000 инструкций на обработку одного пакета. Конечно, это полностью загрузило бы центральный процессор, однако можно было снизить это число до 9000 инструкций при половинной занятости процессора, предоставляя ему возможность выполнять всю необходимую работу.

Сравните эти цифры с цифрами для современных компьютеров, имеющих производительность 1000 MIPS и обменивающихся 1500-байтными пакетами по гигабитной линии. Пакеты могут прибывать с частотой свыше 80 000 пакетов в секунду, поэтому, если мы хотим зарезервировать половину мощности процессора для приложений, обработка пакета должна быть завершена за 6,25 мкс. За это время компьютер с производительностью 1000 млн инструкций в секунду может выполнить 6250 инструкций — лишь треть того, что было доступно хостам сети ARPANET. Более того, современные RISC-инструкции выполняют меньше действий, чем старые CISC-инструкции, так что проблема, на самом деле, оказывается еще тяжелее. Отсюда можно сделать следующий вывод: у протокола осталось меньше времени на обработку пакетов, поэтому протоколы должны стать проще.

Третья проблема состоит в том, что протокол с возвратом на *n* плохо работает в линиях с большим значением произведения пропускной способности на задержку. Рассмотрим, к примеру, линию длиной 4000 км, работающую со скоростью 1 Гбит/с. Время прохождения сигнала в оба конца равно 40 мс. За это время отправитель успевает передать 5 Мбайт. Если обнаруживается ошибка, потребуется 40 мс, чтобы оповестить об этом отправителя. При использовании протокола с возвратом на *n* отправителю потребуется повторять передачу не только поврежденного пакета, но также и до 5 Мбайт пакетов, переданных после поврежденного. Очевидно, что этот протокол использует ресурсы очень неэффективно.

Суть четвертой проблемы состоит в том, что гигабитные линии принципиально отличаются от мегабитных — в длинных гигабитных линиях главным ограничивающим фактором является не пропускная способность, а задержка. На рис. 6.40 изображена зависимость времени, требующегося для передачи файла размером 1 Мбит по линии длиной в 4000 км, от скорости передачи. На скоростях до 1 Мбит/с время передачи, в основном, зависит от скорости передачи данных. При скорости 1 Гбит/с задержка в 40 мс значительно превосходит 1 мс (время, требующееся для передачи битов по оптоволоконному кабелю). Дальнейшее увеличение скорости вообще не оказывает почти никакого действия на время передачи файла.

Изображенная на рис. 6.40 зависимость демонстрирует ограничения сетевых протоколов. Она показывает, что протоколы с ожиданием подтверждений, такие как удаленный вызов процедуры (RPC, Remote Procedure Call), имеют врожденное ограничение на производительность. Это ограничение связано со скоростью света. Никакие технологические прорывы в области оптики здесь не помогут (хотя могли бы помочь новые законы физики).

Пятая проблема, которую стоит упомянуть, связана не с протоколами или технологиями, а с появлением новых гигабитных мультимедийных приложений, для которых постоянство времени передачи пакета так же важно, как и само среднее значение задержки. Низкая, но постоянная скорость доставки часто предпочтительнее высокой, но непостоянной.

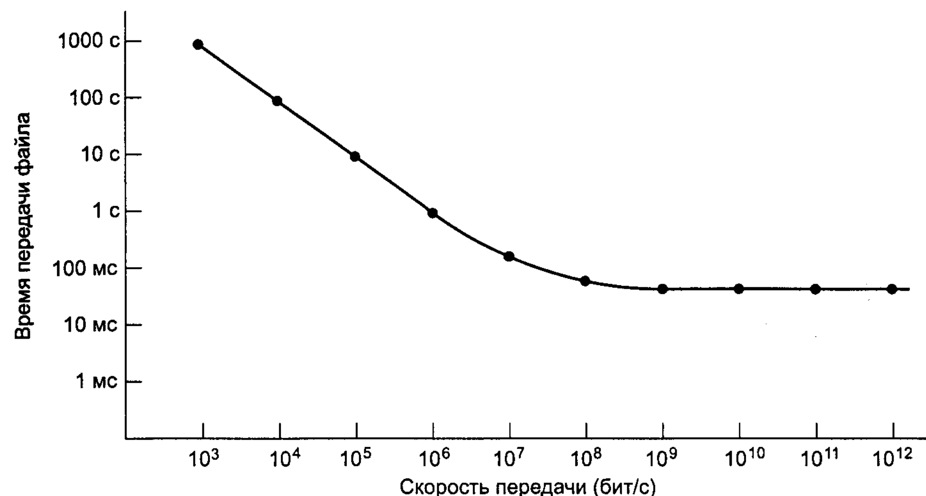


Рис. 6.40. Время передачи и подтверждения файла размером 1 Мбит по линии длиной 4000 км

Перейдем теперь к рассмотрению методов решения всего этого набора проблем. Сначала будет сделано несколько общих замечаний, затем мы рассмотрим механизмы протоколов, формат пакетов и программное обеспечение протоколов.

Главный принцип, который каждый разработчик гигабитных сетей должен выучить наизусть, звучит так:

Проектируя, стремись увеличить скорость, а не оптимизировать пропускную способность.

При разработке старых протоколов обычно ставилась задача минимизировать количество битов, переданных в линию, часто за счет использования полей малого размера и упаковывания нескольких полей вместе в один байт или слово. В настоящее время экономить пропускную способность смысла нет: ее более чем достаточно. Проблема заключается в обработке протоколами пакетов, поэтому при разработке новых протоколов минимизировать нужно именно время обработки. Разработчики IPv6, к счастью, хорошо понимают это.

Конечно, заманчивы попытки ускорить работу сетей, создав быстрые сетевые интерфейсы на аппаратном уровне. Сложность такого подхода состоит в том, что если только протокол не является чрезвычайно простым, аппаратный интерфейс представляет собой дополнительную плату с отдельным процессором и своей программой. Чтобы сетевой сопроцессор не был столь же дорогим, он, как и центральный процессор, часто представляет собой более медленную микросхему. В результате такого подхода быстрый центральный процессор ждет, пока медленный сопроцессор выполнит свою работу. Было бы неверным полагать, что у центрального процессора на время ожидания обязательно найдется другая работа. Более того, для общения двух процессоров общего назначения потребуются тщательно продуманные протоколы, обеспечивающие их корректную синхронизацию. Как правило, наилучший метод заключается в создании простых протоколов, чтобы всю работу мог выполнять центральный процессор.

Рассмотрим теперь вопрос обратной связи в высокоскоростных протоколах. Из-за относительно длинного цикла задержки желателен отказ от обратной связи: на оповещение отправителя уходит слишком много времени. Один пример обратной связи — управление скоростью передачи с помощью протокола скользящего окна. Чтобы избежать долгих задержек, связанных с передачей получателем информации о состоянии окна отправителю, лучше использовать протокол, основанный на скорости. В таком протоколе отправитель может посылать не быстрее, чем с некоторой скоростью, с которой получатель заранее согласился.

Второй пример обратной связи — алгоритм затяжного пуска, разработанный Джекобсоном. Этот алгоритм проводит многочисленные пробы, пытаясь определить пропускную способность сети. В высокоскоростных сетях на проведение пяти-шести проб для определения ответной реакции сети тратится огромное количество сетевых ресурсов. Более эффективная схема состоит в резервировании ресурсов отправителем, получателем и сетью в момент установки соединения. Кроме того, заблаговременное резервирование ресурсов позволяет несколько снизить эффект неравномерности доставки (джиттер). Короче говоря, повышение скоростей передачи в сетях неумолимо заставляет разработчиков выбирать ориентированную на соединение (или близкую к этому) схему работы сети.

Крайне важен в гигабитных сетях формат пакета. В заголовке должно быть как можно меньше полей — это позволит снизить время его обработки. Сами поля должны быть достаточно большими, чтобы нести в себе как можно больше полезной (служебной) информации. Кроме того, они не должны пересекать границы слов, тогда их будет проще обработать. В данном контексте под «достаточно большим» подразумевается такой размер, который исключает возникновение проблем зацикливания порядковых номеров при нахождении в сети старых пакетов, учитывает отсутствие у получателя возможности объявить реально доступный размер окна из-за слишком малого служебного поля размера окна, и т. д.

Кроме того, контрольные суммы заголовка и данных следует считать раздельно. Причин здесь две. Во-первых, чтобы предоставить возможность считать контрольную сумму только заголовка, без данных, что значительно быстрее. Во-вторых, чтобы иметь возможность убедиться в правильности заголовка, прежде чем начать копировать данные в пространство пользователя. Контрольную сумму данных лучше проверять во время копирования данных в пространство пользователя, но если заголовок неверен, то может оказаться, что будет производиться копирование не того процесса. Во избежание ненужного копирования необходимо считать контрольные суммы отдельно.

Максимальный размер поля данных должен быть достаточно большим, чтобы обеспечить возможность эффективной работы сети даже при наличии больших задержек. Кроме того, чем больше размер поля данных, тем меньшую долю в общем потоке данных составляют заголовки.

Еще одно полезное свойство протокола — возможность посылать нормальное количество данных вместе с запросом соединения. При этом можно сэкономить время одного запроса и ответа.

Наконец, следует сказать несколько слов о программном обеспечении протокола. Следует сконцентрировать внимание на успешном варианте работы. Мно-

гие старые протоколы были рассчитаны на работу в условиях плохих линий связи и особое внимание уделяли обработке ошибок (например, при потере пакета). Чтобы сделать протокол быстрее, разработчик должен, в первую очередь, постараться минимизировать время обработки в нормальном случае. Минимизация времени обработки в случае ошибок на линии должна быть на втором месте.

Второй аспект, касающийся программного обеспечения, состоит в минимизации времени копирования. Как уже было сказано ранее, копирование данных часто оказывается основным источником накладных расходов. В идеале, аппаратура должна отображать каждый приходящий пакет в память в виде непрерывного блока данных. Затем программа должна скопировать этот пакет в буфер пользователя за одну операцию блочного копирования. В зависимости от принципа работы кэша может быть желателен отказ от циклической организации копирования. Другими словами, чтобы скопировать 1024 слова, возможно, самым быстрым способом будет использование 1024 инструкций `MOVE` подряд (или 1024 пар инструкций чтения и записи). Процедура копирования является столь критичной, что ее следует очень аккуратно писать вручную на ассемблере, если только нет возможности заставить компилятор произвести самый оптимальный код.

Резюме

Транспортный уровень — это ключ к пониманию многоуровневых протоколов. Он предоставляет различные услуги, наиболее важной из которых является сквозной, надежный, ориентированный на соединение поток байтов от отправителя к получателю. Доступ к нему предоставляется при помощи сервисных примитивов, позволяющих устанавливать, использовать и разрывать соединения.

Транспортные протоколы должны обладать способностью управлять соединением в ненадежных сетях. Установка соединения осложняется возможностью существования дубликатов пакетов, которые могут появляться в самый неподходящий момент. Для борьбы с этими дубликатами при установке соединения применяется алгоритм «тройного рукопожатия». Разрыв соединения проще установки и, тем не менее, далеко не тривиален из-за наличия проблемы двух армий.

Даже если сетевой уровень абсолютно надежен, у транспортного уровня полно работы. Он должен обрабатывать все служебные примитивы, управлять соединениями и таймерами, а также предоставлять и использовать кредиты.

Основными транспортными протоколами Интернета являются TCP и UDP. UDP — это протокол без установления соединения, который работает с IP-пакетами и занимается обеспечением мультиплексирования и демультиплексирования нескольких процессов с использованием единого IP-адреса. UDP может использоваться при клиент-серверных взаимодействиях, например, при удаленном вызове процедур. Кроме того, на его основе можно создавать протоколы реального времени, такие как RTP.

Наиболее распространенным протоколом Интернета является TCP. Он обеспечивает надежную двухстороннюю потоковую байтовую передачу. Он использует 20-байтный заголовок для всех сегментов. Сегменты могут фрагментироваться маршрутизаторами Интернета, поэтому хосты должны уметь восстанавливать исходные сегменты из отдельных фрагментов. Оптимизации производительности протокола TCP было уделено много внимания. Для этого в нем применяются алгоритмы Нагля (Nagle), Кларка (Clark), Джекобсона (Jacobson), Карна (Karn) и др. Беспроводные линии связи приводят к усложнению протокола TCP. Транзакционный TCP — это расширение традиционного протокола TCP, предназначенное для поддержки клиент-серверного взаимодействия с использованием упрощенной процедуры обмена пакетами.

Производительность сети обычно в основном определяется протоколом и накладными расходами по обработке TPDU-модулей, причем с увеличением скорости передачи данных эта ситуация ухудшается. При разработке протоколов следует стараться минимизировать количество TPDU-модулей, количество переключений контекста и время копирования TPDU-модулей. В гигабитных сетях требуются простые протоколы.

Вопросы

1. В нашем примере транспортных примитивов, приведенных в табл. 6.1, `LISTEN` является блокирующим вызовом. Обязательно ли это? Если нет, объясните, как следует пользоваться неблокирующим примитивом. Какое преимущество это даст по сравнению со схемой, описанной в тексте?
2. В модели, лежащей в основе диаграммы состояний на рис. 6.3, предполагается, что пакеты могут теряться на сетевом уровне и поэтому должны подтверждаться индивидуально. Допустим, что сетевой уровень обеспечивает 100-процентную надежность доставки и никогда не теряет пакеты. Нужны ли какие-либо изменения в диаграмме состояний, показанной на рис. 6.3, и если да, то какие?
3. В обеих частях листинга 6.1 значение `SERVER_PORT` должно быть одинаковым у клиента и у сервера. Почему это так важно?
4. Предположим, что используется управляемая часами схема генерирования начальных порядковых номеров с 15-разрядным счетчиком часов. Часы тикают раз в 100 мс, а максимальное время жизни пакета равно 60 с. Как часто должна производится ресинхронизация:
   1) В худшем случае?
   2) Когда данные потребляют 240 порядковых номеров в минуту?
5. Почему максимальное время жизни пакета *T* должно быть достаточно большим, чтобы гарантировать, что не только пакет, но и его подтверждение исчезли?
6. Представьте, что для установки соединений вместо «тройного рукопожатия» использовалось бы двойное (то есть третье сообщение не требовалось). Возможны ли при этом тупиковые ситуации? Приведите пример или докажите, что тупиковых ситуаций нет.

7. Представьте себе обобщенную проблему n армий, в которой договоренность двух любых армий достаточна для победы. Существует ли протокол, позволяющий армиям синих выиграть?
8. Рассмотрим проблему восстановления от сбоев хостов (рис. 6.15). Если бы интервал между записью и отправкой подтверждения, или наоборот, можно было сделать относительно небольшим, какими были бы две лучшие стратегии отправителя и получателя, минимизирующие шансы ошибки протокола?
9. Возможны ли тупиковые ситуации для транспортной сущности, описанной в тексте (листинг 6.2)?
10. Из любопытства разработчик транспортной сущности, представленной в листинге 6.2, решил поместить внутрь процедуры `sleep` счетчики для сбора статистики о массиве *conn*. Среди прочих параметров определялось число соединений в каждом из семи возможных соединений, n_i (i = 1, ..., 7). Написав на языке FORTRAN большую программу для анализа данных, разработчик обнаружил, что соотношение $\Sigma n_i = MAX_CON$ оказывается всегда верным. Есть ли еще другие инварианты, включающие только эти семь переменных?
11. Что происходит, когда пользователь транспортной сущности, представленной в листинге 6.2, посылает сообщение нулевой длины? Обсудите значение этого факта.
12. Для каждого события, которое потенциально может произойти в транспортной сущности, представленной в листинге 6.2, определите, является ли оно разрешенным или нет, когда пользователь находится в состоянии *sending*.
13. Обсудите преимущества и недостатки схемы кредитов по сравнению с протоколами скользящего окна.
14. Зачем нужен протокол UDP? Разве не достаточно было бы просто позволить пользовательским процессам посылать необработанные IP-пакеты?
15. Рассмотрите простой протокол прикладного уровня, построенный на основе UDP, который позволяет клиенту запрашивать файл с удаленного сервера, расположенного по популярному адресу. Клиент вначале посылает запрос с именем файла, а сервер отвечает последовательностью информационных пакетов, содержащих различные части запрошенного файла. Для обеспечения надежности и доставки частей в правильном порядке клиент и сервер используют протокол с ожиданием. Какие проблемы могут возникнуть с таким протоколом, кроме очевидных проблем с производительностью? Обратите внимание на вероятность сбоя процессов.
16. Клиент посылает 128-байтный запрос на сервер, удаленный от него на 100 км, по оптоволокну со скоростью 1 Гбит/с. Какова эффективность линии во время выполнения удаленного вызова процедуры?
17. Рассмотрите снова ситуацию, описанную в предыдущем вопросе. Вычислите минимально возможное время ответа для данной линии со скоростью 1 Гбит/с и для 1-мегабитной линии. Какой вывод можно сделать, исходя из полученных значений?

18. Как в UDP, так и в TCP номера портов используются для идентификации принимающей сущности при доставке сообщения. Назовите две причины того, почему для этих протоколов были изобретены новые абстрактные идентификаторы (номера портов) и не использовались идентификаторы процессов, уже существовавшие на момент появления данных протоколов?
19. Каков суммарный размер минимального MTU протокола TCP, включая накладные расходы TCP и IP, но не включая накладные расходы уровня передачи данных?
20. Фрагментация и дефрагментация дейтаграмм выполняется протоколом IP и невидима для протокола TCP. Означает ли это, что протокол TCP не должен беспокоиться о данных, приходящих в неверном порядке?
21. RTP используется для передачи звукозаписей, по качеству соответствующих компакт-дискам. При этом для передачи одного отсчета каждого из стереоканалов используется пара 16-битных слов, передающихся 44 100 раз в секунду. Сколько пакетов в секунду должен уметь передавать RTP?
22. Возможно ли поместить код RTP в ядро операционной системы наряду с UDP? Ответ поясните.
23. Процессу хоста 1 был назначен порт p, а процессу хоста 2 —порт q. Может ли быть одновременно несколько соединений между этими двумя портами?
24. На рис. 6.23 мы видели, что в дополнение к 32-разрядному полю *Подтверждение* в четвертом слове имеется бит *ACK*? Приносит ли он какую-либо пользу? Ответ поясните.
25. Максимальный размер полезной нагрузки TCP-сегмента может быть равен 65 495 байт. Почему было выбрано такое странное число?
26. Опишите два способа, которыми можно попасть в состояние *SYN RCVD* на рис. 6.26.
27. В чем состоят потенциальные недостатки использования алгоритма Нагля в сильно перегруженной сети?
28. Рассмотрите эффект использования алгоритма затяжного пуска в линии со значением времени прохождения сигнала в оба конца, равным 10 мс, без перегрузок. Размер окна получателя 24 Кбайт, а максимальный размер сегмента равен 2 Кбайт. Через какое время может быть послано полное окно?
29. Предположим, окно перегрузки протокола TCP установлено на 18 Кбайт, когда происходит тайм-аут. Каким будет размер окна, если четыре последующих передачи будут успешными? Максимальный размер предполагается равным 1 Кбайт.
30. Текущее значение оценки времени прохождения сигнала в оба конца протокола TCP *RTT* равно 30 мс, а следующие подтверждения приходят через 26, 32 и 24 мс. Каково будет новое значение *RTT*? Используйте $\alpha = 0{,}9$.
31. TCP-машина посылает окна по 65 535 байт по гигабитному каналу, в котором время прохождения сигнала в один конец равно 10 мс. Какова максимальная достижимая пропускная способность канала? Чему равна эффективность использования линии?

32. Какова максимальная скорость, с которой хост может посылать в линию TCP-пакеты, содержащие 1500 байт полезной нагрузки, если максимальное время жизни пакета в сети равно 120 с? Требуется, чтобы порядковые номера не зацикливались. При расчете учитывайте накладные расходы на TCP, IP и Ethernet. Предполагается, что кадры Ethernet могут посылаться непрерывно.
33. Чему равна максимальная скорость передачи данных для каждого соединения, если максимальный размер TPDU-модуля равен 128 байт, максимальное время жизни TPDU-модуля равно 30 с и используются 8-разрядные порядковые номера TPDU-модулей?
34. Предположим, вы измеряется время, необходимое для получения TPDU-модуля. Когда возникает прерывание, вы читаете показания системных часов в миллисекундах. После полной обработки TPDU-модуля вы снова читаете показания часов. В результате миллиона измерений вы получаете значения 0 мс 270 000 раз и 1 мс 730 000 раз. Какой вывод можно сделать на основании полученных результатов?
35. Центральный процессор выполняет 1000 млн инструкций в секунду (1000 MIPS). Данные могут копироваться 64-разрядными словами. На копирование каждого слова требуется 10 инструкций. Может ли такая система управлять гигабитной линией, если каждый приходящий пакет должен быть скопирован четырежды? Для простоты предположим, что все инструкции, даже обращения к памяти, выполняются с максимальной скоростью 1000 MIPS.
36. Для решения проблемы повторного использования старых порядковых номеров пакетов, в то время как старые пакеты еще существуют, можно использовать 64-разрядные порядковые номера. Однако теоретически оптоволоконный кабель может обладать пропускной способностью до 75 Тбит/с. Каким следует выбрать максимальное время жизни пакетов, чтобы гарантировать отсутствие в сети будущего пакетов с одинаковыми номерами при скорости линий 75 Тбит/с и 64-разрядных порядковых номерах? Предполагается, что порядковый номер дается каждому байту, как в протоколе TCP.
37. Назовите одно преимущество RPC на основе UDP перед T/TCP. Теперь назовите одно преимущество T/TCP перед RPC.
38. На рис. 6.33, *а* видно, что для выполнения удаленного вызова процедуры требуется 9 пакетов. Могут ли возникнуть обстоятельства, при которых понадобятся ровно 10 пакетов?
39. В разделе «Протоколы для гигабитных сетей» мы подсчитали, что гигабитная линия отправляет хосту 80 000 пакетов в секунду, оставляя ему только 6250 инструкций на их обработку — половина производительности процессора отдается приложениям. В результате выяснилось, что размер пакета должен быть 1500 байт. Выполните подсчеты заново для пакетов ARPANET (128 байт). В обоих случаях предполагается, что в размер пакета включены все накладные расходы.
40. В гигабитной линии протяженности более 4000 км ограничивающим фактором является не пропускная способность, а время задержки. Рассмотрим региональную сеть со средней удаленностью отправителя от получателя 20 км. При какой скорости передачи данных время прохождения сигнала в оба конца будет равно времени передачи одного килобайтного пакета?

41. Рассчитайте произведение пропускной способности на задержку в следующих сетях: 1) T1 (1,5 Мбит/с), 2) Ethernet (10 Мбит/с), 3) T3 (45 Мбит/с), 4) STS-3 (155 Мбит/с). Предполагается, что RTT = 100 мс. Не забудьте о том, что в заголовке TCP на размер окна отводится 16-разрядное поле. Как это замечание отразится на результатах вычислений?
42. Чему равно произведение пропускной способности на задержку для 50-мегабитного канала геостационарной спутниковой связи? Если все пакеты имеют размер 1500 байт (включая накладные расходы), какого размера должно быть окно в пакетах?
43. Файловый сервер, моделируемый листингом 6.1, далек от совершенства. Можно внести некоторые улучшения. Проделайте следующие изменения:
    1) пусть у клиента появится третий аргумент, указывающий байтовый диапазон;
    2) добавьте флаг –w в программу клиента, который позволил бы записывать файл на сервер.
44. Измените программу, приведенную в листинге 6.2, таким образом, чтобы она выполняла восстановление после сбоев. Добавьте новый тип пакета *reset*, который может прибывать после того, как соединение было открыто обеими сторонами и никем не закрыто. Это событие, происходящее на обоих концах соединения, означает, что любые пакеты, находившиеся в пути, были либо доставлены, либо уничтожены, но в любом случае они больше не находятся в сети.
45. Напишите программу, моделирующую управление буфером транспортной сущностью и использующую алгоритм скользящего окна вместо примененной в листинге 6.2 системы кредитов. Пусть процесс более высокого уровня случайным образом открывает соединения, посылает данные и закрывает соединения. Для простоты, пусть все данные передаются только с машины *A* на машину *B*. Поэкспериментируйте с различными стратегиями выделения буферов на машине *B*, например, выделяя буферы каждому соединению и организуя общий буферный пул, и измерьте полную пропускную способность, достижимую в каждом случае.
46. Разработайте и реализуйте систему сетевого общения (чат), рассчитанную на несколько групп пользователей. Координатор чата должен располагаться по хорошо известному сетевому адресу, использовать для связи с клиентами протокол UDP, настраивать чат-серверы перед каждой сессией общения и поддерживать каталог чат-сессий. На каждую сессию должен выделяться один обслуживающий сервер. Для связи с клиентами сервер должен использовать TCP. Клиентская программа должна позволять пользователям начинать разговор, присоединяться к уже ведущейся дискуссии и покидать сессию. Разработайте и реализуйте код координатора, сервера и клиента.

Глава 7
Прикладной уровень

- Служба имен доменов DNS
- Электронная почта
- Всемирная паутина (WWW)
- Мультимедиа
- Резюме
- Вопросы

Покончив с изучением базовых сведений о компьютерных сетях, мы переходим к уровню, на котором расположены все приложения. Все уровни, находящиеся в модели OSI ниже прикладного, служат для обеспечения надежной доставки данных, но никаких полезных для пользователя действий не производят. В этой главе мы изучим некоторые реальные сетевые приложения.

Разумеется, даже прикладной уровень нуждается в обслуживающих протоколах, с помощью которых осуществляется функционирование приложений. Соответственно, прежде чем начать рассмотрение самих приложений, мы изучим один из таких протоколов. Речь идет о службе имен доменов, DNS, обеспечивающей присвоение имен в Интернете. Затем мы рассмотрим три реально действующих приложения: электронную почту, Всемирную паутину и, наконец, мультимедиа.

Служба имен доменов DNS

Хотя программы теоретически могут обращаться к хостам, почтовым ящикам и другим ресурсам по их сетевым адресам (например, IP), пользователям запоминать их тяжело. Кроме того, отправка электронной почты на адрес Tanya@128.111.24.41 будет означать, что в случае переезда сервера таниного провайдера или организации на новое место с новым IP-адресом придется изменить ее адрес e-mail. Для отделения имен машин от их адресов было решено использовать текстовые ASCII-имена. Поэтому танин адрес более привычно выглядит в таком виде: Tanya@art.ucsb.edu. Тем не менее, сеть сама по себе понимает только численные адреса, поэтому нужен механизм преобразования ASCII-строк в сетевые адреса. В следующих разделах мы изучим, как производится это отображение в Интернете.

Когда-то давно в сети ARPANET соответствие между текстовыми и двоичными адресами просто записывалось в файле *hosts.txt*, в котором перечислялись все хосты и их IP-адреса. Каждую ночь все хосты получали этот файл с сайта, на котором он хранился. В сети, состоящей из нескольких сотен больших машин, работающих под управлением системы с разделением времени, такой подход работал вполне приемлемо.

Но когда к сети подключились тысячи рабочих станций, всем стало ясно, что этот способ не сможет работать вечно. Во-первых, размер файла рано или поздно стал бы слишком большим. Однако, что еще важнее, если управление именами хостов не осуществлять централизованно, неизбежно возникновение конфликтов имен. В то же время, представить себе централизованное управление именами всех хостов гигантской международной сети довольно сложно. Для разрешения всех этих проблем и была разработана **служба имен доменов** (**DNS**, Domain Name System).

Суть системы DNS заключается в иерархической схеме имен, основанной на доменах, и распределенной базе данных, реализующей эту схему имен. В первую очередь эта система используется для преобразования имен хостов и пунктов назначения электронной почты в IP-адреса, но также может использоваться и в других целях. Определение системы DNS дано в RFC 1034 и 1035.

В общих чертах система DNS применяется следующим образом. Для преобразования имени в IP-адрес прикладная программа обращается к библиотечной процедуре, называющейся **распознавателем**, передавая ей имя в качестве параметра. Распознаватель посылает UDP-пакет локальному DNS-серверу, который ищет имя в базе данных и возвращает соответствующий IP-адрес распознавателю, который, в свою очередь, передает этот адрес вызвавшей его прикладной программе. Имея IP-адрес, программа может установить TCP-соединение с адресатом или послать ему UDP-пакеты.

Пространство имен DNS

Управление большим и постоянно изменяющимся набором имен представляет собой нетривиальную задачу. В почтовой системе на письмах требуется указывать (явно или неявно) страну, штат или область, город, улицу, номер дома, квартиру и фамилию получателя. Благодаря использованию такой иерархической схемы не возникает путаницы между Марвином Андерсоном, живущим на Мейн-стрит в Уайт Плейнс, штат Нью-Йорк, и Марвином Андерсоном с Мейн-стрит в Остине, штат Техас. Система DNS работает аналогично.

Интернет концептуально разделен на 200 **доменов** верхнего уровня. Доменами называют в Интернете множество хостов, объединенное в логическую группу. Каждый домен верхнего уровня подразделяется на поддомены, которые, в свою очередь, также могут состоять из других доменов, и т. д. Все эти домены можно рассматривать в виде дерева, показанного на рис. 7.1. Листьями дерева являются домены, не разделяющиеся на поддомены (но состоящие из хостов, конечно). Такой конечный домен может состоять из одного хоста или может представлять компанию и содержать в себе тысячи хостов.

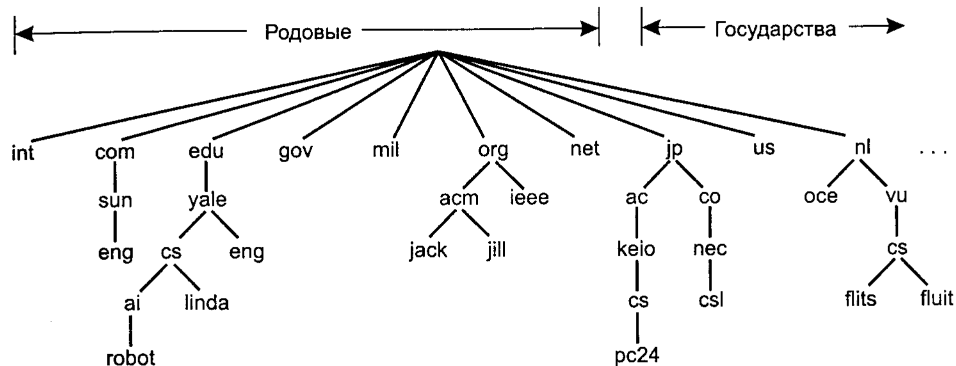


Рис. 7.1. Часть доменного пространства имен Интернета

Домены верхнего уровня разделяются на две группы: родовые домены и домены государств. К родовым относятся домены *com* (commercial — коммерческие организации), *edu* (educational — учебные заведения), *gov* (government — федеральное правительство США), *int* (international — определенные международные организации), *mil* (military — вооруженные силы США), *net* (network — сетевые операторы связи) и *org* (некоммерческие организации). За каждым государством в соответствии с международным стандартом ISO 3166 закреплен один домен государства.

В ноябре 2000 года ICANN было утверждено 4 новых родовых имени доменов верхнего уровня, а именно: *biz* (бизнес), *info* (информация), *name* (имена людей) и *pro* (специалисты, такие как доктора и адвокаты). Кроме того, по просьбе соответствующих отраслевых организаций были введены еще три специализированных имени доменов верхнего уровня: *aero* (аэрокосмическая промышленность), *coop* (кооперативы) и *museum* (музеи). В будущем появятся и другие домены верхнего уровня.

Между прочим, по мере коммерциализации Интернета появляется все больше спорных вопросов. Взять хотя бы домен *pro*. Он предназначен для сертифицированных специалистов. Но кто является специалистом, а кто нет? Кем должны быть эти специалисты сертифицированы? Понятно, что доктора и адвокаты — это профессионалы, спору нет. А что делать со свободными художниками, учителями музыки, заклинателями, водопроводчиками, парикмахерами, мусорщиками, рисователями татуировок, наемниками и проститутками? Имеют ли право квалифицированные представители всех этих и многих других профессий получать домены *pro*? Если да, то кто выдаст сертификат каждому из этих специалистов?

В принципе, получить домен второго уровня типа *name-of-company.com* несложно. Надо лишь проверить, не занято ли желаемое имя домена кем-то другим и не является ли оно чьей-нибудь торговой маркой. Для этого надо зайти на сайт регистрационного бюро верхнего уровня (в данном случае *com*). Если все в порядке, заказчик регистрируется и за небольшую ежегодную абонентскую плату получает домен второго уровня. На сегодняшний день в качестве имен поддоменов *com* уже используются практически все общеупотребительные английские слова. Попробуйте набрать какое-нибудь слово, касающееся домашнего хозяйства, животных, растений, частей тела и т. д. Вряд ли ошибетесь.

Имя каждого домена, подобно полному пути к файлу в файловой системе, состоит из пути от этого домена до (безымянной) вершины дерева. Компоненты пути разделяются точками. Так, домен технического отдела корпорации Sun Microsystems может выглядеть как *eng.sun.com*, а не так, как это принято в стиле UNIX (*/com/sun/eng*). Следует отметить, что *eng.sun.com* не конфликтует с потенциальным использованием имени *eng* в домене *eng.yale.edu*, где он может обозначать факультет английского языка Йельского университета.

Имена доменов могут быть абсолютными и относительными. Абсолютное имя домена всегда оканчивается точкой (например, *eng.sun.com.*), тогда как относительное имя — нет. Для того чтобы можно было единственным образом определить истинные значения относительных имен, они должны интерпретироваться в некотором контексте. В любом случае именованный домен означает определенный узел дерева и все узлы под ним.

Имена доменов нечувствительны к изменению регистра символов. Так, например, *edu* и *EDU* означают одно и то же. Длина имен компонентов может достигать 63 символов, а длина полного пути не должна превосходить 255 символов.

В принципе, новые домены могут добавляться в дерево двумя разными путями. Например, *cs.yale.edu* можно без проблем поместить в домен *us* под именем *cs.yale.ct.us*. На практике, однако, почти все организации в США помещаются под родовыми доменами, тогда как почти все организации за пределами Соединенных Штатов располагаются под доменами их государств. Не существует каких-либо правил, запрещающих регистрацию под двумя доменами верхнего уровня, однако использует эту возможность лишь небольшое число организаций (за исключением интернациональных, например, *sony.com* и *sony.nl*).

Каждый домен управляет доступом к доменам, расположенным под ним. Например, в Японии домены *ac.jp* и *co.jp* соответствуют американским доменам *edu* и *com*. В Голландии подобное различие не используется, и все домены организаций помещаются прямо под доменом *nl*. В качестве примера приведем имена доменов факультетов компьютерных наук трех университетов.

1. cs.yale.edu (Йельский университет, США).
2. cs.vu.nl (университет Врийе, Нидерланды).
3. cs.keio.ac.jp (университет Кейо, Япония).

Для создания нового домена требуется разрешение домена, в который он будет включен. Например, если в Йеле образовалась группа VLSI, которая хочет зарегистрировать домен *vlsi.cs.yale.edu*, ей нужно разрешение от того, кто управляет доменом *cs.yale.edu*. Аналогично, если создается новый университет, например, университет Северной Южной Дакоты, он должен попросить менеджера домена *edu* присвоить их домену имя *unsd.edu*. Таким образом удается избежать конфликта имен, а каждый домен отслеживает состояние всех своих поддоменов. После того как домен создан и зарегистрирован, в нем могут создаваться поддомены, например *cs.unsd.edu*, для чего уже не требуется разрешения вышестоящих доменов.

Структура доменов отражает не физическое строение сети, а логическое разделение между организациями и их внутренними подразделениями. Так, если факультеты компьютерных наук и электротехники располагаются в одном здании и пользуются одной общей локальной сетью, они, тем не менее, могут иметь различные домены. И наоборот, если, скажем, факультет компьютерных наук располагается в двух различных корпусах университета с различными локальными сетями, логически все хосты обоих зданий обычно принадлежат к одному и тому же домену.

Записи ресурсов

У каждого домена, независимо от того, является ли он одиноким хостом или доменом верхнего уровня, может быть набор ассоциированных с ним **записей ресурсов**. Для одинокого хоста запись ресурсов чаще всего представляет собой просто его IP-адрес, но существует также много других записей ресурсов. Когда распознаватель передает имя домена DNS-серверу, то, что он получает обратно, представляет собой записи ресурсов, ассоциированные с его именем. Таким образом, истинное назначение системы DNS заключается в преобразовании доменных имен в записи ресурсов.

Запись ресурса состоит из пяти частей. Хотя для эффективности они часто перекодируются в двоичную форму, в большинстве описаний записи представляются в виде ASCII-текста, по одной строке на запись ресурса. Мы будем использовать следующий формат:

```
Domain_name Time_to_live Class Type Value
```

Поле *Domain_name* (имя домена) обозначает домен, к которому относится текущая запись. Обычно для каждого домена существует несколько записей ресурсов, и каждая копия базы данных хранит информацию о нескольких доменах. Поле имени домена является первичным ключом поиска, используемым для выполнения запросов. Порядок записей в базе данных значения не имеет. В ответ на запрос о домене возвращаются все удовлетворяющие запросу записи требуемого класса.

Поле *Time_to_live* (время жизни) указывает, насколько стабильно состояние записи. Редко меняющимся данным присваивается высокое значение этого поля, например, 86 400 (число секунд в сутках). Непостоянная информация помечается небольшим значением, например, 60 (1 минута). Мы вернемся к этому вопросу позднее, когда будем обсуждать кэширование.

Третьим полем каждой записи является поле *Class* (класс). Для информации Интернета значение этого поля всегда равно *IN*. Для прочей информации применяются другие коды, однако на практике они встречаются редко.

Поле *Type* (тип) означает тип записи. Наиболее важные типы записей перечислены в табл. 7.1.

Таблица 7.1. Основные типы записей ресурсов DNS для IPv4

Тип	Смысл	Значение
SOA	Начальная запись зоны	Параметры для этой зоны
A	IP-адрес хоста	32-разрядное целое число
MX	Обмен почтой	Приоритет, с которым домен желает принимать электронную почту
NS	Сервер имен	Имя сервера для этого домена
CNAME	Каноническое имя	Имя домена
PTR	Указатель	Псевдоним IP-адреса
HINFO	Описание хоста	Описание центрального процессора и ОС в виде ASCII-текста
TXT	Текст	Не интерпретируемый ASCII-текст

Запись *SOA* (Start Of Authority — начальная точка полномочий) сообщает имя первичного источника информации о зоне сервера имен (описанного ниже), адрес электронной почты его администратора, уникальный порядковый номер, различные флаги и тайм-ауты.

Самой важной является запись *A* (Address — адрес). Она содержит 32-разрядный IP-адрес хоста. У каждого хоста в Интернете должен быть по меньшей мере один IP-адрес, чтобы другие машины могли с ним общаться. На некоторых хостах может быть одновременно установлено несколько сетевых соединений. В этом случае им требуется по одной записи типа *A* для каждого сетевого соединения (для каждого IP-адреса). DNS можно настроить на циклический перебор этих записей, чтобы в ответ на первый запрос возвращалась первая запись, в ответ на второй запрос — вторая запись, и т. д.

Следующей по важности является запись *MX*. В ней указывается имя хоста, готового принимать почту для указанного домена. Дело в том, что не каждая машина может заниматься приемом почты. Если кто-нибудь хочет послать письмо на адрес, например, bill@microsoft.com, то отправляющему хосту нужно будет вначале найти почтовый сервер на microsoft.com. Запись *MX* может помочь в этих поисках.

Записи *NS* содержат информацию о серверах имен. Например, в каждой базе данных DNS содержится *NS*-запись для каждого домена верхнего уровня, что позволяет пересылать электронную почту на удаленные участки дерева имен. Позднее мы вернемся к этому вопросу.

Записи *CNAME* позволяют создавать псевдонимы. Представим себе, что человек, знакомый в общих чертах с формированием имен в Интернете, хочет по-

слать сообщение человеку с регистрационным именем *paul* на отделении компьютерных наук Массачусетского технологического института (M.I.T.). Он может попытаться угадать нужный ему адрес, составив строку *paul@cs.mit.edu*. Однако этот адрес работать не будет, так как домен отделения компьютерных наук Массачусетского технологического института на самом деле называется *lcs.mit.edu*. Таким образом, для удобства тех, кто этого не знает, M.I.T. может создать запись *CNAME*, позволяющую обращаться к нужному домену по обоим именам. Такая запись будет иметь следующий вид:

```
cs.mit.edu 86400 IN CNAME lcs.mit.edu
```

Как и *CNAME*, запись PTR указывает на другое имя. Однако в отличие от записи *CNAME*, являющейся, по сути, макроопределением, PTR представляет собой регулярный тип данных DNS, интерпретация которого зависит от контекста. На практике запись PTR почти всегда используется для ассоциации имени с IP-адресом, что позволяет по IP-адресу находить имя соответствующей машины. Это называется **обратным поиском**.

Запись *HINFO* позволяет определять тип машины и операционной системы, которой соответствует домен. Наконец, *TXT*-записи позволяют доменам идентифицировать себя произвольным образом. Оба эти типа записей разработаны для удобства пользователей. Ни один из них не является обязательным, поэтому рассчитывать на их наличие не следует, особенно при обработке записей программами (тем более что программы практически невозможно научить обрабатывать эти текстовые данные).

Наконец, последнее поле записи ресурса — это поле *Value* (значение). Это поле может быть числом, именем домена или текстовой ASCII-строкой. Смысл поля зависит от типа записи. Краткое описание поля *Value* для каждого из основных типов записей дано в табл. 7.1.

Пример информации, хранящейся в базе данных DNS домена, приведен в листинге 7.1. В нем показана часть (почти что гипотетической) базы данных домена *cs.vu.nl*, представленного также в виде узла дерева доменов на рис. 7.1. В базе данных содержится семь типов записей ресурсов.

Листинг 7.1. Часть возможной базы данных домена cs.vu.nl

```
; Официальная информация для cs.vu.nl
cs.vu.nl. 86400 IN SOA star boss (952771,7200,7200,2419200,86400)
cs.vu.nl. 86400 IN TXT "Faculteit Wiskunde en Informatica."
cs.vu.nl. 86400 IN TXT "Vrije Universiteit Amsterdam."
cs.vu.nl. 86400 IN MX 1 zephyr.cs.vu.nl.
cs.vu.nl. 86400 IN MX 2 top.cs.vu.nl.
flits.cs.vu.nl. 86400 IN HINFO Sun Unix
flits.cs.vu.nl. 86400 IN A 130.37.16.112
flits.cs.vu.nl. 86400 IN A 192.31.231.165
flits.cs.vu.nl. 86400 IN MX 1 flits.cs.vu.nl.
flits.cs.vu.nl. 86400 IN MX 2 zephyr.cs.vu.nl.
flits.cs.vu.nl. 86400 IN MX 3 top.cs.vu.nl.
www.cs.vu.nl.86400 IN CNAME star.cs.vu.nl
ftp.cs.vu.nl. 86400 IN CNAME zephyr.cs.vu.nl
rowboat IN A 130.37.56.201
IN MX 1 rowboat
IN MX 2 zephyr
IN HINFO Sun Unix
little-sister IN A 130.37.62.23
IN HINFO Mac MacOS
laserjet IN A 192.31.231.216
IN HINFO "HP Laserjet IIISi" Proprietary
```


В первой строке листинга, не являющейся комментарием, дается основная информация о домене, которая в дальнейшем нас интересовать не будет. В следующих двух строках приводится текстовая информация об организации, которой принадлежит домен. Следующие две строки определяют два хоста, с которыми следует связаться в первую очередь при попытке доставить электронную почту, посланную по адресу person@cs.vu.nl. Хост по имени *zephyr* (специальная машина) следует опросить первым. В случае неудачи следует попробовать доставить письмо машине по имени *top*.

После пустой строки, добавленной для удобства чтения, следуют строки, сообщающие о том, что хост *flits* является рабочей станцией Sun, работающей под управлением операционной системы UNIX, а также даются оба ее IP-адреса. Следующие три строки указывают хосты, которым следует доставлять письма, посылаемые по адресу *flits.cs.vu.nl*. В первую очередь, естественно, следует пытаться доставить письмо самому компьютеру *flits*. Но если этот хост выключен, следует продолжать попытки, обращаясь к хостам *zephyr* и *top*. Следом указаны псевдонимы *www.cs.vu.nl* и *ftp.cs.vu.nl*, позволяющие домену *cs.vu.nl* изменять свой WWW и FTP-серверы, не меняя адресов, по которым пользователи смогут продолжать к ним обращаться.

Следующие четыре строки содержат обычные записи для рабочих станций, в данном случае для *rowboat.cs.vu.nl*. Хранящаяся в базе данных информация содержит IP-адрес, имена первого и второго хостов для доставки почты и информацию о машине. Следом идут две записи о машине *little-sister*, работающей под управлением системы MacOS, отличной от системы UNIX (поэтому эта машина не может сама получать электронную почту). Последние две строки описывают лазерный принтер, подключенный к Интернету.

В этом файле нет IP-адресов доменов верхнего уровня, так как они не принадлежат домену *cs.vu.nl*. Эти адреса поставляются корневыми серверами, чьи IP-адреса присутствуют в файле конфигурации системы и загружаются в DNS-кэш, когда загружается DNS-сервер. Существует около дюжины корневых серверов по всему миру, и каждый из них знает IP-адреса всех серверов доменов верхнего уровня. То есть если машине известен IP-адрес хотя бы одного корневого сервера, она может узнать любое имя DNS.

Серверы имен

Теоретически один сервер мог бы содержать всю базу данных DNS и отвечать на все запросы к ней. На практике этот сервер оказался бы настолько перегруженным, что был бы просто бесполезным. Более того, если бы с ним когда-нибудь что-нибудь случилось, то весь Интернет не работал бы.

Чтобы избежать проблем, связанных с хранением всей информации в одном месте, пространство имен DNS разделено на непересекающиеся **зоны**. Один возможный способ разделения пространства имен, показанного на рис. 7.1, на зоны, изображен на рис. 7.2. Каждая зона содержит часть общего дерева доменов, а также в нее входят серверы имен, хранящие управляющую информацию об этой зоне. Обычно в каждой зоне находится один основной сервер зоны, получающий информацию из файла на своем диске, и несколько дополнительных серверов имен, которые получают информацию от основного сервера имен. Для большей надежности некоторые серверы, обслуживающие зону, могут находиться за пределами самой зоны.

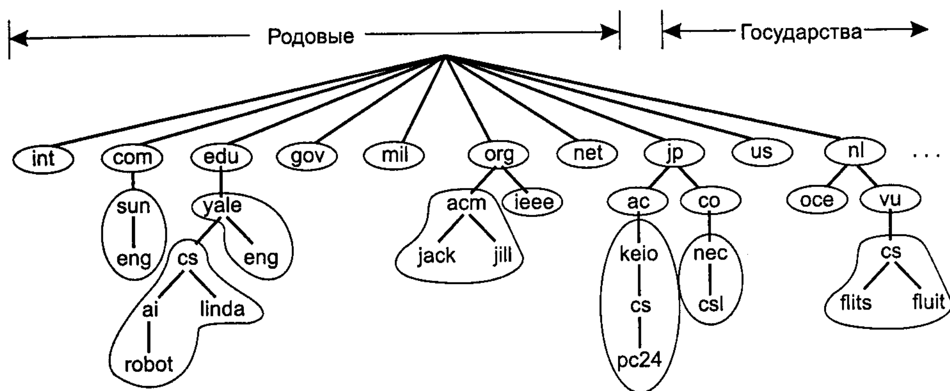


Рис. 7.2. Часть пространства имен DNS, разделенная на зоны

Расстановка границ зон целиком зависит от администратора зоны. Это решение основывается на том, сколько серверов имен требуется в той или иной зоне. Например, на рис. 7.2 у Йельского университета есть сервер для *yale.edu*, управляющий доменом *eng.yale.edu*, но не доменом *cs.yale.edu*, расположенным в отдельной зоне со своими серверами имен. Подобное решение может быть принято, когда факультет английского языка не хочет управлять собственным сервером имен, но этого хочет факультет компьютерных наук. Соответственно, домен *cs.yale.edu* выделен в отдельную зону, а домен *eng.yale.edu* — нет.

Распознаватель обращается с запросом разрешения имени домена к одному из локальных серверов имен. Если искомый домен относится к сфере ответственности данного сервера имен, как, например, домен *ai.cs.yale.edu* подпадает под юрисдикцию домена *cs.yale.edu*, тогда данный DNS-сервер сам отвечает распознавателю на его запрос, передавая ему **авторитетную запись** ресурса. Авторитетной называют запись, получаемую от официального источника, хранящего данную запись и управляющего ее состоянием. Поэтому такая запись всегда считается верной, в отличие от кэшируемых записей, которые могут устаревать.

Однако если домен удаленный, и информацию о запрашиваемом домене нельзя получить от данного сервера имен, последний посылает сообщение с запросом серверу домена верхнего уровня запрашиваемого домена. Поясним данный процесс на примере, показанном на рис. 7.3. В данном случае распознаватель на компьютере *flits.cs.vu.nl* хочет узнать IP-адрес хоста *linda.cs.yale.edu*. На первом шаге он посылает запрос локальному серверу имен, *cs.vu.nl*. Этот запрос содержит имя искомого домена, тип (*A*) и класс (*IN*).

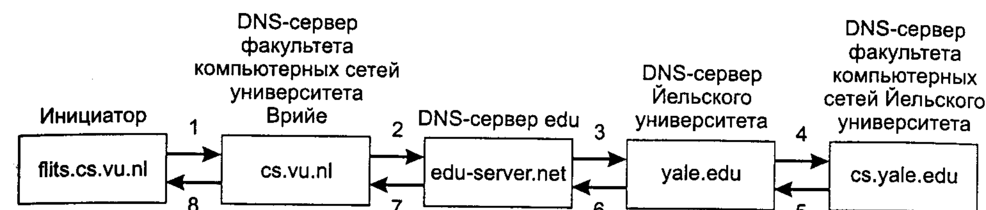


Рис. 7.3. Восемь этапов поиска распознавателем имени удаленного хоста

Предположим, что локальный сервер имен никогда ранее не получал запроса об этом домене и поэтому ничего о нем не знает. Он может опросить несколько находящихся рядом серверов имен, но если они также не знают ответа, данный локальный сервер посылает UDP-пакет серверу домена *edu*, адрес которого содержится в его базе данных. Как видно из рисунка, его имя *edu-server.net*. Маловероятно, чтобы этот сервер знал адрес хоста *linda.cs.yale.edu*. Скорее всего, он даже не знает адреса сервера *cs.yale.edu*, однако он должен знать все свои дочерние домены, поэтому он направляет запрос серверу имен домена *yale.edu* (шаг 3). Тот, в свою очередь, пересылает этот запрос серверу *cs.yale.edu* (шаг 4), у которого должны быть требуемые авторитетные записи ресурсов. Поскольку каждый запрос был от клиента к серверу, посылаемые в ответ записи ресурсов проделывают тот же путь в обратном направлении (шаги 5–8).

Когда записи ресурсов попадают на сервер имен *cs.vu.nl*, они помещаются в кэш на случай, если они понадобятся еще раз. Однако эта информация не является авторитетной, так как изменения в домене *cs.yale.edu* не будут распространяться автоматически на все кэши, в которых может храниться копия этой информации. По этой причине записи кэша обычно долго не живут. В каждой записи ресурса присутствует поле *Time_to_live*. Оно сообщает удаленным серверам, насколько долго следует хранить эту запись в кэше. Если какая-либо машина сохраняет постоянный адрес годами, возможно, будет достаточно надежно хранить эту информацию в кэше в течение одного дня. Для более непостоянной информации, вероятно, более осмотрительно удалять все записи через несколько секунд или одну минуту[1].

Следует отметить, что серия запросов, описанная выше, называется **рекурсивным запросом**, так как каждый сервер, который не владеет запрашиваемой информацией, передает запрос дальше, а затем ответ, также поэтапно, пересылается обратно. Возможны и другие схемы реализации запросов. Так, например,

[1] Практика показывает, что удобнее, наоборот, хранить эту информацию на своем хосте, для чего существуют специальные программы, поддерживающие мини-базы DNS для часто используемых адресов на компьютерах пользователя. — *Примеч. перев.*

возможна форма запроса, в которой в случае отсутствия требуемой информации на локальном сервере клиент сразу получает сообщение о неуспешном выполнении запроса, но при этом ему сообщается имя следующего сервера, который можно об этом спросить. Такая процедура, дающая клиенту больше контроля над процессом поиска, реализована на некоторых серверах. Некоторые серверы вообще не выполняют рекурсивных запросов, а всегда возвращают имя сервера для следующего запроса.

Также следует сказать, что когда DNS-клиент не получает ответа на посланный запрос, он обычно пытается получить искомую информацию у другого сервера, прежде чем истечет период ожидания. Это делается исходя из предположения о том, что сервер, которому адресовался запрос, может в данный момент оказаться выключенным (а не о том, что запрос или ответ потерялся).

Хотя система DNS чрезвычайно важна для обеспечения корректной работы Интернета, основная ее задача заключается в отображении текстовых имен машин на пространство их IP-адресов. Она не помогает найти людей, ресурсы, услуги или другие объекты. Для этого существует иная услуга, называемая **LDAP** (Light-weight Directory Access Protocol — облегченный протокол службы каталогов). Это упрощенная версия стандартной службы каталогов OSI X.500, описание LDAP содержится в RFC 2251. Информация организуется в древовидной форме, и по этому дереву можно осуществлять поиск различных компонентов. Это напоминает телефонную книгу типа «белые страницы». Мы не будем более возвращаться к этой системе, а дополнительную информацию все желающие могут найти в (Weltman и Dahbura, 2000).

Электронная почта

Электронная почта, или **e-mail**, как называют ее многочисленные любители, существует уже более двух десятилетий. До 1990 года она использовалась преимущественно в научных организациях. В 90-е годы она получила широкую известность, и с тех пор количество отправляемых с помощью электронной почты писем стало расти экспоненциально. Среднее число сообщений, посылаемых ежедневно, скоро во много раз превзошло число писем, отправляемых с помощью обычной, бумажной почты.

Электронной почте, как и любой форме коммуникаций, присущ определенный стиль и набор соглашений. В частности, общение по электронной почте носит очень неформальный и демократичный характер. Скажем, человек, который никогда бы не осмелился позвонить или даже написать бумажное письмо какой-нибудь Особо Важной Персоне, запросто может сесть и написать ей небрежное электронное сообщение.

В электронной почте люди обожают использовать особый жаргон и сокращения, такие как BTW (By The Way — между прочим), ROTFL (Rolling On The Floor Laughing — катаюсь по полу от смеха), IMHO (In My Humble Opinion — по моему скромному мнению) и т. д. Кроме того, чрезвычайно популярны так называемые **смайлики**, или **эмотиконы**. Самые интересные из них представлены в табл. 7.2. Чтобы понять, в чем состоит их смысл, бывает полезно повернуть

книгу на 90° по часовой стрелке (или голову — на 90° против часовой стрелки). Существует брошюра (Sanderson и Dougherty, 1993), в которой перечислено свыше 650 смайликов.

Таблица 7.2. Некоторые смайлики. Их не надо зубрить перед выпускным экзаменом :-)

Смайлик	Значение	Смайлик	Значение	Смайлик	Значение
:-)	Не воспринимай всерьез	=:-)	Г-н Линкольн	:+)	Большой нос
:-(	Я зол/огорчен	=):-)	Дядя Сэм	:-))	Двойной подбородок
:-|	Мне безразлично это	*<:-)	Дед Мороз	:-{)	С усами
;-)	Подмигиваю	<:-(	Болван	#:-)	Взъерошенные волосы
:(O)	Громко кричу	(-:	Австралиец (левша)	8-)	Носит очки
:-(*)	Мне тошно от этого	:-)X	С бабочкой	C:-)	Очень умный

Первые системы электронной почты состояли просто из протоколов передачи файлов и договоренности указывать адрес получателя в первой строке каждого сообщения (то есть файла). Со временем недостатки данного метода стали очевидны. Перечислим некоторые из них:

1. Было очень неудобно отсылать сообщения группе получателей. Эта возможность часто требовалась менеджерам для рассылки уведомлений своим подчиненным.
2. Сообщения не обладали внутренней структурой, что затрудняло их компьютерную обработку. Например, если переадресованное сообщение было помещено в тело другого сообщения, извлечь одно сообщение из другого было довольно сложно.
3. Отправитель сообщения никогда не знал, дошло ли его сообщение до адресата.
4. Если кто-либо собирался уехать на несколько недель по делам и хотел, чтобы вся его почта переправлялась его секретарю, организовать это было непросто.
5. Интерфейс пользователя практически отсутствовал. Пользователь должен был сначала в текстовом редакторе набрать сообщение, затем выйти из редактора и запустить программу передачи файла.
6. Было невозможно создавать и посылать сообщения, содержащие смесь текста, изображений (рисунков, факсов, фото) и звука.

Со временем, когда накопился опыт работы, были предложены более сложные системы электронной почты. В 1982 году предложения по работе с электронной почтой, выдвинутые администрацией сети ARPANET, были опубликованы в виде RFC 821 (протокол передачи) и RFC 822 (формат сообщений). Подверг-

шись минимальным изменениям, нашедшим отражение в RFC 2821 и 2822, они фактически стали стандартами Интернета, однако все равно, говоря об электронной почте Интернета, многие ссылаются на RFC 822.

В 1984 году консультативный комитет по международной телефонии и телеграфу (CCITT) впервые представил стандарт X.400. После двух десятков лет борьбы электронная почта, основанная на стандарте RFC 822, получила широкое применение, тогда как системы, базирующиеся на стандарте X.400, практически исчезли. Получилось так, что система, созданная горсткой аспирантов-компьютерщиков, смогла превзойти официальный международный стандарт, имевший серьезную поддержку всех управлений почтово-телеграфной и телефонной связи во всем мире, многих правительств и значительной части компьютерной промышленности. Все это напоминает библейскую историю о Давиде и Голиафе.

Причина успеха стандарта RFC 822 кроется не столько в его достоинствах, сколько в недостатках стандарта X.400. Последний был настолько плохо продуман и так сложен, что никто просто не мог его нормально реализовать. Большинство организаций предпочли простую, но работающую систему электронной почты, основанную на RFC 822, возможно, действительно замечательной, но неработающей системе, в основе которой был стандарт X.400. Может быть, это когда-нибудь послужит кому-нибудь уроком. Итак, далее в нашем обсуждении мы сконцентрируемся на используемых в Интернете системах электронной почты.

Архитектура и службы

В данном разделе мы рассмотрим возможности и организацию систем электронной почты. Обычно они состоят из двух подсистем: **пользовательских агентов**, позволяющих пользователям читать и отправлять электронную почту, и **агентов передачи сообщений**, пересылающих сообщения от отправителя к получателю. Пользовательские агенты представляют собой локальные программы, предоставляющие различные методы взаимодействия пользователя с почтовой системой. Эти методы (или интерфейсы) могут быть командными, графическими или основанными на меню. Агенты передачи сообщений обычно являются системными демонами, работающими в фоновом режиме и перемещающими электронную почту по системе.

Обычно системами электронной почты поддерживаются следующие пять основных функций.

Составление — процесс создания сообщений и ответов. Хотя для создания тела сообщения можно использовать любой текстовый редактор, система поможет в составлении адреса и многочисленных полей заголовков, добавляемых к каждому сообщению. Например, при составлении ответа на сообщение система электронной почты может извлечь адрес отправителя из полученного письма и автоматически поместить его в нужное место в ответе.

Передача — перемещение сообщений от отправителя к получателю. Для этого требуется установить соединение с адресатом или с какой-либо промежуточной машиной, переслать сообщение и разорвать соединение. Система электронной

почты должна выполнять все эти действия автоматически, не беспокоя пользователя.

Уведомление — информирование отправителя о состоянии сообщения. Что с ним стало? Доставлено оно, потеряно или отвергнуто? Существует множество приложений, в которых подтверждение доставки имеет большую важность и даже может иметь юридическую значимость («Ваша честь, моя электронная система не очень надежна, поэтому я полагаю, что повестка с вызовом в суд просто где-то потерялась»).

Отображение приходящих сообщений на экране необходимо, чтобы пользователи имели возможность читать свою электронную почту. Иногда требуется преобразование текста или вызов специальной программы просмотра, например, для просмотра файла формата PostScript или прослушивания оцифрованного звукового сообщения. Иногда также применяются простые преобразования и форматирование текста.

Наконец, на последнем этапе работы с электронным письмом решается дальнейшая судьба полученного сообщения. Получатель может удалить его, не читая, удалить после прочтения, сохранить и т. д. Также должна быть обеспечена возможность найти полученное ранее письмо и прочитать его еще раз, переслать его другому адресату или обрабатывать полученную почту другим способом.

Помимо этих пяти основных услуг большинство систем электронной почты предоставляют великое множество дополнительных функций. Перечислим кратко некоторые из них. Когда пользователи переезжают с места на место или отсутствуют в течение некоторого срока, им может понадобиться автоматическая переадресация их почты.

Большинство систем позволяют пользователям создавать **почтовые ящики** для хранения приходящей почты. Для работы с почтовыми ящиками нужны специальные команды, позволяющие создавать и удалять почтовые ящики, исследовать их содержимое, добавлять и удалять сообщения и т. д.

Менеджерам корпораций часто бывает нужно послать сообщение всем своим подчиненным, заказчикам или поставщикам. Для облегчения выполнения этой задачи применяются **списки рассылки**, представляющие собой список адресов электронной почты. При отправлении сообщения с помощью списка рассылки всем адресатам, числящимся в этом списке, посылаются идентичные копии сообщения.

Среди других полезных дополнительных функций можно перечислить следующие: рассылка копий писем «под копирку» (Carbon copy), рассылка копий без уведомления о других получателях (Blind carbon copy), письма с высоким приоритетом, секретная (то есть зашифрованная) почта, возможность доставки письма альтернативному получателю, если основной временно недоступен, а также возможность перепоручать обработку почты секретарям.

Сегодня электронная почта получила широкое применение в промышленности для обмена информацией внутри компании, что делает возможным совместную работу над сложными проектами далеко удаленных друг от друга (возможно, даже находящихся в различных временных зонах) сотрудников. Более того, общение по электронной почте устраняет концентрацию внимания на различиях

в положении, возрасте и поле, часто мешающих непредвзятому рассмотрению идей. Блестящая идея, посланная студентом-практикантом по электронной почте, имеет шанс выиграть у идеи не столь блестящей, но высказанной вице-президентом компании.

В основе всех современных систем электронной почты лежит ключевая идея о разграничении **конверта** и содержимого письма. Конверт заключает в себе сообщение. Он содержит всю информацию, необходимую для доставки сообщения — адрес получателя, приоритет, уровень секретности и т. п. Все эти сведения отделены от самого сообщения. Агенты передачи сообщений используют конверт для маршрутизации, аналогично тому, как это делает обычная почтовая служба.

Сообщение внутри конверта состоит из двух частей: **заголовка** и **тела письма**. Заголовок содержит управляющую информацию для пользовательских агентов. Тело письма целиком предназначается для человека-получателя. Примеры конвертов и сообщений показаны на рис. 7.4.

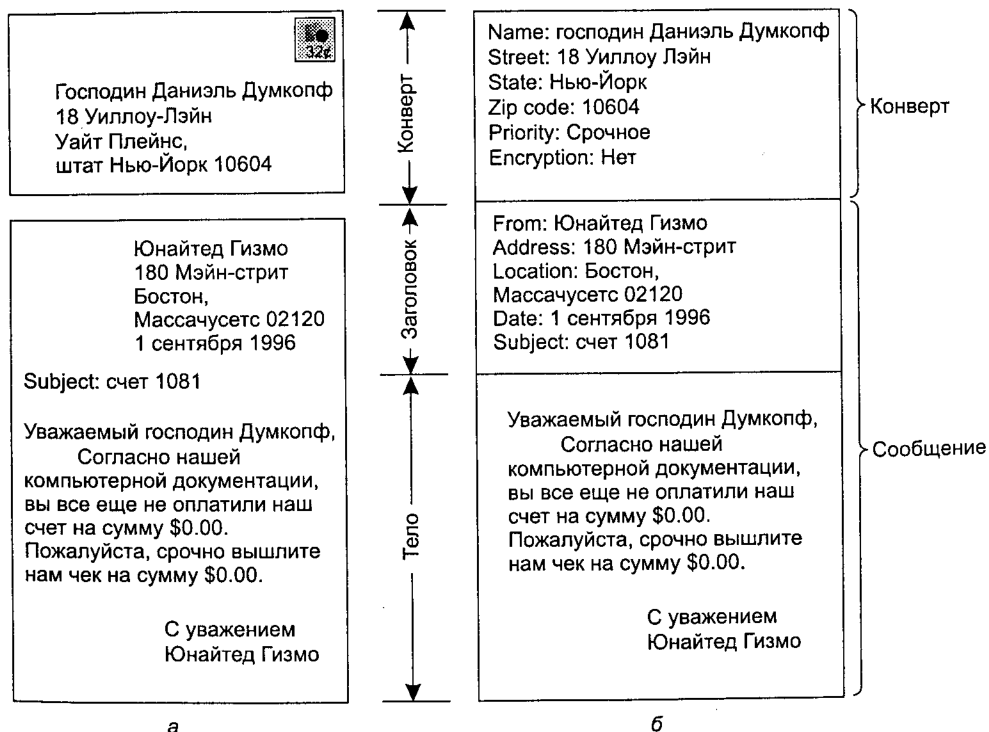


Рис. 7.4. Конверты и сообщения: обычное письмо (*а*); электронное письмо (*б*)

Пользовательский агент

Как уже было показано, системы электронной почты состоят из двух основных частей: пользовательских агентов и агентов передачи сообщений. В данном разделе мы рассмотрим пользовательских агентов. Как правило, пользовательский агент — это программа (иногда называемая почтовым редактором), управляемая множеством команд для составления и получения сообщений, а также для ответа на сообщения и управления почтовыми ящиками. Некоторые пользовательские агенты снабжены причудливым интерфейсом с использованием меню или графическим интерфейсом, для работы с которым требуется мышь, тогда как другие управляются односимвольными командами, вводимыми с клавиатуры. Функционально все они примерно одинаковы. В некоторых из них есть меню или ярлыки, но для основных действий обычно существуют комбинации клавиш.

Отправление электронной почты

Чтобы послать письмо электронной почтой, пользователь должен составить текст сообщения, указать адрес получателя, а также, возможно, некоторые дополнительные параметры. Текст сообщения может быть набран в отдельном текстовом редакторе, с помощью программы изготовления документов или в текстовом редакторе, встроенном в почтовый редактор. Адрес получателя должен быть указан в формате, понятном для пользовательского агента. Многие пользовательские агенты ожидают DNS-адреса вида *пользователь@DNS-адрес*. Система доменных имен DNS уже рассматривалась ранее в этой главе, поэтому сейчас мы не станем подробно останавливаться на этом вопросе.

Однако следует отметить, что существуют и другие формы адресации. В частности, адреса стандарта X.400 абсолютно не похожи на DNS-адреса и состоят из пар *атрибут* = значение, разделенных косыми чертами, например:

```
/C=US/ST=MASSACHUSETTS/L=CAMBRIDGE/PA=360 MEMORIAL DR./CN=KEN SMITH/
```

В этом адресе указаны государство, штат, местоположение, личный адрес и имя получателя (Ken Smith). Возможно также использование различных других атрибутов, что делает возможным отправку электронного письма человеку, чьего имени вы не знаете, при условии, что вам известны другие атрибуты (например, название компании и должность получателя). Хотя форма адресации X.400 значительно менее удобна, чем DNS, большинством систем электронной почты поддерживаются так называемые **псевдонимы**, с помощью которых пользователи могут вводить или выбирать имя адресата и получать корректный электронный адрес. Поэтому даже при использовании адресации X.400 пользователю не приходится вводить такие, мягко говоря, странные строки.

Большинством систем электронной почты поддерживаются списки рассылки, позволяющие пользователю рассылать одно и то же сообщение группе получателей при помощи одной команды. Если список рассылки хранится локально, пользовательский агент может просто послать каждому получателю отдельное сообщение. Однако при удаленном расположении списка рассылки сообщения будут тиражироваться там, где хранится список. Допустим, у группы исследователей птиц есть список рассылки, называющийся *birders* (птицеловы), установленный на домене *meadowlark.arizona.edu*. Тогда любое сообщение, посланное по адресу *birders@meadowlark.arizona.edu*, будет сначала отправляться в университет штата Аризона, а затем рассылаться оттуда в виде отдельных сообщений всем членам списка рассылки, где бы они ни находились. Для отправителя письма список