Мак Мастан

Создание сетевых приложений в среде Linux®

Руководство разработчика

Освойте концепции

- Сетевое/распределенное программирование
- Сетевые клиенты, серверы и одноранговые узлы
- Управление вводом-выводом и многозадачность
- Направленные, широковещательные и групповые сообщения
- Безопасность и протокол SSL (Secure Sockets Layer)

Научитесь создавать

- Клиенты и серверы (простые, многозадачные и многопотоковые)
- Приложения OpenSSL, взаимодействующие с коммерческими системами
- Программы, использующие удаленные вызовы процедур (технология RPC)
- Утилиты групповой и широковещательной доставки сообщений

SAMS
ВИЛЬЯМС

Создание сетевых приложений в среде Linux

Руководство разработчика

Шон Уолтон

Москва • Санкт-Петербург • Киев
2001

ББК 32.973.26-018.2.75

УДК 681.3.07

Издательский дом "Вильяме"

По общим вопросам обращайтесь в Издательский дом "Вильяме"
по адресу: info@williamspublishing.com, http://www.williamspublishing.com

Уолтон, Шон.

Создание сетевых приложений в среде Linux. : Пер. с англ.— М. : Издательский дом "Вильяме", 2001. — 464 с.: ил. — Парал: тит. англ.

ISBN 5-8459-0193-6 (рус.)

Данная книга в основном посвящена программированию сокетов на языке C в среде Linux. В ней шаг за шагом рассказывается о том, как писать профессиональные сетевые клиентские, серверные и одноранговые приложения. Читатель узнает, как работать с существующими клиент-серверными протоколами (в частности, HTTP), взаимодействовать с другими компьютерами по протоколу UDP и создавать свои собственные протоколы. В книге описываются все типы пакетов, поддерживаемых в семействе протоколов TCP/IP, их достоинства и недостатки.

Помимо базового материала представлены сведения о различных методиках многозадачности, рассказывается о средствах управления вводом-выводом и обеспечения безопасности сетевых приложений. В книге также описываются объектно-ориентированные подходы к сетевому программированию на языках Java и C++. Отдельные главы посвящены технологии RPC, протоколу SSL, работе в групповом и широковещательном режимах и стандарту IPv6.

Книга предназначена профессиональным программистам и студентам, которые хотят научиться создавать не только линейные алгоритмы, но и полнофункциональные многозадачные сетевые приложения.

ББК 32.973.26-018.2.75

ISBN 5-8459-0193-6 (рус.)
ISBN 0-672-31935-7 (англ.)

Оглавление

Об авторе

Шон Уолтон получил степень магистра компьютерных наук в 1990 г. в университете Бригема Янга, штат Юта, специализируясь в области теории языков программирования и многозадачности. В 1988 г. он начал работать на факультете вычислительной техники этого же университета в качестве консультанта по разработке теории и методов управления транспьютерами. Перейдя на должность системного администратора факультета, он впервые столкнулся с программированием BSD-сокетов. Впоследствии, будучи сотрудником компании Hewlett-Packard, он разработал средство автоматического распознавания языка (PostScript и PCL), применяемое в принтерах LaserJet 4 и более поздних моделей. Шон также создал микрооперационную систему реального времени для микроконтроллера 8052, которая эмулировала процессоры принтеров.

Шон много лет профессионально занимался программированием и администрированием UNIX, включая операционные системы Linux, HP-UX, Ultrix, SunOS и System V. Работая со столь разнообразными системами, он сосредоточивал свои усилия на создании методик системно-независимого программирования, позволявших писать приложения, без труда переносимые в любую из систем.

В последние несколько лет Шон вел курсы и читал лекции по основам вычислительной техники, управлению проектами, объектно-ориентированному проектированию, языкам Java и C++. В начале 1998 г. он начал заниматься программированием сокетов в Java и позднее включил этот предмет в свой курс лекций по Java. В настоящее время он сотрудничает с компанией Nationwide Enterprise и параллельно занимается преподавательской деятельностью.

Благодарности

Эта книга не появилась бы на свет без помощи нескольких людей. В первую очередь, это моя любимая жена Сьюзан, которая служила мне музой и источником вдохновения. Во-вторых, это мой отец Уэндел, который пытался учить меня организованности и правильному изложению мыслей. В-третьих, это само сообщество Linux, благодаря которому я получил эффективную, надежную операционную систему, давшую мне материал для работы. Наконец, я сам не понимал, насколько эта работа важна для меня в профессиональном плане, пока мои друзья Чарльз Кнутсон и Майк Хольстейн не дали ей свою высокую оценку.

Расширение опыта программирования

Большинство задач, которые приходится решать программисту в повседневной жизни, связано с выполнением локальных действий в системе. Обычно они не выходят за рамки управления мышью, клавиатурой, экраном и файловой системой. Гораздо труднее заставить взаимодействовать несколько программ, которые работают на разных компьютерах, подключенных к сети. Сетевое программирование, несомненно, обогатит ваш опыт, так как вам придется координировать работу множества компонентов приложения и тщательно распределять обязанности между ними.

Фундаментальной единицей всего сетевого программирования в Linux (и большинстве других операционных систем) является сокет. Так же, как функции файлового ввода-вывода определяют интерфейс взаимодействия с файловой системой, сокет соединяет программу с сетью. С его помощью она посылает и принимает сообщения.

Создавать сетевые приложения труднее, чем даже заниматься многозадачным программированием, так как круг возникающих проблем здесь гораздо шире. Необходимо обеспечивать максимальную производительность, координировать обмен данными и управлять вводом-выводом.

В книге рассматривается множество методик и способов решения этих проблем. Она содержит как ответы на конкретные вопросы, так и описание общих стратегий построения профессиональных сетевых программ.

Структура книги

Книга состоит из пяти частей, каждая из которых связана с предыдущим материалом.

- Часть I, "Создание сетевых клиентских приложений"

  Представлены вводные сведения о сокетах, определены основные термины, описаны различные типы сокетов и принципы адресации, изложена базовая теория сетей.

- Часть II, "Создание серверных приложений"

  Рассматриваются серверные технологии, алгоритмы многозадачности, механизмы ввода-вывода и параметры сокетов.

- Часть III, "Объектно-ориентированные сокеты

  Язык C — это не единственный язык программирования сокетов. В этой части рассматриваются объектно-ориентированные подходы к сетевому программированию и описываются достоинства и недостатки объектной технологии в целом.

- Часть IV, "Сложные сетевые методики"

  Рассматриваются сложные методики сетевого программирования, включая безопасность, широковещание и групповое вещание, стандарт IPv6 и низкоуровневые сокеты.

- Часть V,"Приложения"

  В приложения вынесен справочный материал, касающийся сокетов. В приложение А включены таблицы, которые слишком велики для основных глав. В приложениях Б и В описаны системные функции работы с сокетами и функции ядра.

На сопровождающем Web-узле содержатся исходные тексты всех примеров книги, приложения в формате HTML и PDF, а также различные RFC-документы в формате HTML.

Предпочтения профессионалов

Данная книга предназначена для профессиональных программистов. Обычно профессионалам нужно одно из трех: либо изучить новую технологию, либо решить конкретную задачу, либо найти нужные примеры или определения. Главы и приложения книги построены с учетом этих потребностей.

- *Последовательное чтение от начала до конца.* В любом месте книги раскрывается какая-то одна методика. Каждый раздел и глава основаны на предыдущем материале. Вся книга организована по принципу нарастания сложности. Программирование сокетов может показаться поначалу довольно простой задачей. Однако с ним связано множество "подводных камней" и нюансов, пренебрежение которыми приведет к тому, что отличная программа на самом деле окажется неработоспособной.
- *Принцип обзора.* Человек, читающий книгу, хочет быстро получить интересующую его информацию, не останавливаясь на чрезмерных деталях. Основные сведения у него уже есть, и он хочет сразу перейти к главному. Читатель может пропускать разделы до тех пор, пока не найдет то, что ему нужно.
- *Справочный материал.* Экспертам, как правило, требуется быстро найти то, что им нужно (таблицы, фрагменты программ, описания API-функций). Эта информация должна быть лаконичной и легкодоступной.

Многие главы содержат врезки, в которых приводится вспомогательная информация, экспертные замечания и советы.

Что необходимо знать

В книге приводится много примеров программ, иллюстрирующих ту или иную методику. Чтобы работать с ними, необходимо знать следующее:

- как создать исходный файл на языке C или Java (в любом доступном редакторе);
- как скомпилировать полученный файл;
- как запустить скомпилированную программу.

Некоторые примеры требуют модификации ядра (для режимов широковещания и группового вещания). В большинстве дистрибутивов Linux сетевая подсистема инсталлирована и запущена, даже если компьютер не подключен к сети. Чтобы иметь возможность запускать все программы, описанные в книге, необходимо:

уметь создавать и компилировать исходные тексты на языках C, C++ и Java;

иметь в наличии все необходимые компиляторы;

уметь конфигурировать ядро;

уметь компилировать и инсталлировать новое ядро.

Соглашения, используемые в книге

В книге применяются следующие типографские соглашения.

- Примеры программ, названия функций, инструкций и переменных, а также любой текст, который можно ввести или увидеть на экране, набраны моноширинным шрифтом. Полужирный моноширинный шрифт применяется для выделения изменений в примерах.
- Переменные элементы синтаксиса выделены *курсивным моноширинным* шрифтом. Они должны заменяться реальными именами файлов, аргументами функций и т.д.
- *Курсивом* выделены ключевые термины, когда они вводятся впервые.
- В текст иногда включаются ссылки на стандартную документацию по Internet — RFC-документы (Request For Comments — запросы на комментарии). В ссылке указывается номер документа, а сама она заключается в квадратные скобки, например [RFC875].

Часть I

Создание сетевых клиентских приложений

В этой части...

Глава

1 Простейший сетевой клиент

В этой главе...

В темной комнате с плотно затянутыми шторами слабо мерцает экран компьютера. На полу валяются пустые банки из-под пива. На экран уставшими и потускневшими после нескольких бессонных ночей глазами смотрит программист.

— Черт бы побрал этот CMOS! Опять батарея полетела! Который час? Так, звоним в справочную.

— Восемь часов двадцать три минуты и сорок секунд.

— Утра или вечера? Мне что, встать и выглянуть в окно?!

Компьютер, за. которым вы работаете, наверняка подключен к какой-нибудь сети. Это может быть крупная корпоративная сеть с выходом в Internet через прокси-сервер или домашняя микросеть, объединяющая два компьютера в разных комнатах. Сетями соединяются рабочие станции, серверы, принтеры, дисковые массивы, факсы, модемы и т.д. Каждое сетевое соединение использует какие-нибудь ресурсы или само предоставляет их. В некоторых соединениях для получения информации не требуется двустороннее взаимодействие. Подобно описанному выше звонку в справочную, сетевой клиент в простейшем случае просто подключается к серверу и принимает от него данные.

Какие ресурсы, или сервисы, могут предоставляться сервером? Их множество, но все они делятся на четыре категории:

- общие — дисковые ресурсы;
- ограниченные — принтеры, модемы, дисковые массивы;
- совместно используемые — базы данных, программные проекты, документация;
- делегируемые — удаленные программы, распределенные запросы.

В этой главе последовательно рассказывается о том, как написать простейшее клиентское приложение, подключающееся к некоторому серверу. Читатель узнает, что собой представляет процесс написания сетевых программ. Наш клиент первоначально будет обращаться к серверу для того, чтобы определить правильное время (это пример соединения, в котором от клиента не требуется передавать данные на сервер). По ходу главы будут рассмотрены различные функции, их параметры и наиболее часто возникающие ошибки.

Для подключения к серверу клиент должен знать его адрес и предоставить ему свой. Чтобы обмениваться сообщениями независимо от своего местоположения, клиент и сервер используют сокеты. Обратимся еще раз к примеру с телефонным звонком. Телефонная трубка имеет два основных элемента: микрофон (передатчик) и динамик (приемник). А телефонный номер, по сути, представляет собой уникальный адрес трубки.

У сокета имеются такие же два канала: один для прослушивания, а другой для передачи (подобно каналам ввода-вывода в файловой системе). Клиент (звонящий) подключается к серверу (абоненту), чтобы начать сетевой разговор. Каждый участник разговора предлагает несколько стандартных, заранее известных сервисов (см. файл /etc/services), например телефон, по которому можно узнать правильное время.

Выполнение примеров, представленных в книге

Большинство программ, приводимых в книге, можно выполнять, не имея подключения к сети, при условий, что сетевые функции ядра сконфигурированы и демон inetd запущен. В этих программах используется локальный сетевой адрес 127.0.0.1 (так называемый адрес *обратной связи).* Даже если сетевые драйверы отсутствуют, дистрибутивы Linux содержат все необходимые средства для организации, сетевого взаимодействия с использованием адреса Обратной связи.

Клиентская программа должна предпринять несколько действий для установления соединения с другим компьютером или сервером. Причем эти действия следует выполнять в определенном порядке. Конечно, читатель спросит: "А почему нельзя все упростить?" Дело в том, что на каждом из этапов программа может задавать различные опции. Но не пугайтесь: не все действия являются обязательными. Если пропустить некоторые из них, операционная система воспользуется установками по умолчанию.

Базовая последовательность действий имеет такой вид: создание сокета, поиск адресата, организация канала связи с другой программой и разрыв соединения. Ниже в графическом виде представлены действия, которые должен предпринять клиент при подключении к серверу (рис. 1.1).

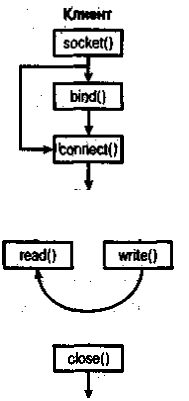

Рис. 1.1. Каждый клиент взаимодействует с операционной системой, вызывая определенные функции в заданном порядке

Опишем каждый из этапов.

1. Создание сокета. Выбор сетевого домена и типа сокета.
2. Задание параметров сокета (необязательно). Поведение сокета регулируется множеством параметров. Пока сокет открыт, программа может менять любой из них (подробно об этом — в главе 9, "Повышение производительности").

3. Привязка к определенному адресу/порту (необязательно). Задание конкретного IP-адреса, а также выбор порта. Если пропустить этот этап, операционная система разрешит связь с любым IP-адресом и назначит произвольный номер порта (подробно об этом — в главе 2, "Основы TCP/IP").

4. Подключение к одноранговому компьютеру/серверу (необязательно). Организация двунаправленного канала связи между клиентской и другой сетевой программой. Если пропустить этот этап, будет создан канал адресной передачи сообщений без установления соединения.

5. Частичный разрыв соединения (необязательно). Выбор одного из двух режимов работы: прием или передача. Этот этап можно выполнить, если создан запасной канал связи.

6. Прием/передача сообщений (необязательно). Этот этап можно пропустить, если требуется всего лишь проверить, доступен ли сервер.

7. Разрыв соединения. Естественно, этот этап важен: долго выполняющиеся программы могут со временем исчерпать лимит дескрипторов файлов, если не закрывать неиспользуемые сеансы.

В следующих параграфах некоторые из этапов описываются подробнее: приводятся примеры и рассматриваются соответствующие системные вызовы.

Связь с окружающим миром посредством сокетов

Несколько лет назад под сетью подразумевался последовательный канал связи между двумя компьютерами. Все компьютеры общались между собой по разным каналам, а для передачи файлов в UNIX применялась система UUCP (UNIX-to-UNIX Copy). С усовершенствованием технологии кабельной передачи данных концепция разделения канала связи стала реальной. Она означала, что каждый компьютер должен был идентифицировать себя уникальным образом и ждать своей очереди для передачи данных. Существуют различные способы совместного использования каналов связи, и многие из них обеспечивают достаточно хорошую производительность. Иногда компьютеры пытаются передавать данные одновременно, в результате чего возникают конфликты пакетов.

За решение подобных проблем и организацию повторной передачи данных отвечают аппаратные и другие низкоуровневые драйверы. Это позволяет программисту сконцентрироваться на решений вопросов приёма и передачи сообщений. Библиотека функций работы с сокетами — Socket API (Application Programming Interface) — является основным инструментом программиста.

Программирование сокетов отличается от прикладного и инструментального программирования, поскольку приходится иметь дело с одновременно выполняющимися программами. Это означает, что требуется дополнительно решать вопросы синхронизации и управления ресурсами.

Сокеты позволяют асинхронно передавать данные через двунаправленный канал. При этом могут возникать различного рода проблемы, например взаимоблокировки процессов и зависания программ. При тщательном проектировании при-

ложений большинство таких проблем вполне можно избежать. О работе в многозадачном режиме рассказывается в главе 7, "Распределение нагрузки: многозадачность", а программирование "надежных" сокетов описывается в главе 10, "Создание устойчивых сокетов".

Обычно перегруженный сервер замедляет работу в сети. Правильная синхронизация процессов и рациональное распределение ресурсов позволяют снизить нагрузку на сервер, повысив пропускную способность сети. Методы повышения производительности рассматриваются в части II, "Создание серверных приложений".

Internet — это сеть с *коммутацией пакетов.* Каждый пакет должен содержать всю необходимую информацию, которая позволит ему достигнуть пункта назначения. Подобно письму, пакет содержит адреса отправителя и получателя. Пакет путешествует от компьютера к компьютеру пр каналам связи *(соединениям).* Если в процессе передачи сообщения происходит разрыв соединения, пакет находит другой маршрут (происходит коммутация) либо маршрутизатор посылает отправителю сообщение об ошибке, информирующее о невозможности обнаружения получателя. Тем самым обеспечивается определенная надежность соединения. Но в любом случае разрывы сети приводят к потерям данных. Читатели наверняка неоднократно с этим сталкивались.

Правила общения: основы адресации в TCP/IP

В сетях применяется множество различных протоколов. Программисты приспособили некоторые протоколы для решения специфических задач, таких как передача данных посредством длинных или ультракоротких волн. Другие протоколы предназначены для повышения надежности сети. Семейство протоколов TCP/IP (Transmission Control Protocol/Internet Protocol) ориентировано на передачу пакетов и выявление нефункционирующих соединений. Если в какой-то момент обнаруживается нарушение сегментации сети, система тут же начинает искать новый маршрут.

Сопровождение пакетов, обнаружение потерь и ретрансляция — это сложные алгоритмы, поскольку в них приходится учитывать множество различных факторов. К счастью, надежность алгоритмов доказана опытом. Обычно в процессе проектирования приложений об этих алгоритмах не вспоминают, поскольку их детали скрыты глубоко в недрах протоколов.

TCP/IP — многоуровневый стек: высокоуровневые протоколы более надежны, но менее гибки, на нижних уровнях повышается гибкость, но за счет надежности. Библиотека Socket API инкапсулирует все необходимые интерфейсы. Это существенный отход от привычной идеологии UNIX, когда за каждым уровнем закреплен собственный набор функций.

Стандартная подсистема функций ввода-вывода также является многоуровневой. Но компьютеры, работающие с TCP/IP, для взаимодействия друг с другом используют почти исключительно сокеты. Это может показаться странным, если учесть, сколько различных протоколов существует, и вспомнить, сколько раз нам говорили о том, что функции `open()` (возвращает дескриптор файла) и `fopen()` (возвращает ссылку на файл) практически несовместимы. В действительности доступ ко всем семействам протоколов (TCP/IP,

IPX, Rose) осуществляется с помощью единственной функции socket(). Она скрывает в себе все детали реализации.

Любой передаваемый пакет содержит в себе данные, адреса отправителя и получателя. Плюс каждый из протоколов добавляет к пакету свою сигнатуру, заголовок и прочую служебную информацию. Эта информация позволяет распространять пакет на том уровне, для которого он предназначен.

Компьютер, подключенный к Internet, обязательно имеет собственный IP-адрес, являющийся уникальным 32-разрядным числом. Если бы адреса не были уникальными, было бы непонятно, куда доставлять пакет.

В TCP/IP концепция адресов расширяется понятием *порта*. Подобно коду города или страны, номер порта добавляется к адресу компьютера. Портов бывает множество, и они не являются физическими сущностями — это абстракции, существующие в рамках операционной системы.

Стандартный формат IP-адреса таков: [0-255].[0-255].[0-255].[0-255], например 123.45.6.78. Значения 0 и 255 являются специальными. Они используются в сетевых масках и в режиме широковещания, поэтому применяйте их с осторожностью (подробнее о деталях IP-адресации рассказывается в главе 2, "Основы TCP/IP"). Номер порта обычно добавляется к адресу через двоеточие:

```
[0-255].[0-255].[0-255].[0-255]:[0-65535]
```

Например, 128.34.26.101:9090 (IP-адрес — 128.34.26.101, порт — 9090)! Но он может добавляться и через точку:

```
[0-255].[0-255].[0-255].[0-255].[0-65535]
```

Например, 64.3.24.24.9999 (IP-адрес — 64.3.24.24, порт — 9999).

Точки и двоеточия в номерах портов

Номер порта чаще отделяется двоеточием, а не точкой.

С каждым IP-адресом может быть связано более 65000 портов, через которые подключаются сокеты (подробно об этом — в главе 2, "Основы TCP/IP").

Прослушивание сервера: простейший алгоритм клиентской программы

Простейшим соединением является то, в котором клиент подключается к серверу, посылает запрос и получает ответ. Некоторые стандартные сервисы даже не требуют наличия запроса, например сервис текущего времени, доступный через порт с номером 13. К сожалению, во многих системах Linux этот сервис по умолчанию недоступен, и чтобы иметь возможность обращаться к нему, требуется модифицировать файл /etc/inetd.conf. Если у вас есть доступ к компьютеру, работающему под управлением операционной системы BSD, HP-UX или Solaris, попробуйте обратиться к указанному порту.

Есть несколько сервисов, к которым можно свободно получить доступ. Запустите программу Telnet и свяжитесь с портом 21 (FTP):

```
* telnet 127.0.0.1 21
```

Когда соединение будет установлено, программа получит приветственное сообщение от сервера. Telnet — не лучшая программа для работы с FTP-сервером, но с ее помощью можно проследить базовый алгоритм взаимодействия между клиентом и сервером, схематически представленный в листинге 1.1. В нем клиент подключается к серверу, получает приветственное сообщение и. отключается.

Листинг 1.1. Простейший алгоритм TCP-клиента

```
/**********************************/
/*** Базовый клиентский алгоритм ***/
/**********************************/
Создание сокета
Определение адреса сервера
Подключение к серверу
Чтение и отображение сообщений
Разрыв                                    соединения.
```

Описанный алгоритм может показаться чересчур упрощенным. В принципе, так оно и есть. Но сама процедура подключения к серверу и организации взаимодействия с ним действительно проста. В следующих разделах рассматривается каждый из указанных выше этапов.

Системный вызов socket()

Функция socket() является универсальным инструментом, с помощью которого организуется канал связи с другим компьютером и запускается процесс приема/передачи сообщений. Эта функция образует единый интерфейс между всеми протоколами в Linux/UNIX. Подобно системному вызову open(), создающему дескриптор для доступа к файлам и системным устройствам, функция socket() создает дескриптор, позволяющий обращаться к компьютерам по сети. Посредством параметров функции необходимо указать, к какому уровню стека сетевых протоколов требуется получить доступ. Синтаксис функции таков:

```
#include <sys/socket.h>
#include <resolv.h>
int socket(int domain, int type, int protocol);
```

Значения параметров функции могут быть самыми разными. Полный их список приводится в приложении А, "Информационные таблицы". Основные параметры перечислены в табл. 1.1.

Таблица 1.1. Избранные параметры функции socket()

Параметр	Значение	Описание
domain	PF_INET	Протоколы семейства IPV4; Стек TCP/IP
	PF_LOCAL	Локальные именованные каналы в стиле BSD; используется утилитой журнальной регистрации, а также при организации очередей принтера
	PF_IPX	Протоколы Novell
	PF_INET6	Протоколы семейства IPv6; стёк TCP/IP
type	SOCK_STREAM	Протокол последовательной передачи данных (в виде байтового потока) с подтверждением доставки (TCP)

Окончание табл. 1.1

	SOCK_RDM	Протокол пакетной передачи данных с подтверждением доставки (еще не реализован в большинстве операционных систем)
	SOCK_RDM	Протокол пакетной передачи данных с подтверждением доставки (еще не реализован в большинстве операционных систем)
	SOCK_DGRAM	Протокол пакетной передачи данных без подтверждения доставки (UDP - User Datagram Protocol)
	SOCK_RAW	Протокол пакетной передачи низкоуровневых данных без подтверждения доставки
protocol		Представляет собой 32-разрядное целое число с сетевым порядком следования байтов (подробно об этом — в главе 2, "Основы TCP/IP"). В большинстве типов соединений допускается единственное значение данного параметра: 0 (нуль), а в соединениях типа SOCK_RAW параметр должен принимать значения от 0 до 255

В примерах, приведенных в данной книге, будут использоваться такие параметры: domain=PF_INET, type=SOCK_STREAM, protocol=0.

Префиксы PF_ и AF_

В рассматриваемых примерах "обозначения доменов в функции socket() даются с префиксом PF_ (protocol family — семейство протоколов). Многие программисты некорректно пользуются константами р префиксом AF_ (address family — семейство адресов). В настоящее время эти семейства констант взаимозаменяемы, но подобная ситуация может измениться в будущем.

Вызов протокола TCP выглядит следующим образом:

```
int sd;
sd = socket(PF_INET, SOCK_5TREAM, 0);
```

В переменную sd будет записан дескриптор сокета, функционально эквивалентный дескриптору файла:

```
int fd;
fd = open(...);
```

В случае возникновения ошибки функция socket() возвращает отрицательное число и помещает код ошибки в стандартную библиотечную переменную errno. Вот наиболее распространенные коды ошибок.

- EPROTONOSUPPORT. Тип протокола или указанный протокол не поддерживаются в данном домене. В большинстве доменов параметр protocol должен равняться нулю.
- EACCES. Отсутствует разрешение на создание сокета указанного типа. При создании сокетов типа SOCK-RAW и PF_PACKET программа должна иметь привилегии пользователя root.
- EINVAL. Неизвестный протокол либо семейство протоколов недоступно. Данная ошибка может возникнуть при неправильном указании параметра domain или type.

Конечно же, следует знать о том, какие файлы заголовков требуется включать в программу. В Linux они таковы:

```
#include  <sys/socket.h>          /* содержит прототипы функций */
#include  <sys/types.h>     /* содержит объявления стандартных
                                        системных типов данных */
#include  <resolv.h>    /* содержит объявления дополнительных
                                        типов данных */
```

В файле sys/socket.h находятся объявления функций библиотеки Socket API (включая функцию socket(), естественно). В файле sys/types.h определены многие типы данные, используемые при работе с сокетами.

Впримерах книги используется файл resolv.h, содержащий объявления дополнительных типов данных. Необходимость в нем возникла, когда при тестировании примеров в системах Mandrake 6.0-7.0 оказалось, что существующий файл sys/types.h некорректен (он не включает файл netinet/in.h, в котором определены типы данных, используемые при работе с адресами). Возможно, в других версиях Linux и UNIX этот файл и с п р а в л е н .

Действие функции socket() заключается в создании очередей, предназначенных для приема и отправки данных. В этом ее отличие от функции open(), которая открывает файл и читает содержимое его первого блока. Подключение очередей к сетевым потокам происходит только при выполнении системного вызова bind().

Если провести аналогию с телефонным звонком, то сокет — это трубка, не подключенная ни к базовому аппарату, ни к телефонной линии. Функции bind(), connect( ) и некоторые функции ввода-вывода соединяют трубку с телефоном, а телефон — с линией. (Если в программе не содержится явного вызова функции bind(), то его осуществляет операционная система; обратитесь к главе 4, "Передача сообщений между одноранговыми компьютерами").

Подключение к серверу

После создания сокета необходимо подключиться к серверу. Эту задачу выполняет функция connect(), действие которой напоминает звонок по телефону.

- Когда вы звоните абоненту, вы набираете его номер, который идентифицирует телефонный аппарат, расположенный где-то в телефонной сети. Точно так же IP-адрес идентифицирует компьютер. Как и у телефонного номера, у IP-адреса есть определенный формат.
- Соединение, телефонное или сетевое, представляет собой канал передачи сообщений. Когда человек на другом конце провода снимает трубку, соединение считается установленным. Ваш телефонный номер не имеет значения, если только человек, с которым вы общаетесь, не захочет вам перезвонить.
- Номер вашего аппарата определяется внутри АТС, где происходит направление потоков сообщений, передаваемых в рамках текущего соединения. В компьютерной сети абонентский компьютер или сервер должен в процессе соединения узнать адрес и порт, по которым можно будет связаться с вашей программой.

  Вы должны сообщить свой телефонный номер людям, которые могут вам позвонить. В случае программы, принимающей входные звонки,

необходимо назначить ей канал (или порт) и сообщить о нем своим клиентам.

Синтаксис функции connect() таков:

```
#include <sys/socket.h>
iinclude <resolv.h>
int connect(int sd, struct sockaddr *server, int addr_len);
```

Первый параметр (sd) представляет собой дескриптор сокета, который был создан функцией socket(). Последний, третий, параметр задает длину структуры sockaddr, передаваемой во втором параметре, так как она может иметь разный тип и размер. Это самый важный момент, делающий функцию socket() принципиально отличной от функций файлового ввода-вывода.

Функция socket() поддерживает по крайней мере два домена: PF_INET и PF_IPX. В каждом из сетевых доменов используется своя структура адреса. Все структуры являются производными от одного общего предка — структуры sockaddr. Именно она указана в заголовке функции connect(). (Полный список определений структур содержится в приложении А, "Информационные таблицы".

Абстрактная структура sockaddr .

Структура sockaddr является абстрактной 8 том смысле, что переменные данного типа почти никогда не приходится создавать напрямую. Существует множество других, специализированных структур, приводимых к типу sockaddr. Подобная методика позволяет работать с адресами различного формата по некоему общему образцу. Аналогичная абстракция используется при организации стеков. В стек могут помещаться данные разных типов, но к ним всегда применяются одинаковые операции: push, (занести), pop (извлечь) и т.д. Во всех структурах семейства sockaddr первое поле имеет суффикс _family и интерпретируется одинаковым образом: оно задает семейство адреса, или сетевой домен. Тип данного поля определяется как 16-разрядное целое число без знака.

Приведем общий вид структуры адреса и рядом для сравнения — структуру адреса в домене PF_INET (взято из файлов заголовков):

```
struct sockaddr {
    unsigned short int sa_family;
    unsigned char sa_data[14];
};
```

```
struct sockaddr_in {
    sa_family_t      sin_family;
    unsigned short int sin_port;
    struct in_addr   sin_addr;
    unsigned char    _   pad[];
```

Взаимосвязь между типом сокета и полем семейства адреса в структуре sockaddr

Тип домена, задаваемый в функции socket( ), должен совпадать со значением, которое записывается в первое поле структуры sockaddr (за исключением префикса: в первом случае это PF_, во втором — AF_). Например, если в программе создается сокет PF_INET6, то в первое поле структуры должно быть помещено значение AF_INET6, иначе программа будет неправильно работать.

Обратите внимание: поля sa_family и sin family в обеих структурах являются общими. Любая функция, получающая подобного рода структуру, сначала прове-

ряет первое поле. Следует также отметить, что это единственное поле с *серверным порядком следования байтов* (подробно об этом — в главе 2, "Основы TCP/IP"). Поля-заполнители (sa_data и _pad) используются во многих структурах. По существующей договоренности структуры sockaddr и sockaddr_in должны иметь размер 16 байтов (в стандарте IPv6 структура sockaddr_in6 имеет размер 24 байта), поэтому такие поля дополняют тело структуры незначащими байтами.

Читатели, наверное, заметили, что размер массива __pad[] не указан. Ничего неправильного в этом нет — таково общепринятое соглашений, Поскольку данный массив заполняется нулями, его размер не имеет значения (в случае структуры sockaddr_in он равен восьми байтам). В некоторых системах в структуре sockaddr_in выделяются дополнительные поля для внутренних вычислений. Не стоит обращать на них внимание, а также использовать их, поскольку нет гарантии, что эти поля будут поддерживаться в другой системе. В любом случае достаточно инициализировать данную структуру нулями.

Ниже описано назначение полей структуры, а также приведены примеры их содержимого.

Поле	Описание	Порядок байтов	Пример
sin family	Семейство протоколов	Серверный	AF_INET
sin port	Номер порта сервера	Сетевой	- 13
sin addr	IP-адрес сервера	Сетевой	127.0.0.1

Прежде чем вызвать функцию connect(), программа должна заполнить описанные поля. В листинге 1.2 показано, как это сделать (полный текст примера имеется на Web-узле). Вообще говоря, в Linux не требуется приводить структуру sockaddr_in к типу sockaddr. Если же предполагается использовать программу в разных системах, можно легко добавить операцию приведения типа.

Приведение к типу sockaddr

В UNIX-системах любую структуру данного семейства можно привести к типу sockaddr. Это позволит избежать получения предупреждений компилятора. В приводимых примерах данная операция не используется только потому, что это делает, примеры немного понятнее (да и Linux этого не требует).

Листинг 1.2. Использование функции connect ()

```
/***    Фрагмент программы, демонстрирующий инициализацию    ***/
/***    параметров и вызов функции connect().                ***/

idefine PORT_TIME    13
struct sockaddr_in dest;
char *host - "127.0.0.1";
int sd;
/**** Создание сокета ****/

bzero(Sdest, sizeof(dest));          /* обнуляем структуру */
dest.sin_family = AF_INET;             /* выбираем протокол */
```

```
dest.sin_port = htons ( PORT_TIME ) ;              /* выбираем порт */
inet_aton(host, &dest.sin_addr) ;                  /* задаем адрес */

if ( connect (sd, &dest, sizeof (dest)) != 0 )   /* подключаемся! */
{
     perror ( "socket connection" ) ;
     abort ();
```

Перед подключением к серверу выполняется ряд подготовительных действий. В первую очередь создается структура sockaddr_in. Затем объявляется переменная, содержащая адрес, по которому будет произведено обращение. После этого выполняются другие, не показанные здесь, операции, включая вызов функции socket(). Функция bzero() заполняет структуру sockaddr_in нулями. Поле sin_family устанавливается равным AF_INET. Далее задаются номер порта и IP-адрес. Функции htons() и inet_aton(), выполняющие преобразования типов данных, рассматриваются в главе 2, "Основы TCP/IP".

Наконец, осуществляется подключение к серверу. Обратите внимание на необходимость проверки значения, возвращаемого функцией connect(). Это один из многих приемов, позволяющих повысить надежность сетевых приложений.

После установления соединения дескриптор сокета, sd, становится дескриптором ввода-вывода, доступным обеим программам. Большинство серверов ориентировано на выполнение единственной транзакции, после чего разрывают соединение (например, сервер HTTP 1.0 отправляет запрашиваемый файл и отключается). Взаимодействуя с такими серверами, программа должна посылать запрос, получать ответ и закрывать сокет.

Получение ответа от сервера

Итак, сокет открыт, и соединение установлено. Можно начинать разговор. Некоторые серверы инициируют диалог подобно людям, разговаривающим по телефону. Они как бы говорят: "Алло!" Приветственное сообщение может включать имя сервера и определенные инструкции.

Когда сокет открыт, можно вызывать стандартные низкоуровневые функции ввода-вывода для приема и передачи данных. Ниже приведено объявление функции read():

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

Эта функция должна быть вам знакома. Вы много раз применяли ее при работе с файлами, только на этот раз необходимо указывать дескриптор не файла (fd), а сокета (sd). Вот как обычно организуется вызов функции read( ):

```
int sd, bytes_read;
sd = socket (PF_INET, SOCK_STREAM, 0);      /* создание сокета */

/**** Подключение к серверу ****/
```

```
bytes_read = read(sd, buffer, MAXBUF);      /* чтение данных .*/
if ( bytes_read < 0 )
    /* сообщить об ошибках; завершить работу */
```

Дескриптор сокета можно даже преобразовать в файловый дескриптор (FILE*), если требуется работать с высокоуровневыми функциями ввода-вывода. Например, в следующем фрагменте программы демонстрируется, как применить функцию fscanf() для чтения данных с сервера (строки, на которые следует обратить внимание, выделены полужирным шрифтом):

```
char Name[NAME], Address[ADDRESS], Phone[PHONE];
FILE *sp;
int sd;
sd = socket(PF_INET, SOCK_STREAM, 0);      /* создание сокета */

/**** Подключение к серверу ****/

if ( (sp = fopen(sd, "r")) == NULL ) /* преобразуем дескриптор
                                        в формат FILE* */
    perror("FILE* conversion failed");
else if ( fscanf(sp, "%*s, %*s,    %*s\n",    /* читаем данные
                                                 из файла */
                NAME, Name, ADDRESS, Address,
                PHONE, Phone) < 0)
{
    perror("fscanf");
```

Только дескрипторы сокетов потокового типа могут быть свободно конвертированы в формат FILE*. Причина этого проста: в протоколе UDP соединение не устанавливается вовсе — дейтаграммы просто посылаются и все. Кроме того, потоковые сокеты обеспечивают целостность данных и надежную доставку сообщений, тогда как доставка дейтаграмм не гарантируется. Применение дейтаграмм подобно вложению письма в конверт с адресом, отправляемый по почте: нельзя быть полностью уверенным в том, что письмо дойдет до адресата. Соединение, имеющее дескриптор типа FILE*, должно быть открытым. Если преобразовать данные в формат дейтаграммы, их можно потерять. (Подробнее о дейтаграммах можно узнать в главе 3, "Различные типы Internet-пакетов".)

Соединения, с которыми связаны файловые дескрипторы, являются для сетевого программиста удобным средством анализа поступающих данных. Следует, однако, быть предельно внимательным: необходимо проверять *все* возвращаемые значения, даже у функций fprintf() и fscanf(). Обратите внимание на то, что в показанном выше примере отрицательный код, возвращаемый функцией fscanf(), сигнализирует об ошибке.

Безопасность и надежность сети

Безопасность и надежность — это основные факторы, которые следует учитывать при написаний сетевых приложений. Не забывайте проверять переполнение буферов и коды завершения функций. Попросите других программистов проверить вашу программу на восприимчивость к произвольным входным данным. Не пренебрегайте советами экспертов.

Возвращаясь к функции read(), отметим, что чаще всего в результате ее выполнения возникают такие ошибки.

- EAGAIN. Задан режим неблокируемого ввода-вывода, а данные недоступны. Эта ошибка означает, что программа должна вызвать функцию повторно.
- EBADF. Указан неверный дескриптор файла, либо файл не был открыт для чтения. Эта ошибка может возникнуть, если вызов функции socket() завершился неуспешно или же программа закрыла входной поток (канал доступен только для записи).
- EINVAL. Указанный дескриптор связан с объектом, чтение из которого невозможно.

Функция read() не имеет информации о том, как работает сокет. В Linux есть другая функция, recv(), которая наряду с чтением данных позволяет контролировать работу сокета: /

```
#include <sys/socket.h>
#include <resolv.h>
int recv(int sd, void *buf, int len, unsigned int flags);
```

Эта функция принимает такие же параметры, как и функция read(), за исключением флагов. Флаги можно объединять с помощью операции побитового сложения *(флаг1 | флаг2 | ...)*. Обычно последний параметр задается равным нулю. Читатели могут поинтересоваться, для чего в таком случае вообще вызывать функцию recv()? Не проще ли вызвать функцию read()? Мое мнение таково: лучше применять функцию recv() — это может помочь вам, если впоследствии работа программы усложнится. Да и, вообще говоря, всегда следует придерживаться какого-то одного стиля.

Ниже перечислены полезные флаги, с помощью которых можно управлять работой сокета. Полный их список представлен в приложении Б, "Сетевые функции".

- MSG_OOB, Обработка внеполосных данных. Применяется для сообщений с повышенным приоритетом. Некоторые протоколы позволяют выбирать, с каким приоритетом следует послать сообщение: обычным или высоким. Установите этот флаг, чтобы диспетчер очереди искал и возвращал только внеполосные сообщения (подробно об этом — в главе 10, "Создание устойчивых сокетов").
- MSG_PEEK. Режим неразрушающего чтения. Заставляет диспетчер очереди извлекать сообщения, не перемещая указатель очереди. Другими словами, при последовательных операциях чтения будут возвращаться одни и те же данные (точнее, должны возвращаться; обратитесь ко врезке "Получение фрагментированных пакетов").
- MSG_WAITALL. Сообщение не будет возвращено до тех пор, пока не заполнится указанный буфер. При отсутствии этого флага возможно получение частично заполненного буфера, поскольку остальные данные еще "в пути". В этом случае программе придется "собирать" их самостоятельно.
- MSG _DONTWAIT. Запрос к сокету не будет блокирован, если очередь сообщений пуста. Аналогичный режим (неблокируемый ввод-вывод) можно

также задать в свойствах самого сокета. Обычно, если данные в очереди отсутствуют, диспетчер очереди ждет до тех пор, пока они не поступят. А когда этот флаг установлен, функция, запрашивающая данные, немедленно завершается, возвращая код ошибки EWOULDBLK. (В настоящее время в Linux не поддерживается этот флаг. Чтобы достигнуть требуемого результата, необходимо вызвать'функцию fcntl() с флагом O_NONBLOCK. Это заставит сокет всегда работать в режиме неблокируемого ввода-вывода.)

Получение фрагментированныхпакетов

Программа может работать гораздо быстрее чем сеть. Иногда пакет приходят по частям, потому что маршрутизаторы фрагментируют их для ускорения передачи по медленным сетям. Если в подобной ситуации вызвать функцию recv(), будет прочитано неполное сообщение. Вот почему даже при наличии флага MSG_PEEK функция recv() при последовательных вызовах может возвращать разные данные: например, сначала 500 байтов, а затем 750. Для решения подобных проблем предназначен флаг MSG_WAITALL.

Функция recv() является более гибкой, чем read(). Ниже показано, как прочитать данные из канала сокета (эквивалентно функции read()):

```
int bytes_read;
bytes_read = recv(sd, buffer, MAXBUF, 0);
```

А вот как осуществить *неразрушающее* чтение:

```
int bytes_read;
bytes_read = recv(sd, buffer, MAXBUF, MSG_PEEK);
```

Можно даже задать режим неразрушающего чтения внеполосных данных:

```
int bytes_read;
bytes_read = recv(sd, buffer, MAXBUF, MSG_OOB | MSG_PEEK);
```

В первом варианте функция просто передает серверу указатель буфера и значение его длины. Во втором фрагменте информация копируется из очереди, но не извлекается из нее.

Во всех трех фрагментах есть одно преднамеренное упущение. Что если сервер пошлет больше информации, чем может вместить буфер? В действительности ничего страшного не произойдет, это не критическая ошибка. Просто программа потеряет те данные, которые не были прочитаны.

Функция recv() возвращает те же коды ошибок, что и функция read(), но есть и дополнения.

- ENOTCONN. Предоставленный дескриптор сокета не связан с одноранговым компьютером или сервером.
- ENOTSOCK. Предоставленный дескриптор не содержит сигнатуру, указывающую на то, что он был создан функцией socket().

Вообще говоря, функция read() тоже может вернуть эти коды, поскольку на самом деле она проверяет, какой дескриптор ей передан, и если это дескриптор сокета, она просто вызывает функцию recv().

Разрыв соединения

Информация от сервера получена, сеанс прошел нормально — настало время прекращать связь. Опять-таки, есть два способа сделать это. В большинстве программ используется стандартный системный вызов close():

```
#include<unistd.h>
int close(int fd);
```

Вместо дескриптора файла (fd) может быть указан дескриптор сокета (sd) — работа функции от этого не изменится. В случае успешного завершения возвращается значение 0.

Всегда закрывайте сокеты

Возьмите за правило явно закрывать дескрипторы, особенно сокетов. По умолчанию при завершении программы операционная система закрывает все открытые дескрипторы и "выталкивает" содержимое буферов. Если дескриптор связан с файлом, все проходит незаметно. 6 случае сокета процесс может затянуться, в результате ресурсы останутся занятыми и другим клиентам будет сложнее подключиться к сети.

Функция close () возвращает всего один код ошибки.

- EBADF. Указан неверный дескриптор файла.

Функция shutdown() позволяет лучше управлять процессом разрыва соединения, поскольку может закрывать отдельно входные и выходные каналы. Эта функция особенно полезна, когда сокет замещает стандартные потоки stdin и stdout.

Путаница с именем shutdown

Функция shutdown () отличается от команды shutdown (см. раздел 8 интерактивного справочного руководства по UNIX), которая завершает работу операционной системы.

С помощью функции shutdown() можно закрыть канал в одном направлении, сделав его доступным только для чтения или только для записи:

```
Iinclude <sys/socket.h>
int shutdown(int s, int how);
```

Параметр how может принимать три значения.

Значение	Выполняемое действие
0	Закрыть канал чтения
1	Закрыть канал записи
2	Закрыть оба канала

Резюме: что происходит за кулисами

Когда программа создает сокет и подключается к TCP-серверу, происходит целый ряд действий. Сам по себе сокет организует лишь очередь сообщений. Основной процесс начинается при подключении. Ниже поэтапно расписано, что происходит на стороне клиента и сервера (табл. 1.2).

Таблица 1.2. Действия, выполняемые при создании сокета и подключении к серверу

Действия клиента	Действия сервера
1. Вызов функции socket(): создание очереди сообщений, установка флагов протокола	(Ожидание подключения)
2. Вызов функции connect(): операционная система назначает сокету порт, если он не был назначен с помощью функции bind()	(Ожидание)
3. Отправка сообщения, в котором запрашивается установление соединения и сообщается номер порта	
(Ожидание ответа сервера)	4. Помещение запроса в очередь порта
(Ожидание)	5. Чтение данных из очереди, прием запроса и создание уникального канала для сокета
(Ожидание)	6. Создание (иногда) уникального задания или потока для взаимодействия с программой
(Ожидание)	7. Отправка подтверждения о том, что соединение установлено. Сервер либо посылает сообщение по указанному порту, либо ожидает запроса от программы. После передачи данных сервер может закрыть канал, если он выдает только односторонние сообщения (например, сообщает текущее время)
8. Начало передачи данных	

Думаю, этого достаточно для простого вызова функции connect(). Описанный процесс может быть гораздо более сложным, если между клиентом и сервером находятся компьютеры, выполняющие маршрутизацию (коммутацию и верификацию пакетов, фрагментацию и дефрагментацию, трансляцию протоколов, туннелирование и т.д.). Библиотека Socket API значительно упрощает сетевое взаимодействие.

Для организации соединения требуется знать язык и правила сетевого общения. Все начинается с функции socket(), которая создает аналог телефонной трубки. Через эту "трубку" программа посылает и принимает сообщения. Чтение и запись данных осуществляются с помощью тех же самых функций read() и write(), которые применяются при работе с файлами. Более сложные системы строятся вокруг функции recv().

В сетях TCP/IP есть много других особенностей, требующих пояснения. В следующей главе рассматриваются принципы IP-адресации, рассказывается о том, какие бывают порты и что такое порядок следования байтов.

Основы TCP/IP

Глава

2

В этой главе...

Работа в Internet требует знания принципов адресации. Адрес — неотъемлемая часть сообщения. Подобно телефону, компьютер должен иметь идентификационный номер, или адрес, чтобы другие компьютеры могли направлять ему данные.

В предыдущей главе рассказывалось о том, что такое сокет. В этой главе мы продолжим знакомство с библиотекой Socket API и узнаем о том, что такое система IP-адресации, как происходит маршрутизация, какие бывают форматы передачи двоичных данных и какие типы сокетов существуют.

IP-адресация

В TCP/IP адрес имеет фиксированную длину. Данное ограничение потребовало от разработчиков протокола IP тщательно продумать особенности его функционирования. В свое время протокол позволял решать целый ряд задач, таких как идентификация компьютера, маршрутизация и преобразование адресов. Сегодня он сталкивается. с новыми проблемами, связанными со взрывоподобным разрастанием сети Internet.

Идентификация компьютера

В сети происходит совместное использование одного ресурса (сетевого кабеля) несколькими компьютерами. Все компьютеры должны быть способны принимать данные, передаваемые по сети. Но если бы каждый компьютер идентифицировался именем Bob или Sasha, невозможно было бы определить, кому из них следует посылать данные. Отсюда возникает необходимость назначения компьютеру уникального идентификатора. Но это еще не все.

Конфликты адресов

Рабочая станция не может правильно функционировать в сети, если она делит с кем-то свой адрес (имеет место *конфликт адресов).* Любой; кто пытался обнаружить данную коллизию, скажет вам, что это не так просто. Ситуация становится еще хуже, если на одном из компьютеров применяется протокол динамического назначения адреса (например, DHCP). Наиболее очевидный (и самый трудоемкий) способ решения проблемы — просмотреть свойства каждого компьютера в данном сетевом сегменте.

Сети — это динамичные структуры, имеющие тенденцию усложняться со временем. Несмотря на увеличение сети, любой входящий в нее компьютер должен легко обнаруживаться с помощью своего идентификатора. Но следует помнить, что все сообщения, передаваемые по сети, занимают 'часть канала с ограниченной пропускной способностью, поэтому чем меньше избыточной информации в них содержится, тем лучше.

В некоторых сетях в сообщение включается карта маршрутизации. Любой сервер, перечисленный в списке, получит сообщение и передаст его следующему серверу. Данный подход позволяет снизить сетевой трафик за счет уменьшения объема служебных данных, передаваемых в заголовке сообщения. Если же закодировать путь к целевому компьютеру в самом адресе, карта маршрутизации станет ненужной.

Компьютер, подключенный к сети, уже имеет уникальный идентификатор, называемый MAC-адресом (Media Access Control — протокол управления доступом к среде). В качестве примера можно привести идентификатор платы Ethernet.

Данный идентификатор применяется в процессе сетевой загрузки бездисковых станций, все постоянные ресурсы которых расположены на сервере. Идентификатор Ethernet имеет длину 6 байтов и обычно записывается в шестнадцатеричном виде, например 00:20:45:FE:A9:OB. Каждая сетевая плата имеет свой уникальный адрес, назначаемый производителем.

К сожалению, применять данный адрес для идентификации компьютера нельзя. Это связано с двумя проблемами. Во-первых, каждому маршрутизирующему серверу пришлось бы запрашивать по сети все идентификаторы. База данных со значениями этих идентификаторов может стать чересчур большой, что существенно замедлит маршрутизацию сообщений. Во-вторых, не во всех протоколах используется MAC-адрес (в PPP, например).

Компьютер должен иметь такой идентификатор, который содержал бы встроенный алгоритм маршрутизации сообщения. Вспомните, как записывается адрес на конверте: от общего (название страны) к частному (номер квартиры). Этому свойству удовлетворяют IP-адреса.

IP-адрес имеет несколько преимуществ по сравнению с аппаратным MAC-адресом. Основное из них заключается в том, что адрес может меняться (он не зафиксирован жестко). Благодаря этому появляется возможность вводить кластеры адресов, а владельцы портативных компьютеров избавляются от проблем, связанных с подключением к сети. Сетевой драйвер преобразует IP-адрес в MAC-адрес с помощью протокола ARP (Address Resolution Protocol — протокол преобразования адресов) [RFC826].

Протокол ARP

В ARP используется простая таблица преобразования IP-адреса в MAC-адрес. Все сетевые сообщения должны нести в себе MAC-адрес, чтобы сетевой адаптер или драйвер могли легко обнаруживать нужные сообщения. Но отправитель может не знать MAC-адрес получателя, далеко спрятанный за несколькими маршрутизаторами. Поэтому ближайший маршрутизатор просто посылает сообщение маршрутизаторам, управляющим работой подсети, которая указана в IP-адресе. Когда сообщение доходит до маршрутизатора, использующего протокол ARP, проверяется адресная таблица. Если, найдено совпадение, MAC-адрес пакета определяет пункт назначения. В противном случае маршрутизатор посылает широковещательное сообщение компьютерам подсети, чтобы определить, кому принадлежит IP-адрес.

Структура адресов Internet

В Internet применяется схема адресации "от старшего к младшему". Каждый идентификатор представляет собой четырехбайтовое число. Первый байт, если читать слева направо, содержит значение *класса сети* (рис. 2.1).

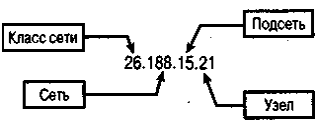

Рис. 2.1. IP-адрес состоит из нескольких элементов, идентифицирующих различные звенья сети

IP-адрес представляет собой нечто вроде дорожной карты для маршрутизатора. Каждый элемент адреса уточняет пункт назначения. Первым элементом, как уже было сказано, является класс сети, последним — номер компьютера в сети. Подобную схему можно сравнить, с адресом на конверте, где сначала указывается страна, затем город, улица, дом, квартира и сам адресат.

В Internet существует пять базовых классов адресов. Каждый класс представляет собой блок адресов, выделенных для сетей разного размера: от сверхбольших до малых. Схема адресации предусматривает возможность покупки компаниями сегментов адресов из общего адресного пространства. Классы нумеруются от A до E.

Класс	Адрес	Число адресов, назначение
	0.0.0.0-0.255.255.255	(Зарезервировано)
A	1.0.0.0 — 126.255.255.255	2^{24}-2, или 16777214 узлов в каждом сегменте. В данном классе имеется 126 сегментов. Используется в очень крупных организациях, которые при необходимости формируют собственные подсети. Пример — провайдеры Internet
	127.0.0.0 — 127.255.255.255	(Зарезервировано для интерфейса обратной связи)
B	128.xxx.0.0-191.xxx.255.255	2^{16}-2, или 65534 узла в-каждом сегменте. В данном классе имеется 64x256 сегментов. Используется в крупных организациях, например в корпорациях и университетах, которые также могут создавать собственные подсети. Адрес *xxx* назначается подсети (к примеру, 129.5.0.0). Данное адресное пространство практически полностью занято
C	192.xxx.xxx.0 - 223.xxx.xxx.255	2^{8}-2, или 254 узла в каждом сегменте. В данном сегменте имеется 32x65536 сегментов. Используется небольшими компаниями и персональными серверами Internet
D	224.0.0.0-239.255.255..255	2^{28}-2, или 268435454 узла. Данные адреса не выделяются, но зарезервированы для режима группового вещания
E	240.0.0.0-255.255.255.255	2^{28}-2, или 268435454 узла. Данная группа адресов зарезервирована на будущее. Обратите внимание на то, что значение 255.255.255.255 является общим широковещательным IP-адресом

В подобной схеме адресации нет ничего загадочного, если учесть, что для определения класса сети проверяются первые биты адреса. Например, в сетях класса A первый бит адреса равен 0. В сетях класса B первый бит равен 1, а следующий за ним — 0.

- Класс A: 0 (00000000) - 126 (01111110)
- Класс B: 128 (10000000) — 191 (10111111)
- Класс C: 192 (11000000) - 223 (11011111)
- Класс D: 224 (11100000) — 239 (11101111)
- Класс E: 240 (11110000) - 255 (11111111)

С момента возникновения Internet маршрутизаторы руководствовались такой схемой для быстрой (оценивалось всего 4 бита) проверки того, нужно ли пропускать сообщение через шлюз. Сегодня маршрутизация выполняется более эффек-

тивно благодаря протоколу CIDR (Classless Internet Domain Routing — бесклассовая междоменная маршрутизация). В нем местоположение компьютера в сети и путь к нему определяются на основании старших битов адреса (определения соответствующих алгоритмов даны в RFC-документах с номерами 1517—1519).

Описанная схема адресации была внедрена в протокол IP для эффективной группировки компьютеров по адресам. Благодаря ей маршрутизатор может быстро определить, что нужно делать: блокировать пакет или передать его интерфейсному устройству, связанному с указанной подсетью. Последовательная передача пакетов очень важна: недостаточно просто принять пакет, проверить его и отправить дальше. Каждый бит, проверяемый маршрутизатором, означает задержку в сети, поэтому маршрутизатор должен просматривать ровно столько битов, сколько необходимо для выяснения приемлемости пакета. Другими словами, работа маршрутизатора не должна вызывать задержек распространения пакетов. К примеру, в типичной схеме коммутации телефонных звонков, применяемой в Соединенных Штатах, весь телефонный номер не анализируется сразу. Сначала проверяются первые три цифры, указывающие регион или штат, а затем — следующие три цифры, определяющие телефонный аппарат.

Маски подсети

В некоторых классах сетей требуется дополнительная фильтрация адресов. Сеть, состоящая из 16 миллионов узлов, будет чересчур громоздкой, если все узлы находятся в одном адресном пространстве. Инсталлируя Linux и конфигурируя сеть, читатели, возможно, обратили внимание на термин *маска подсети*. Он обязан своим появлением протоколу CIDR, который значительно упростил управление крупными подсетями.

Маска подсети идентифицирует непрерывную группу адресов. Она служит дополнительным фильтром, позволяющим пропускать только определенные сообщения. Когда поступает сообщение, маршрутизатор накладывает на его адрес маску. Если маска подходит, сообщение пропускается. Маска может иметь любой вид, но самый младший значащий бит идентифицирует начало адреса подсети.

Например, если имеется небольшая сеть из восьми компьютеров и предполагается, что она может увеличиться не более чем на пять машин, задайте для маршрутизатора маску вида 187.35.209.208. Младший бит адреса находится в числе 208 (11010000). Таким образом, допустимыми адресами будут те, у которых младший байт попадает в диапазон 1101xxxx, где *XXXX*— это сегмент адресов *активной подсети*. Теперь можно назначать адреса: 187.35.209.208 (маршрутизатор), 187.35.209.209 (компьютер № 1) и т.д. до 187.35.209.222. Старшие биты последнего байта маски не имеют значения, так как начало диапазона адресов определяется самым младшим значащим битом.

Маска подсети и нулевой адрес

Использование маски в качестве адреса компьютера может вызвать конфликты и потерю пакетов. Это объясняется наличием специального *нулевого* адреса. Когда маска накладывается на идентичный ей адрес, образуется нулевой адрес активной подсети. Он всегда зарезервирован для сетевой маски. Кроме того, читатели, наверное, обратили внимание на то, что в рассмотренном выше примере пропущен адрес 187.35.209.223. Дело в том, что если наложить на него сетевую маску, полученный адрес активной подсети будет содержать все единицы: 11011111. Подобные адреса (состоящие из одних единиц) являются *широковещательными,* Их нельзя использовать в качестве адресов компьютеров.

В большинстве случаев нет необходимости заниматься настройкой маршрутизаторов, а вопросы их конфигурирования выходят за рамки данной книги. Как правило, если где-либо требуется указать маску подсети, достаточно воспользоваться значением по умолчанию.

Маршрутизаторы и преобразование адресов

В Internet любой компьютер теоретически может получить любое сообщение. Но локальная сеть быстро переполнится сообщениями, передаваемыми между всеми компьютерами сети. Трафик превысит допустимые пределы, пакеты начнут сталкиваться друг с другом (вследствие одновременной передачи), и пропускная способность сети сойдет на нет.

Чтобы этого избежать, в локальных сетях применяется система масок подсетей, позволяющих локализовать потоки сообщений в пределах определенных групп (или *кластеров)* компьютеров. Помимо передачи сообщений требуемым интерфейсным устройствам, маршрутизаторы играют роль информационных шлюзов, фильтруя сетевой трафик. Локальные сообщения остаются в пределах кластера, а внешние пропускаются наружу.

Компьютер, подключенный к сети, использует маску подсети, чтобы определить, следует ли направить сообщение маршрутизатору. Сообщения, адресованные внешним компьютерам, проходят через маршрутизатор, а локальные сообщения остаются в подсети. Тем самым снижается нагрузка на опорную сеть.

Маршрутизаторы направляют сообщения адресатам, руководствуясь таблицами маршрутизации. Эта система широко применяется в глобальных сетях. На каждом этапе передачи сообщения от источника к приемнику проверяется IP-адрес получателя, который сравнивается с адресами, указанными в таблице маршрутизации. Маршрутизаторы шаг за шагом перемещают сообщения в нужном направлении. На первом проходе адрес ищется в кластере, на втором — в подсети и т.д. вплоть до класса. Когда обнаруживается совпадение (на локальном или глобальном уровне), сообщение отсылается в нужном направлении, неся в себе MAC-адрес нужного маршрутизатора. В конце пути, когда сообщение попадет в нужный кластер, с помощью протокола ARP будет произведена замена MAC-адреса маршрутизатора MAC-адресом требуемого компьютера.

Адаптер Ethernet принимает только те сообщения, в которых указан его идентификатор. Компьютер, подключенный к сети, после загрузки посылает в подсеть широковещательное сообщение, в котором указывает свой идентификатор Ethernet и IP-адрес. Обработкой этих сообщений также управляет протокол ARP.

Специальные и потерянные адреса

Как уже упоминалось, существует ряд зарезервированных адресов. В активной подсети таковых два: один содержит все нули (маска подсети), а другой — все единицы (широковещательный адрес). Это означает, что при подсчете числа адресов, имеющихся в вашем распоряжении, необходимо дополнительно отнимать 2. Помните: при создании ста подсетей теряется двести адресов.

Но это только начало. Два блока адресов зарезервированы для внутреннего использования: от 0.0.0.0 до 0.255.255.255 и от 127.0.0.0 до 127.255.255.255.

Первая группа адресов обозначает "текущую" сеть. Например, если адрес компьютера 128.187.0.0, то указанный в сообщении адрес 0.0.25.31 неявно преобразуется в адрес 128.187.25.31.

В 1992 г. Координационный совет сети Internet (TAB — Internet Activities Board) [RFC 1160], вскоре преобразованный в Архитектурный совет сети Internet (Internet Architecture Board), забеспокоился по поводу взрывоподобного роста Internet. Несмотря на применяемые алгоритмы адресации и тщательное планирование, число выделенных адресов превысило 450 миллионов. Адресное пространство катастрофически сокращалось, хотя использовались лишь 2% адресов. Куда исчезли остальные адреса?

Дело в том, что Совет выделял компаниям адреса целыми блоками. При этом компании запрашивали больше адресов, чем реально было необходимо. Это позволяло им в будущем легко наращивать свои сети, оставаясь в пределах непрерывного диапазона адресов.

Инфляция адресов

В одной маленькой компании, где я работал, было зарезервировано 128 адресов. Из них в то время использовалось только 30. Лично для меня выделили 10 адресов, но мне так никогда и не понадобилось больше двух из них. Я полагаю, что эти адреса до сих пор зарезервированы за мной, хотя я покинул компанию более четырех лет назад.

В итоге специальные и неиспользуемые адреса поглотили почти все адресное пространство. Недавние исследования показали, что в сетях класса B и C почти не осталось свободных адресов. Провайдеры Internet задействовали почти все адреса в классе A. Вскоре доступных адресов не останется вовсе.

Что же делать с потерянным адресным пространством? Как видоизменить существующую систему адресации? Организация по назначению имен и адресов в сети Internet (ICANN — Internet Corporation for Assigned Names and Numbers) не может так просто затребовать назад неиспользуемые адреса, которые ранее были проданы компаниям, поскольку информационные подразделения компаний уже провели распределение этих адресов.

Многие компании теперь используют протокол DHCP (Dynamic Host Configuration Protocol — протокол динамической конфигурации сетевых компьютеров) [RFC2131], который назначает компьютеру IP-адрес после начальной загрузки. Ни один компьютер не владеет IP-адресом постоянно. Благодаря этому также решается проблема безопасности, потому что хакеры очень любят фиксированные адреса.

Напомню, что компьютер, подключенный к сети, в процессе загрузки запускает модуль запросов DHCP. Этот модуль посылает широковещательное сообщение, чтобы найти сервер DHCP. Будучи найденным, сервер выделяет адрес из своего пула доступных адресов и посылает его (вместе с соответствующей маской подсети) запрашивающему модулю.

Модуль принимает данные и конфигурирует сетевые протоколы локального компьютера. Иногда из-за большой загруженности сети это может занять несколько секунд. Пока компьютер не сконфигурирован правильно, он подвержен *IP-амнезии* (не знает свой собственный адрес).

IP-амнезия

Амнезия адреса происходит, когда маска подсети и адрес компьютера не заданы должным образом. Это может оказаться серьезной проблемой. Даже команда ping 127.0.0.1 не будет работать. Если компьютер проявляет признаки амнезии, исправьте системные файлы. В Red Hat Linux (и родственных системах) требуемые параметры устанавливаются в файлах /etc/sysconfig/network и /etc/sysconfig/network-scripts/ifcfg-*.

Другое решение заключается в увеличении длины самого IP-адреса. Здесь-то на арену и выходит стандарт IPv6 [RFC2460]. В нем применяются адреса четырехкратного размера: 128 битов, а не 32! Адрес выглядит примерно так:

```
8008:4523:FOE1:23:830:CF09:1:385
```

Естественно, его не так-то легко запомнить.

Основное преимущество протокола IPv6 заключается в существенном увеличении адресного пространства. Наличие $3x10^{38}$ адресов надолго устраняет существующие сегодня проблемы (подробно об этом рассказывается в главе 19, "IPv6: следующее поколение протокола IP").

Номера IP-портов

Все сообщения посылаются по тому или иному заданному адресу. Программа, принимающая сообщения, может в действительности получать сообщения, адресованные другим программам. Операционную систему не интересует, кому направлено пришедшее сообщение. Поэтому в TCP/IP было введено понятие порта. В системе может быть несколько портов, связанных с одним и тем же адресом.

Не стоит путать их с физическими портами. В TCP/IP порты — это каналы, по которым информация направляется нужной программе. Теперь программы не обрабатывают весь поток сообщений: они принимают только сообщения, пришедшие в связанный с программой порт.

Практически в каждом IP-пакете содержится адрес компьютера и номер порта. Последний указан в 16-разрядном поле в заголовке пакета. Операционная система проверяет это поле и помещает пакет в очередь сообщений соответствующего порта. Из нее программа читает данные сокета (с помощью системного вызова read() или recv()). Точно так же, когда программа посылает сообщение (посредством функции write() или send()), операционная система помещает данные в выходную очередь порта.

По умолчанию только одна программа владеет портом. Если попытаться запустить на одном и том же компьютере две программы, которые запрашивают одинаковый номер порта, будет получен код ошибки EINVAL. Чтобы задать режим совместного использования порта, необходимо установить для связанных с ним сокетов параметр SO_REUSEADDR (подробнее об этом речь пойдет в главе 9, "Повышение производительности").

Совместное использование портов в многопроцессорных системах

Правило, по которому два сокета не должны обращаться к одному и тому же порту, применимо и к симметричным Многопроцессорным системам; Причина, этого проста: процессоры, совместно используют память и операционную систему. Если в двух процессорах выполняются две копии операционной системы, то теоретически возможно, что два сокета в разных программах будут обращаться к одному порту, находясь в различных адресных пространствах.

Со всеми стандартными сервисами связаны номера портов. Полный их список содержится в файле /etc/services (обратитесь к приложению А, "Информационные таблицы"). Перечислим некоторые из них.

Порт	Имя сервиса, псевдоним	Описание
1	tcpmux	Мультиплексор сервисов портов TCP
7	echo	Эхо-сервер
9	discard	Аналог устройства /dev/null
13	daytime	Системный сервис даты/времени
20	ftp-data	Порт данных FTP
21	ftp	Основной порт FTP-подключения
23	telnet	Программа Telnet
25	smtp, mail	Почтовый сервер UNIX
37	time, timeserver	Сервер времени
42	nameserver	Сервер имен (DNS)
70	gopher	Программа Gopher
79	finger	Программа Finger
80	www, http	Web-сервер

Некоторые сервисы должны быть знакомы читателю. С большинством из них можно взаимодействовать посредством программы Telnet. Формат файла /etc/services очевиден: имя сервиса, номер порта, псевдоним и описание. Учтите: даже если сервис указан в файле, он может не выполняться в системе (например, в Mandrake Linux сервисы времени недоступны). Все номера портов, перечисленные в файле /etc/services, являются зарезервированными. Попытка создать аналогичные пользовательские порты приведет к конфликту.

Программа взаимодействует с сокетом через какой-нибудь локальный порт. За это отвечает функция bind(). Даже если вызов функции bind() не был произведен, операционная система назначит сокету один из доступных локальных портов.

Привилегированные порты

Система безопасности ядра сконфигурирована таким образом, что для доступа к портам с номерами, меньшими 1024, необходимо иметь права пользователя root. Например, если требуется запустить сервер времени (порт 13), это должен сделать пользователь root или принадлежащая ему программа, для которой установлен бит SUID. Данная аббревиатура расшифровывается как "set user ID" — смена идентификатора пользователя. Программа с установленным .битом SUID всегда будет выполняться так, будто ее запустил владелец, даже если на самом деле это был другой пользователь. Например, общедоступная утилита /usr/bin/at (пользовательский планировщик заданий) должна обращаться к системным cron-таблицам, принадлежащим пользователю root. При этом она автоматически переключается в режим суперпользователя. Подробнее о данной возможности рм. в документации к системе UNIX. Будьте осторожны: бит SUID потенциально несет в себе угрозу безопасности системы. Если вы являетесь суперпользователем, устанавливайте данный бит только для тех программ, о которых известно, что они совершенно безопасны.

Как отмечалось ранее, если с сокетом не был связан порт, операционная система сделает это автоматически. Такого рода динамически назначенные порты называются *эфемерными*. Linux следует правилу BSD-систем: эфемерным портам назначаются номера 1024 и выше.

Безопасная отладка

Экспериментируя с различными портами, следуйте простым правилам:

* создавайте программу, запускайте ее на выполнение и отлаживайте как обычный пользователь (не пользователь root);
* используйте безопасные номера портов и адреса::
* ведите журнал всех поступающих сообщений.

Порядок следования байтов

В сеть могут входить компьютеры разных архитектур и платформ. А самое главное, что в них могут быть разные процессоры. Не все процессоры хранят двоичные числа одинаково. Существуют два базовых формата хранения: *прямой* (от младшего к старшему) и *обратный* (от старшего к младшему). Числа, хранящиеся в обратном порядке, читаются слева направо, а числа, хранящиеся в прямом порядке, — справа налево. Предположим, имеется число 214259635. В шестнадцатеричном виде оно записывается как OxOCC557B3. Процессор с обратным порядком байтов хранит его следующим образом:

```
Адрес:    00    01    02    03    04
Данные:   OC    C5    57    B3    ...
```

Обратите внимание на то, что старший байт (OC) указан первым. Процессор с прямым порядком байтов хранит число по-другому:

```
Адрес:    00    01    02    03    04
Данные:   B3    57    C5    DC    ...
```

Теперь первым указан младший байт (B3).

Необходимость существования различных порядков следования байтов давно оспаривается. Я не хочу возвращаться к этим дискуссиям, остановлюсь лишь на некоторых важных моментах. "Почему, — спросите вы, — нельзя хранить шестнадцатеричные данные в ASCII-формате?" Такое представление неэффективно, так как число используемых байтов удваивается. Компьютеры, которые хотят эффективно взаимодействовать в гетерогенных сетях, должны использовать двоичный формат и выбрать требуемый порядок следования байтов. Порядок байтов, задаваемый компьютером или сервером, называется *серверным*. Порядок байтов, определяемый сетевыми протоколами, называется *сетевым* и всегда является обратным.

Функции преобразования данных

Много лет назад было решено, что в сетях должен применяться обратный порядок следования байтов. Но как быть с процессорами, использующими прямой порядок? Существует ряд средств, позволяющих выполнить соответствующее преобразование. Они находят широкое применение в сетевых приложениях, заполняющих поля структур семейства sockaddr. Тут есть одна тонкость: не все поля структуры имеют сетевой порядок следования байтов. Некоторые из них пред-

ставлены в серверном порядке. Рассмотрим фрагмент программы из главы 1, "Простейший сетевой клиент":

```
/*************************************************************/
/***     Пример преобразования порядка следования байтов:     ***/
/***     заполнение структуры sockaddr_in.                    ***/

struct sockaddr_in dest;
char *dest_addr = "127.0.0.1";

dest.sin_family = AF_INET;
dest.sin_port = htons(13);    /* порт 13 (сервер времени) */
if (inet_aton(dest_addr, &dest.sin_addr) == 1) {
```

Здесь заполняются три поля: sin_family, sin_port и sin_addr. Первое имеет серверный порядок байтов, поэтому не требует преобразования. Следующие два имеют сетевой порядок. Значение двухбайтового поля sin_port (13) преобразуется с помощью функции htons(), входящей в группу из четырех функций.

Функция	Преобразование	Описание
htons ()	Из серверного порядка в сетевой короткий	Представляет 16-разрядное число в обратном порядке
htonl()	Из серверного порядка в сетевой длинный	Представляет 32-разрядное число в обратном порядке
ntohs ()	Из сетевого порядка в серверный короткий	Представляет 16-разрядное число в серверном порядке
ntohl ()	Из сетевого порядка в серверный длинный	Представляет 32-разрядное число в серверном порядке

Эти функции подробно описаны в приложении Б, "Сетевые функции". Возможно, читателям будет интересно узнать, что перечисленные функции не занимают время процессора, если на компьютере используется обратный порядок следования байтов.

Функция inet_aton() преобразует IP-адрес из точечной записи формата ASCII в эквивалентную двоичную форму, представленную в сетевом порядке (т.е. после нее не требуется вызывать функцию htonl()). Эта функция также входит в группу функций преобразования.

Функция	Описание
inet_aton()	Преобразует адрес из точечной записи **(###.###.###.###) в** двоичную форму с сетевым порядком следования байтов; возвращает нуль в случае неудачи и ненулевое значение, если адрес допустимый
inet_addr()	Устарела (аналог inet_aton()), так как неправильно обрабатывает ошибки; при возникновении ошибки возвращает -1 (хотя 255.255.255.255 — обычный широковещательный адрес)
inet_ntoa()	Преобразует IP-адрес, представленный в двоичном виде с сетевым порядком следования байтов, в точечную запись формата ASCII
gethost-byname ()	Просит сервер имен преобразовать имя (такое как www.linux.org) в один или несколько IP-адресов

В библиотеках могут содержаться дополнительные функции преобразования. В данной книге рассматриваются только те из них, которые часто используются. Дополнительная информация об этих функциях содержится в приложении Б, "Сетевые функции".

Как добиться переносимости

Те, кто думают, что обо всех этих преобразованиях можно не заботиться, должны вспомнить: сама идея написания Linux заключалась в том, чтобы сделать данную ОС совместимой с другими системами. Не так уж и трудно использовать указанные функции на разных платформах. Хорошим стилем программирования считается написание программы таким образом, как если бы ее нужно было перенести на другой процессор.

Расширение клиентского приложения

Следующий шаг в расширении функциональных возможностей нашего клиента заключается в добавлении к нему способности послать сообщение и получить ответ. Эта задача решается с помощью функции write():

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

Аналогично функции read(), первый параметр может быть как дескриптором файла, так и дескриптором сокета.

Вот перечень наиболее распространенных кодов ошибок, которые могут возникнуть при работе функции.

- EBADF. Указан неверный дескриптор файла либо файл не открыт для чтения. Эта ошибка появляется, если программа уже закрыла дескриптор сокета или сокет был открыт неправильно (проверьте значение, возвращенное функцией socket()).
- EINVAL. Указанный дескриптор связан с объектом, запись в который невозможна. Это может случиться, если программа закрыла канал записи.
- EFAULT. В качестве параметра задан неправильный адрес. Указатель buf ссылается на недоступную область памяти, при обращении к которой произошло нарушение сегментации,
- EPIPE. Дескриптор связан с каналом -или сокетом, который закрыт на противоположной стороне. В этом случае процессу, осуществляющему запись, посылается сигнал SIGPIPE, и если он его принимает, то функция write() генерирует ошибку EPIPE. Данная ошибка может возникнуть только при втором вызове функции. Примерный сценарий таков:

1) клиент подключается к серверу и посылает данные;

2) сервер принимает часть сообщения, после чего разрывает соединение (или происходит сбой системы);

3) ничего не подозревающий клиент посылает следующую порцию данных.

Ниже показан фрагмент программы, представляющий собой типичный пример записи данных в сокет с помощью функции write():

```
int sd, bytes_written = 0, retval;
sd = socket(PF_INET, SOCK_STREAM, 0);

/*** Подключение к серверу ***/
while ( bytes_written < len)     /* цикл повторяется до тех пор, */
{                    /* пока не будут отправлены все байты сообщения */
    retval = write(sd, buffer+bytes_written, len);
    if ( retval >= 0)
      bytes_written+=retval;
    else
      /* сообщение об ошибке */
}
```

А вот как можно сделать то же самое с помощью функции fprintf(), которой передается дескриптор сокета, приведенный к типу FILE*:

```
FILE  *sp
int sd;
Sd  =  socket(PF_INET, SOCK_STREAM, 0);

/*** Подключение к серверу ***/
sp = fdopen(sd, "w");            /* создание файлового дескриптора
                                                  на основе сокета */
if ( sp == NULL)
  perror("FILE* conversion failed");
fprintf(sp, "%s, %s, %s\n", Name, Address, Phone);
```

Обратите внимание на то, что в первом случае программе приходится выполнять цикл while для передачи всех данных. Хотя мы и имеем дело с сокетом, нет гарантии, что программа сможет послать данные за один прием. Во втором случае подобного ограничения нет, поскольку с дескриптором типа FILE* связана отдельная подсистема буферизации: при записи данных в буфер профамма вынуждена ждать, пока не будут отправлены все данные.

Существует также специализированная функция send(), связанная исключительно с сокетами. Она позволяет профаммисту управлять процессом передачи данных подобно тому, как это происходит в случае чтения данных с помощью функции recv(). Объявление функции выглядит так:

```
#include<sys/socket.h>
#include <resolv.h>
int send(int sd, const void *msg, int len, unsigned int flags);
```

Назначение первых трех параметров такое же, как и в функции write(). С помощью флагов, указанных в четвертом параметре, можно контролировать работу сокета.

- MSG_OOB. Режим внеполосной передачи. Позволяет послать серверу или другому сетевому компьютеру один байт, обозначающий срочное сообщение. Когда приходит внеполосное сообщение, операционная система посылает профамме сигнал SIGURG.
- MSG_DONTROUTE. Запрет маршрутизации пакета. Пакет не проверяется с помощью таблиц маршрутизации и должен быть доставлен адресату на-

прямую. Если адресат недостижим, будет получен код ошибки ENETUNREACH (сеть недоступна). Подобная опция применяется только в программах диагностики и маршрутизации.

- MSG_DONTWAIT. Не ждать завершения функции send(). Это позволяет программе продолжить работу, как будто функция была выполнена. Когда данные будут переданы, операционная система пошлет программе сигнал SIGIO. Если запись невозможна из-за переполнения очереди функции send(), будет возвращено отрицательное число, а в переменную errno будет записано значение EAGAIN.
- MSG_NOSIGNAL. Не посылать сигнал SIGPIPE. Когда по какой-то причине противоположный конец сокета досрочно закрывается, программа получает сигнал SIGPIPE. Если программа не готова его обработать, она будет завершена.

Флаги можно объединять с помощью операции побитового сложения. Вот типичный пример использования функции send():

```
/*************************************************************/
/***              Запись данных в канал сокета             ****/
/*************************************************************/
int bytes_sent;
bytes_sent = send(sd, buffer, MAXBUF, 0);

/*——————————Передача внеполосных данных——————————*/
int bytes_sent;
bytes_sent = send(sd, buffer, MAXBUF, MSG_OOB | MSG_NOSIGNAL);
```

Ниже перечислено несколько кодов ошибок, которые могут возникать при работе функции send().

- EBADF. Указан неверный дескриптор. Очевидно, программа забыла проверить результат функции socket().
- ENOTSOCK. Указанный дескриптор не связан с сокетом.
- EMSGSIZE. Сокет попросил ядро послать сообщение единым блоком, но размер сообщения оказался слишком велик. Эта ошибка может возникать при передаче широковещательных сообщений (которые не могут быть фрагментированы) или в случае, когда программа установила для сокета режим нефрагментированной передачи.
- EAGAIN. Сокет установлен в режим неблокируемой передачи, но запрашиваемая операция приведет к блокировке. В действительности это не ошибка, а признак того, что сокет "не готов". Попробуйте послать данные позднее.
- EPIPE. Противоположный конец сокета был закрыт. Программа также получит сигнал SIGPIPE, если только не установлен флаг MSG_NOSIGNAL.

Функция send() обычно применяется для передачи серверу служебной информации. Например, если клиент хочет вызвать на сервере программу Finger, он открывает порт 79, посылает имя пользователя и ждет ответа. При подобном алгоритме не требуется указания дополнительных опций. Но иногда размер принимаемых данных превышает размер входного буфера. В приведенном ниже

фрагменте программы демонстрируется, как разрешить такого рода проблему (полный текст программы имеется на Web-узле):

```
/*************************************************************/
/*** Расширение клиентской программы: добавлена возможность ****/
/*** доступа к любому порту и отправки сообщения.           ****/

int main(int count, char *strings[])
{   int sockfd;
    struct sockaddr_in dest;
    char buffer[MAXBUF];

    /*— Создание сокета и назначение ему порта —*/
    sockfd = socket(PF_INET, SOCK_STREAM, 0);
    bzero(&dest, sizeof(dest));
    dest.sin_family = AF_INET;
    dest.sin_port = htons(atoi(strings[2]));
    inet_addr(strings[1],   &dest.sin_addr.s_addr);

    /*— Подключение к серверу и передача ему запроса —*/
    if ( connect(sockfd, sdest, sizeof(dest)) != 0 )
      PANIC("connect() failed");
    sprintf(buffer, "%s\n", strings[3]);
    sendfsockfd, buffer, strlen(buffer), 0);
    /*— Очистка буфера и чтение КОРОТКОГО ответа —*/
    bzero(buffer, MAXBUF);
    recv(sockfd, buffer, MAXBUF-1, 0);
    printf("%s", buffer);
    close(sockfd);
    return 0;
}
```

Если необходимо использовать эту программу для приема от сервера больших блоков данных, замените последнюю часть программы:

```
/*************************************************/
/***   Модификация кода, позволяющая принимать   ***/
/***   длинные сообщения.                        ***/
/*************************************************/
/*— Очистка буфера и чтение ДЛИННОГО ответа —*/
do
{
   bzero(buffer, MAXBUF);
   bytes = recv(sockfd, buffer, MAXBUF, 0);
   printf("%s", buffer);
}
while ( bytes > 0);
close(sockfd);
```

Данный код функционирует правильно, если по окончании передачи данных сервер закрывает соединение. В противном случае программа "зависнет".

Ожидание данных — это особая проблема, возникающая при программировании сокетов. Иногда можно быть уверенным в том, что сервер закроет соединение, закончив передавать данные. Но некоторые серверы предпочитают оставлять канал открытым, пока сам клиент не закроет его.

Различные виды пакетов

В сети поддерживается много разных протоколов. У каждого из них имеются свои функции и особенности, но передача данных все равно осуществляется в виде пакетов. Типы пакетов могут быть следующими.

Именованные сокеты	PF_LOCAL	Подключение к сети не происходит; выполняется обработка очередей средствами файловой системы
Стек протоколов TCP/IP	PF_INET	Передача данных по сетям TCP/IP
Стек протоколов Novell	PF_IPX	Передача данных по сетям Novell
Стек протоколов AppleTalk	PF_APPLETALK	Передача данных по сетям AppleTalk

Подробнее о различных протоколах можно узнать в приложении А, "Информационные таблицы". В каждом протоколе существует своя система обозначений и соглашений, но все они используют библиотеку Socket API, что существенно упрощает программирование. К сожалению, создание сокета, работающего со всеми протоколами, — слишком сложная задача, поэтому в данной книге мы сосредоточимся на сокетах TCP/IP.

Именованные каналы в UNIX

Существует проблема: как скоординировать работу программ, посылающих данные и сообщения об ошибках в разное время? Ее можно решить, создав системный журнальный файл. Все необходимые для этого функции имеются в библиотеке Socket API.

Именованные сокеты позволяют нескольким программам передавать сообщения (или пакеты) через одно и то же соединение. Они называются *именованными,* потому что в действительности создают файл на диске. Все взаимодействие локализовано в рамках одной системы; данные не передаются по сети, и ни один сетевой клиент не может подключиться к такому сокету.

Работа с именованным сокетом ведется так же, как и с обычным: создается соединение либо по протоколу TCP, либо UDP. Единственное отличие заключается в структуре sockaddr. Приведем пример:

```
/*****************************************************************/
/***                  Пример именованного канала                 ****/
/*****************************************************************/
linclude <sys/un.h>
int  sockfd;
struct sockaddr_un addr;
sockfd = socket(PF_LOCAL, SOCK_STREAM, 0);
bzerof(&addr, sizeof(addr));
addr.sun_family = AF_LOCAL;
strcpy(addr.sun_path,    "/tmp/mysocket");          /* назначаем файл */
if ( bind(sockfd, Saddr, sizeof(adrr)) != 0 )
  perror("bind() failed");
```

Работа программы должна быть понятна читателям. Поле sun_path может содержать строку длиной до 104 байтов (включая завершающий символ NULL). Далее к

сокету можно применять стандартные библиотечные функции. То же самое справедливо в отношении других протоколов.

После запуска программы загляните в каталог /tmp, и вы увидите там новый файл. Не забудьте его удалить, прежде чем запускать программу повторно.

Резюме: IP-адреса и средства преобразования данных

Библиотека Socket API является чрезвычайно гибким средством сетевого программирования. Она поддерживает множество протоколов, позволяя подключаться к различным сетям и взаимодействовать с ними.

В TCP/IP применяется схема адресации, основанная на маршрутизации сообщений и объединении компьютеров в группы. Это позволяет передавать сообщения без участия отправителя: сообщение, поступившее в сеть, достигает адресата посредством маршрутизации и ARP-таблиц. Однако используемый механизм приводит к тому, что много адресов резервируются и "выпадают" из адресного пространства.

В TCP/IP существует также понятие порта. Большинство клиентских приложений подключается к портам для взаимодействия с конкретными программами на сетевых компьютерах. Это существенно уменьшает поток сообщений.

Данные, передаваемые по сети, могут иметь разный порядок следования байтов и требуют применения специальных функций преобразования. В библиотеке Socket API определен целый ряд таких функций, одни из которых выполняют преобразование адресов (inet_addr(), inet_aton(), inet_ntoa()), а другие меняют порядок следования байтов (htons(), ntohs(), htonl()).

Программа по-разному взаимодействует с протоколами TCP и UDP. В следующей главе будут рассмотрены различные типы протоколов Internet и проанализированы особенности каждого из них.

Глава

3 Различные типы Internet-пакетов

В этой главе:

В физической сети поддерживается несколько типов логических сетевых архитектур, таких как сети Novell (IPX), Microsoft (NetBEUI), AppleTalk и, конечно же, TCP/IP. Но во всех архитектурах данные передаются в виде пакета, который в общем случае состоит из идентификаторов отправителя и получателя, а также собственно данных. У каждой архитектуры есть свой набор функций и стек протоколов, свои слабые и сильные стороны.

В Internet имеется четыре основных типа пакетов: неструктурированный, ICMP, UDP (передача данных без подтверждения доставки) и TCP (потоковая передача). Все они так или иначе связаны с физическим уровнем сети (рис. 3.1). В настоящей главе рассматривается каждый из этих типов, анализируются его преимущества и недостатки, демонстрируются типичные случаи его применения.

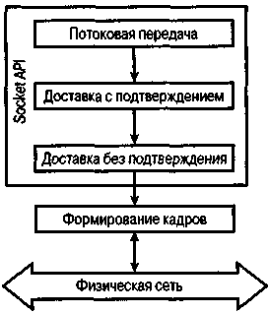

Рис. 3.1. Библиотека Socket API обеспечивает различные уровни доставки сообщений

Базовый сетевой пакет

Если бы нам представилась возможность просмотреть биты, передаваемые от одного компьютера к другому, что бы мы увидели? Все протоколы имеют различную структуру, но одна черта у них общая: они передают данные, посланные программой. Некоторые протоколы включают в сообщение адрес отправителя, другие требуют указания адреса получателя. Последнее кажется совершенно очевидным, но есть протоколы (например, UUCP), в которых получатель идентифицируется автоматически на противоположном конце соединения.

Протокол IP (Internet Protocol) [RFC791] требует, чтобы пакет содержал три базовых элемента: адрес отправителя, адрес получателя и данные. Благодаря наличию такой информации пакет становится автономным. Приняв его в любой точке сети, можно легко узнать, откуда он поступил, куда направляется и каков его размер.

Независимость пакетов является важным свойством сети Internet. Если пакет *релевантен* (содержит актуальные данные, направленные по правильному адресу), маршрутизатор способен переслать его в нужную точку сети.

Замещение пакетов

Автономность пакетов имеет также отрицательную сторону. Поскольку пакет может быть послан откуда угодно куда угодно, злобные хакеры способны обмануть сетевое обеспечение. В сети не требуется проверять достоверность адреса отправителя. Выполнить замещение аппаратного адреса (заменить истинного отправителя другим) трудно, но возможно. Необходимо отметить, что в последних версиях ядра Linux замещение адресов запрещено.

Как рассказывалось в главе 2, "Основы TCP/IP", пакеты имеют сетевой (обратный) порядок следования байтов. Памятуя об этом, давайте рассмотрим программную (листинг 3.1) и физическую (рис. 3.2) структуры пакета.

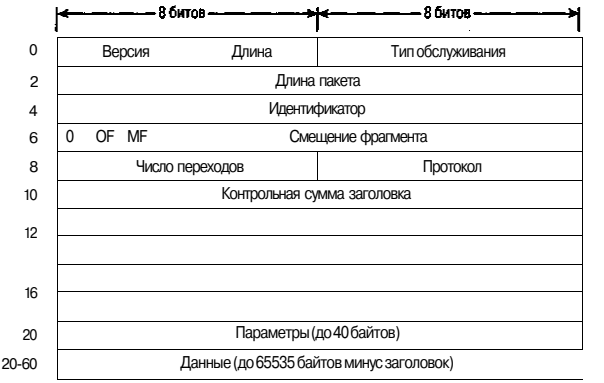

Рис. 3.2. Структура IP-пакета

Листинг 3.1. Определение структуры IP-пакета

```
/*****************************************************************/
/***                    Структура IP-пакета                    ***/
/*****************************************************************/

#typedef unsigned int uint;
#typedef unsigned char uchar;

struct ip_packet {
   uint version:4;      /* версия */
   uint header_len:4;   /* длина заголовка в двухбайтовых
                           словах  */
   uint serve_type:8;   /* правила обслуживания пакета */
   uint packet_len:16;  /* общая длина пакета в байтах */
   uint ID:16;          /* идентификатор пакета */
   uint  _reserved:1;   /* всегда равно 0 */
   uint  dont_frag:1;   /* флаг, запрещающий фрагментацию */
   uint more_frags:1;   /* флаг наличия последующих фрагментов  */
   uint  frag_offset:13; /* смещение фрагмента */
```

```
    uint time_to_live:8; /* число переходов через маршрутизатор */
    uint protocol:8;     /* протокол: ICMP, UDP, TCP */
    uint hdr_chksum:16;  /* контрольная сумма заголовка */
    uint IPv4_source:32; /* IP-адрес отправителя */
    uint IPv4_dest:32;   /* IP-адрес получателя */
    uchar options[];     /* до 40 байтов служебных данных */
    uchar data[];        /* до 64 Кбайт данных */
}
```

Заметьте, что пакет содержит не только те базовые поля, о которых говорилось ранее, но и много других. Все эти дополнительные поля позволяют IP-подсистеме контролировать прохождение пакета. Например, поле dont_frag сообщает о том, что сообщение нельзя разбивать на блоки: оно должно быть передано целиком либо не передано вовсе.

Назначение некоторых полей понятно уже из тех комментариев, которые приведены в листинге 3.1. Остальные поля требуют дополнительного описания. В следующих разделах рассматриваются те поля, которые можно модифицировать или которые представляют интерес для программиста.

Поле version

В первом поле передается номер версии используемого протокола IP. Большинство значений либо зарезервировано, либо не используется. Ниже перечислен ряд доступных значений (табл. 3.1).

Таблица 3.1. Значения поля version

Значение	Протокол
4	IPv4
5	Режим потоковой передачи дейтаграмм (экспериментальный протокол)
6	IPv6
7	TP/IX (Internet-протокол следующего поколения)
8	Internet-протокол "P"
9	Протокол TUBA (TCP and UDP with Bigger Addresses — протоколы TCP и UDP с расширенными адресами)

Заполнить это поле можно только в том случае, если вы создаете неструктурированный сокет *плюс* указываете на то, что будете заполнять заголовок самостоятельно (для этого служит параметр сокета IP_HDRINCL). Но даже в этом случае в поле можно указать значение 0. Нулевой флаг информирует ядро о том, что в данное поле следует подставить нужное значение.

Поле header_len

В данном поле указывается длина заголовка в виде количества двухбайтовых слов. Наибольшее допустимое значение — 15 (60 байтов). Опять-таки, единственная ситуация, при которой необходимо заполнять это поле, — когда создается неструктурированный сокет и установлена опция IP_HDRINCL. Поскольку все IP-

заголовки содержат не менее 20 байтов, минимальное значение данного поля равно 5 (20/4).

Поле serve__type

Данное поле указывает на то, как следует обрабатывать пакет. В нем выделяются два подчиненных поля: первое обозначает приоритет (игнорируется в большинстве систем), а второе — тип обслуживания (TOS — type of service). Последний обычно задается с помощью функции setsockopt() и может иметь четыре значения: минимальная задержка, максимальная пропускная способность, максимальная надежность и минимальная стоимость. Отсутствие атрибута указывает на обычный тип обслуживания (подробно об этом рассказывается в главе 9, "Повышение производительности").

Поле ID

IP-подсистема присваивает каждому пакету уникальный идентификатор. Поскольку данное поле занимает всего 16 разрядов, несложно понять, что идентификаторы быстро исчерпываются. Тем не менее, к тому времени, когда возникнет необходимость повторно задействовать идентификатор, предыдущий пакет с таким же идентификатором, скорее всего, уже достигнет пункта назначения.

Идентификаторы позволяют собирать фрагментированные пакеты. Когда установлен режим ручного заполнения заголовка (IP_HDRINCL), программисту приходится самостоятельно заниматься назначением идентификаторов.

Пользовательские идентификаторы

При самостоятельном создании идентификаторов обязательно помните: ваша программа не единственная, кто может посылать сообщения. IP-подсистема отслеживает идентификаторы всех проходящих пакетов. Будьте предельно внимательны при выборе идентификаторов, чтобы свести к минимуму риск совпадений.

Поля dont_frag, more__frag и frag_offset

Эти поля управляют фрагментацией пакетов. При прохождении длинного пакета через сетевой сегмент с ограниченной пропускной способностью (задано ограничение на размер физического кадра) маршрутизатор может попытаться разбить пакет на составные части *(фрагменты)*. "Сборка" пакета осуществляется уже в пункте назначения. Применение фрагментации снижает производительность, поскольку каждый фрагмент передается в отдельном пакете со своим заголовком.

Системный сборщик пакетов

Если ваш компьютер выполняет функции маршрутизатора, можно осуществлять сборку фрагментированных пакетов с помощью ядра Linux. Обратитесь к утилите конфигурирования ядра (раздел, посвященный брандмауэрам и маршрутизаторам). Учтите, что" сборка пакетов требует времени, особенно если фрагменты поступают со значительным интервалом. Но поскольку получатель все равно должен будет собрать пакет, включение данного режима позволит уменьшить сетевой трафик в пределах брандмауэра.

Флаг dont_frag сообщает о том, что маршрутизатор или сервер не должны разбивать пакет. Если этот флаг установлен, а пакет слишком велик, чтобы пройти через сетевой сегмент, маршрутизатор удалит пакет и вернет ICMP-пакет с сообщением об ошибке.

Флаг more_frags сообщает получателю о том, что следует ожидать дополнительных фрагментов. В последнем фрагменте этот флаг сброшен (как и в нефрагментированных пакетах). При самостоятельном заполнении заголовка всегда устанавливайте этот флаг равным 0.

Поле frag_offset указывает на то, в какой части пакета должен находиться данный фрагмент. Поскольку пакеты распространяются по сети разными путями, они могут достигать адресата в разное время. Зная смещение фрагмента в пакете, получатель может безошибочно собрать пакет. Поле frag_offset занимает всего 13 разрядов — этого слишком мало для пакета размером 64 Кбайт. Поэтому значение поля умножается на 8 при вычислении реального смещения. Это также означает, что размер каждого фрагмента (кроме последнего) должен быть кратен 8. Но не беспокойтесь по поводу различных технических подробностей: IP-подсистема автоматически управляет фрагментацией и сборкой пакетов. Если пакет не удалось собрать в пределах определенного времени, он будет удален и отправитель получит сообщение об ошибке.

Поле time_to_live (TTL)

Первоначально в данном поле задавалось число секунд, в течение которых пакет должен существовать при передаче по сети. Впоследствии в нем стали указывать число переходов. *Переходом* называется этап ретрансляции пакета при передаче через компьютер или маршрутизатор *(узел сети),* когда пакет удаляется из одной сети и попадает в другую.

Поле time_to_live занимает 8 разрядов, что позволяет ретранслировать пакет 255 раз, прежде чем он будет удален. Когда маршрутизатор или ретранслирующий сервер получает пакет, он уменьшает значение данного поля на единицу. Если поле станет равным нулю до того, как пакет достигнет адресата, узел сети удалит пакет и пошлет отправителю сообщение об ошибке. Подобная схема предотвращает бесконечное "петляние" неправильного пакета по сети.

Значение поля time_to_live можно задать с помощью параметра IP_TTL сокета (подробно об этом — в главе 9, "Повышение производительности"). Кроме того, его можно установить в процессе ручного заполнения заголовка пакета.

Поле protocol

Каждый пакет в сети Internet предается по тому или иному протоколу: ICMP (IPPROTO_ICMP, или 1), UDP (IPPROTO_UDP, или 17) или TCP (IPPROTO_TCP, или 6). Данное поле сообщает системе о том, как следует интерпретировать пакет. Этот параметр можно задать вручную, если при вызове функции socket() указать константу SOCK_RAW. Номера протоколов и связанные с ними символические константы определены в системном библиотечном файле netinet/in.h. (Не забывайте о том, что, даже если номер протокола определен в файле, сам протокол может не поддерживаться.)

Поле options

IP-подсистема может задавать в пакете различные параметры. Это может быть информация для маршрутизатора, метка времени, атрибуты безопасности, предупреждение и т.д. Максимальный размер поля составляет 40 байтов. Поскольку некоторые параметры являются системно-зависимыми, устанавливать их вручную нежелательно.

Поле data

В этом поле передается собственно сообщение, занимающее до 65535 байтов (минус 60 байтов — максимальный размер заголовка). В разделе данных могут передаваться заголовки протоколов более высокого уровня. Например, для ICMP требуется 4 байта, для UDP — 8, а для TCP — 20-60.

Описанную структуру имеют все пакеты IPv4. Протоколы верхних уровней добавляют к ней свои данные и средства повышения надежности.

Анализ различных типов пакетов

В Internet имеется несколько типов пакетов, от очень быстрых до очень надежных. Все они основаны на базовой структуре IP-пакета, но в то же время каждый из них предназначен для решения определенных задач. Чтобы правильно выбрать тип пакета, необходимо знать, какого рода данные передаются.

Наиболее распространены пакеты TCP, UDP, ICMP и неструктурированного типа. У каждого имеются свои преимущества и недостатки (табл. 3.2).

Таблица 3.2. Сравнительные характеристики различных типов пакетов

	Неструктурированные данные	ICMP	UDP	TCP
Служебные данные (байты)	20-60	20-60+[4]	20-60+[8]	20-60+[20-60]
Размер сообщения (байты)	65535	65535	65535	(не ограничен)
Надежность	Низкая	Низкая	Низкая	Высокая
Тип сообщения	Дейтаграмма	Дейтаграмма	Дейтаграмма	Поток
Пропускная способность	Высокая	Высокая	Средняя	Низкая
Целостность данных	Низкая	Низкая	Средняя	Высокая
Фрагментация	Поддерживается	Поддерживается	Поддерживается	Низкая

Характеристики протоколов, представленные в таблице, не следует воспринимать буквально. Если надежность отмечена как низкая, то это не означает, что данный протокол не гарантирует доставку сообщений. На основании приведенных данных можно лишь сравнить протоколы друг с другом.

Характеристики пакетов

Передача данных по каждому из протоколов связана с определенными особенностями. В следующих разделах подробно рассмотрены характеристики типов пакетов, перечисленные в табл. 3.2. Приведенная информация поможет читателю понять, почему тот или иной протокол одни возможности реализует, а другие — нет.

Размер служебных данных

К служебным данным относится заголовок пакета, а также данные, передаваемые при инициализации протокола. Большой размер служебных данных приводит к снижению пропускной способности, поскольку сеть вынуждена тратить больше времени на передачу заголовков, чем на чтение данных.

Строгие алгоритмы синхронизации и квитирования (установления связи) увеличивают объем информации, передаваемой при инициализации протокола. Это особенно заметно в глобальных сетях в связи с задержками распространения пакетов. В табл. 3.2 эти данные не учтены.

Размер сообщения

Чтобы вычислить пропускную способность сети, необходимо знать размер пакета и служебных данных. В сумме они определяют максимальный размер передаваемого сообщения. Поскольку во всех протоколах, кроме TCP, сообщения посылаются одноразово, их размер зависит от размера IP-пакета (65536 байтов).

Надежность

Одной из проблем, возникающих в сети, является потеря данных. Сообщение может повредиться или потеряться при переходе от одного компьютера или маршрутизатора к другому. Кроме того, может произойти сбой самих аппаратных средств. В любом случае программе придется передать сообщение повторно.

Необходимо также убедиться в том, что получатель обработает пакет правильным образом. Например, сообщение может не вмещаться в рамки одного пакета. Если второй пакет достигнет пункта назначения раньше первого, получатель должен корректно разрешить данную проблему. С другой стороны, порядок не важен, если сообщения независимы и самодостаточны.

Надежность пакета определяет степень вероятности, с которой можно гарантировать его доставку. Низкая надежность означает, что протокол не гарантирует доставку пакета и правильный порядок в случае фрагментации.

Тип сообщения

Некоторые сообщения являются самодостаточными и не зависят от других сообщений. Небольшие изображения, текстовые файлы, электронные письма могут посылаться одним пакетом. Но есть информация, которая должна передаваться в виде потока, например данные сеанса Telnet, данные канала HTTP, крупные документы, двоичные файлы и т.д. Тип сообщения определяет, в каком виде лучше всего отправлять данные по тому или иному протоколу.

Протокол HTTP

В протоколе HTTP 1.0 данные вполне могли бы передаваться в виде UDP-, а не TCP-пакетов. Клиент просто посылает серверу запрос на Конкретный документ, а сервер в ответ высылает файл. По сути, никакого взаимодействия между клиентом и сервером не происходит.

Пропускная способность

Наиболее важным аспектом передачи данных является пропускная способность сети. Чем она выше, тем спокойнее работается пользователям. Ее необходимо знать, чтобы добиться наилучшей производительности. Число битов в секунду еще ничего не означает, это только часть уравнения: данный показатель указывает лишь на то, как могла бы работать сеть в идеальных условиях.

Пропускная способность определяет, сколько данных реально может быть передано в течение заданного промежутка времени. Если размер заголовка велик, а блока данных — мал, пропускная способность будет низкой. Еще больше она снижается, когда требуется подтверждение доставки каждого сообщения. Как правило, высокая надежность и целостность означают низкую пропускную способность и наоборот.

Целостность данных

Современные сетевые технологии предусматривают множество способов обеспечения целостности данных. В некоторых протоколах в каждое низкоуровневое сообщение включается контрольная сумма или CRC-код (Cyclical Redundancy Check — контроль с помощью циклического избыточного кода). Существуют аппаратные средства фильтрации шумов и выделения значащих данных. Кроме того, каждый протокол содержит механизмы обнаружения ошибок в передаваемых данных.

Важность поддержания целостности данных зависит от самих данных. Приведем примеры.

- Ошибки недопустимы — жизненно важные данные. Все, что может быть связано со здоровьем/жизнью: сигналы от медицинского оборудования, команды ракетных установок и т.п.
- Критические данные — особо важные, высоконадежные данные, которые при неправильной передаче могут нанести вред собственности или безопасности. Например, финансовые транзакции, кредитные карточки, цифровые подписи, обновления антивирусных баз данных.
- Важные данные — данные, связанные с правильным функционированием программ. В качестве примера можно привести файлы, загружаемые по протоколу FTP, Web-страницы, адреса серверов/маршрутизаторов.
- Информационное содержимое — данные, для которых не требуется стопроцентная надежность. Это могут быть электронные сообщения, списки новостей, те же Web-страницы.
- Временные данные — данные, которые важны в течение определенного периода времени. Если до истечения определенного срока программа их не запрашивает, их важность уменьшается. Это могут быть сводки погоды, программы телепередач и др.

- Данные с потерями — данные, которые могут быть частично потеряны без заметного снижения полезности. К таковым относятся аудио- и видеоклипы, изображения, ну и, конечно же, спам.

Прежде чем выбрать тип пакета и протокол, попытайтесь определить тип своих данных согласно приведенной классификации. Кроме того, в каждой программе могут существовать свои собственные ограничения целостности.

Фрагментация

Большие сообщения, передаваемые в медленных сетях, могут тормозить работу других пользователей. Во всех сетях существует ограничение на максимальный размер кадра, чтобы такие сообщения не загружали сеть. Кроме того, маршрутизатор может разбивать сообщение на фрагменты при передаче их в сеть с ограниченной пропускной способностью.

Поскольку сборка фрагментированных сообщений является функцией протокола IP, этот процесс может быть прозрачным для протоколов более высокого уровня. Но в некоторых ситуациях пакет должен поступать целиком. Это особенно важно с точки зрения производительности. Когда маршрутизатор разделяет пакеты на более мелкие блоки, он тратит на это время, к тому же увеличивается число служебных данных, передаваемых в заголовках пакетов. Если фрагментация запрещена, а сообщение слишком велико, оно удаляется, а программе посылается сообщение об ошибке.

Типы пакетов

В следующих разделах рассмотрен каждый из типов пакетов с описанием структуры заголовка, если таковая имеется.

Неструктурированные данные

Неструктурированные данные напрямую записываются в IP-пакет. Это может быть полезно при работе со специальными или пользовательскими протоколами. Атрибуты данного пакета перечислены в табл. 3.3.

Таблица 3.3. Характеристики пакета неструктурированных данных

Служебные данные (байты)	20-60
Размер сообщения (байты)	65535 (65515 — максимальный объем полезных данных)
Надежность	Низкая (сеть может терять или переупорядочивать пакеты)
Тип сообщения	Дейтаграмма
Пропускная способность	Высокая (малая нагрузка на сеть)
Целостность данных	Низкая (система не проверяет сообщения)
Фрагментация	Поддерживается

Linux позволяет работать с протоколами различных уровней стека TCP/IP (подробное описание сетевых уровней и стека Internet-протоколов дано в главе 5, "Многоуровневая сетевая модель"). Базовое сообщение в TCP/IP представлено в виде неструктурированного IP-пакета. Оно не содержит никакой дополнительной информации.

Создать IP-пакет можно самостоятельно, предварительно указав при вызове функции socket() константу SOCK_RAW. По соображениям безопасности пользователь, запускающий программу, в которой создается сокет данного типа, должен иметь привилегии пользователя root.

Неструктурированные сокеты позволяют программисту самостоятельно формировать IP-пакет. Можно сконфигурировать сокет двумя способами: для передачи только данных или данных плюс заголовок. В первом случае передача будет организована по типу протокола UDP, но только не будут поддерживаться порты. Во втором случае необходимо напрямую заполнять поля заголовка пакета.

У пакетов данного типа есть как преимущества, так и недостатки. К недостаткам можно отнести то, что доставка сообщений не гарантируется. Но зато сообщения могут распространяться по сети с максимальной скоростью. Подробнее об обработке неструктурированных данных рассказывается в главе 18, "Неструктурированные сокеты".

Протокол ICMP

ICMP (Internet Control Message Protocol — протокол управляющих сообщений в сети Internet) является надстройкой над протоколом IP. Он используется всеми компьютерами, подключенными к Internet (клиентами, серверами, маршрутизаторами), для передачи управляющих сообщений, а также сообщений об ошибках. Он используется также рядом пользовательских программ, таких как traceroute и ping. Атрибуты данного пакета перечислены в табл. 3.4.

Таблица 3.4. Характеристики пакета ICMP

Служебные данные (байты)	24-64
Размер сообщения (байты)	65535 (65511 — максимальный объем полезных данных)
Надежность	Низкая (то же, что и в неструктурированных пакетах)
Тип сообщения	Дейтаграмма
Пропускная способность	Высокая (то же, что и в неструктурированных пакетах)
Целостность данных	Низкая (то же, что и в неструктурированных пакетах)
Фрагментация	Поддерживается (но мало вероятна)

Если в программе используется протокол ICMP, она может повторно использовать тот же самый сокет для подключения к другому компьютеру. Передача данных осуществляется с помощью функций sendmsg() и sendto() (описаны в следующей главе). Эти функции требуют указания адреса получателя.

Преимущества и недостатки протокола ICMP в основном такие же, как и у других дейтаграммных протоколов. Дополнительно пакет включает контрольную сумму, позволяющую выполнять проверку данных. Кроме того, вероятность фрагментации пакетов ICMP очень мала. Это объясняется природой самого протокола: он предназначен для передачи управляющей информации и сообщений об ошибках. Размер сообщений невелик, поэтому они почти никогда не требуют фрагментации.

Все сообщения об ошибках распространяются по сети в виде ICMP-пакетов. У пакета имеется заголовок, в котором записан код ошибки, а в теле пакета может содержаться сообщение, подробнее описывающее возникшую ошибку.

Будучи частью протокола IP, протокол ICMP использует IP-заголовок, добавляя к нему свой собственный (листинг 3.2).

Листинг 3.2. Определение структуры ICMP-пакета

```
/***                                                          */
/***          Определение структуры ICMP-пакета.              ***/
/***          Формальное определение находится в              ***/
/***          файле netinet/ip_icmp.h.                        ***/

typedef unsigned char ui8;
typedef unsigned short int ui16;

struct ICMP_header {
   ui8 type;      /* тип ошибки */
   ui8 code;      /* код ошибки */
   ui16 checksum; /* контрольная сумма сообщения */
   uchar msg[];   /* дополнительное описание ошибки */
```

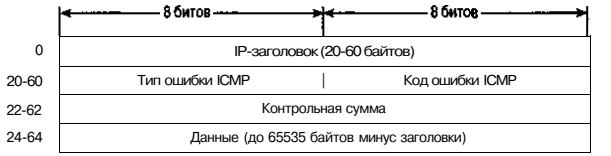

Рис. 3.3. Структура ICMP-пакета

Список типов и кодов ошибок приведен в приложении А, "Информационные таблицы". Поле msg может содержать любую информацию, объясняющую возникновение ошибки.

Протокол UDP

UDP (User Datagram Protocol — протокол передачи дейтаграмм пользователя) применяется для передачи данных без установления соединения (независимые сообщения). Он позволяет повторно использовать один и тот же сокет для отправки данных по другому адресу. Атрибуты пакета UDP перечислены в табл. 3.5.

Таблица 3.5. Характеристики пакета UDP

Служебные данные (байты)	28-68
Размер сообщения (байты)	65535 (65511 — максимальный объем полезных данных)
Надежность	Низкая
Тип сообщения	Дейтаграмма
Пропускная способность	Средняя
Целостность данных	Средняя
Фрагментация	Поддерживается

Чем выше уровень протокола в стеке TCP/IP, тем больше он ориентирован на данные и меньше — на сеть. Протокол UDP скрывает некоторые детали обработ-

ки ошибок и передачи сообщений ядром системы. Кроме того, он обрабатывает фрагментированные сообщения.

Сообщение, отправляемое по протоколу UDP, напоминает электронное письмо. Единственная информация, которая в нем требуется, — это адрес получателя, адрес отправителя и сами данные. Ядро берет сообщение и передает его в сеть, но не проверяет, дошло ли оно до адресата. Как и в случае протокола ICMP, разрешается посылать сообщения через один и тот же сокет различным адресатам.

Благодаря отсутствию проверок обеспечивается почти максимальная производительность, но при этом снижается надежность. Пакеты и их фрагменты могут теряться в сети, а сообщения могут приходить поврежденными. Программы, использующие протокол UDP, либо самостоятельно отслеживают приход сообщений, либо вообще не заботятся о возможной потере данных или их повреждении. (Помните: хотя дейтаграммы считаются ненадежными, это не означает обязательную потерю данных. Просто протокол не гарантирует их доставку.)

Протокол UDP лучше всего подходит для передачи данных трех последних категорий из рассмотренной выше классификации. Приложения, работающие с такими данными, наименее чувствительны к их потерям. Система, передающая в Internet фотографии со спутников, может не успевать обновлять картинку у каждого из клиентов, но вряд ли это кто-либо заметит. Другой хороший пример — сервис текущего времени. Поскольку такого рода данные актуальны в течение очень короткого промежутка времени, сервер вполне может пропустить пару тактов, не нарушив целостность данных.

Преимуществом UDP является высокая скорость. Кроме того, надежность протокола можно повысить самостоятельно следующими способами.

- Разбивайте большие пакеты на части, присваивая каждой из них номер. Компьютер на другой стороне соединения должен будет осуществлять сборку сообщения. Но не забывайте о том, что повышение количества служебных данных приводит к снижению производительности.
- Отслеживайте каждый пакет. Назначьте всем пакетам уникальные номера. Заставьте принимающий компьютер подтверждать доставку пакета, иначе сообщение будет выслано повторно. Если компьютер не получил ожидаемый пакет, он запрашивает повторную доставку, указывая номер последнего сообщения.
- Добавьте поле контрольной суммы или CRC-кода. Проверяйте корректность данных в каждом пакете путем их суммирования. CRC-код более надежен, чем контрольная сумма, но последнюю легче вычислять. Если получатель сталкивается с поврежденными данными, он просит программу повторить сообщение.
- Используйте механизм тайм-аутов. Можно предположить, что истечение заданного времени, в течение которого должна была произойти доставка, означает неудачу. В этом случае отправитель может повторно послать сообщение, а получатель может послать напоминание отправителю.

Перечисленные приемы позволяют имитировать работу протокола TCP, который требуется для доставки важных и критических данных.

Протокол UDP основан на функциях протокола IP. Каждая дейтаграмма UDP добавляет свой заголовок к IP-пакету (рис. 3.4). Структура заголовка приведена в листинге 3.3.

Листинг 3.3. Определение структуры UDP-пакета

```
/*** Определение структуры UDP-пакета.                    ***/
/*** Формальное определение находится в файле netinet/udp.h. ***/

typedef unsigned char ui8;
typedef unsigned short int ui16;

struct UDP_header {
   ui16 src_port;  /* номер порта отправителя */
   ui16 dst_port;  /* номер порта получателя */
   ui16 length;    /* длина сообщения */
   ui16 checksum;  /* контрольная сумма сообщения */
   uchar data[];   /* данные */
```

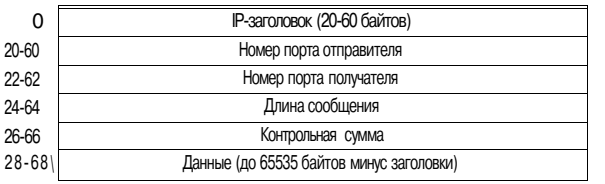

Рис. 3.4. Структура UDP-пакета

В UDP для **каждого сообщения создается виртуальный сетевой приемник, называемый портом. Благодаря портам IP-подсистема может быстро находить владельцев сообщений. Даже** если **порт не был назначен с помощью функции** bind(),система **создаст временный порт из списка эфемерных портов (см. главу 2, "Основы TCP/IP").**

Протокол TCP

TCP (Transmission Control Protocol — протокол управления передачей) является основным протоколом сокетов, используемым в Internet. Он позволяет использовать функции read() **и** write() **и требует повторного открытия сокета при каждом новом подключении. Атрибуты пакета TCP перечислены в табл. 3.6.**

Таблица 3.6. Характеристики пакета TCP

Служебные данные (байты)	40-120
Размер сообщения (байты)	(не ограничен)
Надежность	Высокая (факт получения данных проверяется)
Тип сообщения	Поток
Пропускная способность	Низкая (в сравнении с другими протоколами)
Целостность данных	Высокая (используются контрольные суммы)
Фрагментация	Мало вероятна

Дальнейшее повышение надежности обеспечивается путем проверки того, что адресат действительно получил данные, причем именно в том виде, в каком они были посланы. Протокол UDP работает достаточно быстро, но он не имеет той надежности, которая требуется многим программам. Проблема надежности решается в протоколе TCP.

В самой сети, однако, присутствует ряд факторов, снижающих ее надежность. Это не связано с какими-то внешними ограничениями, а заложено в сетевой архитектуре. Чтобы обеспечить надежную потоковую доставку сообщений в WWW, протокол TCP вынужден реализовывать многие из тех идей, которые были предложены выше для повышения надежности протокола UDP. Ниже рассматриваются три ограничивающих фактора, существующих в Internet: динамические подключения, потери данных и узкие каналы.

Динамические подключения

Один компьютер посылает сообщение другому. Сообщение путешествует по сети, проходя через различные маршрутизаторы и шлюзы. Путь, по которому оно передается, может быть самым разным. В зависимости от того, включен сервер или нет, сетевые сегменты (соединения между компьютерами) могут то появляться, то исчезать. Сила Internet заключается в адаптации к подобным изменениям и способности динамически изменять маршрут сообщения. К сожалению, данное преимущество также означает, что путь между клиентом и сервером может постоянно меняться, то удлиняясь, то укорачиваясь. При удлинении увеличивается время распространения сообщения. Таким образом, программа может послать несколько последовательных сообщений, и все они пойдут разными маршрутами, будут передаваться в разное время и придут в неправильном порядке.

Протокол TCP гарантирует, что прежде чем будет послано следующее сообщение, получатель правильно примет предыдущее. Предположим, программа посылает 10 сообщений подряд. TCP-подсистема перехватывает каждое сообщение, присваивает ему уникальный номер и отправляет его, Получатель принимает сообщение и высылает подтверждение. Только получив его, протокол TCP позволяет программе послать следующее сообщение.

Алгоритм раздвижного окна

Описанная схема "отправил — подожди" работает слишком медленно. В TCP применяется улучшённая методика 'раздвижного окна"; в ней определяется, как «исто и при каких обстоятельствах следует посылать подтверждение. В медленных и'шумящих соединениях число подтверждающих сообщений выше. В более быстрых и надежных соединениях между подтверждениями передается больше данных. Это результат применения алгоритма Нейгла (Nagle). Работу алгоритма можно отменить с помощью параметров сокета (подробнее об этом - в главе 9, "Повышение производительности").

Потери данных

Когда адресат получает сообщение, он проверяет целостность содержащихся в нем данных. Сообщения могут передаваться по плохим линиям, теряющим или повреждающим данные. Протокол TCP сопровождает каждое сообщение контрольной суммой и является последним уровнем, на котором поврежденные данные могут быть обнаружены и исправлены.

Если получатель обнаруживает повреждение, он посылает сообщение об ошибке, запрашивая повторную передачу данных. Кроме того, если компьютер в течение заданного времени не получит подтверждение о том, что сообщение ус-

пешно доставлено, TCP-подсистема автоматически инициирует повторную доставку сообщения без вмешательства программы.

Узкие каналы

Иногда случается, что сообщение слишком велико и не может пройти через встретившийся ему сегмент. Проблема связана с различными технологиями передачи данных. В некоторых сетях допускаются длинные пакеты, в других их размер ограничен.

Протокол UDP пытается передавать настолько большие данные, насколько это возможно, вследствие чего могут возникать проблемы, связанные с ограниченными каналами передачи. В протоколе IP предполагается, что маршрутизаторы могут фрагментировать данные и сообщение потом придется собирать по частям.

В свою очередь, протокол TCP ограничивает каждый пакет небольшим блоком. Длинные сообщения разбиваются прежде, чем это произойдет где-то в сети. Искомый размер выбирается таким, чтобы пакет мог передаваться в большинстве сетей без изменений. По умолчанию размер блока в TCP равен 536 байтам и может быть увеличен до 1500 байтов. Вручную задать это значение можно с помощью параметра MSS (maximum segment size — максимальный размер сегмента) TCP-сокета (обратитесь за детальной информацией к главе 9, "Повышение производительности").

Следует также отметить, что при получении пакетов сообщения в неправильном порядке TCP-подсистема переупорядочивает их, прежде чем передавать программе.

Необходимость решения всех перечисленных проблем приводит к значительному усложнению протокола TCP и структуры его заголовка. Естественно, производительность от этого существенно падает.

Заголовок TCP-пакета

Протокол TCP предлагает много различных возможностей, поэтому в заголовок TCP-пакета добавлен целый ряд дополнительных полей. Размер заголовка в TCP примерно в три раза превышает размер UDP-заголовка. Структура заголовка приведена в листинге 3.4.

Листинг 3.4. Определение структуры TCP-пакета

```
/*** Определение структуры TCP-пакета.                          ***/
/*** Формальное определение находится в файле netinet/tcp.h.  ***/
/*****************************************************************/
typedef unsigned char ui8;
typedef unsigned short int ui16;
typedef unsigned int ui32;
typedef unsigned int uint;

struct TCP_header {
   ui16 src_port;    /* номер порта отправителя */
   ui16 dst_port;    /* номер порта получателя */
   ui32 seq_num;     /* порядковый номер */
   ui32 ack_num;     /* номер подтверждения */
```

```
uint data_off:4; /* смещение данных */
uint   res:6;    /* (зарезервировано) */    .
uint  urg_flag:1; /* срочное, внеполосное сообщение */
uint  ack_flag:1; /* поле подтверждения корректно */
uint  psh_flag:1; /* немедленно выдать сообщение процессу */
uint  rst_flag:1; /* разорвать соединение вследствие ошибок */
uint syn_flag:1; /* открыть виртуальное соединение (канал) */
uint  fin_flag:1; /* закрыть соединение */
ui16 window;      /* число байтов, получаемых адресатом */
ui16  checksum;   /* контрольная сумма сообщения */
ui16  urg_pos;    /* последний байт срочного сообщения */
ui8 options[];    /* опции TCP */
ui8   padding[]; /*  (необходимо для выравнивания
                     массива data[]) */
uchar  data[];    /* данные */
```

Рис. 3.5. Структура TCP-пакета

Заголовок может иметь переменный размер, поэтому поле data_off указывает на начало данных. Для экономии места это поле, подобно полю header_len IP-пакета, определяет не число байтов, предшествующих данным, а число 32-битовых слов.

Некоторые поля используются только для открытия соединения, управления передачей и закрытия соединения. Часть полей в процессе передачи данных не заполняется.

В TCP-заголовках используются те же номера портов, что и в UDP. Поля seq_numи ack_num позволяют выполнять трассировку потока. Когда посылается сообщение, IP-подсистема присваивает ему порядковый номер (seq_num). Получатель уведомляет отправителя о доставке сообщения, посылая ему подтверждение

с номером (ack_num), на единицу большим, чем порядковый номер принятого сообщения. Это позволяет в подтверждениях также передавать данные.

Взаимодействие посредством TCP

При открытии потокового соединения программа и сервер обмениваются сообщениями по схеме, описанной в табл. 3.7.

Таблица 3.7. Трехфазовое квитирование

Клиент посылает	Сервер посылает	Описание
SYN=1 (syn_flag)		Запрос на виртуальное подключение (канал)
ACK=0 (ack_flag)		Задание порядкового номера
	SYN=1 (syn_flag)	Разрешает виртуальное подключение
	ACK=1 (ack_flag)	
SYN=0 (syn_flag)		
ACK=1 (ack_flag)		Устанавливает виртуальное соединение

Это называется *трехфазовым квитированием*. В процессе обмена сообщениями клиент и сервер устанавливают размеры приемных буферов (окон).

Разрыв соединения — тоже не такой простой процесс, как может показаться, ведь в пути могут находиться важные данные. Когда клиент закрывает соединение, обмен сообщениями происходит по схеме, представленной в табл. 3.8.

Таблица 3.8. Разрыв TCP-соединения

Клиент	Сервер	Описание
FIN=1 (syn_flag)	Передает данные	Клиент запрашивает разрыв соединения
	Принимает данные	
ACK=1	Передает дополнительные данные	Происходит очистка каналов сервера
	Принимает дополнительные данные	
ACK=1	FIN=1	Разрыв принят. Сервер разрывает соединение и ждет подтверждения от клиента
ACK=1		Клиент закрывает свой конец соединения

Когда TCP-соединение разорвано, сокет нельзя повторно использовать для организации других соединений. Если требуется подключиться к другому серверу, необходимо создать новый сокет. Остальные протоколы не имеют подобного ограничения.

Взаимосвязь между протоколами

Изучая сетевое программирование, читатели, должно быть, не раз удивлялись: как взаимодействуют между собой все эти протоколы? Иногда кажется, что меж-

ду ними вообще нет никакой связи. Они могут использовать ряд функций друг друга, но все же столь тесного взаимодействия, чтобы считать их неразделимыми, не наблюдается.

Рассмотренные протоколы — ICMP, UDP, TCP и собственно IP — играют в сети каждый свою роль. Об этом следует помнить, проектируя сетевое приложение. Конечно, TCP надежнее и обладает большими возможностями по сравнению с другими протоколами, но его нельзя поменять на ICMP. В различных подсистемах Linux используются разные функции TCP/IP, и каждый протокол важен для корректной работы системы.

Физически пакеты ICMP, UDP и TCP основаны на неструктурированном IP-пакете. Свои заголовки и данные они размещают в разделе данных этого пакета.

Анализ сетевого трафика с помощью утилиты tcpdump

Наблюдая в реальном времени пакеты, проходящие по сети, можно узнать, каким образом ядро обрабатывает сообщения и как сетевая подсистема интерпретирует адреса. Утилита tcpdump отображает данные, находящиеся в текущий момент в сети. Она "видит" как неструктурированные IP-пакеты, так и TCP-пакеты. По соображениям безопасности доступ к ней разрешен только пользователю root.

По умолчанию утилита работает в беспорядочном режиме, перехватывая все сетевые пакеты и взаимодействуя с сетевым интерфейсным устройством напрямую.

Сетевая этика

Знание является грозным оружием и налагает на его обладателя большую ответственность. Имея привилегии пользователя root, можно выполнять много полезной работы, а можно и нанести огромный вред. Будучи инсталлированной на вашем компьютере, система Linux предполагает, что вы используете ее с добрыми намерениями, Наилучший способ дискредитировать сообщество разработчиков открытого программного обеспечения— злоупотребить доверием и властью, которые из самых лучших побуждений были предоставлены вам хорошими людьми.

Обычно сетевой адаптер принимает только те сообщения, которые несут в себе его Ethernet-адрес. Вспомните главу 2, "Основы TCP/IP", в которой говорилось о том, что каждый адаптер Ethernet имеет уникальный 6-байтовый идентификатор. Этот идентификатор служит критерием фильтрации сообщений.

Программируемые идентификаторы Ethernet

Некоторые OEM-производители (Original Equipment Manufacturer — изготовитель комплектного оборудования) предлагают сетевые платы (с интерфейсом PCI или PCMCIA), поддерживающие программируемые MAC-адреса (или идентификаторы Ethernet). Это позволяет нескольким производителям осуществлять массовый выпуск плат, продаваемых сотнями фирм. К сожалению, существует риск получить плату с фиктивным идентификатором, поскольку фирма-продавец не смогла правильно его запрограммировать. При этом идентификатор может оказаться неуникальным в сети.

Беспорядочный режим работы утилиты разрешается отключать. Утилита tcpdump располагает множеством опций, с помощью которых можно фильтровать нежелательные сообщения, выполнять переадресацию и многое другое. Перечислим наиболее интересные опции.

Опция	Директива
-a	Пытаться связывать имена с сетевыми и широковещательными адресами (требуется доступ к серверу имен)
-c <*счетчик*>	Остановиться после получения заданного числа сообщений
-n	Не преобразовывать адреса узлов в символические адреса (полезна, когда нет доступа к серверу имен)
-p	Не переводить плату в беспорядочный режим; при работе в небольшой сети просмотр всех пакетов может показаться интересным, но в крупной сети компьютер быстро "захлебнется" потоком сообщений
-v	Выводить более детальный отчет (отображается значение поля TTL пакета)
-vv	Выводить исчерпывающий отчет
-w <*файл*>	Записывать пакеты в файл

Утилиту tcpdump можно запустить на выполнение безо всяких опций и получить почти всю необходимую информацию. Можно также наблюдать интересные процессы, например, как протокол ARP запрашивает идентификатор Ethernet по заданному IP-адресу. Вот пример перехвата 100 сообщений с выводом детального отчета без указания метки времени:

```
tcpdump -v -t -c 100
```

Опция -t подавляет вывод метки времени. Поскольку сообщения быстро прокручиваются на экране, удобнее направлять результаты работы утилиты в файл.

В работе утилиты tcpdump есть несколько "странностей". Например, она не перехватывает сообщения, адресованные ей самой. Она не видит пакеты, посылаемые командой ping 127.0.0.1, так как сетевая подсистема не передает эти сообщения протоколам нижних уровней.

Создание сетевого анализатора

Чтобы написать свой собственный анализатор сети (который работает подобно утилите tcpdump, но не поддерживает беспорядочный режим), достаточно узнать, как перехватывать сообщения, направляемые компьютеру. В первую очередь необходимо получить привилегии пользователя root. Затем следует вызвать такие функции:

```
sd = socket(PF_INET, SOCK_PACKET, фильтр)-,
bytes_read = recvfrom(sd, buffer, sizeof(buffer), 0, 0, 0);
```

Обратите внимание на новый тип сокета: SOCK_PACKET. Эта константа задает сокет аппаратного уровня, доступный только для чтения. К данному сокету можно

применять несколько фильтров. Фильтр сообщает IP-подсистеме о том, какого рода пакеты следует перехватывать. Перечислим некоторые из фильтров:

ETH_P_802_3	Кадры стандарта 802.3
ETH_P_AX25	Кадры стандарта AX.25
ETH_P_ALL	Все кадры (будьте осторожны!)
ETH_P_802_2	Кадры стандарта 802.2

Нужный нам фильтр задается константой ETH_P_ALL. Вызов функции socket() будет выглядеть следующим образом:

```
sd = socket(PF_INET, SOCK_PACKET, ETH_P_ALL);
```

Теперь при каждом вызове функции recvfrom() будет возвращаться сетевой кадр (сообщение физического уровня). В него входят аппаратный адрес (например, идентификатор Ethernet) и заголовок.

Сокет типа SOCK_PACKET позволяет получить доступ к физическим кадрам, генерируемым в процессе передачи сообщений. *С* помощью этого сокета можно узнать, как в сети создаются кадры. При анализе кадров необходимо помнить, что порядок следования битов может быть разным и зависит от аппаратуры. В структуре IP-адреса предполагается, что бит со смещением нуль является первым битом потока данных.

Имеющаяся на Web-узле программа snooper.c переупорядочивает поля заголовка таким образом, чтобы он соответствовал физическому кадру для процессора с прямым порядком следования байтов (Intel-совместимый) и GNU-компилятора.

Резюме: выбор правильного пакета

С помощью утилиты tcpdump можно узнать структуру пакетов различных типов. В случае протокола IP пакеты бывают неструктурированными, служебными (ICMP), дейтаграммными (UDP) и потоковыми (TCP). Все они играют свою роль в сети.

У каждого пакета есть заголовок. Поскольку пакеты ICMP, UDP и TCP включаются в IP-пакет, общий размер заголовка может составлять от 20 до 120 байтов. Соотношение между размером заголовка и объемом передаваемых данных определяет пропускную способность сети.

У протокола TCP наименьшая пропускная способность, так как у него самый большой размер заголовка. Этот протокол чаще всего применяется в Internet, потому что он обеспечивает надежное соединение между компьютерами. В TCP можно работать с высокоуровневыми функциями ввода-вывода, такими как fprintf() и fgets().

С помощью протокола UDP можно передавать одиночные сообщения между компьютерами, не устанавливая повторное соединение. В нем вводится понятие виртуального порта, посредством которого приложение имеет как бы монопольный доступ к сети. Протокол UDP обеспечивает хорошую пропускную способность, но невысокую надежность передачи данных.

Передача сообщений между одноранговыми компьютерами

Глава

4

В этой главе...

Сообщения могут передаваться двумя различными способами: непрерывным потоком (TCP) или в виде дискретных пакетов (UDP). Первый способ напоминает телефонное соединение, а второй — отправку письма в конверте.

В TCP необходимо, чтобы с адресатом было установлено соединение. Это позволяет гарантировать отсутствие потерь данных и доставку пакетов в нужном порядке. В UDP соединение можно не устанавливать. В этом случае программа должна помещать в каждое сообщение адрес получателя.

В настоящей главе рассматриваются оба интерфейса передачи сообщений.

Сокеты, требующие установления соединения

В Linux (и во всех других операционных системах семейства UNIX) программы могут взаимодействовать на трех различных уровнях, каждый из которых основан на системном вызове socket(). Три базовых протокола (неструктурированный IP, UDP и TCP, о которых рассказывалось в главе 3, "Различные типы Internet-пакетов") позволяют абстрагироваться от низкоуровневых сетевых процессов и повысить надежность соединения, хотя и ценой разумного снижения производительности. Протокол самого верхнего уровня, TCP, обеспечивает наивысшую надежность. Он гарантирует, что данные будут доставлены без потерь и в нужной последовательности. TCP-соединение можно рассматривать как файл или системный канал между процессами. TCP-сокеты ориентированы на установление соединения.

Открытые каналы между программами

TCP-сокет образует открытый двунаправленный канал между двумя программами/Подобно комплекту с наушниками и микрофоном, канал позволяет передавать и принимать данные синхронно. Программы могут посылать друг другу данные одновременно, не занимаясь расшифровкой диалога из отдельных сообщений.

TCP-подсистема запоминает, с кем общается программа. Каждое сообщение на более низком уровне (уровне протокола IP) несет в себе адрес получателя. Это подобно набору номера всякий раз, когда вы хотите поговорить по телефону со своим другом.

При подключении к другой программе (с помощью системного вызова connect()) сокет запоминает адрес получателя сообщений и его порт. В процессе дальнейшего обмена данными можно использовать высокоуровневые функции библиотеки потокового ввода-вывода, такие как fprintf() и fgets(). Это существенно упрощает программирование.

Протокол TCP позволяет избежать проблемы потери данных, существующей в других протоколах, и сконцентрироваться на решении поставленной задачи.

Надежные соединения

В TCP предполагается, что путь, по которому передаются данные, чист и свободен от каких-либо помех. Протокол гарантирует надежность соединения: або-

нент получает все, что посылает ему ваша программа. Сетевая подсистема, объединяющая сетевые устройства и реализующая стек протоколов (на сервере или клиенте), принимает сообщение, проверяет его и передает его нужной программе.

Если сообщение не проходит проверку вследствие обнаружения ошибок, сетевая подсистема запрашивает повторную передачу. Процедура проверки выполняется на обеих сторонах соединения. Как описывалось в предыдущей главе, некоторые сообщения несут в себе уникальный порядковый номер. Он важен для обеспечения последовательного приема фрагментированных пакетов. Порядковые номера сообщений являются отличительной чертой протокола TCP.

С сетевым взаимодействием связаны две основные проблемы: потеря пакетов и их поступление в неправильном порядке. Предположим, программа посылает сообщение. Она не может быть уверенной в том, что абонент его принял, до тех пор пока не получит от него подтверждение. Не обнаруживая поступления данных в течение определенного времени, получатель информирует отправителя о том, пакет с каким номером был принят последним. Таким образом, формируется тесная связь между программами.

С другой стороны, получатель может обнаружить, что некоторые части сообщения не согласуются по номерам с другими частями. Тогда он задерживает фрагмент сообщения и ждет поступления соседних фрагментов. Порядковый номер пакета служит ключом при сборке готового сообщения.

В TCP канал передачи данных организован так, словно информация в нем передается не в виде пакета, а непрерывным потоком, что позволяет применять высокоуровневые функции ввода-вывода. В то же время низкоуровневые протоколы оперируют пакетами, поэтому функции наподобие `printf()` в них недоступны.

Чем менее надежен протокол, тем быстрее осуществляется передача данных. В UDP, например, сообщения отправляются в сеть без учета порядка. Получатель принимает каждое сообщение независимо от других.

Порядок, в котором принимаются данные, важен с точки зрения проектирования приложения. Разработчик может считать, что очередность поступления пакетов не имеет значения, хотя на самом деле это не так. Ниже приводится список вопросов, ответы на которые позволят определить, необходима ли потоковая доставка сообщений (в скобках дается итоговое резюме).

- **Образуют ли данные независимый запрос?** Программа может принимать ответные данные в любом порядке, если сообщение представляет собой независимый запрос. Для несвязанных запросов порядок выполнения неважен: программа может запросить файл, а затем проверить статус абонента соединения. Если же, например, передаются координаты точек для сетевой игры, последовательность, в которой они поступят, чрезвычайно важна. (Да — UDP; нет — TCP.)
- **Если произвольным образом переупорядочить сообщения, изменится ли реакция программы?** Это настоящая лакмусовая бумажка для каналов передачи сообщений. В потоковых соединениях переупорядочение данных не допускается. Протокол TCP следит за тем, чтобы из неупорядоченного потока образовывались правильные, непрерывные сообщения. (Нет – UDP; да – TCP.)
- **Можно ли представить диалог между программами в виде трубопровода, или больше подойдет сравнение со службой доставки Federal Express?** С одной стороны, поток данных между программами можно сравнить с водой, протекающей по трубопроводу. Такая информация упорядочена.

С другой стороны, одиночный пакет данных напоминает бандероль. Такого рода информация не упорядочена. К примеру, бандероли можно принимать в любом порядке и отвечать лишь на некоторые из них. В схеме "трубопровода" это недопустимо, так как приведет к потере данных. (FedEx — UDP; трубопровод — TCP.)

- **Если между передачей двух сообщений отключиться, а затем снова подключиться, должен ли будет сервер проверить ваше местонахождение или состояние транзакции?** Некоторые транзакции организуются по такой схеме: запрос — "Дай мне это", ответ — "Ищи здесь". Сервер не хранит никакие данные в перерывах между транзакциями. В других транзакциях необходимо передавать информацию о состоянии. Например, в сеансе Telnet существуют два состояния: зарегистрирован и не зарегистрирован. (Нет - UDP; да — TCP.)
- **Должна ли программа отслеживать, кто и что передал?** Иногда требуется вести диалоги (с использованием потоков данных или без), при которых сервер поддерживает список клиентов. Примером могут служить системы персональной настройки Web-узлов для их посетителей. (Да — UDP; нет — TCP.)
- **Может ли произойти потеря пакета без ущерба для вычислений на компьютере получателя?** Если потеря данных не играет особой роли с точки зрения безопасной доставки сообщений, можно воспользоваться протоколом пакетной доставки, таким как UDP. Потерять неупорядоченные данные не так страшно, как упорядоченные. Например, информацию о ценах на акции следует доставлять с особой тщательностью, тогда как сводки погоды могут быть неполными. (Да — UDP; нет — TCP.)

Если окажется, что хотя бы для одного вопроса ответ — TCP, необходимо использовать протокол TCP или, по крайней мере, усилить возможности протокола UDP. Это не является жестким требованием, так как играет роль и производительность соединения.

Соединения по протоколам более низкого уровня

Функцию connect() можно вызвать и для UDP-сокета. Это удобно, если не нужно использовать высокоуровневые функции ввода-вывода: протокол UDP обеспечивает существенное повышение производительности благодаря автономным сообщениям. Однако соединение по протоколу UDP функционирует немного по-другому, чем в случае TCP.

В главе 3, "Различные типы Internet-пакетов", описывался процесс трехфазового квитирования, происходящего при установлении соединения. Благодаря квитированию формируется канал потоковой передачи данных между программами. Поскольку в UDP не поддерживаются потоки, функция connect() позволяет лишь немного упростить передачу сообщений.

Будучи вызванной для UDP-соединения, функция connect() просто уведомляет адресата о том, что ему будут посылаться какие-то сообщения. Можно выполнять функции read() и write(), как в TCP, но надежная и последовательная дос-

тавка не гарантируется. Представленный в листинге 4.1 фрагмент программы напоминает те программы, которые мы создавали при TCP-соединении.

Листинг 4.1. Простейшее подключение по протоколу UDP

```
/*************************************************************/
/***                  Пример UDP-соединения                ***/
/***              (из файла connected-peer.c)              ***/

int sd;
struct sockaddr_in addr;
sd = socket(PF_INET, SOCK_DGRAM, 0);    /* дейтаграммный сокет */
bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(DEST_PORT);
inet_aton(DEST_IPADDR,&addr.sin_addr);
if ( connected, &addr, sizeof(addr)) != 0 ) /* подключаемся! */
     perror("connect");
/*— это не потоковое соединение! —*/
send(sd, buffer, msg_len);        /* передача данных как в TCP */
```

Обычно при работе по протоколу UDP используются функции sendto() и recvfrom() (описаны ниже). Функция send() предполагает, что программа зарегистрировала получателя с помощью функции connect!).

Сервер, ожидающий подключения, может использовать различные протоколы, но для программы важно, чтобы он опубликовал номер своего порта с помощью функции bind(). В листинге 4.1 ссылка на номер порта осуществляется с помощью общепринятой константы DEST_PORT. Значение этой константы устанавливает сервер при вызове функции bind(). Когда приходит сообщение, сервер может сам вызвать функцию connect().

В отличие от функции connect() в TCP, программа может многократно подключаться к различным серверам, не закрывая сокет. Это одно из основных преимуществ UDP. В TCP требуется сначала закрыть сокет, а затем открыть его повторно.

Еще раз повторю, что, выбирая протокол UDP, вы сознательно идете на снижение надежности соединения. Возможны потери пакетов или их доставка в неправильном порядке. Функция connect() в UDP лишь регистрирует адресата и не повышает надежность.

Протокол RDP

Протокол RDP (Reliable Data Protocol — протокол надежной доставки данных) [RFC908, RFC1151] не только обеспечивает гарантированную доставку, как в TCP, но и позволяет достичь скорости UDP. Сообщения могут приходить в неправильном порядке, но, по крайней мере, это хороший компромисс между UDP и TCP. К сожалению, хоть этот протокол уже давно описан, Linux и другие UNIX-системы еще не поддерживают его.

Пример: подключение к демону HTTP

Одним из наиболее часто используемых протоколов высокого уровня является HTTP. Он реализует очень простой интерфейс выдачи запроса. Сервер анализирует запрос и отправляет клиенту сообщение. В сообщении может содержаться любой документ.

Несмотря на простоту протокола, на его примере можно проследить различные виды сетевого взаимодействия. Большинство клиент-серверных соединений функционируют в режиме одиночных транзакций. При этом многие элементы TCP, такие как механизм квитирования и проверки, обеспечивающие надежность, становятся избыточными, но таков общепринятый стандарт.

Упрощенный протокол HTTP

Чтобы послать запрос HTTP-серверу, достаточно одной-единственной команды:

GET <запрос> HTTP/1.0<cr><cr>

Таково несколько упрощенное представление протокола HTTP. В главе 6, "Пример сервера", протокол будет проанализирован подробнее.

В запросе обычно содержится путь к каталогу, но могут также указываться различные параметры и переменные. Когда в броузере вводится URL-адрес вида http://www.kernel.org/mirrors/, броузер открывает сокет по адресу www.kernel.org и посылает следующее сообщение:

GET /mirrors/ HTTP/1.0<cr><cr>

Также может быть послана информация о данных, которые требуется получить. Последние два управляющих символа <cr> представляют собой символы новой строки, обозначающие конец сообщения. Без них невозможно было бы обнаружить конец потока, что могло бы привести к бесконечному ожиданию.

Получение HTTP-страницы

Составление запроса — это самая простая часть соединения. Нужно лишь обеспечить, чтобы сервер понял сообщение. В листинге 4.2 продемонстрирован один из способов получения Web-страницы. Эта программа открывает соединение по адресу, указанному в командной строке, и посылает HTTP-запрос. Полученные данные записываются в стандартный выходной поток (stdout).

Листинг 4.2. Получение Web-страницы от HTTP-сервера

```
/***                Взаимодействие с HTTP-сервером                ***/
/*** ***          (из файла http-client.c)                    *** ***/

int sd;
struct servent *serv;
if ( (serv = getservbyname("http", "tcp")) == NULL )
    PANIC("HTTP servent");
if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0 )
```

```
    PANIC("Socket");

/*— Инициализация адресной структуры —*/
bzero(Sdest, sizeof(dest));
addr.sin_family = AF_INET;
addr.sin_port = serv->s_port;                    /* HTTP-сервер */
if ( inet_addr(Strings[1], &dest.sin_addr.s_addr) == 0 )
    PANIC(Strings[1]);

/*— Подключение к серверу—*/
if ( connect(sd, Sdest, sizeof(dest)) != 0 )
    PANIC("Connect");

/*— Составляем запрос и отправляем его —*/
sprintf(buffer, "GET %s HTTP/1.0\n\n", Strings[2]);
send(sd, buffer, strlen(buffer), 0);

/*— Пока есть данные, читаем и отображаем их —*/
do
{
   bytes_read = recv(sd, buffer, sizeof(buffer)-1. 0);
   buffer[bytes_read] = 0;
   if ( bytes_read > 0 )
       printf("%s", buffer);
}
while ( bytes_read > 0 );
```

В программе, представленной в листинге 4.2, устанавливается соединение и посылается запрос, заданный в командной строке. Возвращаемые сервером данные читаются до тех пор, пока сервер не разорвет соединение. В последних версиях протокола HTTP (1.1 и предлагаемая версия HTTP-NG) разрешается оставлять канал открытым. При этом необходимо уведомить получателя об окончании связи. Дополнительную информацию о новых версиях HTTP можно получить на сервере www.w3c.org.

Сокеты, не требующие установления соединения

Не во всех соединениях требуется открытый двунаправленный канал связи между компьютерами. Если телефонный разговор рассматривать как пример потокового соединения, то почтовую службу можно назвать системой, ориентированной на доставку отдельных сообщений (работающую без установления соединения). Подобно отправителю письма, протокол UDP формирует сообщение, записывает в него адрес получателя и отправляет в сеть, не заботясь о том, как оно достигнет адресата. (Повторюсь: ненадежность дейтаграмм означает только то, что факт доставки не проверяется. Это не означает, что сообщение не дойдет.)

Задание адреса сокета

В полную инсталляционную версию Linux обычно входят средства, позволяющие посылать короткие сообщения от одной рабочей станции к другой. При этом необходимо указать лишь адрес компьютера и, возможно, имя пользователя. Подобный механизм можно реализовать самостоятельно с помощью UDP-сокета.

Если для передачи данных соединение не устанавливается, то и функцию connect() вызывать не нужно. Однако в этом случае нельзя будет вызвать функции send() и recv(). В операционной системе существуют две аналогичные низкоуровневые функции, позволяющие задавать адрес получателя: sendto() и recvfrom().

```
#include <sys/socket.h>
#include  <resolv.h>
int sendto(int sd,  char* buffer,  int msg_len,  int options,
           struct sockaddr *addr,  int addr_len);
int recvfrom(int sd,  char* buffer,  int maxsize,  int options,
             struct sockaddr *addr,  int *addr_len);
```

Первые четыре параметра такие же, как и в функциях send() и recv(). Даже опции и возможные коды ошибок совпадают. В функции sendto() дополнительно указывается адрес получателя. При отправке сообщения необходимо заполнить поля структуры sockaddr (листинг 4.3).

Листинг 4.3. Пример функции sendto()

```
/*************************************************************/
/***                  Пример функции sendto()              ***/
/*************************************************************/
int sd;
struct sockaddr_in addr;
sd = socket(PF_INET,  SOCK_DGRAM,  0);
bzero(&addr,  sizeof(addr));
addr.sin_family  =  AF_INET;
addr.sin_port = htons(DEST_PORT);
inet_aton(DEST_ADDR, &addr.sin_addr);
sendto(sd,  "This is a test",  15,  0,  &addr,  sizeof(addr));
```

В данном примере сообщение отправляется прямо по адресу DEST_ADDR:DEST_PORT. В тело сообщения можно включать как двоичные данные, так и ASCII-текст. Это не имеет значения.

В функции recvfrom() первые пять параметров имеют такое же назначение, как и в функции sendto(). Будучи вызванной, она ожидает поступления сообщения от указанного отправителя. Отличие проявляется в шестом параметре, который представляет собой указатель на целое число. Дело в том, что в функции sendto() последние два параметра определяют адрес получателя, а в функции recvfromf) — отправителя. Поскольку в семействе sockaddr существует множество структур разного размера, можно получить сообщение от источника с другим типом сокета (по умолчанию создается сокет, относящийся к семейству адресов AF_INET).

Передача длины адреса

В функцию recvfrom() передается указатель на длину адресной структуры. Это пережиток, доставшийся в наследство от другого семейства протоколов — PF_LOCAL, При вызове функции ей необходимо сообщить (в шестом параметре), каков максимальный размер буфера адреса. По завершении функции в этом же параметре будет содержаться реальная длина полученного адреса. Таким образом, параметр должен передаваться по ссылке, чтобы система могла записывать в него возвращаемое значение.

В связи с тем, что функция recvfrom() может изменять значение параметра addr_len, его необходимо задавать при каждом следующем вызове функции. В противном случае оно будет постоянно уменьшаться. Рассмотрим пример:

```
/*************************************************************/
/***                Пример функции recvfrom()                ***/

int sd;
struct sockaddr_in addr;
sd = socket(PF_INET, SOCK_DGRAM, 0);
/*- привязка к конкретному порту -*/
while ()
{   int bytes, addr_len=sizeof(addr);
    bytes = recvfrom(sd, buffer, sizeof(buffer), 0, fcaddr,
                     &addr_len);
    fprintf(log, "Got message from %s:%d (%d bytes)\n",
            inet_ntoa(addr.sin_addr), ntohs(addr.sin_port),
              bytes);
    /**** обработка запроса ****/
    sendto(sd, reply, len, 0, &addr, addr_len);
}
```

В этом примере в цикле while программа ожидает входящих сообщений, регистрирует каждое из них в файле и возвращает ответные данные. Значение переменной addr_len каждый раз сбрасывается, поэтому не происходит его неявного уменьшения. Значение переменной addr используется в качестве обратного адреса.

Оба параметра, addr_len и addr, в функции recvfrom() должны быть действительными, т.е. они не могут содержать значение NULL (0). Поскольку в UDP соединение не устанавливается, необходимо знать, кто послал запрос. Лучше всего хранить значение параметра addr в течение всего времени, пока обрабатывается запрос.

Упрощение процедуры квитирования

В TCP требуется механизм трехфазового квитирования. Клиент посылает запрос на подключение, сервер принимает его и сам посылает аналогичный запрос. Клиент принимает запрос от сервера, после чего соединение считается установленным.

Протокол, в котором соединение не устанавливается, имеет упрощенную процедуру квитирования. В UDP данная процедура отсутствует вовсе. Единственная информация, которая доступна обеим сторонам, — это собственно сообщение. Если адресат не может быть найден или при передаче возникла ошибка, сетевая подсистема вернет сообщение об ошибке, которое, правда, может прийти с существенной задержкой (от одной до пяти минут после обнаружения ошибки).

Благодаря отсутствию квитирования в UDP существенно повышается быстродействие, так как посылается всего один или два пакета.

Протокол T/TCP

В TCP требуется до десяти инициализирующих пакетов. Это очень накладно, особенно если запрашивающая сторона собирается выполнить одну-единственную транзакцию. В процессе установления соединения обе стороны согласовывают используемые сервисы и порты, а также степень надежности канала. Дополнительный процесс квитирования требуется при разрыве соединения (см. главу 3, "Различные типы Internet-пакетов").

В TCPv3 [RFC 1644] появилась возможность достичь скорости, близкой к UDP, и при этом сохранить надежность, присущую TCP. В протоколе T/TCP (Transaction TCP) подключение, передача данных и разрыв соединения происходят в рамках одной функции sendto(). Как это возможно?

В TCP соединение устанавливается посредством механизма трехфазового квитирования, а при разрыве соединения тоже требуется взаимодействие двух сторон. Это необходимо, чтобы гарантировать получение как клиентом, так и сервером всех данных в правильной последовательности. В процессе квитирования в TCP-пакет включаются специальные флаги, указывающие на установление соединения (SYN), подтверждение приема данных (ACK) и закрытие канала связи (FIN).

Вспомните формат TCP-пакета, который описывался в главе 3, "Различные типы Internet-пакетов". Заголовок пакета включает поля, которые являются взаимоисключающими. Например, флаги SYN, ACK и FIN хранятся по отдельности. Зачем тратить драгоценное пространство пакета, если только один из этих битов может быть установлен в одно и то же время?

В протоколе T/TCP поля устанавливаются одновременно. Когда клиент подключается к серверу, он посылает ему сообщение, в котором содержится запрос на подключение (SYN). Кроме того, устанавливается флаг FIN, означающий, что требуется закрыть соединение сразу по завершении транзакции (рис. 4.1).

Сервер в ответ посылает свой собственный запрос на подключение, запрос на разрыв соединения и подтверждение того, что клиент разорвал соединение (подтверждение на установление соединения не требуется). В этот же пакет включаются данные, которые запросил клиент. Последним посылается пакет, в котором клиент подтверждает разрыв соединения со стороны сервера.

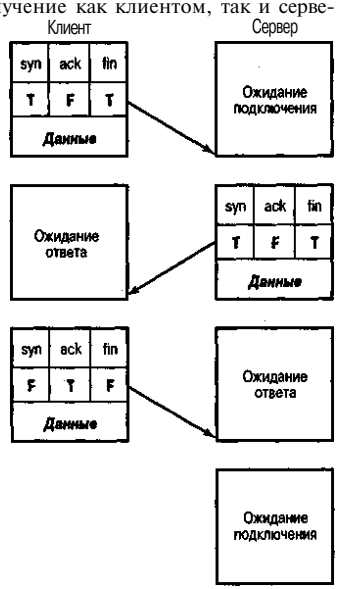

Рис. 4.1. В протоколе T/TCP для установления соединения, передачи данных и разрыва соединения требуются три пакета

Протокол T/TCP обеспечивает очень высокое быстродействие. Существует только одна проблема: вся информация должна быть передана в пределах одного сегмента, максимальный размер которого составляет всего 540 байтов. Правда, существует возможность поднять этот предел до 64 Кбайт. Кроме того, программа может посылать не одно сообщение, а несколько.

Для взаимодействия по протоколу T/TCP от серверной программы ничего не требуется. Алгоритм, реализующий TCP-соединение, прекрасно подходит для этой цели, так как все требуемые функции запрограммированы в сетевой подсистеме. Остальная работа реализуется на стороне клиента. Примерная схема может быть следующей:

```
/*************************************************************/
/***              Базовый алгоритм T/TCP                   ***/
/*************************************************************/

int flag=1;
int sd;
sd = socket(PF_INET, SOCK_STREAM, 0);
if ( setsockopt(sd, IPPROTOJTCP, TCP_NOPUSH, &flag,
                sizeof(flag)) != 0 )
    PANIC("TCP_NOPUSH not supported");
/*** выбор адресата ***/
if ( sendto(sd, buffer, bytes, MSG_FIN, &caddr,
            sizeof(addr)) < 0 )
    PANIC("sendto");
```

При работе по протоколу T/TCP необходимо запретить TCP-подсистеме очищать буферы данных слишком быстро. Для этого в показанном фрагменте программы вызывалась функция setsockopt(). Программа может посылать столько сообщений, сколько ей необходимо, но в последнем сообщении функции sendto() должен передаваться аргумент MSG_FIN.

T/TCP и Linux

К сожалению, протокол T/TCP на сегодняшний день еще не реализован в Linux. Это произойдет в будущем, а пока можно воспользоваться преимуществами протокола в других UNIX-системах. Соответствующие программы находятся на Web-узле.

Как уже упоминалось, в TCP требуется, чтобы для каждого нового соединения сокет создавался повторно, поскольку закрыть соединение можно, только закрыв сам сокет. В T/TCP этого не требуется, так как соединение устанавливается и разрывается неявно.

T/TCP позволяет обмениваться короткими пакетами с сервером при минимальном времени начального и завершающего квитирования. Это дает программе возможность быстрее реагировать на запросы сервера.

Прямая доставка сообщений

До сего момента мы имели дело с потоковыми каналами приема-передачи данных, напоминающими телефонные линии. В случае прямой доставки сообщений не требуется устанавливать соединение, т.е. отсутствует необходимость в

квитировании. Когда сообщение передается напрямую (не через канал), в процесс вовлечены только отправитель и получатель. Отправитель посылает сообщение, а получатель его принимает. Все очень просто.

Каждое сообщение должно нести в себе адрес получателя (ядро системы Linux включает в сообщение обратный адрес программы-отправителя). Функции send() и recv() предполагают, что они связаны с каналом, который автоматически определяет адресата. Поэтому для прямой доставки сообщений необходимо использовать функции sendto() и recvfrom(), которые требуют явного указания адреса.

Прежде чем послать сообщение, необходимо задать адрес получателя, состоящий из адреса компьютера и номера порта. Как объяснялось в предыдущих главах, если не выбрать порт, операционная система назначит его сокету произвольным образом. Это не очень хорошая идея для программ, подключающихся по одноранговым соединениям. Вызывающий компьютер должен знать номер порта получателя, чтобы правильно адресовать сообщение.

Поддержка структур sockaddr в Linux

Ядро Unux поддерживает несколько различных протоколов адресации, но не все они непосредственно включены в дистрибутивы системы (к примеру, протокол радиолюбительской связи). Если вы не уверены в том, какие именно протоколы поддерживаются скомпилированной версией ядра, просто запустите программу. Функции bind(), sendto(), connect() и recvfrom() выдают ошибки, если им передается разновидность структуры sockaddr, которую они не поддерживают. Полный список поддерживаемых структур приведен в приложении А, "Информационные таблицы".

Программы, отправляющие сообщения по протоколам UDP и TCP, должны знать, какой порт прослушивается сервером, чтобы операционная система могла направлять пакеты правильной программе. Номер порта обычно заранее известен как отправителю, так и получателю. Например, в файле /etc/services перечислены порты, обслуживаемые стандартными системными сервисами.

Привязка порта к сокету

Номер порта запрашивается у операционной системы с помощью функции bind(). Почему-то в большинстве справочных руководств по UNIX говорится о том, что эта функция "задает имя сокета". В действительности это относится только к сокетам домена PF_LOCAL или PF_UNIX, связанным с файловой системой. Правильнее говорить о "привязке номера порта к сокету".

Привязка имен и портов

Порядок назначения портов и наименования сокетов существенно различается в структурах семейства sockaddr. Например, в семействах' адресов AF_LOCAL и AF_AX25 используются алфавитно-цифровые имена, а в семействах AF_INET и AF_IPX— порты. Номера портов должны быть уникальными, поэтому, как правило, два сокета (TCP или UDP) не могут быть связаны с одинаковыми номерами портов (по крайней мере, в семействе AF_INET). Но можно заставить UDP-сокет и TCP-сокет совместно использовать один и тот же порт. Вот почему некоторые сервисы (из числа перечисленных в файле /etc/services) предлагают оба типа соединения.

Объявление функции bind() выглядит следующим образом:

```
#include<sys/socket.h>
#include <resolv.h>
int bind(int sd, struct sockaddr *addr, int addr_size);
```

В этом разделе даются лишь общие сведения о данной функции. Подробнее она рассматривается в главе 6, "Пример сервера". Схема ее вызова примерно такая же, как и в случае функции connect(). В первую очередь необходимо инициализировать адресную структуру:

```
/t************************************************************/
/***                  Пример функции bind()                 ****/
/*****************************************************************/
struct sockaddr addr;
int sd;
sd = socket(PF_INET, SOCK_STREAM, 0);
bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(MY_PORT); /* запрашиваем конкретный порт */
addr.sin_addr.s_addr = INADDR_ANY;          /* любой IP-интерфейс */
if ( bindfsd, Saddr, sizeof(addr)) != 0 )
    perror("bind");
```

Если сравнить пример функции bind() с аналогичным примером функции connect(),можно заметить два основных отличия. Во-первых, здесь запрашивается порт MY_PORT, а во-вторых, адрес указан как INADDR_ANY. В реальной программе номер порта будет другим, так как он зависит от компьютера, к которому производится подключение.

Константа INADDR_ANY является специальным флагом (она раскрывается как 0.0.0.0), который указывает на то, что любой компьютер может подключаться по любому интерфейсу. В некоторых компьютерах имеется более одного интерфейсного устройства (например, две сетевые платы, модем с сетевой платой или альтернативные IP-адреса). С каждым аппаратным или логическим интерфейсом связан свой IP-адрес. С помощью функции bind() можно выбрать нужный интерфейс или сразу все интерфейсы. Вторая возможность поддерживается для протоколов TCP и UDP и может использоваться при передаче данных через брандмауэры.

Если нужно выбрать конкретное интерфейсное устройство, задайте адрес следующим образом:

```
if ( inet_aton("128.48.5.161", &addr.sin_addr) == 0 )
    perrorf("address error");
```

В результате будет сделан запрос на прослушивание порта по адресу 128.48.5.161. Преобразование в сетевой порядок следования байтов можно выполнить по-другому:

```
addr.sin_addr.s_addr = htonl(0x803005Al); /* 128.48.5.161 */
```

Оба варианта приведут к одному и тому же результату. Обратите внимание на то, что при использовании константы INADDR ANY вызывать функцию htonl() не

требуется. В данном случае адрес состоит из одних нудей, поэтому порядок следования байтов не важен.

Обмен сообщениями

Передача и прием UDP-сообщений напоминает игру в футбол при сильном тумане. Игроки по очереди бьют или принимают мяч, хотя почти не видят друг друга. Первый "удар по мячу" наносит отправитель сообщения. Получатель должен знать адрес и номер порта, заданные с помощью функции bind(). В этот порт будут поступать данные. Отправителю не нужно указывать свой порт, поскольку в сообщение включается обратный адрес. Существует вероятность того, что игрок отправит мяч в пустую зону, где никого нет.

Отправитель выполняет минимум инициализирующих действий и чаще всего запрашивает одно-единственное сообщение. Можно поместить этот вызов в цикл, чтобы организовать обмен сообщениями ("перепасовку"). В листинге 4.4 показано, как реализуется передача и прием сообщения (этот фрагмент взят из файла connectionless-sender.c на Web-узле).

Листинг 4.4. Пример отправителя дейтаграмм

```
/***                    Пример отправителя                    ***/
/***            (из файла connectionless-sender.c)            ***/
/*****************************************************************/
struct sockaddr addr;
int sd, bytes, reply_len, addr_len=sizeof(addr);
char *request = "select * from TableA where fieldl = 'test';";
char buffer[1024];

/*-  создание  сокета  -*/
sd = socket(PF_INET, SOCK_DGRAM, 0);
bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(9999);     /* запрашиваем конкретный пор* */
if ( inet_aton(DEST_ADDR, &addr.sin_addr) == 0 )    /* IP-адрес
                                                       получателя */
    perror("Network IP bad");

/*-  отправка  сообщения  -*/
if ( sendto(sd, request, strlen(request), 0, saddr,
            addr_len) < 0 )
    perror("Tried to reply with sendto");

/*-  получение  ответа  -*/
bytes = recvfrom(sd, buffer, sizeof(buffer), 0, saddr, &addr_len);
if ( bytes > 0 )
    perror("Reply problem");
else
    printf("%s", buffer);
```

Предполагая, что порт получателя — 9999, программа посылает ему SQL-запрос.

Отправитель не обязан запрашивать конкретный порт, поскольку получатель принимает данные из адресной структуры в функции recvfrom(). (Еще раз напомню, что для всех сокетов требуется порт. Если он не выбран, система назначит его сама.)

Прием сообщения

В отличие от отправителя, получатель должен задать порт с помощью функции bind(). После создания сокета номер порта должен быть опубликован, чтобы компьютер на противоположной стороне мог по нему подключиться. В листинге 4.5 приведен текст программы-приемника (этот фрагмент взят из файла connectionless-receiver.c на Web-узле).

Листинг 4.5. Пример получателя дейтаграмм

```
/*************************************************************/
/***                    Пример получателя                  ***/
/***          (из файла connectionless-receiver.c)         ***/
/*************************************************************/
struct sockaddr addr;
int sd;

/*— создание сокета и назначение порта —*/
sd = socket(PF_INET, SOCK_DGRAM, 0);
bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(9999);      /* запрашиваем конкретный порт */
addr.sin_addr.s_addr = INADDR_ANY;          /* любой IP-интерфейс */
if ( bind(sd, saddr, sizeof(addr)) != 0 )
    perror("bind");

/*— прием и обработка всех запросов —*/
do
{  int bytes, reply_len, addr_len=sizeof(addr);
   char buffer[1024];

   /*— Ожидание сообщения, обработка запроса и ответ —*/
   bytes = recvfrom(sd, buffer, sizeof(buffer), 0, saddr,
                    &addr_len);
   if ( bytes > 0 .)                    /* данные получены */

      printf("Caught message from %s:%d (%d bytes)\n",
             inet_ntoa(addr.sin_addr), ntohs(addr.sin_port),
             bytes);
      /**** обработка сообщения ****/
      if ( sendto(sd, buffer, reply_len, 0, saddr,
                  addr_len) < 0 )
         perror("Tried to reply with sendto");

   else
```

```
        perror( "Awaiting message with RecvFrom");

while ( !quit );
```

В рассмотренном примере программа запрашивает порт и ожидает поступления сообщения. Когда данные приходят, программа обрабатывает запрос и возвращает результат.

Отправитель посылает только одно сообщение. Получатель функционирует подобно серверу, принимая, обрабатывая и отвечая на каждое сообщение. При этом возникает фундаментальная проблема несоответствия между двумя алгоритмами: в обеих программах требуется разный объем памяти для хранения сообщений.

Если программа работает с сообщениями переменной длины, выделите буфер большего размера. Максимальный размер UDP-сообщения составляет 64 Кбайт. Это значение можно принять как верхнюю границу буфера.

Большие UDP-сообщения

Если размер сообщения больше, чем величина буфера, диспетчер очереди отбрасывает лишние байты. Это происходит Во многих протоколах, работающих с дейтаграммами.

Подтверждение доставки UDP-сообщения

Отбрасывание части сообщения не является хорошим решением. Это один из тех моментов, о которых следует помнить при работе по протоколу UDP. Необходимо всегда знать, насколько большим может быть сообщение.

Кроме того, UDP является ненадежным протоколом, т.е. нельзя гарантировать, что адресат получил сообщение. Может показаться, что надежность является неотъемлемым свойством сетевых приложений, но на самом деле не во всех системах она нужна. Например, некоторые сообщения имеют временную привязку. Когда требуемое время истекло, важность сообщения падает до нуля.

Нарастить возможности протокола UDP (но при этом не превратить его в 100-процентный протокол TCP) можно несколькими способами. По сути, не составляет особого труда превысить надежность TCP.

Усиление надежности UDP

Суть протокола TCP заключается в обеспечении потокового интерфейса между клиентом и сервером. Большой объем служебных данных в TCP приводит к ощутимому снижению производительности соединения. Зато, в отличие от UDP, гарантируется доставка каждого сообщения. Чтобы добиться такой же надежности в протоколе UDP, придется писать дополнительный код.

Для примера предположим, что сообщение состоит из нескольких пакетов. При попытке гарантировать его доставку обнаруживаются две интересные проблемы.

- Откуда получатель знает, что пакет должен прийти?

- Как отправитель узнает, что получатель принял пакет?

Решая первую проблему, необходимо помочь получателю отслеживать все пакеты сообщения. Если обнаруживается неправильная последовательность, получатель должен запросить нужный пакет. Кроме того, отправитель должен получить подтверждение по каждому из пакетов.

Один из способов решения проблемы заключается в назначении пакетам уникальных порядковых номеров (подобно механизму подтверждений в TCP). Когда приходит пакет, программа делает пометку. Если пакет потерян, программа запрашивает повторную передачу, указывая номер пакета. Основная трудность заключается в том, что отправитель должен хранить список всех посланных пакетов, который со временем может стать очень большим.

Избежать создания длинного журнала пакетов можно, если посылать подтверждение по каждому пакету. При этом решается вторая из обозначенных проблем. Получатель должен отвечать на каждый пакет своим собственным сообщением, в котором указывается порядковый номер успешно принятого пакета. Такая методика, по сути, как раз и применяется в TCP. Однако одновременно с этим возникает новая проблема, связанная с производительностью.

Дело в том, что большинство физических сетей не являются полнодуплексными. Каждый передаваемый сигнал занимает часть полосы пропускания, препятствуя прохождению других сигналов. Кроме того, у каждого пакета есть IP- и UDP-заголовок (минимум 28 байтов). Если получатель будет отвечать на каждый пакет по отдельности, возникнет переизбыток служебной информации (4 или 5 байтов для подтверждения порядкового номера и 28 байтов для заголовка UDP/IP). Поэтому, чтобы снизить нагрузку на сеть, необходимо заставить получателя "молчать" как можно дольше.

Решение состоит в группировке подтверждений по нескольким пакетам. Например, если размер пакета 1024 байта и посылается сообщение длиной 1 Мбайт, получатель должен принять 1000 пакетов. Имея предел в 1024 байта, получатель может ответить сразу на 10 пакетов. Тогда максимальный размер журнала пакетов будет 10x1024 байта. Чтобы правильно определить предельные размеры, необходимо знать пропускную способность сети и ограничения памяти, существующие на адресуемом компьютере.

И тут мы сталкиваемся с еще одной проблемой: откуда получатель знает, что отправитель закончил передавать данные? Когда в алгоритме получателя заложено, что отвечать нужно на каждый десятый пакет, а отправитель посылает всего пять пакетов, произойдет следующее:

- получатель никогда не пошлет подтверждения;
- отправитель решит, что пакет не дошел до получателя, и осуществит повторную передачу.

Чтобы решить эту проблему, необходимо сообщить получателю общее число пакетов. Когда приходит последний пакет, получатель посылает укороченное подтверждающее сообщение.

Упорядочение пакетов

Каждый пакет UDP-сообщения может достигать адресата в неправильной последовательности относительно других пакетов. Это связано с динамичной природой Internet (см. главу 3, "Различные типы Internet-пакетов"). Разрывы и пе-

регрузки в сети могут заставить маршрутизатор задержать сообщение или послать его по другому маршруту.

Чтобы обеспечить правильную доставку пакетов, необходимо присвоить им уникальные номера. Номера должны идти по возрастанию и быть уникальными в пределах сообщения. Если пакет приходит вне очереди, программа должна задержать его, дожидаясь доставки предыдущих пакетов. Например, когда поступают пакеты 1, 2, 4 и 5, пакеты 4 и 5 откладываются до тех пор, пока не придет пакет 3.

В случае, если пакет 3 потерялся в процессе передачи, получатель может либо явно его запросить, либо послать подтверждение по тем пакетам, которые были получены. В обоих случаях отправитель повторно пошлет недостающий пакет.

Избыточность пакетов

В сети может возникать дублирование пакетов, когда один и тот же пакет дважды достигает адресата. Чаще всего это происходит, если получатель запросил повторную передачу, а в это время появляется запоздавший пакет. Присвоение пакетам порядковых номеров позволяет избежать избыточности. Получатель просто удаляет пакет с повторяющимся идентификатором.

Проверка целостности данных

Пакеты перемещаются от источника к приемнику, проходя через сетевые каналы и маршрутизаторы. В любой момент может произойти повреждение сообщения (так часто бывает в перегруженных сетях). В рассмотренном выше примере потерявшийся пакет 3 наверняка был послан, но "растворился" в сети. Отправитель оказался вынужден повторно выслать пакет.

В UDP и TCP используется обратная контрольная сумма (при ее добавлении к суммарному значению принятых байтов должно получиться число, все биты которого равны 1). В протоколе IP проверяется контрольная сумма заголовка, но не данных. С помощью подобных проверок можно обнаружить несложные ошибки, но очень часто парные ошибки как бы компенсируют друг друга и не проявляются в контрольной сумме. В некоторых сетевых устройствах для нахождения и исправления поврежденных данных применяются аппаратные циклически избыточные коды (CRC — cyclical redundancy check) или коды коррекции ошибок (ECC — error correction code). Все тот же пакет 3 мог на самом деле прийти, но проверка контрольных сумм IP и UDP выявила наличие ошибок.

При передаче пакетов и сообщений можно самостоятельно применять различные контрольные суммы, CRC- и ECC-коды или хеширование. Соответствующие алгоритмы описаны в Internet, там же имеются готовые программы. При выборе алгоритма проверки необходимо учесть следующее:

- объем передаваемых данных — увеличение размеров заголовков приводит к существенному снижению пропускной способности сети;
- возможность исправления данных — некоторые данные можно легко восстановить, поскольку в них имеется избыточность либо они некритичны к ошибкам;
- порядок данных — контрольные суммы хорошо подходят для быстрой проверки итоговых значений, но в них не учитывается порядок следова-

ния байтов. Для получения CRC-кода требуются более трудоемкие вычисления, зато он позволяет обнаруживать ошибки в отдельных битах. "Глубина" поиска определяется длиной кода. Например, 32-разрядный CRC-код гарантирует нахождение ошибки в цепочке длиной до 32 битов.

Надежность важна в большинстве приложений. Проверка контрольных сумм, подтверждение правильности сообщения и упорядочение пакетов по номерам — все это позволяет обеспечить надежную доставку сообщений.

Задержки при передаче данных

Взаимодействие между компьютерами не всегда проходит гладко. Например, каждая из сторон может предположить, что противоположная сторона должна начинать передачу первой (тупик, или *взаимоблокировка).* Другой проблемой является бесконечное ожидание. Это напоминает ситуацию, когда вы звоните в офис, секретарь просит подождать, а сам уходит пить кофе.

Зависание происходит, когда запрашивающая сторона бесконечно долго ждет ответа. Причиной может быть медленная сеть, перезагрузка компьютера и т.д. В большинстве случаев проблему можно решить путем написания подпрограммы, посылающей напоминания отправителю о том, что вы все еще ждете ответа.

Прежде чем начинать работу, такая подпрограмма должна выждать определенное время. В текст напоминания могут включаться идентификатор последнего полученного пакета и метка времени. Не получив ничего в ответ, получатель может сбросить связь. Описанная методика подразумевает, что отправитель понимает оба сообщения: "напоминание" и "сброс".

Взаимоблокировка может произойти в случае потери одного из сообщений, когда отправитель ждет ответа, а получатель — потерянного сообщения. Подпрограмма напоминания также может разрешить эту ситуацию, сообщая отправителю идентификатор последнего принятого пакета.

Переключение задач: введение в многозадачность

Зачем нужно создавать две разные программы: отправителя и получателя? Почему не сделать одну? В принципе, такое возможно, но ведь нельзя одновременно и посылать, и принимать данные.

Предположим, имеется распределенная программа, выполняющая очень быструю обработку изображений. Обработка осуществляется в несколько этапов: захват изображения, ретуширование, сглаживание и т.д. Все этапы реализуются на разных компьютерах, и данные перетекают из одного этапа в другой. Модуль ретуширования принимает данные от утилиты захвата, обрабатывает их и передает сглаживающему фильтру. На каждом этапе данные должны приниматься и передаваться практически одновременно.

Совмещать два этапа нельзя: на каждом этапе входная информация не должна смешиваться с выходной. Можно обнаружить еще одну проблему: как передать сообщение об ошибке вверх по цепочке?

Разделение задания на четко определенные компоненты является одним из базовых принципов *многозадачности.* Можно запустить несколько программ одно-

временно, причем каждую в своем адресном пространстве, и обеспечить тем самым изоляцию входных и выходных потоков.

Программы, рассматривавшиеся в данной главе, столь похожи, что можно объединить их в одну приемопередающую систему, работающую в многозадачном режиме. Одна копия программы будет работать на прием, а другая — на передачу, причем взаимодействовать они будут друг с другом. Необходимо только выбрать, кто первый начнет посылать сообщения.

Многозадачность является неотъемлемым элементом сетевого программирования. Подробнее она рассматривается в главе 8, "Механизмы ввода-вывода".

Резюме: модели взаимодействия с установлением и без установления соединения

Передача сообщений между одноранговыми компьютерами или между клиентом и сервером может требовать либо не требовать установления соединения. Когда соединение устанавливается, клиент открывает канал, подключается к другому компьютеру и начинает передавать и принимать данные. Адрес получателя запоминается, поэтому его не нужно каждый раз указывать заново. Соединения поддерживаются как в протоколе TCP, так и в UDP, но лишь в TCP гарантируется надежная и последовательная доставка пакетов. В UDP соединения существуют просто для удобства, чтобы можно было работать с высокоуровневыми функциями ввода-вывода, а не с функциями sendto() и recvfromf).

При отсутствии соединения передача сообщения подобна отправке письма: в каждом письме необходимо указывать адрес. Если в TCP прием и передача данных осуществлялись с помощью функций send() и recv(), то в UDP используются функции sendto() и recvfrom(), дополнительно требующие указания адреса.

Протокол T/TCP (Transaction TCP) является версией TCP, в которой не поддерживаются соединения. Данный протокол позволяет избежать длинных процедур установления и разрыва соединения, что ускоряет процесс передачи сообщения. Кроме того, передача данных осуществляется в виде пакета, а не потока, поэтому взаимодействие между клиентом и сервером ограничено всего тремя пакетами. В Linux этот протокол пока еще не реализован.

По умолчанию в UDP не устанавливается соединение, но можно произвести подключение сокета, чтобы упростить программирование. В отличие от TCP, в UDP можно последовательно подключать один и тот же сокет к различным компьютерам.

Протокол TCP обеспечивает надежность, но снижает производительность. Можно повысить надежность UDP, внедрив в программу средства проверки доставки, идентификации пакетов, устранения избыточности, проверки целостности и предотвращения зависаний. Протоколы TCP и UDP обеспечивают разный уровень межпрограммного взаимодействия. В TCP данные рассматриваются как поток, а в UDP происходит манипулирование отдельными сообщениями. Оба протокола базируются на архитектуре протокола IP.

Многоуровневая сетевая модель

Глава

5

В этой главе...

Одним из любимых десертных блюд в Европе является торт. Он обычно состоит из нескольких (от пяти до восьми) прослоек. Каждая из них имеет свой вкус и цвет. Сверху торт может быть залит глазурью.

Если проанализировать архитектуру сетевой подсистемы (или *стека протоколов),* то обнаружится очевидная аналогия с тортом. На поверхности находится общедоступный программный интерфейс, под которым располагаются слои протоколов. Каждый сетевой слой основан на другом слое, без которого он не сможет функционировать. Эта зависимость очень важна: если в одном из слоев возникнут проблемы, вся сеть "рухнет" или, по крайней мере, начнет функционировать неоптимально.

Сетевая подсистема представляет собой сложное объединение, в которое входят аппаратные устройства, ядро системы, подсистема ввода-вывода, механизмы обеспечения безопасности, данные и приложения. Координация работы всех перечисленных компонентов составляет главную сетевую задачу.

В этой главе рассматриваются основные проблемы, возникающие при решении сетевой задачи, и описываются две сетевые модели, в которых это решение найдено.

Решение сетевой задачи

С сетевым программированием связано столько технологий и методик проектирования, что заниматься им на низком уровне очень сложно. Подобно строительству дома, нужно думать не столько об укладке кирпичей, сколько о проекте в целом. Трудности, возникающие в сетевом программировании, делают данную область особенно интересной, но, чтобы создавать сетевое приложение "с нуля", нужно быть гуру или специалистом практически по всем компьютерным специальностям. Весь круг решаемых вопросов можно разделить на следующие категории: аппаратная среда, передача данных, взаимодействие с операционной системой и взаимодействие с приложениями.

Аппаратная среда

Существует много технологий построения сетей. В основе сети может находиться проводящая или непроводящая среда (физический канал передачи данных); данные могут передаваться электрическим или неэлектрическим способом, прямыми или непрямыми маршрутами, на короткие или длинные расстояния. В табл. 5.1 перечислены характеристики основных передающих сред.

Таблица 5.1. Характеристики сетевых информационных сред

Среда	Проводник	Электрический канал	Направленность	Расстояние	Максимальная пропускная способность
Коаксиальный кабель	Да	Да	Прямое подключение	<2км	10 Мбит/с
Витая пара	Да	Да	Прямое подключение	<150м	100 Мбит/с
Оптоволокно	Да	Нет	Прямое подключение	(Не ограничено)	100 Гбит/с

Окончание табл. 5.1

Среда	Проводник	Электрический канал	Направленность	Расстояние	Максимальная пропускная способность
Беспроводная связь: HF	Нет	Да	Широковещание	> 1000 км	< 10 Кбит/с
Беспроводная связь: VHF/UHF	Нет	Да	Широковещание в пределах прямой видимости	<30км	< 40 Кбит/с
Беспроводная связь: микроволны	Нет	Да	Да, в пределах прямой видимости	<30км	< 1 Мбит/с
Спутник	Нет	Да	Да	(Не ограничено)	< 10 Мбит/с
Инфракрасное излучение	Нет	Нет	Да, широковещание	<10м	< 1 Мбит/с
Лазер	Нет	Нет	Да	Очень большое	< 100 Гбит/с

К счастью, подобные детали (скрываются на уровне ядра системы. Представьте, насколько усложнилась бы наша задача, если бы пришлось учитывать тип передающей среды. Тем не менее в каждой среде существуют общие проблемы.

Одной из них является *затухание сигнала,* вызываемое сопротивлением среды. Данная проблема особенно часто возникает в средах электрического типа (коаксиальный кабель, витая пара). В них сообщение может разрушаться непосредственно в процессе передачи.

Другая возможная проблема — *взаимные помехи сигналов.* Они возникают, когда несколько компьютеров, совместно использующих общую среду (проводящую или непроводящую), одновременно отправляют сообщения. Если компьютер посылает пакет в то время, когда передается другой пакет, в результате возникшего конфликта искаженными окажутся оба пакета. Обе передающие стороны должны обнаружить конфликт, отменить передачу, восстановить пакет и послать его повторно.

Помехи в передающем канале приводят к потере сигнала и разрушению пакетов. Если отсоединить коаксиальный кабель от сети, любой распространяющийся электрический сигнал отразится от свободного конца кабеля как от зеркала. Все сообщения окажутся искаженными.

От помех страдают также среды с направленной передачей: лазер, микроволны, инфракрасное излучение. Преграда, возникшая между приемником и передатчиком, может полностью блокировать сигнал. Помехи могут возникать вследствие сигаретного дыма, тумана, полетов птиц и т.д.

Компьютеры подключаются к единой передающей среде, совместно используя сетевые ресурсы, поэтому иногда проблемой оказывается идентификация аппаратных устройств. Если сетевое соединение установлено всего между двумя компьютерами (сеть "точка-точка"), адресат не нужен, так как он очевиден. Но в сети общего доступа каждый пакет должен передаваться по конкретному аппаратному адресу, чтобы сетевая плата могла выбирать предназначенные ей сообщения из проносящегося мимо потока данных.

Ethernet-плата имеет свой собственный идентификатор: шестибайтовый MAC-адрес. Когда компьютер посылает сообщение, сетевая подсистема разделяет его

на кадры, или фреймы, — наименьшие единицы передаваемой информации. В сетях Ethernet каждый кадр содержит MAC-адрес источника и приемника. Впоследствии сетевая подсистема связывает логический IP-адрес с физическим MAC-адресом. Сетевая плата принимает только те сообщения, которые несут ее идентификатор.

Передача данных в сети

Выше были рассмотрены вопросы физической организации каналов связи. Следующий круг проблем возникает при передаче данных по сети. Существует множество факторов, способных привести к повреждению или потере пакета. Часто отправитель и его адресат не получают уведомления о сбое системы. Иногда получатель превышает лимит времени, отведенный на ожидание сообщения, поскольку произошел разрыв сети.

На сетевом уровне проблемы передачи пакетов обычно связаны с маршрутизацией. Поскольку сеть может меняться в процессе перехода сообщения от одного компьютера к другому, полезно заранее знать, что может произойти.

Изменения в сетевой топологии (или *динамика распространения сигнала)* возникают даже в самых надежных сетевых архитектурах. Причины этих изменений разные, но в результате происходит потеря данных. Читатели наверняка сталкивались с этим сами, когда кто-то неосторожно зацепил сетевой кабель и разорвал сегмент. Конфигурация сети часто меняется, соединения между компьютерами могут то появляться, то исчезать.

Другая проблема связана со старением пакетов. Когда пакет продвигается по маршруту, на каждом принимающем узле с помощью поля TTL (time-to-live) отслеживается время жизни пакета. Пакет устаревает, когда число переходов через маршрутизаторы превышает заданное число. Каждый пакет может осуществлять до 255 переходов. По умолчанию задан лимит 64 перехода. Обычно этого достаточно, чтобы пакет мог попасть в пункт назначения.

Ограничение числа переходов очень важно, так как позволяет избавить сеть от переполнения давно забытыми пакетами. Пакет, за которым не ведется слежение, легко может "зациклиться". Предположим, маршрутизатор А получает сообщение. На основании его таблицы адресов выясняется, что наилучший путь пролегает через маршрутизатор Б, которому и передается сообщение. Но маршрутизатор Б потерял связь с адресатом, поэтому он отправляет сообщение обратно. У маршрутизатора А не записано, что пакет был принят, и в результате пакет передается туда-сюда до тех пор, пока счетчик TTL не достигнет предельного значения или маршрутизаторы А и Б не согласуют свои таблицы. Поле TTL определяет, насколько долго пакет может находиться в цикле.

Таблицы маршрутизации могут стать очень большими. Тщательное управление ими позволяет избежать длительных задержек в процессе вычисления оптимального пути к адресату.

Трудности возникают не только в связи со сложной сетевой топологией, но также при обнаружении потерь и циклов. Часто, когда исчезает сетевой сегмент, образуется цикл, в котором пакет быстро устаревает. Сетевая подсистема далеко не всегда уведомляет о связи с проблемными узлами. Как правило, пакет просто устаревает и удаляется из сети.

Иногда неоднозначность сетевых маршрутов приводит к тому, что пакет множится в процессе передачи *(зеркальное двоение).* Столь странное явление невозможно обнаружить, если только с каждым пакетом не связан идентификатор или

порядковый номер. В сети может происходить не только ненужное дублирование информации, но также ее потеря при прохождении через ненадежные маршрутизаторы или при повреждении пакета, вследствие чего отправитель никогда не получит сообщения об ошибке.

Потери пакетов происходят постоянно. Не ведя учет пакетов и не принимая регулярно подтверждения, отправитель не сможет определить, получил ли адресат нужное сообщение.

Наконец, проблемы могут возникать из-за несовместимости маршрутизаторов. Некоторые маршрутизаторы и сети не поддерживают 8-битовый режим передачи или пакеты больших размеров. В таких случаях маршрутизатор должен выполнять преобразование пакета или возвращать сообщение об ошибке.

К счастью, с описанными проблемами приходится сталкиваться только в протоколах очень низкого уровня. Если у вас нет желания заниматься ими и вас устраивает не очень высокая пропускная способность, используйте испытанные и проверенные протоколы высокого уровня. На них основано большинство сетевых приложений. Протестируйте свой алгоритм на высоком уровне и лишь затем переходите к низкоуровневой настройке производительности.

Взаимодействие сети и операционной системы

Сетевая подсистема должна взаимодействовать с операционной системой по нескольким причинам. Во-первых, операционная система управляет сетевыми ресурсами, такими как прерывания, порты и память. Во-вторых, сама сетевая подсистема размещается в системном ядре, чтобы обеспечить высокую производительность.

Интерфейс между ядром и сетью сложен, особенно если ядро не является реентерабельным (не допускает повторное использование точки входа в ядро). Однако в ядре Linux проблема реентерабельности уже решена, поэтому общая задача упрощается.

Когда приходит сообщение, сетевое устройство посылает центральному процессору запрос на прерывание, чтобы как можно быстрее вытолкнуть сообщение из буфера. Если ядро (в котором располагается обработчик прерываний) хоть немного запоздает, может прийти следующее сообщение, которое займет место первого. Это особенно часто происходит, когда сетевая плата работает в *беспорядочном режиме* (принимает все сообщения, проходящие по сети; см. главу 3, "Различные типы Internet-пакетов").

За исключением сетей наподобие PPP, центральный процессор редко взаимодействует с сетевым оборудованием напрямую. Чаще всего он работает с сопроцессором, располагающимся на сетевой плате. Процессор загружает данные в буфер сетевой платы и посылает ей команду на передачу. Когда отправка данных завершена, плата посылает процессору запрос на прерывание. Это служит для ядра сигналом о том, что плата готова принять следующую порцию данных. В PPP требуется более интенсивное взаимодействие с процессором, но проблем с наложением сообщений не возникает, так как это более медленный протокол.

В очереди сообщений хранятся как исходящие, так и входящие пакеты. Когда ядро получает сигнал о том, что сетевая плата готова принять очередной блок данных, оно извлекает сообщение непосредственно из очереди. Если программа посылает сообщение, а плата не готова его обработать, ядро помещает сообщение в очередь. Сеть является ограниченным ресурсом: никакие две программы не

могут обратиться к нему одновременно. Поэтому операционная система должна обрабатывать запросы по одному за раз.

Кроме того, ядро должно подготавливать сообщение к передаче и разупаковывать его при получении. Буферная подсистема отслеживает сформированные сетевые кадры (фреймы) и подготавливает их для сетевой подсистемы или клиентского процесса (например, для функций дискового ввода-вывода).

Частью процесса упаковки сообщения является назначение идентификаторов. Работать с аппаратными идентификаторами (в частности, с MAC-адресами) слишком неудобно в больших сетях, поэтому в сетевой подсистеме применяется логическая идентификация (посредством IP-адресов).

Каждый пакет должен содержать идентификатор, чтобы операционная система могла быстро определить, куда его следует отправить. IP-адреса позволяют сетевой подсистеме группировать компьютеры в подсети.

Взаимодействие сети и приложений

Для программистов основная задача заключается в организации взаимодействия сети и приложения. Здесь скрыто много проблем, с каждой из которых необходимо разбираться по отдельности.

Первая проблема связана с приемом и обработкой сообщений об ошибках и исключительных ситуациях. Некоторые служебные сообщения поступают в программу в асинхронном режиме. Механизмы их обработки существуют в C и Java, но не в Pascal, например. Поэтому при разработке приложения следует учитывать, что не все языки подходят для сетевого программирования.

Вторая проблема связана с надежностью данных и пакетов. В одних программах требуется высокая надежность, в других — нет. Проанализируйте, с какого рода данными вам предстоит работать и насколько надежными они должны быть. В главе 3, "Различные типы Internet-пакетов", приводился перечень категорий данных в зависимости от их важности. Руководствуйтесь этим список при выборе нужного протокола.

Третьей проблемой является синхронизация. Любое сетевое приложение взаимодействует с другой одновременно выполняемой программой. Необходимо координировать их работу, чтобы избежать взаимоблокировок и зависаний (подробнее об этом рассказывается в главе 10, "Создание устойчивых сокетов").

Наконец, следует различать работу по реальным и виртуальным соединениям. Сетевая подсистема упрощает схему функционирования программы, позволяя ей запрашивать сетевое соединение для монопольного использования. Через эти виртуальные соединения (порты) в программу поступают только те сообщения, которые адресованы непосредственно ей.

Хотя организация межпрограммного взаимодействия достаточно сложна, именно она делает сетевое программирование захватывающе интересным. Список перечисленных проблем не является исчерпывающим, но многие из них скрыты на уровне библиотеки Socket API.

Несколько лет назад группы программистов взялись за решение сетевой задачи и разработали ряд стандартных сетевых моделей. Стандарты обычно базируются друг на друге, образуя слои. Если, как в луковице, снять слой за слоем, то в середине обнаружится среда физической передачи данных (электричество, радиоволны, свет). В этой главе будут рассмотрены две основные модели: OSI и IP.

Сетевая модель OSI

Наиболее известная сетевая модель— OSI (Open Systems Interconnection — взаимодействие открытых систем) — основана на многоуровневом подходе и имеет 7 уровней (слоев), описывающих как физическое взаимодействие программы и сети, так и пользовательское взаимодействие с сетевым приложением. Модель OSI реализует единый межплатформенный интерфейс доступа к сети, в котором скрыты все детали аппаратной реализации.

Модель OSI предназначена для решения всех задач, возникающих в сетевом программировании. Вместе с тем она предоставляет доступ к протоколам низкого уровня, благодаря чему профессиональные программисты могут создавать приложения любой степени сложности. В модели имеется семь уровней, начиная с аппаратного (рис. 5.1). Каждый следующий уровень все более скрывает от пользователя и сетевого приложения детали организации сети.

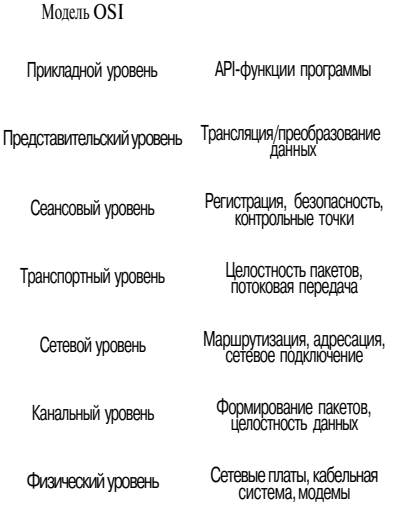

Рис. 5.1. Уровни модели OSI: с каждым следующим уровнем возрастает функциональность и надежность, но снижается производительность

Уровень 1: физический

Физический уровень охватывает все аппаратные интерфейсы, описывая среду передачи данных и способы распространения сигнала. Среда — это носитель (например, витая пара, коаксиальный кабель, оптоволокно), по которому передается сигнал.

На физическом уровне работает сетевой адаптер. Он функционирует в качестве посредника между ядром системы и физическим носителем. На одном конце соединения он принимает запросы на передачу данных от драйверов ядра и посылает пакеты сообщений (кадры, или фреймы). По окончании передачи он уведомляет ядро, посылая запрос на прерывание.

Микроконтроллер сетевого адаптера проверяет состояние сети перед отправкой кадра. Он обнаруживает конфликты и обрабатывает запросы на ретрансляцию, а если необходимо — уведомляет ядро о возникших проблемах. Драйверы ядра либо самостоятельно организуют повторную передачу данных, либо посылают сообщение об ошибке в стек протоколов.

На другом конце соединения адаптер прослушивает сеть, ожидая сообщений, в которых указан его аппаратный (Ethernet) адрес. Когда приходит сообщение, оно помещается во внутренний буфер, после чего генерируется запрос на прерывание. На физическом уровне имеются собственные механизмы идентификации, позволяющие реализовать описанную схему взаимодействия.

Если в сети есть повторитель, то он принимает все сообщения и передает их в следующий сегмент. Повторитель является "неинтеллектуальным" устройством в том смысле, что он не осуществляет фильтрацию пакетов или проверку аппаратных адресов. Повторитель функционирует на физическом уровне и предназначен для увеличения длины кабельной системы, так как он усиливает затухающие сигналы.

Следует также упомянуть о том, что в настоящее время некоторые адаптеры Ethernet включают в передаваемые сообщения собственные контрольные суммы или CRC-коды, выполняя таким образом проверку данных. Когда обнаруживается ошибка целостности, плата помечает данные как подозрительные, с тем чтобы на канальном уровне ошибка была исправлена. Это позволяет немного повысить надежность протоколов передачи дейтаграмм.

В различных сетях используются разные механизмы проверки. Например, в PPP-соединении осуществляется контроль четности. В FDDI применяются контрольные суммы и CRC-коды. В некоторых протоколах радиовещания каждый символ просто посылается дважды.

Уровень 2: канальный

Основное предназначение канального уровня заключается в управлении передачей данных от узла к узлу. Здесь происходит разбивка сообщений на физические кадры, а также осуществляется обнаружение и исправление ошибок (если это возможно). Если аппаратное устройство не поддерживает контрольные суммы или CRC-коды, канальная подсистема вычисляет их самостоятельно.

Канальный уровень образует интерфейс между ядром системы и сетевым адаптером. Обычно он реализуется в виде сетевого драйвера, размещаемого в ядре. Драйвер скрывает в себе все детали взаимодействия с физическим уровнем, благодаря чему ядро может работать с самыми разными сетевыми устройствами.

На этом уровне кадры данных подготавливаются к передаче и происходит восстановление полученных сообщений. Драйвер ожидает запросов на прерывание, сигнализирующих об отправке или получении сообщения. Получив уведомление о том, что сообщение отправлено, драйвер загружает следующий кадр.

Канальная подсистема тесно взаимодействует с буферной подсистемой ядра. В большинстве случаев она использует 10-60 Кбайт системной памяти. Утилита конфигурирования ядра Linux позволяет при наличии достаточного объема памяти установить размер буфера и кадра равным более 1 Мбайт для сверхбыстрых соединений (100 Мбайт/с и выше). Это может потребовать перекомпиляции ядра.

На канальном уровне работают сетевые мосты. Функция программного моста в ядре Linux на момент написания данной книги все еще являлась экспериментальной. Чтобы ее активизировать, необходимо переконфигурировать и заново скомпилировать ядро, подключив к нему необходимые драйверы.

Уровень 3: сетевой

Сетевой уровень отвечает за преобразование адресов и маршрутизацию. Его функция заключается в поиске узла-адресата с помощью шлюзов и маршрутизаторов. В сетях, не поддерживающих маршрутизацию, данный уровень неактивен.

Сетевой уровень обеспечивает единый механизм адресации компьютеров в гетерогенных сетях. Если, к примеру, сообщение передается из сети AppleTalk в сеть Ethernet, будет осуществлена автоматическая трансляция кадров.

На данном уровне работают сетевые маршрутизаторы, соединяющие между собой гомогенные и гетерогенные сети.

Уровень 4: транспортный

Транспортный уровень отвечает за доставку сообщения в неповрежденном виде, в правильном порядке и без дублирования. На данном уровне появляются функции, обеспечивающие потоковую передачу данных и целостность потока.

Это последний уровень модели, на котором проверяются ошибки данных и сеанса связи. (Под *ошибками данных* здесь понимается только целостность данных, а не их смысловая непротиворечивость. Например, на данном уровне не выявляются неправильно сформированные HTTP-запросы, зато обнаруживаются ошибки контрольных сумм.) Если выявлена потеря пакета или его повреждение, транспортная подсистема просит отправителя произвести ретрансляцию. С целью проверки ошибок в пакет добавляются дополнительные контрольные суммы.

В данном уровне вводится понятие *виртуальной сети,* или *мультиплексирования.* Здесь каждое соединение функционирует так, как будто имеет монопольный доступ к сети (вспомните концепцию порта в TCP/IP). Программа получает только сообщения, адресованные непосредственно ей, хотя на данном компьютере может одновременно выполняться несколько сетевых приложений.

Транспортный уровень наиболее часто используется приложениями, взаимодействующими друг с другом в среде Internet. В нем имеются все средства, необходимые для установления соединений.

Уровень 5: сеансовый

Основное предназначение сеансового уровня заключается в контроле над соединениями и потоками данных. Поскольку соединение может быть нестабильным, вводится механизм *контрольных точек.* Предположим, например, что в процессе передачи файла происходит разрыв соединения. Время, потраченное на доставку первой части файла, окажется потраченным впустую, если при повторном подключении файл придется передавать заново. Благодаря контрольным точкам состояние сеанса фиксируется в определенные моменты времени, поэтому в новом сеансе достаточно будет передать недостающую часть файла.

Из сказанного следует, что сеансовый уровень отвечает за возобновление прерванного соединения. Он также реализует процедуры аутентификации и регистрации в системе.

Другой функцией сеансового уровня является управление потоком данных. Одна из проблем клиент-серверного взаимодействия состоит в определении порядка ведения диалога. В сеансовом уровне эта проблема решается путем передачи маркера: компьютер, получивший маркер, может осуществлять критические

операции. Использование маркеров позволяет уменьшить вероятность возникновения взаимоблокировок, зависаний и повреждения данных (подробнее об этом — в главе 10, "Создание устойчивых сокетов").

Уровень 6: представительский

Представительский уровень определяет способы обработки данных: шифрование, кодирование, форматирование, сжатие и т.д. Технология RFC (Remote Procedure Calls — удаленные вызовы процедур) реализована на этом уровне (за подробной информацией обратитесь к главе 15, "Удаленные вызовы процедур (RPC)").

Уровень 7: прикладной

Последний, прикладной, уровень предоставляет различные сетевые сервисы, такие как пересылка файлов, эмуляция терминала, электронная почта и сетевое управление. Данный уровень используется практически всеми приложениями, поскольку в нем вводятся библиотеки API-функций, помогающих программе взаимодействовать с сетью. На прикладном уровне реализована сетевая файловая система (NFS), являющаяся частью архитектуры операционной системы и построенная на основе RPC.

Набор протоколов Internet

Linux (как и большинство других UNIX-систем) не использует модель OSI напрямую. Но эта модель является отправным пунктом для понимания стека протоколов TCP/IP. В Linux применяется набор протоколов Internet, позволяющих управлять собственными сетевыми интерфейсами Linux.

Примечание

Протокол IP появился в 1972 г. в сети ARPAnet, основанной организацией DARPA (Defense Advanced Research Projects Agency— Управление перспективных исследовательских профамм министерства обороны США). Корпорация BBN реализовала первые варианты протокола. Позднее UNIX, бесплатная операционная система, разработанная компанией Bell Labs, также приняла модель IP. Университеты, входившие в сеть ARPAnet,, использовали UNIX для проверки того, как могут компьютеры взаимодействовать и между собой в пределах США. Библиотека Socket API появилась в BSD 4.2 в 1983 г.

Набор протоколов Internet состоит из четырех уровней, которые тесно связаны с моделью OSI. Самый верхний уровень называется прикладным. Он охватывает уровни OSI с пятого по седьмой (рис. 5.2).

Уровень 1: доступ к сети

Первый уровень семейства протоколов Internet соответствует физическому и канальному уровням модели OSI. Поскольку аппаратные устройства и их драйверы тесно связаны между собой, **их** нельзя рассматривать по отдельности.

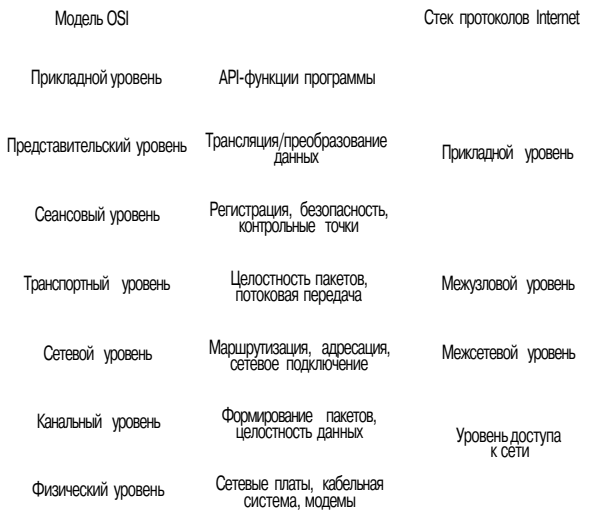

Рис. 5.2. Стек протоколов Internet напоминает модель OSI; библиотека Socket API связана с транспортным уровнем OSI и ниже, а более высокие уровни оставлены для таких программ, как Telnet, FTP и Lynx

Характеристики данного уровня такие же, как и у его "собратьев" в OSI. Драйвер напрямую взаимодействует с сетевым интерфейсным устройством, предоставляя ядру системы набор функций. Ядро, в свою очередь, обеспечивает драйверу прямой доступ к портам и вызовам прерываний. Если аппаратное устройство не поддерживает какую-то возможность, необходимую ядру, ее эмулирует драйвер.

Linux усложняет данный уровень собственным расширением: сетевыми адаптерами "горячей" замены. Эта технология применяется в PCMCIA-устройствах. Когда плата PCMCIA вставляется в компьютер, диспетчер PCMCIA распознает ее, назначает ей порт и запрос на прерывание, загружает соответствующие модули ядра и конфигурирует сетевую подсистему. Аналогичным образом при удалении платы диспетчер деактивизирует сетевую подсистему и выгружает модули ядра из памяти.

Уровень 2: межсетевой (IP)

Оба IP-стандарта, IPv4 и IPv6, связаны с драйверами устройств и предоставляют открытые интерфейсы и наборы функций. Эти стандарты отличаются друг от друга на межсетевом уровне, образуя различные стеки протоколов, между которыми при необходимости может выполняться преобразование.

Межсетевой уровень соответствует сетевому уровню модели OSI. В нем выполняется обработка логических адресов и маршрутизация. Единственное, что отсутствует на данном уровне, — это механизм обработки ошибок.

Уровень 2: управляющие расширения (ICMP)

ICMP (Internet Control Message Protocol — протокол управляющих сообщений в сети Internet) обеспечивает недостающую обработку ошибок. Он позволяет выявлять ошибки, а также исключительные ситуации в процессе передачи и маршрутизации сообщений. Типичными примерами управляющих сообщений являются "network not reachable" (сеть недоступна) и "host not found" (узел не найден).

Этот протокол работает совместно с другими подсистемами, выполняя различные функции. Например, он используется, когда в протоколе ARP просматривается таблица адресов и обнаруживается, что узел недоступен. Кроме того, он применяется в TCP в алгоритме "раздвижного окна" для ускорения или замедления соединения. В стандарте IPv6 управляющие сообщения посылаются при многоадресной (групповой) передаче пакетов.

Особенность протокола ICMP заключается в том, что он обособлен. Ни в UDP, ни в TCP он не используется. Эти протоколы принимают ICMP-сообщения и обрабатывают их по своему усмотрению.

Уровень 3: межузловой (UDP)

Вопреки распространенному мнению, UDP (User Datagram Protocol — протокол передачи дейтаграмм пользователя) далеко не в полной мере соответствует транспортному уровню модели OSI (рис. 5.3). На транспортном уровне гарантируется доставка, правильный порядок пакетов, отсутствие ошибок и потоковая передача. Всего этого в UDP нет (табл. 5.2).

Таблица 5.2. Сравнительные характеристики транспортного уровня и протокола UDP

Транспортный уровень	UDP
Надежные данные	Надежные данные (контрольные суммы)
Надежная доставка	Не гарантированная доставка
Согласуемый размер окна	Фиксированное окно (устанавливается программой)
Ориентирован на записи	Ориентирован на пакеты
Основан на сетевом уровне	Основан на протоколе IP

В UDP имеется одно существенное дополнение к транспортному уровню: виртуальные порты. Как описывалось в главе 4, "Передача сообщений между одноранговыми компьютерами", они позволяют программе вести себя так, как если бы она монопольно владела сетью. Все остальные свойства UDP соответствуют параметрам межсетевого уровня стека протоколов Internet. Правильнее всего UDP позиционируется между сетевым и транспортным уровнями модели OSI, больше попадая в сетевой уровень.

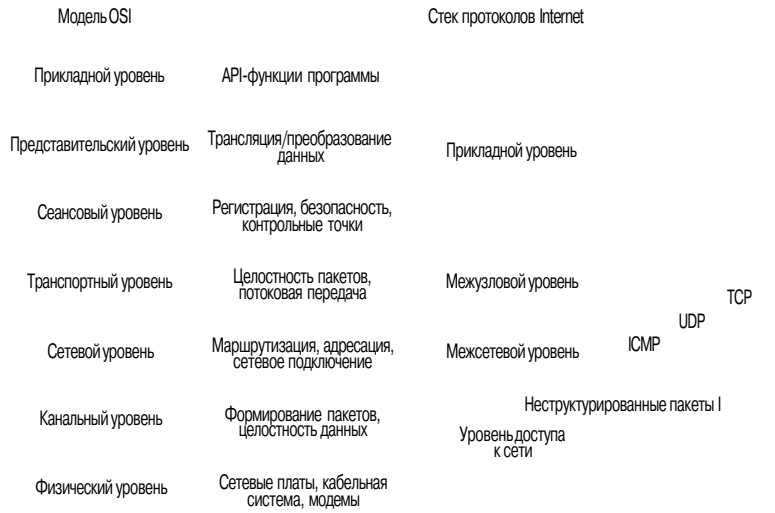

Рис. 5.3. Стек протоколов Internet больше ориентирован на расширение функциональности, а не на инкапсуляцию; как можно заметить, протокол UDP частично опускается до сетевого уровня модели OSI

Уровень 3: потоки данных (TCP)

Протокол TCP четко соответствует транспортному уровню модели OSI. Он обеспечивает все необходимое для организации сеанса: гарантированную доставку сообщений, потоковую передачу данных, правильный порядок пакетов и обработку ошибок. Следует подчеркнуть, что протокол TCP не основан на UDP. У TCP-пакета свой заголовок и канал распространения. В табл. 5.3 дается сравнение протокола TCP и транспортного уровня модели OSI.

Таблица 5.3. Сравнительные характеристики транспортного уровня и протокола TCP

Транспортный уровень	TCP
Надежные данные	Надежные данные (контрольные суммы)
Надежная доставка	Гарантированная доставка
Согласуемый размер окна	"Раздвижное" окно
Ориентирован на записи	Ориентирован на потоки
Основан на сетевом уровне	Основан на протоколе IP

Уровень 4: прикладной

На прикладном уровне модель TCP/IP заканчивается. Он охватывает сеансовый, представительский и прикладной уровни модели OSI. На данном уровне работают Web-броузеры, шлюзы, Telnet и FTP (File Transfer Protocol — протокол передачи файлов). Технология RPC соответствует представительскому уровню OSI. Сетевая файловая система (NFS) основана на RPC и находится на прикладном уровне OSI.

Программы прикладного уровня работают с протоколами UDP и TCP, но могут принимать сообщения непосредственно от протокола ICMP.

Фундаментальные различия между моделями OSI и IP

В моделях OSI и IP весь стек протоколов распределен по уровням. Обе они ориентированы на то, чтобы не писать машинно-зависимый код, а создавать абстрагированные приложения. Тем не менее между ними можно заметить некоторые отличия.

В OSI по уровням распределены не только протоколы, но и данные. На каждом уровне к пакету добавляется свой заголовок. По мере того как данные продвигаются вниз по стеку протоколов к физическому уровню, сообщение последовательно инкапсулируется в пакет все более низкого уровня.

Например, когда посылается сообщение сеансового уровня, на физическом уровне можно будет наблюдать заголовки в таком порядке: канальный, сетевой, транспортный и сеансовый. Вслед за ними в конце блока данных располагается собственно сообщение. Процесс инкапсуляции повторяется на каждом уровне, вплоть до физического. Таким образом, у одного пакета может быть семь заголовков.

На принимающей стороне происходит обратный процесс: данные последовательно извлекаются, чтобы определить, следует ли передать сообщение на следующий уровень. В предыдущем примере сообщение поступает на физическом уровне. На каждом уровне удаляется соответствующий заголовок. Если за ним обнаруживается еще один заголовок, сообщение передается вверх по стеку. В конце концов, сообщение попадет на сеансовый уровень.

В модели IP все происходит по-другому. Если на каждом уровне добавлять заголовок, сообщение может стать очень большим, в результате чего снизится пропускная способность. Вместо этого тип протокола указывается в отдельном поле заголовка IP-пакета. Когда IP-подсистема принимает сообщение, она проверяет данное поле и направляет сообщение непосредственно указанному протоколу, предварительно удалив свой собственный заголовок.

Кроме того, как рассказывалось в конце главы 3, "Различные типы Internet-пакетов", каждый протокол из стека TCP/IP играет свою конкретную роль. В отличие от модели OSI, где все протоколы последовательно располагаются друг на друге, в Internet все они основаны на одном протоколе IP. Протокол ICMP предназначен для обработки ошибок. Протокол UDP используется для направленной отправки сообщений без установления соединения. Протокол TCP ориентирован

на потоковую передачу данных. Сам протокол IP предназначен для разработки новых протоколов.

Что чему служит

Теперь необходимо разобраться, как использовать сетевую модель Internet. Чтобы получить доступ к различным уровням стека IP, следует вызвать функцию socket() (табл. 5.4).

Таблица 5.4. Доступ к протоколам семейства Internet

Уровень стека TCP/IP	Программный/пользовательский доступ
4 — прикладной	FTP, Gopher, Lynx, IRC
3 - межузловой (TCP)	socket(PF_INET, SOCK_STREAM, 0);
3 — межузловой (UDP)	socket(PF_INET, SOCK_DGRAM, 0)-,
2 — межсетевой (ICMP)	socket(PF_INET, SOCK_RAW, IPPROTO_ICMP);
2 — межузловой (IP)	socket(PF_INET, SOCK_RAW, *протокол);*
1— доступ к сети	socket(PF_INET, SOCK_PACKET, фильтр);

Linux разрешает доступ для чтения к низкоуровневым сообщениям драйверов с помощью функции socket() с аргументом SOCK_PACKET. Благодаря этому можно перехватывать все сообщения, передаваемые по сети. Данная возможность рассматривалась нами при построении сетевого анализатора в главе 3, "Различные типы Internet-пакетов".

Резюме: от теории к практике

В данной главе анализировалась взаимосвязь между различными элементами сети. Рассматривались вопросы аппаратного взаимодействия, подключения к сети, связи с операционной системой и межзадачного общения. В каждой из этих областей существуют свои проблемы, которые решаются с помощью тщательно разработанных сетевых моделей, предназначенных для упрощения сетевого программирования.

В обеих сетевых моделях, OSI и IP, круг решаемых задач разбит на уровни. Все уровни, или слои, основаны друг на друге, подобно прослойкам торта. Первый уровень отвечает за организацию физической связи между компьютерами. С каждым следующим уровнем повышается надежность данных, но снижается скорость взаимодействия.

Часть II

Создание серверных приложений

В этой части...

Глава 6

Пример сервера

В этой главе...

В сетевом соединении всегда есть отправитель и получатель. В общем случае отправителем является клиент, который запрашивает сервис, предоставляемый сетевым компьютером. В части I, "Создание сетевых клиентских приложений", рассматривались основы клиентского программирования: как подключить клиента к серверу, как организовать прямую доставку сообщений без установления соединения и как работать с протоколами стека TCP/IP. С этой главы начинается знакомство с другой стороной соединения — приемником, или сервером.

Чтобы понять схему взаимодействия клиента и сервера, представьте, что сеть — это телефонная система большой компании, в которой сервер является центральным телефонным номером, направляющим звонки конкретным служащим. Клиент связывается с требуемым служащим, набирая центральный и дополнительный номера. Теперь ситуация проясняется. Центральный номер является адресом сетевого узла, а дополнительный номер — это порт конкретного сервиса.

Клиент должен знать номер порта, по которому обращается. Это похоже на телефонный номер, который должен быть где-то опубликован: если клиент не знает номер, он не сможет по нему позвонить.

Если в части I речь шла о том, как запрашивать сервисы, то теперь мы остановимся на том, как предоставлять их. В главе шаг за шагом рассматривается процесс создания сервера. В конце главы приводится пример небольшого HTTP-сервера, в котором демонстрируется, как связываться с Web-клиентом и упаковывать HTML-сообщения.

Схема работы сокета: общий алгоритм сервера

Процесс построения сервера всегда начинается с создания сокета. Подобно тому как в клиентской программе требуется определенная последовательность системных вызовов, аналогичная последовательность необходима и на сервере, только здесь она длиннее. Если некоторые функции клиенту вызывать не обязательно, то для серверного приложения все они нужны (рис. 6.1).

Клиентская программа, которую мы писали в первых главах, вызывала функции в такой последовательности: socket(), connect(), read(), write() и close(). Системный вызов bind() был необязательным, так как эту функцию вызывала операционная система. Номер порта не требовался, поскольку программа обращалась напрямую к серверу. Клиент всегда создает *активное соединение,* потому что он постоянно его занимает.

С другой стороны, серверные программы должны предоставлять своим клиентам неизменные, четко заданные номера портов. Базовая последовательность вызовов здесь будет такой: socket(), bind(), listen(), accept() и close(). В то время как клиент создает активное соединение, серверное соединение пассивно. Функции listen() и accept() устанавливают соединение только тогда, когда приходит запрос от клиента.

Знакомство с функцией bind() состоялось в главе 4, "Передача сообщений между одноранговыми компьютерами". В этой главе она будет описана более формально. Кроме того, будут представлены две новые функции: listen() и accept().

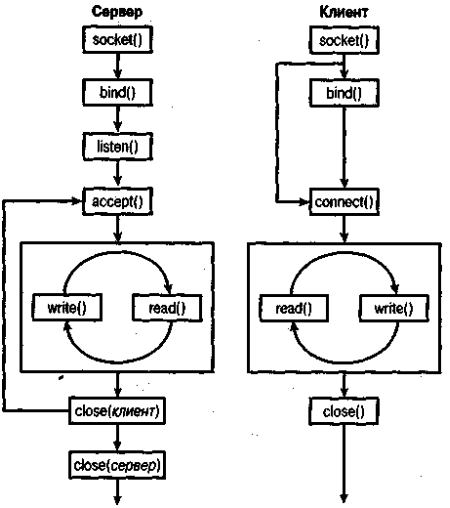

Рис. 6.1. Алгоритмы построения клиента и сервера сходны, но схема подключения к сети в них разная

Простой эхо-сервер

Прежде чем перейти к рассмотрению системных функций, следует рассказать о том, какого рода сервер мы будем создавать. В качестве образца был выбран стандартный эхо-сервер. Это основа основ серверного программирования, подобно приложению "Hello, World" в программировании на языке C. Полный текст примера находится на Web-узле в файле simple-server.c.

Большинство соединений можно проверить, послав данные и запросив их назад в неизменном виде (эхо). Это хорошая идея для создания простейшего сервера. Аналогичным образом пишутся и отлаживаются даже самые сложные приложения.

Парадигма построения и отладки

В сетевом программировании приходится очень много заниматься тестированием и отладкой. Это столь сложная область, что с целью минимизации ошибок следует придерживаться простейших подходов к построению приложений. Парадигма построения и отладки (одна из составных частей методологии ускоренной разработки программ) предписывает сконцентрироваться на решении конкретной проблемы. Когда она будет решена, полученный программный модуль станет строительным блоком для остальной части приложения.

В общем случае в серверной программе требуется в определенной последовательности вызвать ряд системных функций. На примере эхо-сервера можно наглядно увидеть эту последовательность, не отвлекаясь на решение других, более специфических задач. Ниже описан общий алгоритм работы эхо-сервера.

1. Создание сокета с помощью функции socket().
2. Привязка к порту с помощью функции bind().
3. Перевод сокета в режим прослушивания с помощью функции listen().
4. Проверка подключения с помощью функции accept().
5. Чтение сообщения с помощью функции recv() или read().
6. Возврат сообщения клиенту с помощью функции send() или write().
7. Если полученное сообщение не является строкой "bye", возврат к п. 5.
8. Разрыв соединения с помощью функции close() или shutdown().
9. Возврат к п. 4.

В приведенном алгоритме четко видны отличия от протокола UDP и других протоколов, не ориентированных на установление соединений. Здесь сервер не закрывает соединение до тех пор, пока клиент не пришлет команду bye.

Благодаря алгоритму становится понятно, что необходимо предпринять дальше при создании сервера. Первый очевидный шаг (создание сокета) рассматривался в главе 1, "Простейший сетевой клиент". Как уже упоминалось, с этого начинается любая сетевая профамма. Следующий шаг — выбор порта — обязателен для сервера.

Привязка порта к сокету

Работа с TCP-сокетами начинается с вызова функции socket)), которой передается константа SOCK_STREAM. Но теперь требуется задать также номер порта, чтобы клиент мог к нему подключиться.

Функция bind() спрашивает у операционной системы, может ли профамма завладеть портом с указанным номером. Если сервер не указывает порт, система назначает ему ближайший доступный порт из пула номеров. Этот номер может быть разным при каждом следующем запуске профаммы.

Если профамма запрашивает порт, но не получает его, значит, сервер уже выполняется. Операционная система связывает порт только с одним процессом.

Объявление функции bind)) выглядит так:

```
tinclude <sys/socket.h>
linclude <resolv.h>
int bind(int sd, struct sockaddr *addr, int addr_size);
```

Параметр sd является дескриптором ранее созданного сокета. В параметре addr передается структура семейства sockaddr. В ней указывается семейство протоколов, адрес сервера и номер порта (см. главу 1, "Простейший сетевой клиент"). Последний параметр содержит размер структуры sockaddr. Его необходимо задавать, потому что такова концепция библиотеки Socket API: один интерфейс, но много архитектур. Операционная система поддерживает множество протоколов, у каждого из которых своя адресная структура.

Перед вызовом функции bind() необходимо заполнить поля структуры sockaddr (листинг 6.1).

Листинг 6.1. Вызов функции bind() в TCP-сервере

```
/***        Пример TCP-сокета: заполнение структуры          ***/
/***        sockaddr_in                                      ***/

struct sockaddr_in addr;                  /* создаем TCP-сокет */
bzero(&addr,  sizeof(addr));             /* обнуляем структуру */
addr.sin_family = AF_INET;             /* выбираем стек TCP/IP */
addr.sin_port = htons(MY_PORT);          /* задаем номер порта */
addr.sin_addr.s_addr = INADDR_ANY;          /* любой IP-адрес */
if ( bind(sd, saddr, sizeof(addr)) != 0 ) /* запрашиваем порт */
    perror("Bind  AF_INET");
```

В следующем фрагменте программы (листинг 6.2) осуществляется инициализация именованного сокета (семейство AF_UNIX или AFJGOCAL).

Листинг 6.2. Вызов функции bind() в локальном сервере

```
/***    Пример локального сокета: заполнение структуры        ***/
/***    sockaddr_ux                                           ***/

#include <linux/un.h>
struct sockaddr_ux addr; /* создаем локальный именованный сокет */
bzero(Saddr,  sizeof(addr));                  /* обнуляем структуру */
addr.sun_family = AF_LOCAL;        /* выбираем именованные сокеты */
strcpy(addr.sun_path,   "/tmp/mysocket");          /* выбираем имя */
if ( bind(sd, saddr, sizeof(addr)) != 0 )    /* привязка к файлу */
    perror("Bind   AF_LOCAL");
```

Если запустить на выполнение эту программу, то после ее завершения в каталоге /tmp появится файл mysocket. Именованные сокеты используются системным демоном регистрации сообщений, syslogd, для сбора информации: системные процессы устанавливают соединение с сокетом демона и посылают в него сообщения.

В результате выполнения функции bind() могут возникнуть перечисленные ниже ошибки.

- EBADF. Указан неверный дескриптор сокета. Эта ошибка возникает, если вызов функции socket() завершился неуспешно, а программа не проверила код ее завершения.
- EACCES. Запрашиваемый номер порта доступен только пользователю root. Помните, что для доступа к портам с номерами 0—1023 программа должна иметь привилегии пользователя root. Подробнее об этом рассказывалось в главе 2, "Основы TCP/IP".
- EINVAL. Порт уже используется. Возможно, им завладела другая программа. Эта ошибка может также возникнуть, если сервер завис и вы тут же запускаете его повторно. Для операционной системы требуется время, чтобы освободить занятый порт (до пяти минут!).

Функция bind() пытается зарезервировать для серверного сокета указанное имя файла или порт (список доступных или стандартных портов содержится в файле /etc/services). Клиенты подключаются к данному порту, посылая и принимая через него данные.

Создание очереди ожидания

Сокет обеспечивает интерфейс, посредством которого одна программа может взаимодействовать с другой по сети. Соединение является эксклюзивным: после того как программа подключилась к порту, никакая другая программа не может к нему обратиться. Для разрешения подобной ситуации на сервере создается очередь ожидания.

Очередь сокета активизируется при вызове функции listen(). Когда сервер вызывает эту функцию, он указывает число позиций в очереди. Кроме того, сокет переводится в режим "только прослушивание". Это очень важно, так как позволяет впоследствии вызывать функцию accept().

```
#include <sys/socket.h>
linclude <resolv.h>
int listen(int sd, int numslots);
```

Параметр sd является дескриптором сокета, полученным в результате вызова функции socket(). Параметр numslots задает число позиций в очереди ожидания. Приведем пример (листинг 6.3).

Листинг 6.3. Пример функции listen()

```
/*************************************************************/
/***    Пример функции listen(): перевод сокета            ***/
/***    в режим прослушивания клиентских подключений       ***/
y*************************************************************/
int sd;
sd = socket(PF_INET, SOCK_STREAM, 0);
/*** Привязка к порту ***/
if ( listen(sd, 20) != 0 )              /* перевод сокета в режим */
    perror("Listen"); /* прослушивания очереди с 20-ю позициями */
```

Как правило, размер очереди устанавливается равным от 5 до 20. Больший размер оказывается избыточным в современной многозадачной среде. Если многозадачный режим не поддерживается, может потребоваться увеличить размер очереди до величины периода тайм-аута (например, 60, если тайм-аут составляет 60 секунд).

Функция listen() может генерировать следующие ошибки.

- EBADF. Указан неверный дескриптор сокета.
- EOPNOTSUPP. Протокол сокета не поддерживает функцию listen(). В TCP (SOCK_STREAM) очередь ожидания поддерживается, а в протоколе UDP (SOCK_DGRAM) – нет.

После перевода сокета в режим ожидания необходимо организовать цикл получения запросов на подключение.

Прием запросов от клиентов

На данный момент программа создала сокет, назначила ему номер порта и организовала очередь ожидания. Теперь она может принимать запросы на подключение. Функция accept() делает указанный сокет диспетчером соединений. Здесь привычный ход событий нарушается. Когда сокет переводится в режим прослушивания, он перестает быть двунаправленным каналом передачи данных. Программа не может даже читать данные из него. Она может только принимать запросы на подключение. Функция accept() *блокирует* программу до тех пор, пока не поступит такой запрос.

Когда клиент устанавливает соединение с сервером, сокет, находящийся в режиме прослушивания, организует новый двунаправленный канал между клиентом и своим собственным портом. Функция accept)) неявно создает в программе новый дескриптор сокета. По сути, при каждом новом подключении создается выделенный канал между клиентом и сервером. С этого момента программа взаимодействует с клиентом через новый канал.

Можно также узнать, кто устанавливает соединение с сервером, поскольку в функцию accept() передается информация о клиенте. Аналогичный процесс рассматривался в главе 4, "Передача сообщений между одноранговыми компьютерами", когда функция recvfrom() получала не только данные, но и указатель на адрес отправителя.

```
#include <sys/socket.h>
#include <resolv.h>
int accept(int sd, sockaddr *addr, int *addr_size);
```

Как всегда, параметр sd является дескриптором сокета. Во втором параметре возвращается адрес клиента и номер порта, а в третьем — размер структуры sockaddr. В отличие от функции recvfrom(), последние два параметра являются необязательными. Если в программе не требуется знать адрес клиента, задайте эти параметры равными нулю.

Необходимо убедиться, что размер буфера адреса достаточен для размещения в нем полученной адресной структуры. Беспокоиться о повреждении данных из-за переполнения буфера не стоит: функция задействует ровно столько байтов, сколько указано в третьем параметре. Параметр addr_size передается по ссылке, поэтому программа может легко узнать реальный размер полученной структуры (листинг 6.4).

Листинг 6.4. Пример функции accept()

```
/*****************************************************************/
/***   Пример функции accept(): ожидание и принятие запросов  ***/
/***   на подключение от клиентов                              ***/
/*****************************************************************/

int sd;
struct sockaddr_in addr;
/*** Создание сокета, привязка его к порту и
     перевод в режим прослушивания ***/
for (;;)                          /* цикл повторяется бесконечно */
{  int clientsd;                      /* новый дескриптор сокета */
   int size = sizeof(addr);      /* вычисление размера структуры */
```

```
clientsd = accept(sd, &addr, &size); /* ожидание подключения */
if ( clientsd > 0 )                          /* ошибок нет */
{
   /*** Взаимодействие с клиентом ***/
   close(clientsd);                /* очистка и отключение */
}
else                                    /* произошла ошибка */
   perror("Accept");
```

Взаимодействие с клиентом

Обратите внимание на то, что в приведенном выше фрагменте программы закрывался дескриптор clientsd, который отличается от основного дескриптора сокета. Это очень важный момент, поскольку для каждого соединения создается отдельный дескриптор. Если забыть их закрыть, лимит дескрипторов может со временем исчерпаться.

Повторное использование адресной структуры

В функции accept() можно использовать адресную структуру, инициализированную еще при вызове функции bind( ). По завершении функции bind( ) хранящаяся в этой структуре информация больше не нужна серверу.

Помните, что большинство полей структуры имеет сетевой порядок следования байтов. Извлечь адрес и номер порта из переменной addr можно с помощью функций преобразования (листинг 6.5).

Листинг 6.5. Пример функции accept( ) с регистрацией подключений

```
/*****************************************************************/
/***   Расширенный пример функции accept(): информация        ***/
/***   о каждом новом подключении отображается на экране      ***/
/*****************************************************************/
/*** (Внутри цикла) ***/

client = accept(sd, saddr, &size);
if ( client > 0 )
{
   if ( addr.sin_family == AF_INET)
      printf("Connection[%s]: %s:%d\n",          /* регистрация */
             ctime(time(0)),                     /* метка времени */
             ntoa(addr.sin_addr), ntohs(addr.sin_port));
   /*— взаимодействие с клиентом —*/
```

Если в процессе выполнения функции accept() происходит ошибка, функция возвращает отрицательное значение. В противном случае создается новый дескриптор сокета. Ниже перечислены коды возможных ошибок.

- EBADF. Указан неверный дескриптор сокета.
- EOPNOTSUPP. При вызове функции accept() сокет должен иметь тип SOCK_STREAM.
- EAGAIN. Сокет находится в режиме неблокируемого ввода-вывода, а очередь ожидания пуста. Функция accept() блокирует работу программы, если не включен данный режим.

Настало время вернуться к эхо-серверу, который возвращает клиенту полученное сообщение до тех пор, пока не поступит команда bye (листинг 6.6).

Листинг 6.6. Пример эхо-сервера

```
/***      Пример эхо-сервера: возврат полученного сообщения      ***/
/***  до  тех  пор  пока  не  поступит  команда "bye<ret>"       ***/

/*** (Внутри цикла после функции accept()) ***/
...
if ( client > 0 )
{  char buffer[1024];
   int  nbytes;

   do
   {
      nbytes - recv(client, buffer, sizeof(buffer), 0);
      if ( nbytes > 0 )  /* если получены данные, возвращаем их */
          send(client, buffer, nbytes, 0);
   }
   while ( nbytes > 0 && strncmp("bye\r", buffer, 4) 1=0);
   close(client);
}
```

Заметьте, что признаком окончания сеанса является строка "bye\r", а не "bye\n". В общем случае это зависит от того, как выполняется обработка входного потока. Из соображений надежности следует проверять оба случая. Попробуйте протестировать данную программу, использовав в качестве клиента утилиту Telnet.

Общие правила определения протоколов

Взаимодействуя с другими компьютерами, программа должна следовать определенным правилам общения. Два основных вопроса, на которые необходимо ответить: "Кто начинает первым?" и "Когда мы закончим?"

Придерживаясь описываемых правил, клиент и сервер могут быть уверены в том, что они не начинают передачу одновременно и не ждут бесцельно друг друга.

Какая программа должна начинать передачу первой?

БОЛЬШИНСТВО серверов первыми начинают сеанс. Но в некоторых системах с высоким уровнем безопасности предполагается, что клиент должен отправить первое сообщение. Сервер может заставить клиента идентифицировать себя (указать не только адрес узла и порт).

Следует избегать ненужного взаимодействия. Если сервер начинает первым, он, как правило, выдает одну и ту же информацию при каждом подключении. Нужно ли это клиенту? Не замедлит ли это работу?

Какая программа должна управлять диалогом?

Чаше всего диалогом с сервером управляют клиенты. Клиент подключается к серверу и посылает запросы. Сервер, в свою очередь, обрабатывает запросы и выдает ответ.

Но иногда необходимо меняться ролями. Например, клиент запрашивает информацию из базы данных сервера. После того как данные были переданы, другой клиент обновляет часть полей, с которыми работает первый клиент. Если первый клиент принимает на основании имеющейся информации какие-то решения, они могут быть неправильными. В такой ситуации сервер должен самостоятельно послать клиенту обновления, а клиент должен их принять.

Какой уровень сертификации требуется?

Пи создании высоконадежных систем важно знать, с кем общается сервер. Это означает, что сервер должен определить или сертифицировать пользователя или, по крайней мере, его компьютер.

Процесс сертификации включает передачу имени пользователя и пароля. Кроме того, может потребоваться наличие цифрового сертификата. В главе 16, "Безопасность сетевых приложений", описывается протокол SSL (Secure Sockets Layer — протокол защищенных сокетов), а также рассматриваются вопросы безопасности.

С другой стороны, сертификация не всегда нужна, а вместо этого необходимо регистрироваться в системе. Как часто пользователи посещают сервер? Требуется ли настраивать работу сервера в соответствии с предпочтениями отдельных пользователей? Как незаметно собрать информацию о посетителе? Ответы на эти вопросы важны при создании серверных приложений, особенно Web-серверов.

Какой тип данных используется?

БОЛЬШИНСТВО серверов использует кодировку ASCII, а большинство Web-страниц представлено в формате текст/HTML. Полезно задать себе вопросы: "Является ли это наиболее эффективной формой представления данных?" и "Поддерживает ли клиент сжатие данных?" Подумайте, как можно уменьшить задержки на сервере, в сети и в клиентской системе.

Сжатие данных имеет существенные преимущества. Компрессированные ASCII-потоки уменьшаются в размере на 50-80%.

Как следует обрабатывать двоичные данные?

Передавать двоичные данные — особенно в сжатом виде — намного эффективнее, чем работать с ASCII-текстом. Но существует одна проблема: некоторые сети поддерживают только 7-битовую кодировку байта. Такие сети являются пережитками прошлого, но их все еще слишком дорого демонтировать. К счастью, маршрутизаторы, подключенные к этим сетям, выявляют подобную несовместимость и автоматически преобразуют данные в том или ином направлении. Это требует дополнительного времени и замедляет продвижение пакетов. Дополнительная нагрузка ложится также на узел-получатель, так как он должен восстанавливать данные.

Случается ли так, что программа начинает передачу данных в текстовом виде, а затем переключается в двоичный режим? В этом случае необходимо, чтобы клиент или сервер посылал соответствующее уведомление.

Как обнаружить взаимоблокировку?

Бывает, что клиент и сервер ждут друг друга (это называется *взаимоблокировкой,* или тупиковой ситуацией). При этом бесцельно расходуются ценные ресурсы и испытывается терпение пользователей. Когда возникает тупик, единственное решение заключается в том, чтобы разорвать соединение и подключиться заново, смирившись с возможной потерей данных.

Как клиент, так и сервер могут *зависать,* входя в режим бесконечного ожидания ресурсов. Зависание обычно происходит, когда клиент или сервер выполняет какие-то другие действия помимо передачи данных. Как и при взаимоблокировке, стандартным решением является разрыв соединения по тайм-ауту и повторное подключение.

Необходима ли синхронизация по таймеру?

Синхронизация необходима, когда выполняются действия, имеющие привязку по времени, например финансовые транзакции. Это сложная задача, требующая координации таймеров на разных компьютерах. В первую очередь необходимо выяснить, насколько точной должна быть синхронизация.

Сервер должен инициализировать свой таймер в соответствии с таймером клиента. Но поскольку большинство клиентов не синхронизируют свои таймеры по сетевому времени, требуется помощь третейского судьи (сервера времени). При этом возникают дополнительные проблемы (задержки в получении синхросигналов по сети).

Одно из решений проблемы синхронизации заключается в том, чтобы проводить все транзакции на сервере (максимально снимая ответственность с клиента). Когда сервер завершает транзакцию, он сообщает клиенту дату и время ее окончания.

Как и когда переустанавливать соединение?

Иногда в процессе взаимодействия клиенту и серверу может потребоваться начать передачу заново без разрыва соединения. Разрыв соединения может означать существенную потерю данных, поэтому он неприемлем.

В TCP/IP существует понятие приоритетного сообщения, с помощью которого можно просигнализировать об отмене. Подробная информация о приоритетных сообщениях и передаче внеполосных данных приводится в главе 9, "Повышение производительности". Но отправка приоритетного сообщения — это только полдела: как сервер, так и клиент должны вернуться к некой начальной точке, что представляет собой серьезную проблему в структурном программировании.

Повторное открытие соединения позволяет начать сеанс сначала. Посредством приоритетного сообщения клиент или сервер уведомляется о том, что необходимо закрыть соединение. Затем соединение снова открывается, при этом часть информации теряется и происходит откат к предыдущему состоянию.

Когда завершать работу?

Итак, между клиентом и сервером установлено соединение и передаются пакеты. Пришло время прощаться. Определить конец сеанса может быть не так просто, как кажется. Например, при взаимодействии с HTTP-сервером сеанс завершается в момент получения двух символов новой строки. Но иногда запроса от клиента можно ждать бесконечно, если чтение данных осуществляется с помощью функции read() или recv(), а буфер имеет недостаточный размер. В этом случае сервер превысит время ожидания и объявит о разрыве соединения.

Кроме того, бывает трудно определить, какая программа должна прервать соединение первой. Клиент получит сообщение о разрыве канала (EPIPE), если сервер закроет соединение до того, как клиент закончит передачу данных.

Более сложный пример: сервер HTTP

Эхо-сервер представляет собой отличный отправной пункт для создания различных видов серверов. Одним из них является HTTP-сервер. Полная его реализация выходит за рамки данной книги, но можно создать уменьшенный вариант сервера, который отвечает на запросы любого броузера. Текст этого примера имеется на Web-узле (файл html-ls-server.c).

Сервер генерирует HTML-код динамически, а не загружает его из файла. Это упрощает программу (листинг 6.7).

Листинг 6.7. Пример простого HIP-сервера

```
/*************************************************************/
/***                    Простой HTTP-сервер                ***/
/*************************************************************/

while(1)
{  int client;
   int size = sizeof(addr);

   client = accept(sd, &addr, &size);
   if ( client > 0 )
   {  char buffer[1024];
      /* Сообщение клиенту */
      char  *reply  =  "<html><body>Hello!</body></html>/n";
```

```
        bzero(buffer, sizeof(buffer) );          /* очистка буфера */
        recv(client, buffer, sizeof(buffer), 0);   /* получение
                                                      сообщения */
        send(client, reply, strlen(reply), 0); /* ответ клиенту*/

        /*— отображение клиентского сообщения —*/
        fprintf(stderr, "%s", buffer);
        close(client);
    }
    else
        perror("Accept");
```

Каждый раз, когда клиент подключается, он посылает запрос примерно такого вида:

```
GET  /dir/document  HTTP/1.0
(определение протокола)
```

Первая строка представляет собой запрос. Все последующие сообщения информируют сервер о том, какого рода данные готов принимать клиент. В первую строку может входить конфигурационная информация, позволяющая серверу определить, как следует взаимодействовать с клиентом. Метод GET принимает два параметра: собственно запрос и используемый протокол. Сервер может анализировать запрос, выделяя имя каталога и имя документа (естественно, запросы бывают гораздо более сложными). Протокол HTTP 1.0 допускает наличие пробелов в путевом имени, поэтому запрос включает в себя все, что находится от начального символа косой черты до строки HTTP/.

При ответе сервер может дополнительно передавать MIME-заголовок, указывая на статус сообщения и тип возвращаемого документа:

```
HTTP/1.1 200 OK
Content-Type: text/html

(пустая строка)
(пустая строка)
<html>
<head>
```

Первая строка является статусной. Она информирует клиента о том, насколько успешно выполнен запрос. Именно здесь передается печально известное сообщение об ошибке 404 ("Not Found"). Полный список кодов завершения HTTP 1.1 представлен в приложении А, "Информационные таблицы".

В действительности этот этап можно пропустить, поскольку по умолчанию клиент ожидает поступления HTML-документа. Таким образом, достаточно просто послать сгенерированный HTML-код.

При написании HTTP-сервера следует учитывать ряд моментов. В частности, сервер не знает заранее, насколько большим получится результирующий документ, поэтому дескриптор сокета необходимо привести к типу FILE* (листинг 6.8).

Листинг 6.8. Расширенный алгоритм HTTP-сервера

```
/*************************************************************/
/***  Пример сервера HTTP 1.0: устанавливаем соединение,  ***/
/***   принимаем запрос, открываем каталог и создаем      ***/
/***   для него список файлов в формате HTML              ***/
/*************************************************************/

/*** Создание сокета, привязка его к порту
     и перевод в режим прослушивания ***/
for(;;)
{  int client;
   int size = sizeof(addr);

   client = accept(sd, &addr, &size);      /* ожидание запроса
                                              на подключение */
   if ( client > 0 )
   {  char buf[1024];
      FILE *clientfp;

      bzero(buf, sizeof(buf));              /* очистка буфера */
      recv(client, buf, sizeof(buf), 0);    /* получение
                                               сообщения */
      clientfp = fdopen(client, "w");  /* приведение к типу
                                          FILE* */
      if ( clientfp != NULL )           /* если преобразование
                                           прошло успешно */
      {
          /**** Извлекаем путевое имя из сообщения ****/
          /**** открываем каталог ****/
          /**** для каждого файла... ****/
          /****   Читаем имя файла ****/
          /****   Генерируем HTML-таблицу ****/
          fclose(clientfp);    /* закрываем указатель на файл */
      }
      else
          perror("Client FILE");    /* приведение к типу FILE*
                                       невозможно */
       close(client);             /* закрываем клиентский сокет */
   }
   else
      perror ("Accept") ;          /* ошибка в функции accept () */
```

Эту программу можно улучшить, сортируя список файлов по именам, распознавая тип каждого файла, добавляя коды ошибок HTTP 1.1 и т.д.

Резюме: базовые компоненты сервера

В этой главе рассматривались основы создания серверных приложений. Серверы позволяют централизованно управлять данными и операциями, обслуживать множество клиентов и распределять нагрузку между ними.

В серверных приложениях используются три новые функции: bindf), listen() и accept() — помимо тех функций, которые обычно вызываются клиентами. С помощью этих функций осуществляется выбор номера порта (bind()), создание очереди подключений (listen()) и прием запросов на подключение (accept()). Функция accept() создает новый сокет для каждого соединения, позволяя программе обслуживать несколько соединений по одному порту.

При создании сервера необходимо учитывать, как осуществляется взаимодействие клиента и сервера и как должна вести себя каждая из сторон. Пошаговый анализ используемого протокола позволит определить, как наилучшим образом обрабатывать каждый запрос.

Распределениенагрузки многозадачность

Глава 7

В этой главе...

Представим себе, что мы одновременно выполняем множество различных задач. Работая в параллельном режиме, мы сосредоточиваем внимание на каждой конкретной задаче, не отвлекаясь на остальные. В действительности наш мозг способен делать это. Например, можно мыть пол, одновременно продумывая сложный программный.алгоритм, или решать кроссворд, слушая музыку. В программировании это называется *многозадачностью.*

На данный момент мы написали ряд клиентских приложений, подключающихся к серверам, рассмотрели алгоритм обмена информацией между одноранговыми компьютерами и создали несколько простых серверов. Но во всех этих случаях каждая программа одновременно выполняла только одно действие. Сервер мог взаимодействовать только с одним клиентом, а клиент, устанавливающий соединение, вынужден был ждать, пока сервер обслужит текущего клиента. Как было бы здорово принимать несколько запросов на подключение одновременно! А как насчет того, чтобы подключаться сразу к нескольким серверам?

Многозадачность — это очень мощная методика, позволяющая существенно упростить программирование, если только вы способны разделить общий алгоритм на несколько одновременно выполняющихся модулей, каждый со своими обязанностями. Без тщательного планирования попытка реализовать многозадачность приведет к написанию трудноуправляемой и громоздкой программы.

В этой главе рассматриваются вопросы программирования процессов и отдельных потоков, рассказывается, когда и как их следует использовать, в чем отличие между ними, каковы их сильные и слабые стороны. Кроме того, приводится информация по обработке сигналов и механизмам блокировки.

Как можно догадаться, представленный материал достаточно обширен. Существуют целые книги, написанные по данной теме. Мы же коснемся только вопросов, связанных с сетевым программированием.

Для ясности термин *задание* употребляется в данной главе по отношению к любому исполняемому системному объекту. Термины *процесс* и *поток* обозначают конкретные разновидности заданий.

Понятие о многозадачности: процессы и потоки

Многозадачность — это одна из важнейших особенностей систем Linux и UNIX. Она позволяет выделять каждой программе свою долю процессорного времени *(квантование времени)* и других системных ресурсов. Программы могут работать намного эффективнее, если они написаны с учетом многозадачности.

Задания представляют собой отдельные исполняемые модули в системе. Каждая выполняемая команда является заданием. В общей концепции многозадачности выделяются два основных понятия: процессы и потоки (или *облегченные процессы).* Они определяют два различных способа совместного использования данных. Чтобы понять суть многозадачности, необходимо разобраться в том, как операционная система отделяет одно задание от другого.

Каждое задание хранит свою информацию в нескольких разделах памяти *(страницах).* Операционная система назначает заданию *таблицу страниц* — набор страниц, каждая из которых выполняет отдельную функцию. Работа со страницами осуществляется через подсистему виртуальной памяти, которая реализуется в

виде таблицы, преобразующей программные адреса в физические. Когда операционная система начинает переключать задания, она сохраняет информацию о текущем задании — *контекст* — и загружает в виртуальную память таблицу страниц следующего задания.

Назначение виртуальной памяти

Таблица страниц виртуальной памяти содержит информацию не только о преобразовании адресов. В ней есть также ссылки на атрибуты прав доступа (чтение/запись/выполнение). Кроме того, операционная система помечает те страницы, которые выгружены во внешнее хранилище. Когда программа обращается к такой странице, диспетчер памяти генерирует ошибку страницы (сигнализирует об отсутствии страницы в памяти). Ее перехватывает обработчик страничных ошибок, который загружает недостающую страницу с диска в ОЗУ.

Задания могут совместно использовать различные страницы, но это зависит от типа задания. Для процессов в Linux применяется алгоритм *копирования при записи.* Процессы не допускают совместного доступа к одинаковым областям памяти, поэтому при запуске процесса вся память, к которой он обращается, должна быть скопирована на диск. В действительности же копируются только страницы, модифицируемые родительским или дочерним процессом. Такая методика позволяет существенно уменьшить время, требуемое для создания процесса, что увеличивает производительность рабочей станции. Своей высокой производительностью система Linux во многом обязана именно отложенному копированию страниц памяти.

У процессов и потоков свои задачи, которые редко пересекаются. Например, процесс создается для запуска внешней программы и получения от нее информации. С помощью отдельного потока можно отображать графический файл по мере его загрузки. Таким образом, выбор между процессом и потоком делается на основании простого правила: если необходим совместный доступ к данным, используйте поток.

На рис. 7.1 представлено, какие компоненты задания допускают совместное использование. В любом случае совместный доступ к стеку и контексту задания запрещен. В текущей реализации библиотеки потоковых функций потоку разрешается использовать все, кроме идентификатора процесса (PID — process ID).

При переключении заданий операционная система заменяет текущую таблицу страниц таблицей активизируемого задания. Это может потребовать нескольких циклов работы процессора. Обычно переключение занимает от 1 мкс до 0,1 мс, в зависимости от процессора и тактовой частоты. Задержка бывает достаточно большой, особенно если переключение осуществляется 100 раз в секунду (каждые 10 мс). Любое задание занимает долю процессорного времени, часть которого отводится на собственно переключение задания.

В некоторых версиях UNIX потоки выполняются быстрее, потому что диспетчер заданий должен выгружать меньшее число записей. В версиях ядра Linux 2.0 и 2.2 скорость переключения заданий почти такая же. Сходство возникает из-за четко отлаженного алгоритма переключения.

В Linux также поддерживается симметричная мультипроцессорная обработка. Если программа написана с учетом многозадачности, то в мультипроцессорной системе она получает дополнительное ускорение. (На момент написания книги в Linux могло одновременно поддерживаться максимум 16 процессоров.)

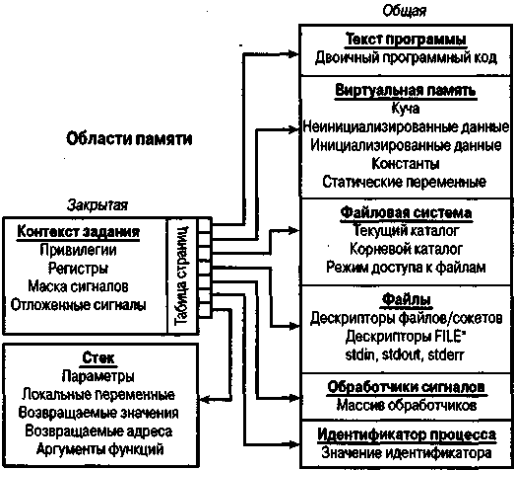

Рис. 7.1. Задания в Linux имеют несколько областей памяти

Когда следует применять многозадачный режим

Когда необходима многозадачность? В общем случае пользователь должен всегда контролировать выполнение программы. Иногда программа вынуждена ждать завершения других операций, и переход в многозадачный режим позволяет ей продолжить взаимодействие с пользователем во время простоя. Подобно тому как броузер Netscape позволяет вызывать команды меню в процессе загрузки Web-страницы, родительская программа должна поручать все операции сетевого ввода-вывода дочерним заданиям. Учитывая, что у различных серверов разное время ответа, можно эффективнее организовать использование сетевого канала, если с каждым сервером связать отдельный поток загрузки данных.

С помощью приведенного ниже правила можно быстро определить, когда необходим многозадачный режим. Ожидая завершения операции ввода-вывода, программа может одновременно:

- делать другую работу — обрабатывать информацию или поручать задания другим программам;
- взаимодействовать с пользователем — принимать от него данные или отображать информацию о состоянии;
- обслуживать другие программы или других пользователей. Например, одно задание может принимать запросы на подключение, а другое — управлять существующими соединениями.

Смешение потоков и процессов в рамках одной программы может показаться непривычным. Однако так часто происходит в больших интерактивных приложениях. В броузерах, к примеру, каждое окно является отдельным процессом, а для каждого запроса, такого как загрузка страницы, создается несколько потоков.

Характеристики многозадачного режима

У всех заданий в списке процессов (выводится с помощью системной команды top или ps aux) имеются общие атрибуты. Благодаря им можно лучше понять сущность многозадачности.

Во-первых, у каждого задания имеется предок. (Необходимо добавить слово "почти". В списке процессов можно заметить программу init. Она является прародителем всех заданий в системе и отвечает за их выполнение.) Родительское задание создает дочерние задания, которым передает часть ответственности. Когда задание-потомок завершается, его предок должен выполнить финальную очистку. Если он этого не делает, вмешивается программа init.

Каждое задание использует память и другие ресурсы ввода-вывода. Большинство программ работает с информацией большего объема, чем может вместить контекст задания (16—32 регистра). Эта информация размещается в ОЗУ и файле подкачки.

Программа должна с чем-то или кем-то взаимодействовать. Это подразумевает осуществление операций ввода-вывода. Каждому заданию предоставляются три общедоступных канала:

- stdin — стандартный входной поток (только для чтения), обычно связанный с клавиатурой;
- stdout — стандартный выходной поток (только для записи), обычно связанный с экраном;
- stderr — стандартный поток ошибок (только для записи), обычно связанный с экраном или журнальным файлом.

Направление всех стандартных потоков можно изменить (выполнить *переадресацию)* непосредственно в программе или в командной строке. Они могут быть связаны с другими устройствами, файлами и даже заданиями. При создании *(порождении)* дочернее задание наследует дескрипторы всех открытых файлов своего предка.

С каждым заданием связан отдельный аппаратный стек. Об этом важно помнить, особенно при выполнении низкоуровневого системного вызова clone() (рассматривается ниже), который создает новое задание. Программы используют аппаратные стеки для хранения результатов завершения функций, локальных переменных, параметров и возвращаемых адресов. Если бы задания решили разделить стек между собой, их работа немедленно нарушилась бы.

Наконец, у каждого задания имеется уникальный приоритет, представляющий собой число. Повышая или понижая приоритет программы, можно контролировать, сколько времени процессора она использует.

Планирование заданий в Linux

В многозадачных операционных системах применяются различные методики планирования заданий. В Linux используется схема приоритетного кругового обслуживания. В этой схеме каждое задание по очереди получает свою долю процессорного времени. Задания с высоким приоритетом перемещаются по списку быстрее, чем те, у которых низкий приоритет.

Сравнение процессов и потоков

Различия между процессами и потоками не всегда очевидны. В следующей таблице проведено их детальное сравнение.

Процессы	Потоки
После успешного вызова функции fork() существуют два процесса, выполняющихся параллельно	Родительская программа указывает имя функции, которая будет выполняться в качестве дочернего потока
Дочерний процесс должен быть явно завершен с помощью системного вызова exit()	Дочерний поток можно завершить явно либо неявно с помощью функции pthread_exit(void* arg) или инструкции return
Общих данных нет; единственная информация, передаваемая потомку, — это снимок данных родительского процесса	Потомок имеет доступ к данным предка, принимая от него параметры и возвращая значения
Дочерний процесс всегда связан с родительским; когда процесс-потомок завершается, его предок должен произвести очистку	Дочерний поток может выполняться независимо от родительского и завершиться без его вмешательства (если поток не является независимым, родительская программа также должна производить очистку после него)
Поскольку данные процесса недоступны другим процессам, не происходит конфликтов при доступе к ресурсам	Все совместно используемые данные должны быть идентифицированы и заблокированы, чтобы не произошло их повреждение
Независимая работа с файловой системой	Потомок реагирует на все изменения текущего каталога (команда chdir), корневого каталога (команда enroot) и стандартного режима доступа к файлам (команда umask)
Таблицы дескрипторов открытых файлов не являются общими; операционная система копирует таблицы, поэтому если в двух процессах открыт один и тот же файл, то закрытие его в одном процессе не приведет к изменению работы другого процесса	Совместное использование таблиц дескрипторов; если дочерний поток закрывает файл, родительский поток теряет к нему доступ
Сигналы обрабатываются независимо	Один поток может блокировать сигнал с помощью функции sigprocmask(), не влияя на работу других потоков

Создание процесса

Многозадачность чаще всего реализуется с помощью процессов. Процесс представляет собой новый экземпляр программы, наследующий от нее копии дескрипторов открытых каналов ввода-вывода и не обменивающийся никакими

другими данными. Для порождения нового процесса предназначен системный вызов fork():

```
#include <unistd.h>
pid_t fork(void);
```

Функция fork() проста и "немногословна": вы просто вызываете ее, и внезапно у вас появляются два идентичных процесса. Она возвращает значения в трех диапазонах:

- нуль — означает, что функция завершилась успешно и текущее задание является потомком; чтобы получить идентификатор процесса-потомка, вызовите функцию getpid();
- положительное число — означает, что функция завершилась успешно и текущее задание является предком; значение, возвращаемое функцией, представляет собой идентификатор потомка;
- отрицательное число — произошла ошибка; проверьте значение переменной errno или вызовите функцию perror(), чтобы определить причину ошибки.

В большинстве программ функция fork() помещается в условную конструкцию (например, if). Результат проверки позволяет определить, кем стала программа — предком или потомком. В листингах 7.1 и 7.2 приведены два типичных примера использования функции.

Листинг 7.1. Пример разделения заданий

```
/*********************************************************/
/***  Предок и потомок выполняются каждый по-своему  ***/
/*********************************************************/
int pchild;

if ( (pchild = fork()) == 0 )
{   /* это процесс-потомок */
    /*— выполняем соответствующие действия —*/
    exit(status); /* Это важно! */
}
else if ( pchild > 0 )
{   /* это процесс-предок */
    int retval;
    /*— выполняем соответствующие действия —*/
    wait(&retval); /* дожидаемся завершения потомка */
}
else
{   /* произошла какая-то ошибка */
    perror("Tried to fork() a process");
```

Листинг 7.2. Пример делегирования полномочий

```
/*********************************************************/
/*** Предок (сервер) и потомок (обрабатывает задание)  ***/
/*********************************************************/
int pchild;
for (;;) /* бесконечный цикл */
{
    /*— ожидаем получения запроса —*/
    if ( (pchild = fork()) == 0 )
    {   /* это процесс-потомок */
        /*— обрабатываем запрос —*/
        exit(status);
    }
    else if ( pchild > 0 )
    {   /* предок выполняет очистку */
        /* функция wait() НЕ НУЖНА */
        /* используйте сигналы (см. далее) */
    }
    else
    {   /* произошла какая-то ошибка */
        perror("Can't process job request");
    }
```

В программе, представленной в листинге 7.1, процесс-предок выполняет какую-то работу, а затем дожидается завершения процесса-потомка. В листинге 7.2 происходит распределение полномочий. Когда какая-то внешняя программа посылает запрос, предок создает потомка, который обрабатывает запрос. Большинство серверов работает именно по такой схеме.

Часто требуется одновременно выполнять разные действия. Они могут быть одинаковыми с алгоритмической точки зрения, но использовать отличающиеся наборы данных. Не имея подобной возможности, программам пришлось бы тратить время на повторное выполнение одних и тех же функций. *Дифференцирование* означает разделение заданий таким образом, чтобы они не дублировали друг друга. Хотя программа, представленная в листинге 7.3, корректна с синтаксической точки зрения, она уничтожает суть многозадачности, так как в ней не происходит дифференцирования.

Листинг 7.3. Ветвление без дифференцирования

```
/*********************************************************/
/***   Это пример дублирования. В подобном варианте вызова ***/
/***   функции fork() задания дублируют друг друга, что    ***/
/***   приводит к бессмысленной трате ресурсов процессора. ***/
/*********************************************************/
/*— какие-то действия —*/
fork();
/*— продолжение —*/
```

В нашей книге многозадачность без дифференцирования называется *дублированием* или *слиянием*. Как правило, подобной ситуации следует избегать. Дублирование может также произойти, когда процесс не завершился корректно путем явного вызова функции exit().

Устойчивость к ошибкам за счет слияния

Дублирование процессов может применяться при реализации отказоустойчивых систем. В отказоустойчивой системе вычисления дублируются с целью повышения достоверности результатов. Запускается несколько одинаковых заданий, каждое из которых закрепляется за отдельным процессором {эта возможность еще не реализована в Linux). Через определенный промежуток времени все задания посылают свои результаты модулю проверки. Если в каком-то процессоре произошел сбой, полученные данные будут отличаться. Соответствующее задание выгружается.

Если при выполнении функции fork() произошла ошибка, то, очевидно, возникли проблемы с таблицей страниц процесса или с ресурсами памяти. Одним из признаков перегруженности системы является отказ в предоставлении ресурсов. Виртуальная память и таблицы страниц являются основой правильного функционирования операционной системы. Поскольку эти ресурсы очень важны, Linux ограничивает общий объем ресурсов, которыми может владеть процесс. Когда невозможно выделить блок памяти требуемого размера или нельзя создать новое задание, значит, система испытывает острую нехватку памяти.

Создание потока

Благодаря потокам можно организовать совместный доступ к ресурсам со стороны родительской программы и всех ее дочерних заданий. Создавая потоки, программа может поручить им обработку данных, с тем чтобы самой сосредоточиться на решении основной задачи. Например, один поток может читать графический файл с диска, а другой — отображать его на экране. В процессах столь тесного взаимодействия, как правило, не требуется.

Одной из наиболее известных реализаций многопотоковых функций является библиотека Pthreads. Она совместима со стандартом POSIX 1c. Программы, в которых используется эта библиотека, будут выполняться в других операционных системах, поддерживающих стандарт POSIX. В библиотеке Pthreads новый поток создается с помощью функции pthread_create ( ):

```
#include <pthread.h>
int pthread_create(pthread_t* child, pthread_attr_t* attr,
                   void* (*fn)(void*), void* arg);
```

Различия между библиотечными и системными вызовами

Библиотечные и системные потоковые функции отличаются объемом выполняемой работы. Функция fork() является интерфейсом к сервисам ядра. Вызов функции pthread_create() преобразуется в системный вызов _clone(). В этом случае для компиляции программы необходимо *в* качестве последнего аргумента командной строки компилятора cc указать переключатель -lpthreads. Например, чтобы скомпилировать файл mythreads.c и подключить к нему библиотеку Pthreads, выполните такую команду:

```
cc mythreads.c -o mythreads -lpthreads
```

Как и в случае системного вызова fork(), после завершения функции pthread create() начинает выполняться второе задание. Однако создать поток сложнее, чем процесс, так как в общем случае требуется указать целый ряд параметров (табл. 7.1).

Таблица 7.1. Параметры функции pthread_create()

Параметр	Описание
child	Дескриптор нового потока; с помощью этого дескриптора можно управлять потоком после завершения функции
attr	Набор атрибутов, описывающих поведение нового потока и его взаимодействие с родительской программой (может быть равен NULL)
fn	Указатель на функцию, содержащую код потока; в отличие от процессов, каждый поток выполняется в отдельной подпрограмме родительской программы, и когда эта подпрограмма завершается, система автоматически останавливает поток
arg	Параметр, передаваемый функции потока и позволяющий конфигурировать его начальные установки; необходимо, чтобы блок данных, на который осуществляется ссылка, был доступен потоку, т.е. нельзя ссылаться на стековую переменную (этот параметр тоже может быть равен NULL)

Как уже было сказано, после завершения функции существуют два потока: родительский и дочерний. Оба они совместно используют все программные данные, кроме стека. Родительская программа хранит дескриптор дочернего потока (child), который выполняется в рамках своей функции (fn) с конфигурационными параметрами (arg) и атрибутами (attr). Даже если параметры потока задать равными NULL, его поведение можно будет изменить впоследствии. Но до того времени он будет выполняться в соответствии с установками по умолчанию.

В итоге вызов функции сводится к указанию всего двух параметров. В следующих двух листингах можно сравнить алгоритмы создания процесса и потока (листинг 7.4 и 7.5).

Листинг 7.4. Пример создания процесса

```
/***          В этом примере создается процесс          ***/

voidChild_Fn(void)
{
   /* код потомка */


int main(void)
{  int pchild;

   /*-  Инициализация  -*/
   /*-  Создание нового процесса  -*/
   if ( (pchild = fork(» < 0 )
        perrorf ("Fork error");
   else if ( pchild == 0 )
   {    /* это процесс-потомок */
        /* закрываем ненужные ресурсы ввода-вывода */
        Child_Fn();
        exit(0);
```

```
   else if ( pchild > 0 )
   {   /* это процесс-предок */
       /* закрываем ненужные ресурсы ввода-вывода */
       /* дожидаемся завершения потомка */
       wait();
   }
   return 0;
```

Листинг 7.5. Пример создания потока

```
/***              В этом примере создается поток              ***/


void *Child_Fn(void *arg)
{  struct argstruct *myarg = arg;
   /* код потомка */
   return NULL; /* произвольное значение */


int main(void)
{   struct argstruct arg = {};
   pthread_t tchild;

   /*-  Инициализация  -*/
   /*-  Создание нового потока  -*/
   if ( pthread_create(&tchild, NULL, &Child_Fn, &arg) != 0 )
       perror("Pthreads error");  /* ошибка */

   /*** обратите внимание на то, что остальных проверок нет ***/
   /* Мы по-прежнему находимся в родительском модуле (неявно) */

   /* выполняем другие действия */
   /* дожидаемся завершения потомка */
   pthread_join(tchild, NULL);

   return 0;
}
```

При создании процесса вызывается функция `fork()`, а затем проверяется, кем — предком или потомком — стала программа. При создании потока указывается функция, в которой он выполняется, атрибуты, задающие поведение потока, и инициализирующий параметр.

В первом случае требуется, чтобы дочерний процесс завершился явным вызовом функции exit() во избежание дублирования. При работе с библиотекой Pthreads об этом можно не беспокоиться. Инструкция return (и даже просто достижение конца функции) приводит к неявному завершению потока.

Системный вызов clone()

В Linux имеется низкоуровневая функция _clone(), позволяющая гораздо сильнее управлять созданием процессов и потоков. С ее помощью можно задавать 6 различных режимов совместного доступа к памяти. Если снова обратиться к диаграмме страниц виртуальной памяти (см. рис. 7.1), то окажется, что функция _clone() дает возможность указывать любую комбинацию общих областей памяти.

Будьте осторожны при работе с функцией __clone ()

Системный вызов __clone {) предназначен для истинных профессионалов. Учитывая мощь этой функции, можно легко разрушить работу программы и сделать отладку практически невозможной.

Синтаксис функции __clone () таков:

```
#include<sched.h>
int __clone(int (*fn)(void*), void* stacktop, int flags,
            void* arg);
```

Подобно функции fork(), она возвращает идентификатор дочернего задания или - 1 в случае ошибки. В табл. 7.2 описано назначение каждого параметра.

Таблица 7.2. Параметры функции __clone()

Параметр	Описание
fn	Как и в функции pthread_create(), это указатель на функцию потока; когда она завершается (с помощью инструкции return или системного вызова exit()), поток останавливается
stacktop	Указатель на вершину стека дочернего задания; в большинстве процессоров (за исключением HP/PA RISC) стек заполняется в направлении сверху вниз, поэтому необходимо задать указатель на первый байт стека (чтобы добиться совместимости, воспользуйтесь директивами условной компиляции)
flags	Набор флагов, определяющих, какие области памяти используются совместно (табл. 7.3) и какой сигнал посылать при завершении дочернего задания (по умолчанию - SIGCHLD)
arg	Аналогично функции pthread_create(), передается в качестве параметра функции fn

Работа со стеками

Поскольку совместное использование аппаратного стека недопустимо, родительский процесс должен зарезервировать в программе дополнительную память для стека дочернего задания. Эта область памяти будет общедоступной.

Флаги, представленные в табл. 7.3, позволяют выбрать, какие из общих областей памяти задания будут доступными для совместного использования (см. рис. 7.1). Если какой-то из флагов не указан, операционная система подразумевает, что соответствующая область памяти должна копироваться между заданиями.

Таблица 7.3. Флаги функции clone ()

Флаг	Описание
CLONE_VM	Совместное использование области данных между заданиями; если флаг указан, будут доступны все статические и предварительно инициализированные переменные, а также блоки, выделенные в куче, в противном случае в дочернем задании будет создана копия области данных
CLONE_FS	Совместное использование информации о файловой системе: текущем каталоге, корневом каталоге и стандартном режиме доступа к файлам (значение umask); если флаг не указан, задания будут вести себя независимо друг от друга
CLONE_FILES	Совместное использование открытых файлов; когда в одном задании перемещается указатель текущей позиции файла, в другом задании отразится это изменение, и если закрыть файл в одном задании, то и в другом он станет недоступным (если флаг не указан, в дочернем задании создаются новые ссылки на открытые индексные дескрипторы)
CLONE_SIGHAND	Совместное использование таблиц сигналов; каждое задание может запретить обработку того или иного сигнала с помощью функции sigprocraaskf), и это не отразится на других заданиях (если флаг не указан, в дочернем задании создается копия таблицы сигналов)
CLONE_PID	Совместное использование идентификатора процесса; применять данный флаг следует осторожно, так как он не всегда поддерживается (как это имеет место в случае библиотеки Pthreads); если флаг не указан, в дочернем задании создается новый идентификатор процесса

Функция _clone() является универсальным средством создания заданий. Если все области памяти доступны для совместного использования, создается поток, если ни одна из областей недоступна, создается процесс. Будучи правильно сконфигурированной, функция заменяет собой как системный вызов fork(), так и функцию pthread_create().

В листингах 7.6 и 7.7 сравнивается, как создавать процессы и потоки с помощью функций fork() и pthread_create() с одной стороны, и функции__clone() — с другой.

Листинг 7.6. Пример функций fork() и pthread_create()

```
/*****************************/
/***  Системный вызов fork()  */
/*****************************/
void Child(void)
{
   /* код потомка */
   exit(0);
}

int main(void)
{  int pchild;

   if ( (pchild = fork()) == 0 )
       Child();
   else if ( pchild > 0 )
```

```
        wait();
    else
        perror( "Can't  fork process");


/*********************************************/
/***  Библиотечная функция pthread_create ()  ***/
/*********************************************/
void* Child (void *arg)
{
   /* код потомка */
   return &Result;
}

int main (void)
{   pthread_t tchild;

   if ( pthread_create(Stchild, 0, &Child, 0) != 0
       perror ( "Can't create thread" ) ;
   pthread_join(tchild, 0);
}
```

Листинг 7.7. Эквивалентный пример функции clone ()

```
/*****************************************/
/*** Эквивалент системному вызову fork()  */
/*****************************************/
void Child(void *arg)
{
   /* код потомка */
   exit(0);
}
idefine STACK   1024
int main(void)
{   int cchild;
   char *stack = mallocfSTACK);
   if ((cchild = _clone(&Child, stack+STACK-1, SIGCHLD, 0)) == 0)
       {/** секция дочернего задания — недоступна **/}
   else if ( cchild > 0 )
       wait();
   else
       perror("Can't  clone  task");
}
/*****************************************/
/*** Эквивалент функции pthread_create()   ***/
/*********************                    "~
void* Child(void *arg)
{
   /* код потомка */
   exit(0);
}
```

```
tdefine STACK    1024
int  main(void)
{   int cchild;
    char *stack = malloc(STACK);
    if ((cchild = _clone(&Child, stack+STACK-1, CLONE_VM |
                         CLONE_FS | CLONE_FILES | CLONE_SIGHAND |
                         SIGCHLD, 0)) < 0)
        perror("Can't clone");
    wait();
}
```

Функцию _clone() можно использовать практически везде, где стоит вызов одной из двух эквивалентных ей функций. Следует, однако, отметить, что она не полностью совместима с библиотекой Pthreads. Например, клонированное задание может возвращать только целочисленное значение (как и процесс), тогда как в библиотеке Pthreads допускается возвращение произвольного значения из потока. Кроме того, отличается процедура завершения задания. Если создается программа, которая будет выполняться не только на компьютерах Linux, лучше воспользоваться стандартными библиотечными функциями.

Взаимодействие заданий

При создании заданий необходимо позаботиться об их дифференцировании, чтобы они не выполняли дублирующуюся работу. Это можно сделать как путем полного разделения процессов, так и путем организации взаимодействия потоков. К примеру, рассмотрим игру в покер. Игра делится на несколько фаз, в каждой из которых игроки должны подчиняться определенным правилам. Поскольку у игроков на руках разные карты, а каждый из участников имеет свой стиль игры и неодинаковую сумму денег, игра становится интересной. Точно так же и в программировании можно достичь оптимальных результатов, если поручить нескольким заданиям выполнять различные функции.

Инициализация потомка

В покере каждый игрок покупает набор фишек, а сдающий раздает карты. У каждого из игроков свои принципы торговли, а на сдающего возложена дополнительная обязанность управлять колодой карт. В программировании одним из способов организации взаимодействия является запуск задания в определенном состоянии. После запуска как процессы, так и потоки следуют свои путем, обычно не пересекаясь с родительской программой.

Родительская программа передает дочернему заданию данные, формируя среду, в которой выполняется потомок. Предок располагает лишь незначительными возможностями корректировать работу потомка. Как и в покере, управление носит директивный ("Этого делать нельзя!"), а не процедурный ("Ставь все деньги!") характер.

Процедура запуска процессов и потоков в целом похожа. В первую очередь необходимо определить совместно используемые области памяти, а затем создать новое задание. Потоки дополнительно могут принимать параметр типа void*. Тип данных void* определен в языке C для работы с абстрактными структурами. Благодаря ему можно передавать в потоковую функцию любое значение, а уже при-

нимающая сторона будет анализировать это значение. Получение данного параметра подобно сдаче карт в покере.

Тип данных void

Может показаться, что передача параметра типа void* — очень мощная возможность, но в действительности необходимо внимательно следить за тем, какие именно данные передаются. Это должен быть либо блок памяти, выделенный с помощью функции malloc(), либо глобальная или статическая переменная. Другими словами, это не должны быть данные, помещаемые в стек (имеющие неявный спецификатор auto). Причина этого проста: при вызове/завершении различных функций содержимое стека меняется. Дочернее задание может не получить требуемые данные.

Общедоступная память

В покере имеются два ресурса: кон и колода. От них зависит, как играть и когда говорить "пас". Эти ресурсы ограничены и используются игроками совместно. Сдающий берет карты из колоды и распределяет их между игроками. Игроки оценивают полученные карты и делают ставки, бросая фишки на кон. Точно так же необходимо помнить о ресурсах памяти, выделенных программе.

Можно запрограммировать процессы и потоки таким образом, что они будут просто передавать информацию в общедоступную область памяти, принадлежащую кому-то другому. Совместное использование памяти заложено в концепцию потоков: родительская программа и дочерний поток (а также все параллельные потоки) по умолчанию имеют доступ к одним и тем же областям памяти. Единственным уникальным ресурсом является стек (необходимо убедиться, что игроки не делают ставки одновременно).

Создать общедоступную область памяти в процессах можно с помощью системного вызова shmget(). Через этот блок памяти процессы могут обмениваться данными, однако их работа замедлится, поскольку управление доступом к памяти берет на себя операционная система.

Взаимодействие процессов

Во время игры в покер игроки обмениваются сообщениями друг с другом и со сдающим. Некоторые сообщения являются инструкциями, а некоторые носят эмоциональный характер. Один игрок может передать сообщение другому игроку или всем присутствующим.

Проектируемое задание может взаимодействовать с другими процессами. Обычно подобное взаимодействие организуется в форме *каналов:* сообщение направляется от одного задания к другому через канал. Канал является однонаправленным, подобно стандартным потокам ввода-вывода stdin и stdout, и создается с помощью системного вызова pipe():

```
linclude <unistd.h>
int pipe(int fd[2]);
```

Параметр fd является массивом, состоящим из двух целых чисел. В этот массив записываются два дескриптора файла. В результате выполнения функции могут возникать следующие ошибки.

- EMFILE. Уже открыто слишком много файлов.
- EFAULT. Параметр fd не указывает на массив из двух целых чисел.

Ниже представлен пример использования функции pipe():

```
/*****************************************************************/
/***                  Пример функции pipe()                    ***/
/*****************************************************************/

int fd[2];      /* создание массива, содержащего два дескриптора */
if ( pipe(fd) != 0 )                          /* создание канала */
    perror("Pipe creation error");

read(fd[0], buffer, sizeof(buffer)); /* чтение данных из канала */
```

Индексация дескрипторов

Каждому каналу ввода-вывода назначается номер, который является индексом в таблице дескрипторов файлов. По умолчанию у каждого процесса, есть три канала ввода-вывода: stdin(0), stdout (1) и stderr (2). Когда создается новый канал, ему назначаются две ближайшие позиции в таблице (одна — для входного конца, другая — для выходного). Например, если у задания нет открытых файлов, то в результате вызова функции pipe() будет создан канал чтения (3) и канал записи (4).

Создавать канал в многозадачной среде довольно сложно. Но достаточно сделать это несколько раз, и вы избавитесь от головной боли.

В листингах 7.8 и 7.9 демонстрируется, как создавать каналы в родительских процессах и потоках. Аналогичные операции в дочерних заданиях рассматриваются чуть ниже.

Листинг 7.8. Создание канала в родительском процессе

```
/*****************************************************************/
/***                  Родительский процесс                     ***/
/*****************************************************************/

int FDs[2];    /* создание массива, содержащего два дескриптора */
pipe(FDs); /* создание канала: FDs[0] - чтение, FDs[1] - запись */
char buffer[1024];

/*— Создание процесса с помощью функции fork() —*/
close(FDs[0]);      /* делаем канал доступным только для записи */

write(FDs[1], buffer, buffer_len); /* передача сообщения
                                      дочернему заданию */

/*— дальнейшая обработка —*/
wait();
close(FDs[1]);
```

Листинг 7.9. Создание канала в родительском потоке

```
/*****************************************************************/
/***                  Родительский поток                       ***/
/*****************************************************************/
```

```
int FDs[2];    /* создание массива, содержащего два дескриптора */
pipe(FDs); /* создание канала: FDs[0] - чтение, FDs[1] - запись */
char buffer[1024];

/*— Создание потока с помощью функции pthread create() —*/

write(FDs[1], buffer, buffer_len); /* передача сообщения
                                      дочернему заданию */

/*— закрытие канала предоставляется потомку —*/
pthread_join(pchild,  arg);
```

Дочерние процессы и потоки решают другие задачи и программируются по-разному. Сравните листинги 7.10 и 7.11.

Листинг 7.10. Создание канала в дочернем процессе

```
/***                    Дочерний процесс                          ***/

int FDs[2];    /* создание массива, содержащего два дескриптора */
pipe(FDs); /* создание канала: FDs[0] - чтение, FDs[1] - запись */
char buffer[1024];

/*— Создание дочернего процесса —*/

dup(FDs[0], 0); /* замещаем поток stdin */•
close(FDs[0]);  /* больше не нужен */
close(FDs[1]);  /* передача данных родительскому процессу
                   не производится */

read(0, buffer, sizeof(buffer)); /* чтение сообщения от
                                    родительского процесса */

/*— дальнейшая обработка —*/
printf("My report...");
exit(0);
```

Листинг 7.11. Создание канала в дочернем потоке

```
/*********«
/***                    Дочерний поток                            ***/

int FDs[2];    /* создание массива, содержащего два дескриптора */
pipe(FDs); /* создание канала: FDs[0] - чтение, FDsfl] - запись */
char buffer[1024];

/*___ создание дочернего потока —*/
```

```
read(FDs[0], buffer, sizeof(buffer)); /* чтение сообщения от
                                         родительского процесса */

/*— дальнейшая обработка —*/
printf("My report...");

close(FDs[0]);   /* закрываем входной канал */
close(FDs[1]);   /* закрываем выходной канал */

pthread_exit(arg);
```

При создании канала между двумя заданиями пользуйтесь схемой, представленной на рис. 7.2, в качестве руководства. Ниже описана общая последовательность действий.

1. Родительская программа объявляет массив дескрипторов файлов. Этот массив будет заполняться функцией pipe().
2. Родительская программа создает канал. При этом ядро создает очередь ввода-вывода и помещает дескриптор канала чтения в элемент fd[0], а дескриптор канала записи — в элемент fd[1].

При работе с каналом необходимо закрывать один из его концов

Когда система создает канал, она связывает его входной конец с выходным, т.е. фактически канал замыкается — все, что программа запишет в канал, будет ею же прочитано, хотя сообщение дойдет и до потомка. В большинстве своем коммуникационные каналы являются односторонними, поэтому ненужный конец канала должен быть закрыт, Если же требуется полноценное двустороннее взаимодействие, необходимо вызвать функцию `pipe()` дважды и создать для каждого направления свой канал (рис. 7.3).

3. Родительская программа создает новое задание. При этом процесс получает точную копию родительских данных (включая канал). Канал между предком и потомком является замкнутым.
4. Родительская программа взаимодействует с дочерним заданием благодаря имеющимся дескрипторам канала.
5. Дочерний процесс перенаправляет свой собственный входной поток (это необязательно) и получает данные от родительской программы через открытый канал. Выходной канал закрывается.
6. Родительской программе требуется только выходной конец канала для взаимодействия с потомком, поэтому она закрывает входной конец.

Перенаправлять каналы ввода-вывода в потоках нельзя, поскольку они совместно используют таблицы дескрипторов файлов. Это означает, что если закрыть файл в одном потоке, то он автоматически закроется во всех остальных потоках программы. В то же время процессы создают копии таблиц дескрипторов, так что закрытие одного из каналов не влияет на работу процессов программы.

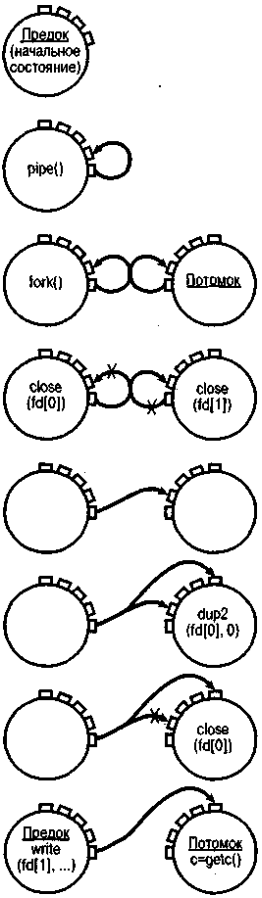

Рис. 7.2. При создании канала между заданиями необходимо придерживаться определенной последовательности действий по перенаправлению и закрытию каналов

На рис. 7.2 была представлена схема создания одностороннего соединения между предком и потомком. Для создания двунаправленного соединения необходимо придерживаться схемы, изображенной на рис. 7.3.

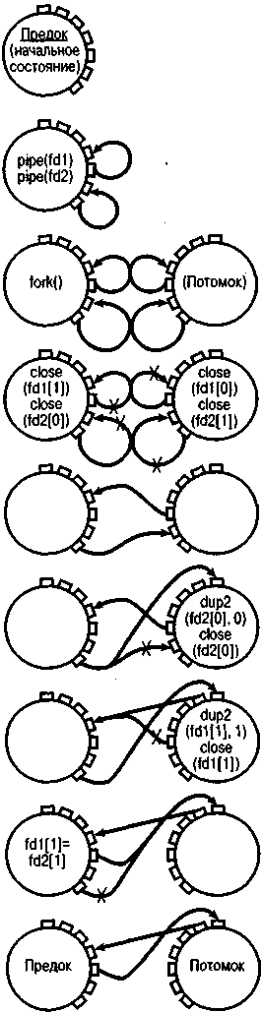

Рис. 7.3. Если между предком и потомком необходимо организовать двунаправленное соединение, создайте дополнительный канал

Сигнализация о завершении

В покере есть несколько прямых управляющих команд. В любой момент игрок может сказать "пас" (выйти из текущей игры). Если бы игра происходила на Диком Западе, обманывающий игрок мог подвергнуться внезапному исключению из игры. В любом случае брошенные карты забирает сдающий. Точно так же процесс с помощью сигнала может уведомить программу о своем завершении.

Любое задание в программе должно обрабатывать все полезные сигналы. Всего существует около 30-ти сигналов (два сигнала определяются пользователем). Большинство из них можно игнорировать или не перехватывать, поскольку вероятность их возникновения ничтожно мала или их обработка не представляет особого смысла. Многие сигналы, будучи не перехваченными, приводят к завершению программы, а некоторые из этих сигналов важны в многозадачных программах. Например, операционная система уведомляет родительскую программу о завершении задания-потомка, посылая сигнал SIGCHLD.

Сигнал напоминает запрос на аппаратное прерывание. О нем известно только то, что он произошел. Когда появляется сигнал, задание прекращает выполнять свои действия и переходит к специальной подпрограмме (называемой *обработчиком сигналов).* При получении сигнала SIGCHLD можно вызвать функцию wait(), чтобы провести дополнительную очистку после завершения потомка. Можно перехватывать и другие сигналы, в частности, те, которые возникают в случае математических ошибок или после нажатия клавиш <Ctrl+C>.

В Linux поддерживаются сигналы двух разновидностей: в стиле System V (однократный сигнал, который возвращается стандартному обработчику, когда система вызывает пользовательский обработчик) и в стиле BSD (посылается обработчику до тех пор, пока не будет явно остановлен). Если сигнал посылается с помощью системного вызова signal(), он будет однократным. Но на это можно не обращать внимания: следует ожидать, что сигнал будет перехватываться многократно.

Сброс сигнала

Для сброса сигнала некоторые программисты помещают вызов системного обработчика сигналов непосредственно в тело собственного обработчика. К сожалению, это может привести к возникновению такого состояния, как "гонка", когда сигнал приходит раньше, чем вызывается сбрасывающая его функция.

Вместо функции signal() лучше пользоваться системным вызовом sigaction(), который позволяет лучше контролировать поведение сигнальной подсистемы. Прототип этой функции таков:

```
iinclude  <signal.h>
int  sigaction(int  sig_num,  const  struct  sigaction  *action,
                const struct sigaction *old);
```

Первый параметр определяет номер перехватываемого сигнала. Второй параметр задает способ обработки сигнала. Если последний параметр не равен NULL, будет запомнено последнее выполненное действие. Ниже приведено определение структуры sigaction:

```
struct sigaction
{
   /* Указатель на функцию обработки */
   void (*sa_handler)(int signum);
   /* Специальная функция обратного вызова */
   void (*sa_sigaction)(int, siginfo t *, void *);
   /* Массив битовых флагов, указывающих, какие сигналы
      следует игнорировать, находясь в теле обработчика */
   sigset_t sajnask;
   /* Выполняемое действие */
   int sa_flags;
   /* (больше не используется - должно быть равно 0) */
   void (*sa_restorer)(void);
};
```

Для активизации третьего параметра необходимо, чтобы первое поле (sa_handler) отличалось от указателя на функцию во втором параметре. Если поместить в это поле константу SIG_IGN, программа проигнорирует указанный сигнал, а если задать константу SIG_DFL, будет восстановлена стандартная процедура обработки.

Чтобы разрешить или запретить каскадирование сигналов (один сигнал прерывает другой, вследствие чего возникает цепочка вызовов обработчиков), воспользуйтесь третьим полем структуры, sa mask. Каждый бит (всего их 1024) обозначает разрешение (1) или запрет (0) обработки сигнала. По умолчанию обработчик сигнала игнорирует другие аналогичные сигналы. Например, если обрабатывается сигнал SIGCHLD и в это же время завершается другой процесс, повторный сигнал будет проигнорирован. Подобный режим можно изменить с помощью флага SA_NOMASK.

Поле sa_flags содержит набор флагов, определяющих поведение обработчика. В большинстве случаев это поле можно задать равным нулю.

- SA_ONESHOT. Режим System V: сброс сигнала после того, как он перехвачен.
- SA_RESETHAND. То же, что и SA_ONESHOT.
- SA_RESTART. Повторный запуск некоторых системных функций, если сигнал прервал их выполнение. Это позволяет восстанавливать работу таких функций, как, например, accept().
- SA_NOMASK. Разрешить обработку того же самого сигнала во время обработки более раннего сигнала.
- SA_NODEFER. То же, что и SA_NOMASK.
- SA_NOCLDSTOP. Не уведомлять родительскую программу о прекращении работы дочернего задания (сигнал SIGSTOP, SIGTSTP, SIGTTIN или SIGTTOU). Этот флаг важен для данной главы.

В листинге 7.12 демонстрируется, как перехватывать сигнал SIGFPE (исключительная ситуация в операции с плавающей запятой) и игнорировать сигнал SIGINT (запрос на прерывание от клавиатуры).

Листинг 7.12. Обработчик сигналов SIGFPE и SIGINT

```
/*** Перехват сигнала SIGFPE и игнорирование сигнала SIGINT ***/

#include<signal.h>

/* Определение обработчика сигналов */
void sig_catcher(int sig)

   printf("I caught signal #%d\n", sig);


int main(void)
{  struct sigaction act;

   bzero(&act, sizeof(act));
   act.sa_handler = sig_catcher;
   sigaction(SIGFPE,  act,  0);        /* перехватываем ошибку в
                                    операции с плавающей запятой */

   act.sa_handler = SIG_IGN;              /* игнорируем сигнал */
   signal(SIGINT,&act,0);               /* игнорируем сигнал ^C */
   /*—  тело  программы  —*/
}
```

Потеря сигналов в обработчиках

Если обработчик сигналов выполняется слишком долго, программа может потерять сигналы, ожидающие обработки. В очереди сигналов только одна позиция — когда приходят два сигнала, записывается только один из них. Поэтому старайтесь минимизировать время, затрачиваемое на обработку сигналов.

Серверы и клиенты могут принимать несколько различных сигналов. Чтобы сделать программу более отказоустойчивой, следует обрабатывать все сигналы, от которых потенциально зависит работа программы. (Некоторые сигналы, например SIGFAULT, лучше всего не трогать. Данный сигнал свидетельствует о наличии ошибки в тексте программы или в ее данных. Такую ошибку нельзя исправить.)

Уменьшение текста программы за счет совместного использования обработчиков сигналов

Можно объединить несколько обработчиков сигналов в одной подпрограмме. Распределение обязанностей несложно организовать внутри подпрограммы, так как система передает ей номер сигнала.

Дочернему заданию можно послать любой сигнал. Из командной строки это можно сделать с помощью команды kill. В программе доступен системный вызов kill(). Его прототип выглядит следующим образом:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t PID, int sig_num);
```

Детальное описание параметров этой функции можно найти в интерактивном справочном руководстве. По сути, программа вызывает функцию kill(), указывая идентификатор задания и сигнал, который следует ему послать.

Получение данных от потомка

Вернемся к игре в покер. Когда игра заканчивается или последний из оппонентов говорит "пас", сдающий должен собрать колоду и заново ее перетасовать. Некоторые игроки могут потребовать посмотреть карты победителя, прежде чем он заберет себе выигрыш. Точно так же родительская программа должна проверять результаты завершения каждого из потомков.

По коду завершения программа может определить, успешно ли выполнилось дочернее задание. Когда процесс завершается, он всегда возвращает целое число со знаком. В то же время поток может вернуть абстрактный объект типа void*. В этом случае необходимо соблюдать осторожность, чтобы не произошло потери данных. Не следует возвращать объект, созданный на основании значений стековых переменных. Лучше передавать данные через кучу или глобальную переменную.

Обгоняя время: исключающие семафоры и гонки

Сила, заключенная в потоках, очень привлекательна. Если правильно управлять ими, можно заставить программу выполняться быстрее и реже "зависать". Тем не менее есть один подводный камень — соперничество за право обладания ресурсами. Когда два потока одновременно обновляют одни и те же данные, они почти наверняка будут повреждены. Отлаживать такие потоки можно часами. Ниже рассматриваются вопросы, связанные с одновременным доступом к ресурсам.

Гонки за ресурсами

Возможно, вам знакомо *состояние гонки, в* котором оказываются два потока, пытающиеся сохранить свои данные. Ранее в этой главе рассматривался пример, когда гонка возникала при сбросе сигнала. *Критической секцией* называется раздел программы, где происходит "столкновение" потоков. Рассмотрим листинги 7.13 и 7.14, предполагая, что оба потока выполняются одновременно.

Листинг 7.13. Состояние гонки в потоке 1

```
/*****************************************************************/
/***  Пример гонки, в которой два потока соперничают           ***/
/***  за право доступа к массиву queue                         ***/
```

```
int queue[10];
int in, out, empty;
/************** поток 1 **************/
/* Чтение данных из очереди */
if ( !empty ) /* избегаем чтения пустой очереди */
{   int val = queue[out];
    out++;
    if ( out >= sizeof(queue) )
        out = 0; /* начинаем заново */
    empty = (out == in);
}
```

Листинг 7.14. Состояние гонки в потоке 2

```
/***  Пример гонки, в которой два потока соперничают          ***/
/***  за право доступа к массиву queue                        ***./

int queue[10];
int in, out, empty;
/************** поток 2 **************/
/* Запись данных в очередь */
if ( Jempty && out != in ) /* избегаем переполнения очереди */
{   queue[in] = 50;

    if ( in >= sizeof(queue) )
        in = 0; /* начинаем заново */
    empty = (out == in);
}
```

Обратите внимание на то, что приращение индексных переменных происходит после обращения к очереди. Все кажется правильным, если предположить, что потоки выполняются параллельно, строка за строкой. К сожалению, это происходит очень редко. Что если первый поток отстает от второго на несколько строк? Поток 2 мог выполнить проверку переменной empty как раз перед тем, как поток 1 сбросил значение переменной out. Возникнет проблема, так как переменная out никогда не станет равной переменной in.

Другая проблема возникнет в случае переключения между потоками (если они выполняются не строго параллельно, а в многозадачной среде). Переключение заданий может произойти сразу после завершения операции out++ в потоке 1. При этом существует вероятность, что поток 2 получит неправильные значения переменных out и empty, так как не все проверки были завершены.

В этом примере две подпрограммы соперничают за четыре ресурса: переменные queue, in, out и empty.

Исключающий семафор

Работать с критическими секциями можно, блокируя другие процессы при обращении к ним. Подобный процесс называется *взаимным исключением* (кто пер-

вый захватил ресурс, тот блокирует остальных) или *сериализацией* (разрешение одновременного доступа к ресурсу только для одного задания). Эта методика позволяет предотвращать повреждение данных в критических секциях. *Исключающий семафор* — это флаг, разрешающий или запрещающий монопольный доступ к ресурсу. Если флаг сброшен (семафор опущен), поток может войти в критическую секцию. Если флаг установлен (семафор поднят), поток блокируется до тех пор, пока доступ не будет разрешен.

Существуют две методики блокировки: грубая и точная. В первом случае, когда программа входит в критическую секцию, блокируются все остальные выполняемые задания. При этом может отключаться режим квантований времени. С данной методикой связаны две большие проблемы: во-первых, блокируются задания, не относящиеся к текущей программе, и, во-вторых, она не поддерживается в многопроцессорных системах.

Точная блокировка применяется по отношению к ресурсам, а не заданиям. Поток запрашивает доступ к общему ресурсу. Если он не используется, поток захватывает его в свое распоряжение. Если оказывается, что ресурс уже зарезервирован, поток блокируется, ожидая освобождения ресурса.

В библиотеке Pthreads имеется множество средств управления потоками. Существуют также функции работы *с* исключающими семафорами. Использовать их очень просто (листинг 7.15).

Листинг 7.15. Пример исключающего семафора

```
/*************************************************************/
/***          Создание глобального исключающего семафора          ***/
/г*************************************************************/
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

/* Начало критической секции */
pthread_mutex_lock(&mutex);

/*— Работа с критическими данными —*/

pthread_mutex_unlock(&mutex);
/* Конец критической секции */
```

Полный текст этого примера содержится на Web-узле в файле thread-mutex.c. Параметр mutex является семафором, блокирующим доступ к секции. Он может быть инициализирован тремя способами.

- Быстрый (по умолчанию) — PTHREAD_MUTEX INITIALIZER. Выполняется простая проверка наличия блокировки. Если один и тот же поток попытается дважды заблокировать исключающий семафор, возникнет тупиковая ситуация (взаимоблокировка).
- Рекурсивный — PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP. Проверяется, не блокирует ли владелец повторно тот же самый исключающий семафор. Если это так, включается счетчик *(считающий семафор),* определяющий число блокировок. Исключающий семафор должен быть разблокирован столько раз, сколько было сделано блокировок.

- С проверкой ошибок — PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP. Проверяется, тот ли самый поток пытается разблокировать исключающий семафор, что и поток, который заблокировал его. Если это другой поток, возвращается ошибка и блокировка не снимается.

В библиотеке имеется дополнительная функция pthread_mutex_trylock(), которая запрашивает блокировку семафора. Если она невозможна, возвращается ошибка EBUSY.

Как избежать проблем с критическими секциями

Старайтесь, чтобы в критической секции выполнялись операции лишь над нужными ресурсами. В частности, не следует вызывать функции ввода-вывода или манипулировать другими данными, если только этого совершенно невозможно избежать. При необходимости можно скопировать данные в локальные переменные и обработать их за пределами секции. Помните, что включение операций ввода-вывода в критические секции может привести к возникновению взаимоблокировок.

Проблемы с исключающими семафорами в библиотеке Pthreads

При работе с библиотекой Pthreads следует помнить о некоторых ограничениях. Во-первых, исключающий семафор не содержит ссылку на блокируемую область памяти. Он представляет собой всего лишь флаг. Поэтому существует возможность, что два разных потока используют один и тот же семафор для блокирования несвязанных данных. Это не опасно — не произойдет ни повреждения данных, ни взаимоблокировки. Просто можно блокировать поток тогда, когда в этом нет никакой необходимости.

Во-вторых, рассмотрим ситуацию, когда несколько потоков работают с большим массивом данных, например с таблицей. Ее ячейки не связаны друг с другом, и потоки работают в разных сегментах таблицы. Необходимо блокировать не всю таблицу, а лишь отдельные ее секции *(зонная блокировка)*. Библиотека Pthreads не позволяет определить, в каком месте ресурса работает поток.

Наконец, может потребоваться определить приоритеты доступа. В некоторых случаях вполне допускается одновременное чтение данных несколькими потоками *(нежесткая блокировка)*. Но ни один из потоков не может осуществлять запись данных. В библиотеке Pthreads возможен только монопольный доступ — вы либо владеете ресурсом, либо нет.

Предотвращение взаимоблокировки

Представьте двух детей, играющих одними и теми же игрушками. Каждый ребенок видит игрушку другого и хочет ее, но не желает отдавать свою. В программировании это называется *взаимоблокировкой*.

Создавая потоки, обязательно выявляйте критические секции и возможные конфликты ресурсов. Обнаружив критические данные, определите, кому и когда они могут понадобиться. Может оказаться, что два ресурса должны быть забло-

кированы, прежде чем работа продолжится. Если проявить невнимательность, возникнет взаимоблокировка.

Рассмотрим следующий пример.

Поток 1	Поток 2
1. Блокирует семафор Funds_Mutex_1.	1. Блокирует семафор Funds_Mutex_2.
2. Блокирует семафор Funds_Mutex_2.	2. Блокирует семафор Funds_Mutex_1.
3. Используя семафор Funds_Mutex_2, изменяет семафор Funds_Mutex_1.	3. Используя семафор Funds_Mutex_1, изменяет семафор Funds_Mutex_2.
4. Разблокирует семафор Funds_Mutex_2.	4. Разблокирует семафор Funds Mutex_2.
5. Разблокирует семафор Funds_Mutex_1.	5. Разблокирует семафор Funds Mutex_1.

Взаимоблокировка в потоке 2 произойдет на втором этапе. Она возникает из-за того, что оба потока ожидают ресурсов друг друга. Ниже перечислены правила, которые позволяют снизить вероятность возникновения взаимоблокировок:

- наименование ресурсов по группам — идентифицируйте взаимосвязанные группы ресурсов и присвойте соответствующим исключающим семафорам сходные имена (например, Funds_Mutex_l и Funds_Mutex_2);
- правильный порядок блокировки — блокируйте ресурсы по номерам от наименьшего к наибольшему;
- правильный порядок разблокирования — разблокируйте ресурсы по номерам от наибольшего к наименьшему.

Если придерживаться этих правил, можно избежать утомительного процесса отладки, который требуется для поиска взаимоблокировки.

Управление дочерними заданиями и задания-зомби

Итак, мы создали несколько потоков и процессов. Всеми дочерними заданиями можно управлять. Вопрос в том, как это делать. Выше уже говорилось о том, что взаимодействовать с дочерними заданиями можно посредством сигналов, передаваемых данных и возвращаемых значений.

Приоритеты и планирование дочерних заданий

Можно понизить приоритет дочернего задания, чтобы другие задания получили больше процессорного времени. Для этого предназначены системные вызовы getpriority() и setpriority() (они доступны только заданиям, имеющим привилегии пользователя root).

В отличие от процессов, практически не имеющих контроля над своими дочерними заданиями, для потоков можно изменять алгоритм планирования, а так-

же отказываться от владения ими. В Linux применяется алгоритм приоритетного кругового обслуживания. В библиотеке Pthreads поддерживаются три алгоритма:

- обычный — аналогичен алгоритму планирования в Linux (принят по умолчанию);
- круговой — планировщик игнорирует значение приоритета, и каждый поток получает свою долю времени, пока не завершится (этот алгоритм применяется в системах реального времени);
- FIFO — планировщик помешает каждый поток в очередь и выполняет его до тех пор, пока он не завершится (этот алгоритм также применяется в системах реального времени).

Уничтожение зомби: очистка после завершения

Возможно, от внимания читателей ускользнул тот факт, что в некоторых из рассмотренных примеров могут появляться процессы-зомби. Это не триллер "Хеллоуин", а настоящий кошмар для любого системного администратора. Появившись в таблице процессов, зомби не исчезают вплоть до перезагрузки компьютера.

Если вы любите риск и не знаете, что такое зомби, попробуйте создать его в своей системе. *Не делайте этого, если не имеете возможности перегрузиться.* Зомби не причиняют вреда системе, но занимают ценные ресурсы (память и место в таблице процессов). Запустите многозадачный эхо-сервер (текст имеется на Web-узле), подключитесь к нему и введите команду "bye". Затем закройте соединение. Теперь введите команду `ps aux | grep` *<имя пользователя>* (заменив параметр своим пользовательским именем). В полученном списке будет присутствовать задание, имеющее статус Z (зомби). Обычно его можно уничтожить, уничтожив предка (эхо-сервер).

Когда процесс завершается, он возвращает целочисленный код. Это значение сигнализирует об успешном окончании работы или о наличии ошибки. Обычно родительское задание дожидается завершения потомка с помощью системного вызова `wait()`. Данная функция принимает от потомка код завершения и передает его родительской программе. Если забыть вызвать эту функцию, дочерний процесс перейдет в режим бесконечного ожидания.

Появление зомби

Предок должен заботиться обо всех своих потомках. Но он не всегда это делает. Если родительский процесс завершился, оставив после себя дочерние процессы, ждущие подтверждения о завершении, управление над ними принимает программа **init**. В ее обязанности входит планирование, выполнение и завершение процессов, однако она не всегда справляется с последней частью задачи. В этом случае в таблице процессов появляются зомби. Их нельзя удалить даже с помощью команды **kill**. (Можно попробовать выполнить команду **init s** или **init 1** для очистки таблицы процессов, но нет гарантии, что она будет работать.)

В отличие от процессов, создаваемых с помощью системного вызова `fork()` или `_clone()`, библиотека Pthreads позволяет отказаться от владения потоком *(отсоединить* его). Отсоединив поток, можно продолжить выполнение програм-

мы, не дожидаясь его завершения. Объявление соответствующей библиотечной функции выглядит так:

```
#include <pthread.h>
int pthread_detach(pthread_t tchild);
```

Параметр tchild является ссылкой, получаемой от функции pthread_create(). В результате выполнения функции могут возникать следующие ошибки.

- ESRCH. Для заданного параметра tchild не найден поток.
- EINVAL. Поток уже был отсоединен.

Общая схема использования функции pthread_detach() такова:

```
/*****************************************************************/
/***   Пример отсоединения потока от родительского задания.    ***/
/***   это позволяет заданию продолжить свою работу,           ***/
/***   не проверяя завершение дочернего потока.                ***/
/*****************************************************************/

/* Создание переменной, содержащей ссылку на поток */
pthread_t tchild;

/* Создание потока */
if ( pthread_create(&tchild, 0, Child_Fn, Child_Arg) != O )
    perror( "Could not create thread");
else
    /* Отсоединяем поток */
    pthread_detach(tchild);
```

После вызова функции pthread_detach() родительское задание может продолжить свою работу. Процессы лишены подобной гибкости, а ведь необязательно ждать завершения каждого потомка — это противоречит принципу распределения полномочий.

Перехватывать сообщения о завершении потомков можно в асинхронном режиме с помощью сигналов. Когда потомок завершается, операционная система посылает родительской программе сигнал SIGCHLD. Все, что требуется сделать, — это перехватить его и вызвать в обработчике функцию wait( ).

```
/*** Пример обработчика сигналов, перехватывающего сообщения  ***/
/*** о завершении дочернего задания.                           ***/
/*****************************************************************/

#include<signal.h>

void sig_child(int sig)
{   int status;

    wait(&status);   /* получаем окончательные результаты */
    fprintf(stderr, "Another one bytes the dust\n");
```

Чтобы иметь возможность получать подобные уведомления, необходимо связать этот обработчик с требуемым сигналом, например:

```
{   struct sigaction act;

    bzero(&act, sizeof(act));
    act.sa_handler = sig_child;
    act.sa_flags = SA_NOCLDSTOP | SA_RESTART;
    sigaction(SIGCHLD, &act, 0); /* перехватываем сигнал
                                    о завершении */
```

Зомби и функция exec()

При запуске внешней программы с помощью функции exec() (рассматривается ниже) следует также перехватывать сигнал о завершении потомка. Может показаться, что внешняя программа больше не связана с родительским заданием, но в действительности ее адресное пространство все еще принадлежит предку.

Узнать о том, как завершился потомок, можно с помощью переменной status. Естественно, родительская программа должна понимать ее значения. Например, потомок может возвращать 0 в случае успешного завершения и отрицательное число при возникновении ошибки.

Обработка кодов завершения, возвращаемых процессами

Полученное значение сдвигается на 8 битов влево, чтобы осталось только 8 значащих битов. Таким образом, если возвращается код 500 (0x01 F4), функция wait() запишет в переменную status значение OxF400, остальные биты будут потеряны.

Значение, возвращаемое функцией wait(), нельзя интерпретировать напрямую. В нем закодирована информация о состоянии процесса, а также собственно код его завершения. Извлечь последний можно с помощью макросов WIFEXITED() и WEXITSTATUS(). Чтобы получить о них более подробную информацию, обратитесь к разделу справочного руководства, касающемуся функции waitpid().

Расширение существующих версий клиента и сервера

Концепции, рассмотренные в этой главе, можно применить к серверам и клиентам из предыдущих глав, чтобы дополнить их процессами и потоками. Например, чтобы сделать эхо-сервер многозадачным, следует добавить в него вызов функции fork() или pthread_create(). В первом случае требуется также включить в программу средства обработки сигналов о завершении дочерних заданий.

После компиляции и запуска программы к ней смогут подключаться сразу несколько клиентов telnet. Это именно то, чего ждут пользователи, — работа с многозадачным сервером в псевдомонопольном режиме.

Вызов внешних программ с помощью функций семейства exec()

Никому не нравится изобретать колесо. Несомненно, удобнее собирать информацию о заданиях *с* помощью команды ps, чем делать это программным способом. Кроме того, писать анализатор строк намного удобнее на Perl, чем на C. Каким образом можно воспользоваться преимуществами готовых программ?

Набор системных вызовов семейства exec() расширяет возможности функции fork(), позволяя вызывать внешние программы и взаимодействовать с ними. Это напоминает использование команд CGI в Web-среде.

- На одном из сетевых узлов содержится CGI-сервер, принимающий команды. У команды могут быть параметры, например:

"http://www.server.com/cgi/<команда>?лараметр1+парметр2+..."

- Клиент посылает серверу сообщение следующего вида:

"http://www.server.com/cgi/ls?tmp+proc"

- Сервер создает дочерний процесс и перенаправляет потоки stdin, stdout и stderr в клиентский канал.
- Затем сервер вызывает указанную программу с помощью функции exec().

Следует отметить, что запуск внешних программ невозможен без многозадачности. Существует пять разновидностей функции exec().

- execl(). Принимает список параметров переменного размера, в котором первым указано полное путевое имя программы. Все следующие параметры являются аргументами командной строки, начиная с нулевого аргумента (arg[0]). Список заканчивается параметром 0 или NULL.

```
/* Например: */
if ( execl("/bin/ls", "/bin/ls", "-aF", "/etc", NULL) != 0 )
    perror("execl failed");
exit(-1);
/* в действительности проверка if избыточна - все функции
   семейства exec() не возвращают значений, если только
   не завершились неуспешно */
```

Первые два параметра функции execl()

Функция execl() выглядит избыточной. Почему первые два параметра одинаковые? Дело в том, что первый параметр указывает на имя исполняемого файла, а второй является нулевым аргументом командной строки (как аргумент arg[0] функции main() в программе на языке C). Это не одно и то же. Не забывайте, что некоторые программы проверяют, под каким именем они вызваны, и в зависимости от этого могут выполнять разные действия.

- execlp(). Аналогична функции exec(), но полное путевое имя не указывается.

```
/* Путь к команде ls ищется в переменной PATH */
if ( execlp("ls", "ls", "-aF", "/etc", NULL) != 0 )
    perror("execlp failed");
exit(-1);
```

- execle(). Аналогична функции execl(), но дополнительный параметр представляет собой массив строк, содержащих установки переменных среды (не забывайте оканчивать его значением 0 или NULL).

```
/* Например: */
char *env[] = {"PATH=/bin:/usr/bin", "USER=gonzo",
               "SHELL=/bin/ash", NULL};
if ( execle("/bin/ls", "/bin/ls", "-aF", "/etc", NULL, env) != 0 )
    perror("execle failed");
exit(-1);
```

- execv(). Принимает два параметра. Первый представляет собой полное путевое имя программы. Второй является массивом аргументов командной строки (последний элемент равен 0 или NULL).

```
/* Например: */
char *args[] = {"/bin/ls", "-aF", "/etc", NULL};
if ( execv(args[0], args) != 0 )
    perror("execv failed");
exit(-1);
```

- execvp(). Аналогична функции execv(), но полное путевое имя не указывается (оно ищется в переменной PATH).

```
/* Например: */
char *args[] = {"ls", "-aF", "/etc", NULL};
if ( execvp(args[0], args) != 0 )
    perror("execvp failed");
exit(-1);
```

В случае успешного завершения функции семейства exec() не возвращаются в программу. Операционная система замещает контекст задания контекстом внешней программы, поэтому не нужна даже проверка if. Если после нее все же выполняются какие-либо инструкции, то это означает, что функция exec() потерпела неудачу.

Чтобы применить эту функцию в рассматриваемом нами сервере, замените цикл ожидания клиентских подключений следующим кодом:

```
/***    Фрагмент программы, в котором принимаются запросы    ***/
/***    на подключение и вызывается внешняя программа        ***/
/***  ("ls -ai /etc").                                       ***/


while(1)
{   int client, addr_size = sizeof(addr);
```

```
    client = acceptfsd, &addr, &addr_size);
    printf("Connected:  %s:%d\n",  inet_ntoa(addr.sin_addr),
                                   ntohs(addr.sin_port));
    if ( fork() )
        close(client);
    else
    {
        close(sd); /* клиенту не нужен доступ к сокету */
        dup2(client, 0); /* замещаем поток stdin */
        dup2(client, 1); /* замещаем поток stdout */
        dup2(client, 2); /* замещаем поток stderr */
        execl("/bin/ls", "/bin/ls", "-al", "/etc", 0);
        perror("Exec failed!"); /* что-то случилось */
    }
}
```

Разница между функциями fork() и vfork()

В некоторых версиях UNIX можно встретить системный вызов vfork(). Его существование связано с тем, что в процессах не происходит совместного использования данных, поэтому потомок принимает копию сегмента данных своего предка. Если же сразу вслед за функцией fork() предполагается вызвать функцию exec(), весь этот блок данных останется в памяти, просто занимая место. Для решения этой проблемы и появилась функция vfork(), которая подавляет создание копии. Однако в Linux применяется алгоритм копирования при записи (родительские страницы виртуальной памяти копируются только в том случае, если предок или потомок модифицирует их). Дочерний процесс, запущенный с помощью функции exec(), не сможет обновить родительские страницы, поэтому система их не копирует. Следовательно, отпадает необходимость в системном вызове vfork(). В Linux он транслируется в вызов fork().

Если сравнить эти примеры с обычной схемой создания дочернего задания, то можно заметить два основных отличия: перенаправление потоков ввода-вывода и вызов функции семейства exec(). Перенаправлять стандартные потоки, включая поток stderr, необходимо, чтобы предоставить внешней программе все данные, которые требуются ей для нормальной работы.

Функции семейства exec() можно вызывать как в процессах, так и в потоках, что очень удобно. Но с потоками связан один нюанс. Поскольку родственные потоки совместно пользуются ресурсами, в них нельзя выполнить независимую переадресацию ввода-вывода. Вследствие этого потоки, в которых вызывается функция exec(), вынуждены делить каналы с внешней программой.

Вызов функции exec() в потоке

Интересный вопрос; что происходит, когда в потоке вызывается функция exec()? Ведь внешняя программа не может заместить контекст потока, не повлияв на работу других связанных р ним потоков. В ядре Linux применяется методика, при которой внешняя программа не получает доступ к данным потока, но совместно с ним использует существующие файловые каналы. Поэтому когда внешняя программа завершается и закрывает дескрипторы каналов stdin и stdout, это изменение отражается на всех остальных потоках. Отсюда вывод: вызывайте функцию fork() перед запуском внешней программы из потока.

Резюме

Многозадачность является средством ускорения работы программ и повышения их гибкости. Процессы и потоки позволяют заметно повысить скорость работы клиентов, не говоря уже о серверах, где многозадачность просто необходима. Переход на многозадачную методику позволяет не только упростить программирование, но и существенно повысить надежность программ.

В данной главе рассматривались вопросы многозадачности и работы с двумя типами заданий: процессами и потоками. Рассказывалось о том, как создавать их на высоком уровне с помощью библиотеки Pthreads или функции fork() и на низком уровне с помощью системного вызова _clone().

Работой потоков можно управлять с помощью средств синхронизации. Благодаря синхронизации можно упорядочить доступ к общим ресурсам и обеспечить целостность данных.

Любое задание может принимать данные через каналы (стандартные или программные) и получать сигналы для обработки асинхронных событий. Создав канал и передав по нему информацию дочернему заданию, можно выгрузить родительскую программу. Кроме того, посредством каналов можно осуществлять переадресацию стандартных потоков ввода-вывода.

В следующей главе будут описаны расширенные концепции ввода-вывода, представляющие собой альтернативу многозадачности.

Механизмы ввода-вывода

Глава 8

В этой главе...

Учитывая современные требования к производительности, не помешает изучить методы экономии времени при работе в сети. Важно помнить, что самый критический, самый лимитированный ресурс компьютера — это центральный процессор. Любому заданию, выполняющемуся в системе, требуется получить доступ к процессору, причем доступ должен быть ограничен небольшими промежутками времени. На самом простом уровне многозадачность реализуется путем квантования времени процессора между заданиями. В Linux используется более динамичный алгоритм планирования, учитывающий зафуженность процессора и задержки в каналах ввода-вывода.

Когда профамма запущена, она либо выполняется, либо ожидает чего-то *(блокирована)*. Существуют два вида блокирования: в ходе операций ввода-вывода (профамма тратит больше времени, осуществляя ввод-вывод) и при доступе к процессору (профамма тратит больше времени на вычисления). В случае приостановки операций ввода-вывода другие задания получают возможность выполнить свои собственные действия. Именно так большинство заданий кооперируется при доступе к ограниченным ресурсам процессора. Когда одно задание ожидает ввода-вывода, оно приостанавливается (блокируется), вследствие чего активизируется другое задание.

Не всегда требуется дожидаться завершения операций ввода-вывода, чтобы продолжить выполнение задания. Представьте, как утомительно было бы работать с броузером, который позволяет закрыть себя только *после* того, как страница будет полностью загружена. А подумайте о клиенте, который одновременно подключается к нескольким серверам и загружает из них данные. При правильном управлении таким конвейером он всегда будет заполнен, поэтому можно добиться максимальной пропускной способности сети.

Если продолжать удерживать контроль над центральным процессором, находясь в режиме ожидания, то можно повысить скорость реакции приложения на запросы пользователя. Чтобы добиться этого, следует перейти в режим неблокируемого ввода-вывода и задать предельное время ожидания (тайм-аут). Для определения степени готовности данных Следует либо опросить подсистему ввода-вывода, либо приказать ядру послать профамме сигнал при наличии данных. Это позволит профамме сосредоточиться на взаимодействии с пользователем или на выполнении основного вычислительного алгоритма.

Блокирование ввода-вывода может применяться в сочетании с многозадачностью или заменять ее. Это еще одно мощное средство в арсенале профаммиста.

Определить, когда следует применять режим блокирования, а когда — нет, не всегда легко. Этой проблеме и посвящена данная глава.

Блокирование ввода-вывода: зачем оно необходимо?

В многозадачной среде можно выполнять множество действий одновременно. Правильно спроектированная профамма должна на 100% подходить для симметричной мультипроцессорной обработки. Но многозадачность имеет свои офаничения и правила. Если их не придерживаться, это отразится на производительности системы.

Одно из правил гласит, что любое задание периодически должно давать возможность выполниться другим заданиям. В системе это правило применяется повсеместно. Например, каждое задание получает свою долю процессорного времени. Если в этот момент задание простаивает, оно отказывается от своей доли. Но как оно узнает, когда нужно что-то сделать? И разве задание не всегда чем-то занято

Ответ будет "и да, и нет". Одной из причин простоя является ожидание завершения какой-либо операции ввода-вывода. Чтобы понять это, сравните скорость процессора со скоростью работы жесткого диска или сети.

Даже в самых быстрых дисковых массивах скорость передачи данных составляет 160 Мбайт/с, а время позиционирования головки — 5 мс. Если задание выполняется на процессоре Pentium III с частотой 500 МГц, для выполнения каждой инструкции в общем случае требуется один такт процессора. В среднем это 2—4 нс. За то время, пока происходит позиционирование головки диска, программа может выполнить 1250000 ассемблерных инструкций (машинных кодов).

Дополнительные задержки в стандарте EnergyStar

Большинство современных компьютерных систем соответствует стандарту Green или *EnergyStar*. Одно из его требований заключается в том, что питание жесткого диска выключается, если к нему нет обращения в течение определенного промежутка времени. Возврат из "спящего" режима обычно занимает 1-2 секунды.

В сети с пропускной способностью 10 Мбит данные передаются от компьютера к компьютеру со средней скоростью 300 Кбайт/с (это оптимальный измеренный показатель для сетей TCP/IP). В сети с пропускной способностью 100 Мбит этот показатель увеличивается лишь в 10 раз (3 Мбайт/с). Даже если не учитывать время маршрутизации и время, требуемое для обработки запроса, клиенту придется ожидать минимум 1000 инструкций (на аналогично сконфигурированной системе). С учетом дополнительных факторов этот показатель смело можно умножать на 1000-5000.

Среднее число машинных кодов, приходящееся на одну строку скомпилированной программы, которая написана на языке С, равно 10. Другими словами, задержка в сети, составляющая 20 мс, эквивалентна 5000000 машинных кодов или 500000 строк на языке С.

Каким образом операционная система справляется с подобными задержками? Попросту говоря, она переводит задание в "спящий" режим *(блокирует* его). Когда обработка системного запроса завершается, задание "пробуждается" и продолжает ВЫПОЛНЯТЬСЯ:

Блокирование заданий, осуществляющих ввод-вывод, — это, скорее, правило, чем исключение. Чтобы узнать, сколько заданий в действительности выполняются в системе, запустите команду ps aux. Те из них, которые находятся в активном режиме, будут помечены буквой R (running). Задания, обозначенные буквой S (stopped), являются остановленными. Вполне вероятно, что активным в списке будет только одно задание — сама команда ps.

Когда следует переходить в режим блокирования?

Программа может блокироваться (перестать выполняться и перейти в режим ожидания), если для завершения какой-либо операции ввода-вывода требуется время. Каждый раз, когда программа делает системный вызов, операционная система может заставить задание дождаться завершения транзакции. В большинстве случаев программа ожидает, когда будет отправлена или получена информация. Блокирование происходит в следующих ситуациях.

- Чтение — блокирование по чтению случается, если данные еще не были получены. Отсутствие даже одного байта может заставить функцию read() завершиться после определенного периода ожидания.
- Запись — блокирование по записи возникает, когда внутренние буферы подсистемы ввода-вывода переполнены, а программа запрашивает передачу следующей порции данных. Linux отдельно выделяет память для буферов всех подсистем ввода-вывода. Они используются с целью временного хранения данных в течение процесса передачи. Если буферы заполнены, все последующие вызовы блокируются до тех пор, пока какой-нибудь из буферов не освободится.
- Подключение — блокирование данного типа происходит, когда функции accept() и connect() не обнаруживают поступивших запросов на подключение в очереди порта. В этом случае можно либо блокировать задание, либо поручить обработку запросов ввода-вывода отдельному потоку или процессу.

Альтернативы блокированию

Какие имеются альтернативы блокированию? Можно заставить программу выполнять другие действия, пока завершается обработка системного запроса. Программа может:

- проверить целостность данных;
- инициировать другие запросы или отслеживать их появление;
- обслуживать несколько других соединений;
- выполнять вычисления, требующие интенсивного использования процессора.

Устранение возможного блокирования может показаться очень привлекательным решением, учитывая, сколько всего можно сделать во время вынужденного простоя. Но в действительности довольно трудно написать неблокируемую программу, если только тщательнейшим образом не проектировать ее с нуля. (Переделать программу так, чтобы в ней не возникало блокирование, слишком сложно и может потребовать огромных усилий.) Избежать блокирования можно с помощью одной из трех методик: опрос каналов, тайм-ауты и асинхронный ввод-вывод.

Асинхронный и сигнальный ввод-вывод

Алгоритмы асинхронного ввода-вывода, представленные в этой главе, в действительности работают на основе сигналов, которые посылаются, когда буферы готовы для чтения или записи. При истинно асинхронном вводе-выводе, который определен в стандарте POSIX.1, никогда не возникает блокирование. Например, вызов функции read() немедленно завершается. Буферы считаются незаполненными до тех пор, пока не завершится операция чтения и программа не получит сигнал. Linux (как и многие другие операционные системы) не соответствует стандарту POSIX.1, касающемуся асинхронного ввода-вывода для сокетов. Чтобы не грешить против истины, лучше употреблять термин "сигнальный ввод-вывод".

Сравнение различных методик ввода-вывода

Чтобы увеличить производительность программы и повысить ее способность к реагированию на запросы, следует творчески подойти к взаимодействию с подсистемой ввода-вывода. Программа может получать данные из внешних источников. Четкий контроль над тем, когда и как следует переходить в режим блокирования, позволит программе оперативнее обрабатывать запросы пользователя. В Linux существуют четыре методики ведения операций ввода-вывода. У каждой из них есть свои преимущества и недостатки. На рис. 8.1 изображена схема выполнения задания, читающего пакет данных четырьмя различными способами. Процесс начинается, когда в буфере еще нет данных.

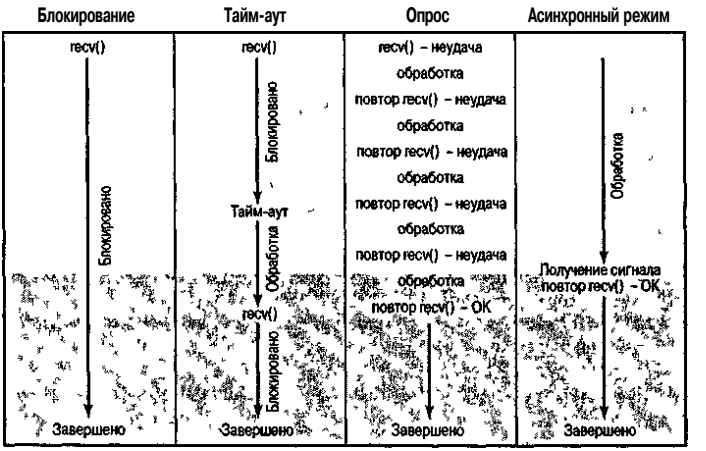

Рис. 8.1. В каждой из методик ввода-вывода у задания разные периоды простоя

Работая в режиме блокирования, программа ожидает поступления данных. В режиме опроса программа вызывает функцию recv() до тех пор, пока данные не появятся. В режиме тайм-аута программа может вообще не получить данные, если они не пришли вовремя. Тем не менее ядро сохраняет сообщение до момента последующего вызова функции recv(). Наконец, в асинхронном режиме ядро посылает программе сигнал о поступлении данных.

Следует отметить, что в случае отправки данных процесс будет выглядеть по-другому. Основную роль здесь играют буферы, заполнения которых вынуждена ожидать программа. В любом случае в режиме блокирования программа останавливается, ожидая, пока ядро полностью отправит сообщение. (Это не совсем верно, так как в действительности ядро ожидает момента, когда можно будет завершить передачу сообщения, после чего сразу же посылает заданию сигнал "пробуждения", но в целях упрощения программирования можно предполагать, что на момент получения сигнала данные уже отправлены.) В режиме опроса и в асинхронном режиме функция send() помещает сообщение в буфер и немедленно завершается. В схеме, изображенной на рис. 8.2, предполагается, что данные слишком велики и не помещаются целиком в буфере.

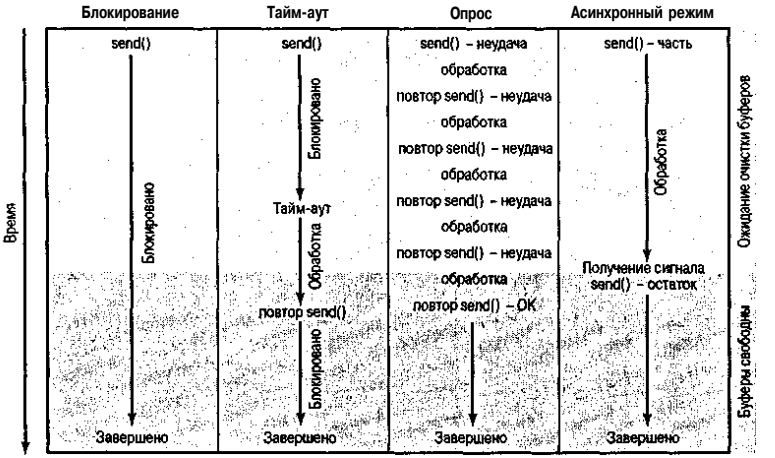

Рис. 8.2. В процессе записи задание выполняется быстрее благодаря наличию внутренних буферов

В следующих разделах рассматриваются особенности каждого из трех неблокируемых режимов.

Опрос каналов ввода-вывода

Программа может выполнять множество других действий во время ожидания пакета. В конце концов, после того как вызвана функция send(), программу не

интересует, как и когда сообщение покинет компьютер. С ее точки зрения подсистема ввода-вывода отправляет сообщение целиком и за один прием. Но часто это не так. Большинство сообщений разбивается на фрагменты из-за ограниченной пропускной способности сети. Точно так же при приеме сообщения оно может поступать по частям, что не всегда желательно. В обоих случаях программа останавливается и дожидается заполнения буфера.

Методика опроса позволяет избежать блокирования при осуществлении ввода-вывода. Ее суть заключается в периодической проверке готовности канала. В первую очередь сокет должен быть переведен в неблокируемый режим. Затем необходимо проверить значение, возвращаемое системным вызовом send() или recv(). Если возникает ошибка EWOULDBLK или EAGAIN, следует повторить попытку позднее. Чаще всего подобные действия выполняются в цикле, например таком:

```
/***                Общий алгоритм опроса                ***/
/*****************************************************/

while( /*** передача данных ***/ )
{
   if ( /*** проверка готовности канала ***/ )
      /*** обработка данных ***/
   /*** другие вычисления ***/
```

Сутью приведенного фрагмента является наличие секции вычислений. Напомним, что в Linux применяется алгоритм приоритетного кругового обслуживания заданий. Если программа больше занята обращением к процессору, чем к каналам ввода-вывода, ее эффективность возрастает. Но если не задать секцию вычислений, возникнет *поглощающий цикл,* в котором приоритет программы увеличивается, хотя она, по сути, ничего не делает. Поглощающие циклы представляют собой серьезную проблему с точки зрения планирования заданий. Программа может занимать ресурсы процессора, выполняя слишком малый объем работы.

Если вам не приходит в голову, что делать в секции вычислений, не используйте дополнительный цикл задержки или функцию sleep(). Они уничтожают суть методики опроса, если только речь не идет о программе, работающей в реальном времени. Вместо них следует либо перейти в режим тайм-аута, либо позволить программе блокироваться.

Чтение данных по методике опроса

Методика опроса позволяет собирать блоки поступающих данных и обрабатывать их по частям, пока передаются остальные данные. Поскольку процессы чтения и обработки обычно связаны между собой, программа должна перевести сокет в режим неблокируемого ввода-вывода. Это делается с помощью функции fcntl():

```
linclude <fcntl.h>
in  fcntl(int  fd,  int command,  int option);
```

Подобно системному вызову read(), эта функция принимает в первом параметре либо дескриптор файла, либо дескриптор сокета. Параметр command должен

быть равен F_SETFL, а параметр option — 0_NONBLOCK. (У этих параметров очень много возможных значений, которые перечислены в приложении B, "API-функции ядра")- Формат использования функции таков:

```
if ( fcntl(sd, F_SETFL, 0_NONBLOCK) != O )
    perror("Fcntl — could not set nonblocking");
```

Для примера предположим, что создается модуль музыкального проигрывателя. Вместо того чтобы ждать, пока весь аудиоклип будет загружен из сети Internet, можно воспроизводить поступающие данные. Естественно, следует также учитывать, что данные могут приходить в сжатом виде. Общий алгоритм программы представлен в листинге 8.1.

Листинг 8.1. Пример чтения данных по методике опроса

```
/*************************************************************/
/***     Пример алгоритма опроса: чтение аудиопотока,      ***/
/***     обработка данных и их воспроизведение             ***/
/*************************************************************/


if ( fcntl(sd, F SETFL, 0 NONBLOCK) != O )
    perror("FcntI -- could" not set nonblocking");

done = 0;
while( !done )
{   int bytes;
    Queue ProcessingQueue;
    Queue OutputQueue;

    /*— Получаем данные из буфера и помещаем их —*/
    /*— в очередь обработки —*/
    if ( (bytes = recv(sd, buffer, sizeof(buffer), 0)) > 0 )
        QueueData(ProcessingQueue,OutputQueue);

    /*— Преобразуем определенное число байтов из  —*/
    /*— очереди обработки в аудиоформат (обычно   —*/
    /*— это выполняется быстрее, чем прием данных —*/
    /*— с помощью функции recv())                 —*/
    ConvertAudio( ProcessingQueue, OutputQueue );

    if ( /*** в выходной очереди накоплено достаточно данных ***/ )
        PlayAudio(OutputQueue);

    /*— Если входной поток закончился —*/
    /*— и выходная очередь пуста —*/
    if ( bytes == 0 && /*— выходная очередь пуста —*/ )
        done = 1;
}
```

Программа запрашивает данные, помещает их в очередь и обрабатывает. Но есть одна проблема: если обе очереди пусты, цикл while станет поглощающим. Чтобы решить эту проблему, можно добавить в программу небольшой цикл задержки, так как она должна работать в реальном времени.

Запись данных по методике опроса

Программный код выполняется в несколько раз быстрее, чем любая функция ввода-вывода. (По сути, чтобы производительность компьютера повышалась, необходимо покупать дополнительную память и заменять медленные устройства ввода-вывода более новыми. Приобретение более быстродействующего процессора не дает такого эффекта.) Поэтому когда программа посылает сообщение или выдает какой-нибудь запрос, операционная система заставляет ее ждать достаточно долго.

Алгоритм записи данных по методике опроса напоминает рассмотренный выше алгоритм чтения, так как подготовка данных, их обработка и отправка осуществляются в одном цикле. Подготовительные действия здесь такие же: вызывается функция `fcntl()` с аналогичными аргументами.

Вспомним: когда в программе чтения обработка данных завершается быстрее, чем будут получены новые данные, возникает поглощающий цикл. В программе записи существует прямо противоположная проблема: подготовка данных может произойти быстрее, чем завершится функция `send()` *(накладка записи)*.

Для примера рассмотрим программу, представленную в листинге 8.2. Она посылает фотографии, получаемые от цифровой камеры, сразу нескольким клиентам.

Листинг 8.2. Пример записи данных по методике опроса

```
/*************************************************************/
/***    Пример алгоритма опроса: отправка изображения      ***/
/***    нескольким клиентам                                ***/
/*************************************************************/


int pos[MAXCLIENTS];
bzero(pos, sizeof(pos));
for ( i = 0; i < ClientCount; i++ )
     if ( fcntl(client[i], F_SETFL, O_NONBLOCK) != 0 )
         perrorf'Fcntl – could not set nonblocking");
...
done = 0;
/*– повторяем до тех пор, пока все клиенты –*/
/*– не получат сообщение целиком –*/
while( !done )
{  int bytes;

   done = 0;
   /*– для всех клиентов –*/
   for { i = 0; i < ClientCount; i++ )
      /*– если имеются неотправленные данные... –*/
      if ( pos[i] < size )
      {
          /*– отправляем сообщение, отслеживая, –*/
          /*– сколько байтов послано –*/
          bytes = send(client[i], buffer+pos[i], size-pos[i], 0);
          if ( bytes > 0 )
          {
              pos[i] += bytes;
              /*--- если сообщение благополучно отправлено –*/
              /*– всем клиентам, завершаем работу –*/
```

```
if ( pos[i] < size )
    done = 0;
```

В этой программе каждое клиентское соединение должно обрабатываться отдельно, поскольку передача данных в разных направлениях может происходить с разной скоростью. Целочисленный массив pos предназначен для хранения номеров позиций в буфере, по которым в настоящий момент выполняется обработка данных для каждого из клиентов.

Еще одна возможная проблема заключается в том, что программа может получить следующее изображение от камеры, в то время как отправка предыдущей фотографии еще не была завершена. Пути решения этой проблемы могут быть такими:

- присоединить новое сообщение к старому;
- отменить последнее сообщение и начать передачу нового;
- изменить частоту работы камеры.

В любом случае выбор зависит от того, насколько критичными являются данные, сколько данных может быть потеряно, насколько неустойчивым является соединение и т.д. Общие соображения по этому поводу приводились в главе 4, "Передача сообщений между одноранговыми компьютерами".

Установление соединений по методике опроса

Одним из редко применяемых алгоритмов опроса является прием запросов на подключение по разным портам. Как правило, серверная программа работает только с одним портом. Но ничто не мешает открыть в программе столько портов, сколько необходимо. Рассмотрим пример:

```
/*** Пример алгоритма опроса: проверка поступления запросов  ***/
/*** на подключение по нескольким портам и создание для      ***/
/*** каждого запроса нового задания                           ***/

...
/*- Установка неблокируемого режима для каждого сокета -*/
for ( i = 0; i < numports; i++ )
     if ( fcntl(client[i], F_SETFL, O_NONBLOCK) != 0 )
         perror("Fcntl -- can't set nonblocking on port #%d", i);

for (;;) /* повторяем бесконечно */
{  int client;

   for ( i = 0; i < numports; i++ )
        if ( (client = accept(sd[i], &addr, &size)) > 0 )
            SpawnServer(sd[i], i);
   /*** служебные действия ***/
```

Представленный в программе алгоритм может показаться очень привлекательным, но в действительности создавать новое задание для каждой функции accept() не совсем удобно, так как требуется выполнить много вспомогательной работы для каждого задания. В действительности наилучшим решением является использование системного вызова select(), который дожидается изменения состояния хотя бы одного из каналов ввода-вывода в указанном наборе дескрипторов (дополнительная информация об этой функции представлена ниже).

Асинхронный ввод-вывод

Проблему файлового ввода-вывода можно решить, если заставить операционную систему сообщать программе о готовности канала обрабатывать запросы. Это позволит программе больше времени тратить на вычисления и меньше — на выполнение системных вызовов.

Режим асинхронного чтения напоминает мерцание лампочки на автоответчике, свидетельствующее о наличии сообщения. Аналогичным образом процедуру обратного вызова можно сравнить с режимом асинхронной записи. В обоих случаях программе посылается уведомление о том, что канал доступен.

В Linux асинхронный ввод-вывод реализуется с помощью сигнала SIGIO (поэтому иногда данный режим называется *сигнальным вводом-выводом)*. Программа получает сигнал SIGIO, когда данные накоплены в буфере чтения или буфер записи готов для приема следующей порции данных. Как и в случае со всеми остальными сигналами, программа не получает никакой дополнительной информации помимо того, что произошло интересующее ее событие. Поэтому если программа работает с двумя каналами, при получении сигнала SIGIO еще не ясно, к какому из каналов он относится.

Другой проблемой являются исключительные ситуации. Ядро не посылает сигнал, если в канале произошла ошибка. В связи с этим не всегда можно обнаружить потерю связи с клиентом, если только программа сама не проверяет состояние канала.

Программа переходит в режим сигнального ввода-вывода, уведомляя ядро о том, что она готова обрабатывать сигнал SIGIO. Когда поступает сигнал, его обработчик устанавливает специальный флаг, понятный программе. В это время в теле программы выполняется цикл, в котором периодически проверяется состояние этого флага. Обработчик сигнала может сам управлять вводом-выводом или предоставлять эту задачу программе.

Чем данный режим отличается от методики опроса? В обоих случаях периодически выполняется проверка готовности канала ввода-вывода. В режиме опроса системные вызовы не блокируются, то же самое имеет место и в асинхронном режиме.

Основное отличие заключается в том, сколько времени программа тратит на выполнение системных вызовов. В асинхронном режиме время работы функций send() и recv() очень невелико, так как ядро само уведомляет программу об их завершении. С другой стороны, в режиме опроса программа вынуждена периодически вызывать их.

В листинге 8.3 представлен общий алгоритм асинхронного ввода-вывода.

Листинг 8.3. Алгоритм асинхронного ввода-вывода

```
/*****************************************************************/
/***      Общий алгоритм асинхронного, или сигнального,       ***/
/***      ввода-вывода                                        ***/
/*****************************************************************/
int ready=0;

void sig_io(int sig)
{
   /*** функция recv(): получаем все данные из буфера ***/
   /*** функция send(): отправляем все обработанные данные ***/
   ready =1; /* сообщаем программе о завершении транзакции */
}

for (;;)
{
   if ( ready > 0 )
   {
      /*** Временно блокируем сигнал SIGIO ***/
      ready = 0;
      /*** функция recv(): копируем данные в буферы
           для обработки ***/
      /*** функция send(): заполняем выходной буфер из очереди
           обработанных данных ***/
      /***  Разблокируем сигнал SIGIO ***/
   }
   /*** Обработка поступающих данных ***/
   /*** -ИЛИ- ***/
   /*** Подготовка новых данных для отправки ***/
}
```

Блокирование сигнала SIGIO может показаться несколько необычным. Оно необходимо из-за того, что обработчик сигнала и основная программа имеют доступ к одной и той же переменной, поэтому фактически в данном месте программы присутствует критическая секция (см. главу 7, "Распределение нагрузки: многозадачность"). Отключение обработчика подобно применению исключающего семафора.

Сообщения, помещенные в очередь

Для обслуживания всех отложенных сообщений может потребоваться несколько раз вызвать в обработчике функцию send( ) или recv(). Ядро может послать программе несколько сигналов одновременно, но сама программа в конкретный момент времени получает сигнал только одного типа. Если два сигнала поступают с очень маленьким интервалом, программа распознает их как один сигнал.

Программа заявляет о том, что она готова принимать сигналы SIGIO, вызывая функцию fcntl(). Эта функция не только переводит сокет в асинхронный режим, но также просит систему направлять сигналы заданному процессу. Например, в следующем фрагменте включается обработка сигналов SIGIO, и система получает указание направлять сигналы текущему заданию.

```
/***        Запуск обработчика сигналов SIGIO        ***/
/*****************************************************/

if ( fcntl(sd, F_SETFL, O_ASYNC | O_NONBLOCK) < 0 )
    PANIC("Can't make socket asynch & nonblocking");
if ( fcntl(sd, F_SETOWN, getpid()) < 0 )
    PANIC("Can't own SIGIO");
```

Когда программа запрашивает получение сигналов ввода-вывода, это касается всех сигналов данной группы, а не только SIGIO. Другим возможным сигналом является SIGURG, посылаемый при передаче внеполосных данных.(Дополнительная информация приводится в главе 9, "Повышение производительности".)

Чтение данных по запросу

Программа может обрабатывать данные в асинхронном режиме по мере их поступления *(по запросу)*. Когда приходит новая порция данных, ядро посылает программе сигнал. В ответ на это программа извлекает данные из буфера и обрабатывает их. Это напоминает конвейерную линию, в которой изделия собираются по мере продвижения.

Лучше всего применять чтение по запросу в тех программах, которые выполняют много вычислений и ожидают поступления данных из одного источника. (Помните: можно открыть несколько каналов ввода-вывода, но придется вручную проверять, для какого из каналов был послан сигнал.) В качестве примера рассмотрим процедуру обработки VRML-документа (Virtual Reality Modeling Language — язык моделирования виртуальной реальности), представленную в листинге 8.4.

Листинг 8.4. Алгоритм асинхронного чтения

```
/*****************************************************************/
/*** Пример асинхронного чтения VRML-документа: обработка      ***/
/*** порции данных во время ожидания следующей порции          ***/

int ready=0, bytes;

void sig_io(int sig)
{
   /*— чтение отложенных сообщений —*/
   bytes = recv(server, buffer, sizeof(buffer), 0);
   if ( bytes < 0 )
       perror("SIGIO");
   ready =1; /* сообщаем программе о завершении транзакции */
}


/*— Разрешаем асинхронный, неблокируемый ввод-вывод —*/
if ( fcntl(sd, F_SETFL, O_ASYNC | O_NTONBLOCK) < 0 )
    PANIC("Can't make socket asynch & nonblocking");
/*— Заявляем о готовности обрабатывать —*/
```

```
/*— сигналы SIGIO и SIGURG —*/
if ( fcntl(sd, F_SETOWN, getpid()) < 0 )
    PANIC("Can't own SIGIO");
while ( Idone )

   if ( ready > 0 )

      /*** Временно блокируем сигнал SIGIO ***/
      ready = 0;
      FillQueue(Queue, buffer, bytes);
      /*** Разблокируем сигнал SIGIO ***/

   /*** Обработка поступающих данных в модуле растрирования ***/
   /*** в течение короткого промежутка времени или до тех   ***/
   /*** пор, пока переменная ready не изменится             ***/
}
```

Избегайте выполнения большого количества действий в обработчике сигналов. В нашем случае обработчик мог бы просто уведомить программу о том, что она должна загрузить данные, но поскольку менеджер очереди может выполнять свои собственные операции ввода-вывода, функции recv() и FillQueue() следует разнести между собой.

Когда буфер считается пустым?

Когда уровень заполнения входного буфера достигает нижней отметки (минимальное число байтов, после которого система начинает посылать сигнал SIGIO), программе посылается уведомление. Этот уровень по умолчанию равен 1, т.е. всякий раз, когда в буфер записывается хотя бы один байт, программе будет отправлен сигнал. В некоторых системах это значение можно увеличить с помощью функции setsockopt{) (описана в главе 9, "Повышение производительности"). Но похоже, что в Linux данная возможность не поддерживается.

Асинхронная запись данных

Асинхронная запись реализуется немного по-другому, чем асинхронное чтение. Когда программа впервые вызывает функцию send(), то существует вероятность, что все сообщение целиком поместится в буферах ядра. Но возможно также, что этого не произойдет и сохранена будет только часть данных. Поэтому необходимо проверять код завершения каждой функции send(), чтобы выяснить, отправлены ли данные.

В качестве примера можно привести запрос на выборку к базе данных. Его результаты можно принимать и обрабатывать достаточно долго, посылая по ходу запросы на модификацию базы данных. В листинге 8.5 представлен общий алгоритм асинхронной записи.

Листинг 8.5. Алгоритм асинхронной записи

```
/***     Пример асинхронной записи: генерирование ответа     ***/
/***     в процессе обработки запроса                         ***/
```

```
int ready=0, bytes, size=0, pos=0;

void sig_io(int sig)
{
   if ( size > 0 )
   {
       bytes = send(client, buffer, size+pos, 0);
       if ( bytes < 0 )
       {
           pos += bytes;
           ready = 1;
       }
   }

/*— Разрешаем асинхронный, неблокируемый ввод-вывод и —*/
/*— заявляем о готовности обрабатывать сигнал SIGIO —*/
while ( Idone )

   if ( /*** канал доступен ***/ )
      /*** отправляем сообщение ***/
   else
      /*** помещаем сообщение в очередь ***/
}
```

Каждый раз, когда канал свободен и готов передавать данные, ядро посылает процессу сигнал SIGIO. В ответ на это происходит передача очередной порции данных.

Подключение по запросу

Теоретически можно создать сервер, устанавливающий соединения с клиентами по запросу. Это напоминает алгоритм чтения по запросу, но вместо чтения файла программа вызывает функцию accept(). Для взаимодействия с основной профаммой обработчику сигналов требуются три глобальные переменные: дескриптор текущего сокета, массив дескрипторов и текущее число соединений. Все локальные переменные будут потеряны при завершении обработчика. Текст обработчика может выглядеть так:

```
/*** Пример подключения по запросу: установление соединения    ***/
/*** в обработчике сигналов. В основной программе в цикле      ***/
/*** опрашиваются все сокеты. (Взято из файла demand-accept.c  ***/
/*** на Web-узле.)                                             ***/


int   Connections[MAXCONNECTIONS];
int sd,  NumConnections=0;
void  sig_io(int  sig)
{  int client;
```

```
    /*- прием запросов на подключение; если запросов -*/
    /*- слишком много, выдаем сообщение об ошибке    -*/
    /*- и разрываем связь                            -*/
    if ( (client = acceptf sd, 0, 0) > 0 )
        if ( NumConnections < MAXCONNECTIONS )
            Connections [NumConnections++] = client;
        else
        {
            sendfclient, "Too many connections !\n, 22, 0");
            close(client);
        }
    else
        perror("Accept");
}
```

Хотя такой сервер может оказаться полезным, преимущество многозадачности заключается в том, что она позволяет, во-первых, сохранять значения локальных переменных, а во-вторых, обрабатывать каждый запрос на подключение по отдельности, по-своему.

Устанавливать соединения по запросу трудно с точки зрения программирования и не дает особых преимуществ, если только сервер не перегружен слишком большим числом заданий (программа создает столько процессов, что ее производительность падает). Гораздо проще работать в многозадачном режиме или применять методику опроса.

Устранение нежелательного блокирования с помощью функций poll() и select()

В Linux есть две функции, которые помогают работать с несколькими открытыми каналами одновременно. Они обеспечивают более высокую эффективность (и более просты в работе), чем при самостоятельном опросе каналов. Идея этих функций заключается в том, что системный вызов блокируется до тех пор, пока не изменится состояние любого из каналов.

Под изменением состояния подразумеваются различные события, включая наличие данных для чтения, доступность канала для записи и возникновение ошибки. Если функция завершилась, значит, состояние одного или нескольких каналов изменилось. Обе функции возвращают число каналов, подвергшихся изменению.

Функция select() довольно сложна. С ней связан ряд дополнительных макросов, управляющих списками дескрипторов.

```
#include <sys/time.h>
#include <sys/types.h>
linclude <unistd.h>
int select(int maxfd, fd_set *to_read, fd_set *to write,
           fd_set *except, struct timeval *timeout);
```

```
FD_CLR(int fd, fd_set *set);    /*. удаляет дескриптор из списка */
FD_ISSET(int fd, fd_set *set);              /* проверяет наличие
                                              дескриптора в списке */
FD_SET(int fd, fd_set *set);  /* добавляет дескриптор в список */
FD_ZERO(fd_set *set);      /* инициализирует список дескрипторов */
```

Описание параметров функции select() приведено в табл. 8.1.

Таблица 8.1. Параметры функции select() и связанных с ней макросов

Параметр	Описание
maxfd	Число, на единицу большее номера самого старшего дескриптора в списке
to_read	Список дескрипторов каналов, из которых производится чтение данных
to_write	Список дескрипторов каналов, в которые осуществляется запись данных
except	Список дескрипторов каналов, предназначенных для чтения приоритетных сообщений
timeout	Число микросекунд, в течение которых необходимо ждать
fd	Дескриптор добавляемого, удаляемого или проверяемого канала
set	Список дескрипторов

Параметр maxfd равен порядковому номеру старшего дескриптора в списке плюс 1. Каждому заданию выделяется пул дескрипторов ввода-вывода (обычно 1024), а каждому из его каналов назначается один дескриптор (первыми идут стандартные каналы: stdin — 0, stdout — 1, stderr — 2). Если в список to_read входят дескрипторы [3,6], в список to_write — дескрипторы [4—6], а в список except — дескрипторы [3,4], то параметр maxfd будет равен 6 (номер старшего дескриптора) плюс 1,т.е. 7.

Параметр timeout задает временной интервал, в течение которого функция выполняется. Он может принимать следующие значения:

- NULL — функция ждет бесконечно долго;
- положительное число — функция ждет указанное число микросекунд;
- нуль — после проверки всех каналов функция немедленно завершается.

Предположим, в программе открыты три канала с дескрипторами [3,4,6], по которым посылаются данные:

```
/*****************************************************************/
/***                  Пример функции selectf)                  ***/
/*****************************************************************/
int count;
fd_set set;
struct timeval timeout;

FD_ZERO(&set);   /* очищаем список */
FD_SET(3, &set); /* добавляем канал #3 */
FD_SET(4, &set); /* добавляем канал #4 */
FD_SET(5, &set); /* добавляем канал #5 */

timeout.tv_sec = 5; /* тайм-аут длится 5,25 с */
```

```
timeout.tv_usec = 250000;

/*— Ожидаем завершения функции select() —*/
if ( (count = select(6+1, &set, 0, 0, &itimeout)) > 0 )
    /*** Находим канал, состояние которого изменилось ***/
else if ( count == 0 )
    fprintf(stderr, "Timed out!");
else
    perror("Select");
```

В данном примере если функция select() возвращает положительное число, значит, в одном из трех каналов появились данные для чтения. Чтобы определить, какой именно из каналов готов, следует написать дополнительный код. Если функция возвращает 0, был превышен интервал ожидания, равный 5,25 секунды.

Функция poll() проще, чем select(), и ею легче управлять. В ней используется массив структур, определяющих поведение функции:

```
struct pollfd
{
    int   fd;          /* проверяемый дескриптор */
    short events;  /* интересующие нас события */
    short revents; /* события, которые произошли в канале */
}
```

В первом поле, fd, содержится дескриптор файла или сокета. Второе и третье поля, events и revents, являются битовыми масками, определяющими события, за которыми требуется следить.

- POLLERR. Любая ошибка. Функция завершается, если в канале возникла ошибка.
- POLLHUP. Отбой на другом конце канала. Функция завершается, если клиент разрывает соединение.
- POLLIN. Поступили данные. Функция завершается, если во входном буфере имеются данные.
- POLLINVAL. Канал fd не был открыт. Функция завершается, если канал не является открытым файлом или сокетом.
- POLLPRI. Приоритетные сообщения. Функция завершается, если поступило приоритетное сообщение.
- POLLOUT. Канал готов. Функция завершается, если вызов функции write() не будет блокирован.

Объявление функции poll() выглядит так:

```
linclude <sys/poll.h>
int poll(struct pollfd *list, unsigned int cnt, int timeout);
```

Программа заполняет массив структур pollfd, прежде чем вызвать функцию. Параметр cnt определяет число дескрипторов в массиве. (Учтите, что массив дескрипторов должен быть непрерывным. Если один из каналов закрывается, необходимо переупорядочить, т.е. сжать, массив. Не во всех системах функция poll()

может работать с пустыми структурами.) Параметр timeout аналогичен одноименному параметру функции select(), но выражается в миллисекундах.

Обе функции, select() и poll(), позволяют одновременно контролировать несколько каналов. Тщательно спроектировав программу, можно избежать некоторых проблем избыточности, связанных с многозадачностью.

Реализация тайм-аутов

Вместо опроса каналов можно сообщить операционной системе о том, что ждать требуемого события нужно не дольше указанного времени. В Linux это нельзя сделать напрямую. Существуют два способа реализации механизма тайм-аутов:

- путем задания параметра timeout в функции select() или poll();
- путем посылки программе сигнала SIGALRM по истечении заданного времени.

Поддержка тайм-аутов для сокетов в Linux

Среди атрибутов сокета есть значения тайм-аутов для операций чтения (SO_RCVTIMEO) и записи (SQ_SNDTIMEO). К сожалению, в настоящее время эти атрибуты нельзя модифицировать.

Функция select() задает значение тайм-аута в микросекундах, а функция poll() — в миллисекундах. Эти функции использовать проще всего, но тайм-аут будет применен ко всем каналам, перечисленным в списке.

Подобная ситуация может представлять проблему. Предположим, что имеются три канала, по одному из которых связь очень плохая. Посредством этих функций невозможно определить, что произошло в проблемном канале: тайм-аут или закрытие из-за отсутствия связи. Можно самостоятельно определять время работы функции и сравнивать его с параметром timeout, но это требует большого количества дополнительных действий.

В качестве альтернативы можно воспользоваться сигналом таймера SIGALRM. *Таймер* — это часы, которые запущены в ядре. Когда таймер срабатывает, ядро посылает заданию команду пробуждения в виде сигнала. Если задание в этот момент находится в ожидании завершения системного вызова, сигнал прерывает соответствующую функцию, вызывается обработчик сигналов и генерируется ошибка EINTR.

Любое задание может послать сигнал самому себе, но оно должно знать, как его обрабатывать. В некоторых системах тайм-аут задается с помощью функции alarm(), но предварительно следует включить обработку сигнала SIGALRM. Рассмотрим пример:

```
/*** Пример реализации тайм-аута с помощью функции alarm(). ***/
/*** (Взято из файла echo-timeout.c на Web-узле.)            ***/
/*****************************************************************/

int sig_alarm(int sig)
{/*** ничего не делаем ***/}
```

```
void reader()
{   struct sigaction act;

    /*— Инициализируем «структуру —*/
    bzero(&act, sizeof(act));
    act.sa_handler = sig_alarm;
    act.sa_flags = SA_ONESHOT;

    /*— Активизируем обработчик сигналов,—*/
    if ( sigaction(SIGALARM, &act, 0) != 0 )
        perror( "Could not set up timeout");
    else
        /*— Если обработчик активизирован, —*/
        /*— запускаем таймер —*/
        alarm(TIMEOUT_SEC);

    /*— Вызываем функцию, которая может —*/
    /*— завершиться по тайм-ауту —*/
    if ( recv(sd, buffer, sizeof(buffer), 0) < 0 )
    {
        if (errno == EINTR)
            perror ("Timed out!");
    }
}
```

В этой программе активизируется обработчик сигналов, запускается таймер, после чего вызывается функция recv(). Если за указанный промежуток времени функция не прочитает данные, программа получит сигнал SIGALRM. Выполнение функции recv() будет прервано, а в библиотечную переменную errno запишется код ошибки EINTR.

Чтобы добиться этого, из поля sa_flags структуры sigaction следует удалить флаг SA_RESTART. В главе 7, "Распределение нагрузки: многозадачность", говорилось о том, что при обработке сигналов данный флаг следует задавать. Но как видно из примера, это не относится к режиму тайм-аутов.

И последнее замечание: избегайте использовать функцию alarm() вместе с системным вызовом sleep(), так как это может привести к возникновению проблем. Следует либо обрабатывать сигнал SIGALRM, либо вызывать функцию sleep(), но не смешивать их.

Резюме: выбор методик ввода-вывода

Сетевая программа взаимодействует с другими программами по сети. Поскольку производительность и время ответа каждого компьютера различны, важно знать, когда и как переходить в режим неблокируемого ввода-вывода, чтобы увеличить производительность работы клиентов и серверов и повысить их восприимчивость к командам пользователя.

В этой главе рассматривались вопросы, связанные с блокированием ввода-вывода (что это такое, когда оно необходимо и для чего оно нужно), а также описывались методики, позволяющие его избежать. Мы познакомились с функцией fcntl(), ифающей важнейшую роль при переходе в режим неблокируемого ввода-вывода.

Две основные методики неблокируемого ввода-вывода — это режим опроса и асинхронный режим. В первом случае периодически вызывается функция, осуществляющая чтение или запись данных. В асинхронном (или сигнальном) режиме основная нафузка возлагается на ядро системы, которое должно отслеживать, когда канал освобождается. В режиме опроса профамма больше сосредоточена на вводе-выводе, тогда как в асинхронном режиме больше времени уделяется локальным вычислениям.

Еще одним важным инструментом являются тайм-ауты. Их можно реализовать посредством функции select() или poll() либо с помощью сигнала таймера, который "будит" программу, если системный вызов выполняется слишком долго.

Глава

9 Повышение производительности

В этой главе...

Как добиться максимальной производительности сервера или клиента? В библиотеке Socket API имеется ряд средств, позволяющих решить эту задачу достаточно просто. Тем не менее следует рассмотреть проблему под разными углами, поскольку все ее аспекты тесно связаны друг с другом.

В сетевой программе можно выделить три основных компонента, требующих отдельного анализа: задание (главный процесс), соединение (сокет) и сообщение. Первый компонент отвечает за управление всеми связанными процессами и потоками. Второй компонент — это сам сокет и его параметры. Третий компонент — это процедуры взаимодействия с каналами ввода-вывода, отвечающие за прием и передачу сообщений. Чтобы добиться максимальной производительности от сетевой программы, необходимо сбалансировать работу первого и третьего компонентов. Что касается второго компонента, то с помощью параметров сокета можно настроить соединение под особенности решаемой задачи.

В этой главе приводятся практические советы относительно того, как повысить производительность приложения.

Подготовка к приему запросов на подключение

К настоящему моменту мы узнали, как создать сервер, как сделать его многозадачным и как управлять вводом-выводом. В главе 7, "Распределение нагрузки: многозадачность", рассказывалось о том, как поручать задачу приема и обслуживания запросов на подключение дочерним процессам, создавая их по мере необходимости. Однако в некоторых системах создавать каждый раз новое задание расточительно с точки зрения времени и ресурсов. Да и не всегда хочется писать отдельную программу, управляющую только сетевыми клиентами. В любом случае необходимо контролировать как размер серверной программы, так и число одновременных подключений, что требует дополнительного программирования.

Ограничение числа клиентских соединений

Как описывалось в предыдущих главах, сервер создает новый дочерний процесс, когда от клиента поступает запрос на подключение. Нам потребуется модифицировать нашу программу, чтобы в ней не создавалось слишком много процессов (иногда их называют *процессами-"кроликами",* поскольку они размножаются слишком быстро). Ниже приведен стандартный алгоритм.

```
/*************************************************************/
/***        Стандартный алгоритм обслуживания клиентов      ***/
/*************************************************************/
int sd;
struct sockaddr_in addr;

if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0 )
    PANIC("socket()  failed");
bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(MY_PORT);
```

```
addr.sin_addr = INADDR_ANY;
if (bind(sd, &addr, sizeof(addr)) != 0 )
    PANIC("bind()  failed");
if (listen(sd, 20)  != 0)
    PANIC ("listen() failed");
for (;;)
{   int client, len=sizeof(addr);
    client = accept(sd, &addr, &len);
    if (client > 0)
    {  int pid;
        if ( (pid = fork()) == 0 )
        {
            close(sd);
            Child(client); /*   Обслуживаем нового клиента   */
        }
        else if( pid > 0 )
            close(client);
        else
            perror("fork() failed");
    }
}
```

Этот алгоритм приводился в главе 7, "Распределение нагрузки: многозадачность". Он хорошо подходит для обслуживания длительных соединений, таких как процесс регистрации пользователя в системе, сеанс работы с базой данных и т.д. Однако у него есть один существенный недостаток: что если исчерпается список доступных идентификаторов процессов? Что произойдет, если ресурсов ОЗУ окажется недостаточно для обслуживания существующего числа процессов и программе придется выгружать часть данных в файл подкачки? Эта проблема характерна для процессов-"кроликов".

Разница между ОЗУ и виртуальной памятью

Может показаться, что выделение для файла подкачки большого раздела на диске решает многие проблемы в системе, но это обманчивое впечатление. На самом деле, чтобы добиться высокой производительности, необходимо держать все активные процессы вместе со своими данными в ОЗУ. Если процесс был выгружен на диск, то время, затрачиваемое ядром на его последующую загрузку обратно в память, сводит на нет скорость работы программы. Для решения проблемы производительности необходимо в первую очередь определить, какие ограничения существуют с точки зрения памяти, процессора и устройств ввода-вывода.

Можно осуществлять подсчет числа процессов:

```
/*************************************************************/
/***       Алгоритм, в котором ограничено число потомков.      ***/
/***          (Взято из файла capped-servlets.c на Web-узле.)      ***/
/*************************************************************/
int ChildCount=0;

void sig_child(int sig)
{
    wait(0);
    ChildCount--;
}
```

```
for (;;)
{   int client, len=sizeof(addr);
    while ( ChildCount >= MAXCLIENTS )
       sleep(1);
    client = accept(sd, &addr, &len);
    if ( client > 0 )
    { int pid;
        if ( (pid = fork()) == 0 )
        {/*- Потомок -*/
            close(sd);
            Child(client);  /*- Обслуживаем нового клиента -*/
        }
        else if ( pid > 0 )
        {/*- Предок -*/
            ChildCount++;
            close(client) ;
        }
        else
            perror("fork() failed");
    }
}
```

В этом примере программа отслеживает число дочерних процессов. Когда их становится слишком много, программа просто переходит в "спяший" режим, пока соответствующее число соединений не будет закрыто и не завершатся "лишние" процессы. Реализовать этот метод несложно. К переменной ChildCount обращаются только сервер и его обработчик сигналов, поэтому можно не беспокоиться о возникновении состояния гонки.

Опять-таки, этот алгоритм хорошо работает в случае длительных соединений. Если же имеется множество коротких соединений, предназначенных для выполнения одной-единственной транзакции, накладные расходы, связанные с созданием и уничтожением дочерних процессов, могут свести производительность сервера к нулю. Идея заключается в том, чтобы иметь набор уже выполняющихся процессов, ожидающих поступления запросов на подключение.

Предварительное ветвление сервера

На сервере могут выполняться 5—20 процессов, предназначенных для приема запросов от клиентов. Именно так распределяет нагрузку HTTP-сервер. (Если в системе инсталлирован и запущен демон httpd, загляните в таблицу процессов, и вы обнаружите, что выполняется несколько его экземпляров.) Как потомок получает необходимую информацию, если в процессах не происходит совместного использования данных? Ответ заключен в следующем фрагменте программы:

```
/*************************************************************/
/***                Фрагмент дочернего процесса            ***/
/*************************************************************/
if ( (pid = fork()) == 0 )
{
    close(sd);
    Child(client) ;  /*- Обслуживаем нового клиента -*/
}
```

Как можно заметить, дочерний процесс закрывает дескриптор сокета (sd). Но ведь клиентский дескриптор идентифицирует само соединение! Зачем процесс его закрывает?

Дело в том, что, когда создается дочерний процесс, он принимает дескрипторы открытых файлов от своего родителя. Если какой-либо из файлов потомку не нужен, он может его закрыть — это не повлияет на работу предка. В то же время файл можно вполне оставить открытым и продолжать читать или записывать в него данные параллельно с родительским процессом. Вот как создается несколько процессов, обслуживающих один и тот же сокет:

```
/***   Создание набора серверов-потомков, ожидающих запросов   ***/
/***   на установление соединения.                              ***/
/***   (Взято из файла preforking-servlets.c на Web-узле.)      ***/
/*****************************************************************/
int  ChildCount=0;

void sig_child(int sig)
{
    wait(0);
    ChildCount--;


main()

    /*** Регистрируем обработчик сигналов;
         создаем и инициализируем сокет    ***/
    for (;;)

        if ( ChildCount < MAXCLIENTS )

            if  (  (pid = fork())  ==  0  )  /*- Потомок -*/
                for (;;)

                    int client = accept(sd, 0, 0);
                    Child(client);  /*- Обслуживаем нового
                                       клиента   -*/

            else if  ( pid > 0  )  /*- Предок -*/
                ChildCount++;
            else
                perror("fork()  failed");

        else
           sleep(1);
    }
}
```

Здесь применяется обратный порядок вызова функций accept() и fork(): вместо того чтобы создавать новый процесс после подключения, программа сначала запускает дочерний процесс, который затем ожидает поступления запросов.

В этом фрагменте может быть создано, к примеру, 10 процессов. Все они входят в бесконечный цикл ожидания запросов. Если запросы не поступают, про-

цессы блокируются (переходят в "спящий" режим). Когда появляется запрос, все процессы "пробуждаются", но только один из них устанавливает соединение, а остальные девять снова "отходят ко сну". Подобный цикл продолжается бесконечно долго.

Что произойдет, если один из процессов завершит работу? Именно по причине возможности такого события в родительском процессе не происходит вызова функции accept(). Основная задача предка состоит в том, чтобы поддерживать нужное число потомков.

В первую очередь предок задает лимит дочерних процессов. Когда один из них завершается, обработчик сигналов получает уведомление и понижает на единицу значение счетчика процессов. В результате родительский процесс создает нового потомка. Если лимит потомков исчерпан, родительский процесс переходит в "спящий" режим, отдавая свою долю процессорного времени другому заданию.

Адаптация к различным уровням загруженности

Главная обязанность родительского задания заключается в том, чтобы гарантировать наличие достаточного количества дочерних процессов, обслуживающих поступающие запросы на подключение к серверу. При этом следует соблюдать баланс в отношении загруженности системных ресурсов. Если не проявить бдительность, то начиная с определенного момента времени клиенты не смогут получить доступ к серверу из-за отсутствия обслуживающих процессов.

Наличие заранее заданного числа процессов, принимающих запросы от клиентов (такие процессы называются *сервлетами),* ограничивает количество активных соединений. Тем не менее это хорошая идея, поскольку предотвращается "раздувание" таблицы процессов программы и снижается нагрузка на системные ресурсы. Как минимизировать число создаваемых процессов и в то же время удовлетворить потребности клиентов?

Можно сделать алгоритм создания сервлетов адаптируемым к уровню загруженности системы. Сервер должен решать две разные задачи: он должен определять, когда допускается создавать новые сервлеты и когда требуется уничтожать лишние процессы. Вторая задача проще: процесс-сервлет сам себя уничтожает, если находится в неактивном режиме дольше определенного промежутка времени.

Первую задачу нельзя решить напрямую. Вспомните, что у каждого TCP-сервера имеется очередь ожидания, создаваемая с помощью системного вызова listen(). Сетевая подсистема помещает в нее каждый поступающий запрос на подключение, а уже функция accept() принимает запрос и создает выделенный канал связи с клиентом. Если клиент ожидает своей очереди слишком долго, его терпение в конце концов заканчивается.

Чтобы правильно настроить работу сервера, было бы неплохо иметь возможность определить, сколько отложенных запросов находится в очереди и какой из них пребывает в ней дольше всего. Подобная информация позволила бы выяснить степень реагирования системы на поступающие запросы. Когда очередь заполняется, сервер может создать новые сервлеты. Если же запрос находится в очереди слишком долго, можно скорректировать предельное время пребывания сервлета в неактивном состоянии. К сожалению, подобный алгоритм не реализуется с помощью API-функций. Необходимо использовать другой подход.

Один из возможных методов напоминает ловлю рыбы на приманку. Рыбак периодически забрасывает наживку, а рыба ее съедает. Частота, с которой повторяется этот процесс, позволяет определить, сколько рыбы водится в пруду или данном месте реки.

Словно на рыбалке, сервер периодически "забрасывает" новый сервлет. Если в ответ на создание определенного количества сервлетов аналогичное число сервлетов завершается (из-за слишком долгого пребывания в неактивном состоянии), сервер продолжает данный процесс. Когда сервлеты перестают завершаться, сервер увеличивает число создаваемых сервлетов, пока ситуация не стабилизируется.

Суть алгоритма предварительного ветвления заключается в том, чтобы минимизировать время, затрачиваемое на создание сервлетов и освобождение ресурсов после их завершения. Если бессистемно создавать и уничтожать сервлеты, внезапно может оказаться, что функция `fork()` просто вызывается для каждого нового соединения, как и раньше. Необходимо тщательно спроектировать программу, чтобы избежать этого. Кроме того, требуется свести к минимуму число выполняющихся сервлетов.

Именно здесь на помощь приходит статистика. В адаптивном алгоритме возможны три ситуации: число соединений стабильно, растет или уменьшается. В первом случае в течение заданного промежутка времени число созданных сервлетов равно числу завершившихся. Например, если сервер порождает один сервлет каждые 60 секунд (с 30-секундным предельным временем простоя), схема этого процесса будет соответствовать верхней части рис. 9.1.

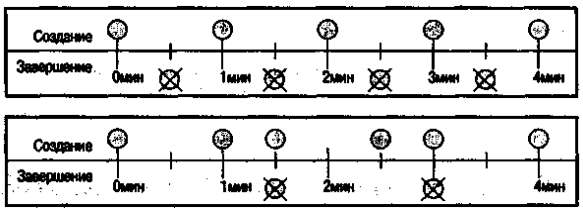

Рис. 9.1. Когда имеются необслуженные запросы, сервер создает дополнительные сервлеты

Если число соединений увеличивается, то количество завершившихся сервлетов будет меньше, чем созданных. Отсутствие в течение заданного промежутка времени сигнала о завершении сервлета можно считать признаком того, что он обслуживает запрос на подключение. В этом случае сервер удваивает скорость создания сервлетов (см. рис. 9.1, внизу). Процесс продолжается до тех пор, пока не будет достигнута максимальная частота вызова функции `fork()` (к примеру, каждые 5 секунд).

Наконец, если число соединений начинает уменьшаться, дополнительные сервлеты станут не нужны и сервер начнет получать сигналы об их завершении. В этом случае необходимо восстановить нормальную частоту создания сервлетов.

```
/***          Адаптивный алгоритм создания сервлетов          ***/
/***     (Взято из файла servlet-chumner.c на Web-узле.)      ***/
/*****************************************************************/
```

```
int delay=MAXDELAY;                        /* например, 5 секунд */
time_t  lasttime;
void  sig_child(int  signum)
{
    wait(0);                              /* подтверждаем завершение  */
    timej&lasttime);                     /* определяем текущее время */
    delay =MAXDELAY;                  /* сбрасываем значение задержки */
}

void Chuiraner(void  (*servlet)(void))
{
    time(&lasttime)  ;            /* запоминаем дату начала процесса */
    for (;;)
    {
        if ( !fork() )                      /* создаем новый сервер */
            servlet();  /* вызываем потомка (должен завершиться/
                                        с помощью функции exit()) */
        sleep(delay);                             /* берем тайм-аут */
        /* если ни один из сервлетов не завершился,
           удваиваем частоту */
        if ( times[0] - timesll] >= delay - 1 )
            if ( delay > MINDELAY )        /* не опускаемся ниже
                                              минимального порога */
                delay /= 2;                  /* удваиваем частоту */
    }
}
```

В этом фрагменте программы демонстрируется, как управлять группой сервлетов, проверяя время последнего завершения сервлета. При завершении также происходит сброс таймера (переменной delay), чтобы процесс начался заново.

Расширение возможностей сервера с помощью функции select()

Применение процессов для управления соединениями — это достаточно понятный и эффективный способ распределения задач на сервере. Эффективность обработки запросов еще более повышается, если ограничить число одновременно выполняющихся процессов и осуществлять их предварительное ветвление. Однако в рассмотренном алгоритме есть одна фундаментальная проблема: стихийное "затопление" планировщика лавиной процессов.

Давиноподобная загрузка планировщика

В действительности существуют две проблемы, связанные с ограничениями алгоритмов многозадачности. Они позволяют понять, почему однопроцессорные системы уходят в небытие. Первая проблема возникает, когда сервлетам приходится иметь дело с большой таблицей процессов. Вторая заложена в самой концепции предварительного ветвления.

Таблица процессов легко может включать несколько сотен процессов. В зависимости от объема имеющейся оперативной памяти и размера файла подкачки число процессов может быть еще большим. Но подумайте о следующем ограничении: Linux переключается между процессами каждые 10 мс. Сюда еще не входят задержки, связанные с выгрузкой контекста старого процесса и загрузкой нового.

Например, если работают 200 сервлетов, каждый процесс будет ждать 2 с, чтобы получить маленькую долю процессорного времени в системе, где переключение задач происходит каждые 0,01 с. При условии, что в этом промежутке передается 1 Кбайт данных, общая скорость передачи данных составит 200 Кбайт/с в сети с пропускной способностью 10 Мбит/с (примерная норма в сетях TCP/IP). Но в каждом конкретном соединении скорость передачи будет лишь 512 байт/с. Это основное следствие неконтролируемого роста числа процессов.

Вторая проблема связана с сутью механизма предварительного ветвления: несколько процессов могут ожидать одного и того же запроса на подключение. Предположим, сервер создал 20 сервлетов и заблокировал их в ожидании запроса. Когда приходит запрос, ядро посылает сигнал пробуждения всей двадцатке, но только один сервлет (самый первый из пробудившихся) принимает запрос, а остальные возвращаются в блокированное состояние. Некоторые программисты называют это *лавинным сходом процессов*. Данная проблема подчеркивает необходимость минимизации числа ожидающих сервлетов и адаптации их количества к уровню загруженности сервера.

Решение проблемы "лавинного схода" в ядре

Трудность, связанная с написанием книги по такой стремительно меняющейся системе, как Linux, заключается в том, что информация очень быстро устаревает. В Linux 2.4 проблема "лавинного схода" решена за счет того, что ядро посылает сигнал пробуждения только одному из заблокированных процессов.

Чрезмерное использование функции select()

Одно из решений перечисленных проблем заключается в отказе от многозадачности. В предыдущей главе рассказывалось о двух системных вызовах, позволяющих реализовать переключение каналов ввода-вывода: select() и poll(). Эти функции могут заменить собой сервлеты.

Вместо того чтобы создавать 20 сервлетов, ожидающих поступления запросов на подключение, можно обойтись всего одним процессом. Когда приходит запрос, процесс помещает дескриптор соединения (клиентского канала) в список дескрипторов ввода-вывода, после чего функция select() или poll() завершается, возвращая список программе. Это дает серверу возможность принять и обработать запрос.

Однако соблазн отказаться от многозадачности и перейти к методике опроса каналов может привести к потере производительности. Когда за обработку информации отвечает одно задание, серверу приходится периодически ожидать ввода-вывода. При этом теряется возможность выполнять другие действия. К примеру, тот, кому когда-либо приходилось компилировать ядро, наверняка замечал, что компиляция протекает быстрее, если задать многозадачный режим с помощью опции -j утилиты make. (При наличии достаточного количества оперативной памяти распределение обязанностей между двумя или тремя заданиями на каждый процессор позволяет существенно сократить время компиляции.)

Разумное использование функции select()

При создании мощного многопользовательского сервера распределять нагрузку можно различными способами. Два из них — это многозадачность и опрос каналов ввода-вывода. Но если использовать только многозадачный режим, можно потерять контроль над системным планировщиком и драгоценное время на бесконечном переключении задач. Если работать только по методике опроса, будут возникать периоды простоя и сократится пропускная способность сервера. Решение заключается в объединении двух методик. *Разумно* используя функцию select(), можно понизить влияние "узких мест" обеих методик, не теряя при этом никаких преимуществ.

В рассматриваемом алгоритме создается несколько процессов. С каждым из них связан набор открытых клиентских каналов. Переключением каналов управляет функция select().

```
/*** Пример разумного использования функции select(): каждый  ***/
/*** потомок пытается принять запрос на подключение и в       ***/
/*** случае успеха добавляет дескриптор соединения в список.  ***/
/*** (Взято из файла smart-select.c на Web-узле.)              ***/

int sd, maxfd=0;
fd_set set;
FD_ZERO(&set);

/*** Создание сокета и вызов функции fork() * * * / > :

/*- в дочернем процессе -*/
maxfd ~ sd;
FD SET(sd, &set);
for(;;)
{   struct timeval timeout={2,0}; /* 2 секунды */
    /*- Ожидаем команды -*/
    if ( select(maxfd+1, &set, 0, 0, stimeout) > 0 )
    {
        /*- Если новое соединение, принимаем его --*/
        /*- и добавляем дескриптор в список -*/
        if ( FD_ISSET(sd, &set) )
        {   int client = accept(sd, 0, 0);
            if ( maxfd < client )
                maxfd = client;
            FD SET(client, &set);
        }
        /*- Если запрос от существующего клиента, -*/
        /*- обрабатываем его-*/
        else
            /*** обработка запроса ***/
            /*- ЕСЛИ клиент завершил работу, --*/
            /*- удаляем дескриптор из списка -*/
    }
```

В этой программе создается небольшое число процессов (намного меньше, чем в чисто многозадачном сервере), например 5—10 сервлетов, С каждым из сервлетов, в свою очередь, может быть связано 5-10 соединений.

Коллизии выбора

Возможно, читателям доводилось слышать о проблеме в BSD-системах, которая называется *коллизия выбора. В* рассмотренной программе предполагается, что функция select() "пробуждается" только в том случае, когда меняется состояние одного из дескрипторов. Однако в BSD4.4 пробуждаются одновременно все процессы, заблокированные функцией select(). Похоже, что ОС Linux лишена подобного ограничения.

Подобный алгоритм позволяет обрабатывать сотни соединений без каких бы то ни было конфликтов. Благодаря балансу между числом заданий и обслуживаемых ими соединений сокращается размер таблицы процессов. В то же время управление таким количеством соединений может привести к возникновению проблем, если необходимо отслеживать состояние соединений.

Данный алгоритм без труда реализуется в серверах, не хранящих информацию о состоянии транзакции. В этом случае транзакции, выполняемые на сервере, не должны иметь никакой связи друг с другом, а в каждом соединении должна выполняться только одна транзакция. По такой схеме работают серверы с кратковременными соединениями, например HTTP-сервер, сервер запросов к базе данных и модуль переадресации Web-броузера.

Проблемы реализации

Несложно заметить, что и в описанном алгоритме есть ряд недостатков. Первый из них заключается в том, сервер не может гарантировать одинаковую загрузку сервлетов. Чтобы устранить этот недостаток, можно добавить в программу ограничение на число создаваемых соединений в каждом сервлете:

```
/*** Ограничение числа соединений в сервлете ***/
/***********************************************/
if ( FD_ISSET(sd, &set) )
    if ( ceiling < MAXCONNECTIONS )
    {   int client = accept(sd, 0, 0);
        if ( maxfd < client )
            maxfd = client;
        FD_SET(client, &set);
        ceiling++;
    }
```

Когда лимит соединений в данном сервлете исчерпан, запрос передается другому сервлету. Тем не менее распределение соединений между сервлетами осуществляется случайным образом.

Другой недостаток связан с контекстом соединения. Когда клиент подключается к серверу, последний проходит через несколько режимов работы, например режим регистрации и режим сеанса (если только сервер не работает в режиме отдельных транзакций). В предыдущем фрагменте профамма всегда возвращается в состояние ожидания после приема запроса. Если требуется отслеживать состоя-

ние каждого соединения, необходимо проверять, какое именно сообщение приходит по заданному соединению. Это не сложно реализовать, но необходимо тщательно спланировать программу.

Последний недостаток связан с долгосрочными соединениями. Программа хорошо работает, когда клиент и сервер взаимодействуют друг с другом посредством одиночных транзакций или коротких соединений. При более длительных соединениях, как правило, приходится хранить промежуточную информацию. Это существенно усложняет серверную программу. Кроме того, возрастает вероятность дисбаланса нагрузки.

Перераспределение нагрузки

Проблему распределения нагрузки можно решить, если создавать потоки, а не процессы. Как описывалось выше, сервер не может просто так назначить заданию фиксированное число соединений. (Этого можно добиться, обмениваясь информацией между процессами, но вряд ли стоит заниматься столь сложными вещами.) Работая с потоками, можно легко организовать совместное использование данных, в том числе таблиц дескрипторов.

Необходимо создать массив дескрипторов и воспользоваться функцией poll(). (Подойдет и функция select(), но алгоритм программы при этом усложнится.) Каждый поток проверяет свою часть массива. Когда поступает запрос от клиента, родительский поток принимает его и помещает новый дескриптор в массив. Данный алгоритм работает только в том случае, если родительский поток способен принять все запросы.

Способ вызова функции poll() может быть разным. Как правило, период тайм-аута устанавливается достаточно коротким, чтобы потоки быстро реагировали на любые изменения в массиве дескрипторов. Когда родительский поток добавляет в массив новый дескриптор, функция poll() в дочернем потоке должна быть вызвана повторно.

```
/*****************************************************************/
/***     Пример распределения нагрузки: родительский поток     ***/
/***    принимает запросы на подключение и распределяет их     ***/
/***  между дочерними потоками.                                ***/
/***     (Взято из файла fair-load.c на Web-узле.)             ***/
/*****************************************************************/

int  fd_count=0;
struct pollfd fds[MAXFDs];  /* обнуляется с помощью
                               функции bzero() */

/*—   Родительский поток —*/
int sd = socket(PF_INET, SOCK_STREAM, 0);
/*** Вызов функций bind() и llsten() ***/
for (;;)
{   int i;
    /*— Проверяем наличие свободных позиций в очереди, —*/
    /*— прежде чем принять запрос —*/
    for ( i=0; i<fd_count; i++ )
         if ( fds[i].events == 0 )
              break;
    if ( i == fd_count && fd_count < MAXFDs )
```

```
        fd_count++;
    /*- ЕСЛИ свободные позиции имеются, -*/
    /*- устанавливаем соединение -*/
    if (i < fd_count)
    {
        fds[i].fd = accept(sd, 0,0);
        fds[i].events = POLLIN | POLLHUP;
    }
    else          /* в противном случае ненадолго переходим */
        sleep(1);                       /* в "спящий" режим */
}

/*- Дочерний поток -*/
void *Servlet(void *init)
{   int start = *(int*)init; /* начало диапазона дескрипторов */
    for (;;)
    {   int result;
        /* ожидаем 0,5 секунды */
        if ( (result = poll(fds+start, RANGE, 500)) > 0 )
        {   int i;
            for (i = 0; i < RANGE; i++)
            {
                if ( fds[i].revents & POLLIN )
                    /*** Обрабатываем сообщение .***/,
                else if ( fds[i].revents & POLLHUP )
                    /*** Разрываем соединение ***/
            }
        }
        else if ( result < 0 )
            perror("poll() error");
```

Из приведенного фрагмента видно, как распределяются обязанности между предком и потомком. Родительский поток принимает запросы и помещает дескрипторы соединений в массив fds[]. Дочерний поток ожидает изменения состояния дескрипторов во вверенном ему диапазоне. (Если поле events в структуре pollfd равно нулю, функция poll() пропустит этот элемент массива. Если активных соединений нет, функция просто дождется окончания тайм-аута и вернет 0.)

Работать с потоками можно и в клиентской программе. Как рассказывалось в главе 7, "Распределение нагрузки: многозадачность", потоки очень полезны, когда требуется одновременно отправлять несколько независимых запросов, принимать данные и обрабатывать их. Клиент может все это делать, поскольку большая часть времени в сетевых соединениях тратится на ожидание ответа сервера.

Анализ возможностей сокета

Повышение производительности клиентских и серверных программ тесно связано с настройкой параметров сокетов. В предыдущих главах упоминался ряд параметров, с помощью которых можно модифицировать работу сокета. В нескольких последующих разделах рассматривается множество других параметров.

Управление параметрами сокетов осуществляется с помощью функций getsockopt() и setsockopt(). Они позволяют конфигурировать как общие параметры, так и те, что зависят от протокола (полный список всех возможных параметров представлен в приложении А, "Информационные таблицы"). Прототипы этих функций таковы:

```
int getsockopt(int sd, int level, int optname, void *optval,
               socklen_t *optlen);
int setsockopt(int sd, int level, int optname, const void *optval,
               socklen_t optlen);
```

Каждый параметр относится к определенному уровню. В настоящее время в Linux определены 4 уровня: SOL_SOCKET, SOL_IP, SOL_IPV6 и SOL_TCP. В основном все параметры являются целочисленными или булевыми. При вызове любой из функций значение параметра передается через аргумент optval, а размерность значения указывается в аргументе optlen. В последнем аргументе функция getsockopt() возвращает реальное число байтов, занимаемое параметром.

Например, ниже показано, как сбросить флаг SOJCEEPALIVE:

```
/***                Пример функции setsockopt()                ***/

int value=0; /* FALSE */
if ( setsockopt(sd, SOL_SOCKET, SO_KEEPALIVE, Svalue,
                sizeof(value)) != 0 )
    perror("setsockopt() failed");
```

А вот как можно узнать тип сокета:

```
/***                Пример функции getsockopt()                ***/

int value;
int val size = sizeof(value);
if ( getsockopt(sd, SOL_SOCKET, SO_TYPE, &value, &value)) != 0 )
    perror("getsockopt() failed");"
```

Общие параметры

Общие параметры применимы ко всем сокетам. Они относятся к уровню SOL_SOCKET.

- SO_BROADCAST. Позволяет сокету посылать и принимать широковещательные сообщения. В широковещательном адресе все биты маски активной подсети равны единице (см. главу 2, "Основы TCP/IP"). Режим широковещания поддерживается не во всех сетях. Там, где он допустим, в этом режиме могут передаваться только дейтаграммы. (Булево значение, по умолчанию False)
- SO_DEBUG. Включает/отключает режим регистрации всех отправляемых и принимаемых сообщений. Эта возможность поддерживается только в протоколе TCP. (Булево значение, по умолчанию False.)

SO_DONTROUTE. Включает/отключает маршрутизацию. В редких случаях пакеты не должны подвергаться маршрутизации. Например, это могут быть пакеты конфигурирования самого маршрутизатора. (Булево значение, по умолчанию False.)

SO_ERROR. Содержит код последней ошибки сокета. Если не запросить этот параметр до момента выполнения следующей операции ввода-вывода, будет установлена библиотечная переменная errno. (Целочисленное значение, по умолчанию 0, только для чтения.).

SO_KEEPALIVE. Если TCP-сокет не получает от внешнего узла сведений в течение двух часов, он посылает серию сообщений, пытаясь повторно установить соединение или определить причину проблемы. (Булево значение, по умолчанию True.)

SO LINGER. Срket не будет закрыт, если в его буферах есть данные. Функция close() помечает сокет как подлежащий закрытию (но не закрывает его) и немедленно завершается. Этот параметр сообщает сокету о том, что программа должна дождаться его закрытия. Значение параметра представляет собой структуру типа linger, в которой есть два поля: l_onoff (включить/отключить режим задержки) и l_linger (максимальная задержка в секундах). Когда режим задержки включен, а поле l_linger равно нулю, при закрытии сокета произойдет потеря содержимого буферов. Если же это поле не равно нулю, функция close() будет ждать в течение указанного периода времени. (Структура типа linger, по умолчанию режим отключен.)

SO_OOBINLINE. Через сокет можно посылать очень короткие сообщения, которые принимающая сторона не будет помещать в очередь, — они передаются отдельно (вне основной полосы пропускания). Режим внеполосной передачи можно применять для передачи срочных сообщений. Установка этого флага приводит к помещению внеполосных данных в обычную очередь, откуда сокет может прочесть их традиционным способом. (Булево значение, по умолчанию False.)

SO_PASSCRED. Включает/отключает режим передачи пользовательских идентификаторов. (Булево значение, по умолчанию False.)

SO_PEERCRED. Задает атрибуты идентификации передающей стороны (идентификаторы пользователя, группы и процесса). (Структура Типа ucred, по умолчанию обнулена.)

SO_RCVBUF. С помощью этого параметра можно изменить размер входного буфера. Для TCP это значение должно быть в 3—4 раза больше максимального размера сегмента (см. параметр TCP MAXSEG). В UDP подобная возможность недоступна, и все данные, не помещающиеся в буфере, будут потеряны. (Целочисленное значение, по умолчанию 65535 байтов.)

SO_RCVLOWAT. Применяется в функциях, связанных с опросом каналов, и в режиме сигнального ввода-вывода. Задает минимальное число байтов, после приема которых сокет будет считаться доступным для чтения. В Linux этот параметр доступен только для чтения. (Целочисленное значение, по умолчанию 1 байт.)

SO_RCVTIMEO. Задает предельную длительность тайм-аута при чтении. Когда функция чтения (read(), readv(), recv(), recvfrom() или recvmsg()) превышает указанное время ожидания, генерируется сообщение об ошибке. (Структура типа timeval, по умолчанию 1 с, только для чтения.)

SO_REUSEADDR. С помощью этого параметра можно создать два сокета, которые совместно используют соединение по одному и тому же адресу/порту. Допускается совместное использование порта несколькими сокетами в пределах одного процесса, разных процессов и разных программ. Данный параметр полезен, когда сервер "рухнул" и его необходимо быстро перезапустить. Ядро обычно резервирует порт в течение нескольких секунд после завершения его программы-владельца. Если в результате вызова функции bind() возникает ошибка "Port already used" (порт уже используется), установите рассматриваемый флаг, чтобы избежать этой ошибки. (Булево значение, по умолчанию False.)

SO_SNDBUF. Позволяет задать размер выходного буфера. (Целочисленное значение.)

SO_SNDLOWAT. Задает минимальный размер выходного буфера. Функция, связанная с опросом каналов, сообщает о том, что сокет готов для записи, когда в буфере освобождается указанное число байтов. В режиме сигнального ввода-вывода программа получает сигнал, когда в буфер записывается данное количество байтов. В Linux этот параметр доступен только для чтения. (Целочисленное значение, по умолчанию 1 байт.)

SO_SNDTIMEO. Задает предельную длительность тайм-аута при записи. Когда функция записи (write(), writev(), send(), sendto() или sendmsg()) превышает указанное время ожидания, генерируется сообщение об ошибке. (Структура типа timeval, по умолчанию 1 с, только для чтения.)

SO_TYPE. Содержит тип сокета. Это значение соответствует второму параметру функции socket(). (Целочисленное значение, по умолчанию не инициализировано, только для чтения.)

Параметры протокола IP

Перечисленные ниже параметры применимы к дейтаграммным и неструктурированным сокетам. Они относятся к уровню SOL_IP.

- IP_ADD_MEMBERSHIP. С помощью этого параметра происходит добавление адресата к группе многоадресной доставки сообщений. (Структура типа ip_mreq, по умолчанию не инициализирована, только для записи.)
- IP_DROP_MEMBERSHIP. С помощью этого параметра происходит удаление адресата из группы многоадресной доставки сообщений. (Структура типа ipjnreq, по умолчанию не инициализирована, только для записи.)
- IP_HDRINCL. Установка этого флага свидетельствует о создании заголовка неструктурированного IP-пакета. Единственное поле, которое не нужно заполнять, — это контрольная сумма. Данный параметр предназначен только для неструктурированных сокетов. (Булево значение, по умолчанию False.)

IP_MTU_DISCOVER. Позволяет начать процесс определения максимального размера передаваемого блока (MTU - Maximum Transmission Unit). При этом отправитель и получатель договариваются о размере пакета. Данный параметр может принимать три значения:

- IP_PMTUDISC_DONT (0) — никогда не посылать нефрагментированные пакеты;
- IP_PMTUDISC WANT (1) — пользоваться подсказками конкретного узла;
- IP_PMTUDISC DO (2) — всегда посылать нефрагментированные пакеты. (Целочисленное значение, по умолчанию режим отключен.)

IP_MULTICAST_IF. Задает исходящий групповой адрес сообщения, представляющий собой адрес в стандарте IPv4, связанный с аппаратным устройством. У большинства машин есть только одна сетевая плата и один адрес, но у некоторых их больше. С помощью данного параметра можно выбрать, какой из адресов следует использовать. (Структура типа in_addr, по умолчанию в поле адреса записана константа INADDR_ANY.)

IP_MULTICAST_LOOP. Разрешает обратную групповую связь. Входной буфер передающей программы получит копию отправляемого сообщения. (Булево значение, по умолчанию False.)

IP_MULTICAST_TTL. Задает максимальное число переходов (TTL — Time To Live) для сообщений, отправляемых в режиме групповой передачи. Если требуется вести групповое вещание в Internet, необходимо правильно инициализировать этот параметр, так как по умолчанию разрешен только один переход. (Целочисленное значение, по умолчанию 1.)

IP_OPTIONS. Позволяет выбирать конкретные IP-опции. Эти опции передаются в заголовке IP-пакета и сообщают принимающей стороне различную служебную информацию (метки времени, уровень безопасности, предупреждения и т.д.). (Массив байтов, по умолчанию пуст.)

IP_TOS. Позволяет определить тип обслуживания (TOS — Type Of Service), которое требуется для исходящего пакета. Может принимать четыре значения:

- IPTOS_LOWDELAY — минимальная задержка;
- IPTOS_THROUGHPUT — максимальная пропускная способность;
- IPTOS_RELIABILITY — максимальная надежность;
- IPTOS_LOWCOST — минимальная стоимость. (Целочисленное значение, по умолчанию дополнительное обслуживание не предусмотрено.)

IP_TTL. Задает предельное время жизни всех пакетов. Равен максимальному числу переходов, после которого пакет удаляется из сети. (Целочисленное значение, по умолчанию 64.)

Параметры стандарта IPv6

Параметры данной группы применимы к сокетам, работающим по стандарту IPv6. Они относятся к уровню SOL_IPV6.

- IPV6_ADD_MEMBERSHIP. Как и в IPv4, с помощью этого параметра можно добавить адресата к группе многоадресной доставки сообщений. (Структура типа ipv6_mreq, по умолчанию не инициализирована, только для записи.)
- IPV6_ADDRFORM. С помощью этого параметра можно задать преобразование сокета из стандарта IPv4 в стандарт IPv6. (Булево значение, по умолчанию False.)
- IPV6_CHECKSUM. При работе с неструктурированными сокетами стандарта IPv6 с помощью этого параметра можно задать смещение поля контрольной суммы. Если он равен —1, ядро не вычисляет контрольную сумму, а принимающая сторона ее не проверяет. (Целочисленное значение, по умолчанию —1.)
- IPV6_DROP_MEMBERSHIP. Как и в IPv4, с помощью этого параметра можно удалить адресата из группы многоадресной доставки сообщений. (Структура типа ipv6_mreq, по умолчанию не инициализирована, только для записи.)
- IPV6_DSTOPTS. С помощью этого параметра можно извлечь все опции из принятого пакета. Получить эту информацию в программе можно с помощью функции recvmsg(). (Булево значение, по умолчанию False.)
- IPV6_HOPLIMIT. Если этот флаг установлен и вызывается функция recvmsg(), из вспомогательного поля сообщения будет получено число переходов, в течение которых пакет еще может существовать. (Булево значение, по умолчанию False.)
- IPV6_MULTICAST_HOPS. Как и в IPv4 (параметр IP_MULTICAST_TTL), задает максимальное число переходов для сообщений, отправляемых в режиме групповой передачи. (Целочисленное значение, по умолчанию 1.)
- IPV6_MULTICAST_IF. Как и в IPv4, задает, какой IP-адрес (определяемый номером интерфейса) использовать для групповых сообщений. (Целое число, по умолчанию 0.)
- IPV6_MULTICAST_LOOP. Как и в IPv4, задает режим "эха" исходящих сообщений при групповой передаче. (Булево значение, по умолчанию False.)
- IPV6_NEXTHOP. Если этот флаг установлен, можно задать направление следующего перехода дейтаграммы. Чтобы выполнить эту операцию, необходимо иметь привилегии пользователя root. (Булево значение, по умолчанию False.)
- IPV6_PKTINFO. Если этот флаг установлен, программе будет передан номер интерфейса и целевой адрес IPv6. (Булево значение, по умолчанию False.)
- IPV6_PKTOPTIONS. С помощью этого параметра можно задать опции пакета в виде массива байтов. Данный массив передается с помощью функции sendmsg(). (Массив байтов, по умолчанию пуст.)

IPV6_UNICAST_HOPS. Как и в IPv4 (параметр IPJTTL), задает максимальное число переходов для одноадресных сообщений. (Целочисленное значение, по умолчанию 64.)

Параметры протокола TCP

Параметры данной группы применимы к TCP-сокетам. Они относятся к уровню SOL_TCP.

- TCP_KEEPALIVE. Сокет, для которого установлен флаг SO_KEEPALIVE, ожидает 2 часа, после чего пытается повторно установить соединение. С помощью параметра TCP_KEEPALIVE можно изменить длительность ожидания. Единицей измерения являются секунды. В Linux вместо этого параметра используется функция sysctl(). (Целочисленное значение, по умолчанию 7200 с.)
- TCP_MAXRT. С помощью этого параметра можно задать длительность ретрансляции в секундах. Если указано —1, сетевая подсистема будет осуществлять ретрансляцию бесконечно долго. (Целочисленное значение, по умолчанию 0.)
- TCP_MAXSEG. В TCP поток данных разбивается на блоки. Данный параметр задает максимальный размер сегмента данных в каждом блоке. Сетевая подсистема проверяет, не превышает ли это значение аналогичный параметр самого узла. (Целочисленное значение, по умолчанию 540 байтов.)
- TCP_NODELAY. В TCP применяется алгоритм Нейгла, который запрещает отправку сообщений, размер которых меньше максимального, до тех пор, пока принимающая сторона не подтвердит получение ранее посланных сообщений. Если установить этот флаг, алгоритм Нейгла будет отключен, вследствие чего можно посылать короткие сообщения до получения подтверждений. (Булево значение, по умолчанию False.)
- TCP_STDURG. Этот параметр задает, где во входном потоке следует искать байт внеполосных данных. По умолчанию это байт, следующий за байтом, который был получен при вызове функции recv() с флагом MSG_OOB. Поскольку во всех реализациях TCP такая установка поддерживается, использовать данный параметр нет особой необходимости. В Linux он заменен функцией sysctl(). (Целочисленное значение, по умолчанию 1.)

Восстановление дескриптора сокета

При написании серверных программ можно столкнутся с ситуацией, когда в результате вызова функции bind() возникает ошибка "Port already used" (порт уже используется). Это одна из самых распространенных ошибок (даже опытные программисты ее допускают), о ней чаще всего спрашивают в Usenet. Проблема связана с тем, как ядро назначает порт сокету.

В большинстве случаев ядро ждет несколько секунд, прежде чем повторно выделить порт (иногда пауза затягивается до минуты). Это делается из соображений

предосторожности. Задержка необходима, чтобы пакеты, которые еще находятся в пути, были удалены, прежде чем будет установлено новое соединение.

Проблемы можно избежать, если установить флаг SO_REUSEADDR, Считается, что он должен быть установлен на всех серверах. Как уже говорилось, это позволяет быстро создавать повторное подключение, даже если ядро все еще не освободило порт. Ниже показано, как задать данный флаг.

```
/***    Пример повторного использования порта (для сервера,    ***/
/***    который завис и должен быть запущен повторно)          ***/

int value=1; /* TRUE */
int sd = socket(PF_INET, SOCK_ADDR, 0);
if ( setsockopt(sd, SOL SOCKET, SO_REUSEADDR, &value,
                sizeof(value)) != 0 )
    perror("setsockopt() failed");
/*** Вызовы функций bind(), listen() и accept() ***/
```

Можно также попробовать вызвать функцию bind() повторно, если возникает ошибка EAGAIN. Необходимо только быть уверенным в том, что никакая другая программа не использует этот же порт.

Когда флаг SO_REUSEADDR установлен, могут возникнуть другие проблемы. Например, не нужно, чтобы два HTTP-сервера работали одновременно по одному и тому же порту 80. Кроме того, попытка запуска сервера, который уже запущен, является серьезной ошибкой системного администратора. В этом случае можно, например, проверить идентификаторы выполняющихся процессов в каталоге /var.

Досрочная отправка: перекрытие сообщений

Для сервера важно быстро восстанавливать свою работу в случае сбоев. (Имеет также значение, насколько быстро клиент способен послать запрос серверу. Можно установить флаг TCP_NODELAY, чтобы максимально ускорить отправку клиентских запросов.

Как уже упоминалось, в TCP применяется алгоритм Нейгла, позволяющий ограничить число маленьких сообщений, передаваемых в глобальной сети. Этот алгоритм запрещает передавать любое сообщение, чей размер меньше максимального, до тех пор, пока не будут получены подтверждения на ранее посланные сообщения. Конечно, если данных в буфере больше, чем максимальный размер сегмента, они будут отправлены без промедления, Это означает, что, когда есть несколько маленьких сообщений, они будут передаваться дольше, чем одно большое, потому что функция write() отправляет их не сразу, а дожидается получения подтверждений.

В данной ситуации потери данных не происходит, а лишь снижается пропускная способность сети, но это можно контролировать. Помните: если размер заголовка пакета начинает превышать размер блока данных, снижается пропускная способность. Можно ограничить минимальный размер сообщений, создавая их самостоятельно в буфере и отправляя в виде одного пакета.

Проблемы файлового ввода-вывода

Когда смотришь на обилие параметров, связанных с передачей сообщений, хочется вернуться к стандартным системным или высокоуровневым библиотечным функциям файлового ввода-вывода. Системные функции хорошо подходят для быстрой разработки приложений. Они хорошо документированы и имеют ряд возможностей, упрощающих программирование. По сути, при создании шаблона приложения следует использовать функции, которые наиболее соответствуют аналогичным функциям библиотеки Socket API. В то же время применение таких функций, как, например, printf(), следует ограничить, поскольку потом их труднее преобразовывать.

Однако функции файлового ввода-вывода нежелательно применять, если производительность приложения играет важную роль. Когда программа вызывает одну из таких функций, передавая ей дескриптор сокета, система может по нескольку раз копировать данные из файловых буферов в буферы сокета. Это наносит существенный удар по производительности. Даже низкоуровневые функции read() и write() проверяют тип дескриптора и, если он относится к сокету, вызывают соответствующие функции Socket API.

При работе с сокетами лучше полагаться только на библиотеку Socket API. Это также сделает программу понятнее и упростит ее отладку. Можно будет легко увидеть, где программа работает с файлом, а где — с сокетом.

Ввод-вывод по запросу: рациональное использование ресурсов процессора

У всех сетевых программ есть два общих компонента: собственно алгоритм и подсистема ввода-вывода, с которой они работают. И если саму программу можно тщательно спроектировать и отладить, то второй компонент не является столь же гибким. Но одной из наиболее привлекательных черт Linux является то, что работу этой подсистемы можно настраивать. Функции ввода-вывода библиотеки Socket API имеют множество параметров, позволяющих управлять приемом и передачей данных. Кроме того, очень мощными возможностями располагает функция fcntl().

В большинстве случаев ядро просто буферизует отправляемые сообщения. Это означает, что можно сосредоточиться на сборе информации и ее обработке. В главе 8, "Механизмы ввода-вывода", рассказывалось о том, как с помощью режима асинхронного ввода-вывода поручить ядру делать часть работы за вас. В этом случае программа возлагает задачу по генерированию сигналов на подсистему ввода-вывода.

Ускорение работы функции send()

При отправке сообщения ядро копирует его в свои буферы и начинает процесс создания пакета. После его окончания ядро возвращает в программу код завершения. Таким образом, задержка связана с формированием результатов работы функции send(). Если возникают какие-то проблемы, ядро выдает сообщение об ошибке.

Однако большинство ошибок происходит в процессе передачи, а не в самой функции send(). Проблемы, связанные, например, с неправильным использованием указателей, функция обнаруживает достаточно быстро, а вот о других ошибках будет сообщено только по завершении функции. С помощью параметра SO_ERROR сокета о них можно узнать даже раньше, чем будет установлена библиотечная переменная errno.

Работу функции send() можно ускорить, если задать в ней опцию MSG_DONTWAIT. В этом случае функция копирует сообщение в буфер и немедленно завершается. С этого момента можно выполнять другие операции, дожидаясь получения сигнала о завершении функции. Однако в обязанности программиста входит проверка кодов ошибок, возвращаемых функцией.

Единственная ситуация, когда операция записи блокируется, — это переполнение буфера. Оно нечасто происходит в системе, где много свободной памяти. Если же такая ситуация возникает, необходимо применять асинхронный ввод-вывод, обрабатывая сигналы об освобождении буфера.

Разгрузка функции recv()

В отличие от функции send(), большинство вызовов функции recv() блокируются, поскольку программа, как правило, выполняется быстрее, чем приходят данные. Когда во входных буферах имеются данные, функция recv() копирует их в память и завершается. Даже если в буфере всего один байт, функция вернет этот байт и завершится. (Изменить подобное поведение можно, установив флаг MSG_WAITALL.)

Как правило, не нужно ждать поступления всех данных, так как программа в это время может выполнять множество других действий. Есть два выхода из ситуации: создавать потоки, управляющие работой отдельных каналов, или использовать сигналы. В первом случае следует помнить о системных ресурсах. Во-первых, обязательно будет существовать задание, на какое-то время заблокированное функцией recv(). Во-вторых, разрастается таблица процессов, что ведет к снижению производительности.

Применение асинхронного, или сигнального, ввода-вывода позволяет программе продолжать выполняться, не дожидаясь поступления данных. Когда сообщение придет, ядро пошлет программе сигнал SIGIO. Обработчик сигналов его примет и вызовет функцию recv(). По окончании чтения обработчик установит флаг, свидетельствующий о том, что данные подготовлены.

Не забывайте о том, что сигнал служит лишь признаком поступления данных; он не говорит о том, сколько именно данных прибыло. Кроме того, если выполнять ввод-вывод непосредственно в обработчике, можно не успеть обслужить другие сигналы, которые поступают в это же время. Решить данную проблему можно, если установить в обработчике флаг, информирующий программу о том, что она должна вызвать функцию recv() со сброшенной опцией MSG_WAITALL.

Отправка приоритетных сообщений

В процессе обмена данными программе может потребоваться "разбудить" принимающую сторону или отменить какую-то операцию. В протоколе TCP поддерживаются срочные сообщения, которые проходят "сквозь" входную очередь.

Такие сообщения называются *внеполосными* (OOB — out-of-band). Несмотря на заманчивое название, действительность несколько разочаровывает: согласно спецификации, срочное сообщение может занимать всего один байт (такие сообщения поддерживаются и в других протоколах, но они иначе реализованы).

Если программа получает два срочных сообщения подряд, второе из них затирает первое. Это связано с тем, как сетевая подсистема хранит срочные сообщения: для каждого сокета зарезервирован буфер размером один байт.

Первоначально срочные сообщения применялись для соединений, работающих по принципу транзакций. Например, в Telnet требовалось передавать сигнал о прерывании сеанса (^C) по сети. С помощью срочных сообщений можно было сообщить клиенту или серверу тип операции, скажем reset-transaction или restart-transaction. Всего существует 256 возможных вариантов.

Если флаг SO_OOBINLINE сокета не установлен, ядро уведомит программу о поступлении внеполосных данных с помощью сигнала SIGURG (по умолчанию этот сигнал игнорируется). В обработчике сигналов можно прочитать данные, задав в функции recv() опцию MSG_OOB.

Чтобы отправить срочное сообщение, необходимо установить флаг MSG OOB в функции send(). Кроме того, как и в асинхронном режиме, необходимо с помощью функции fcntl() включить обработку сигналов ввода-вывода:

```
/*********************************************************/
/***      Запуск обработчика сигналов SIGIO и SIGURG      ***/
/*********************************************************/
if ( fcntl(sockfd, F_SETOWN, getpid()) != 0 )
    perror("Can't claim SIGURG and SIGIO");
```

Эта функция сообщает ядру о том, что программа хочет получать асинхронные уведомления в виде сигналов SIGIO и SIGURG.

С помощью срочных сообщений можно проверять, не пропала ли принимающая сторона, так как, в отличие от обычных данных, внеполосные данные не блокируются. Вот как можно реализовать алгоритм "перестукивания" между сервером и клиентом:

```
/*****************************************************************/
/***    Обмен срочными сообщениями между клиентом и сервером    ***/
/***    (сервер отвечает на сигналы).                            ***/
/***    (Взято из файла heartbeat-server.c на Web-узле.)         ***/

int clientfd;
void sig_handler(int signum)
{
    if ( signum == SIGURG )
    {   char c;
        recv(clientfd, &c, sizeof(c));
        if (c == '?')                          /* Ты жив? */
            send(clientfd, "Y", 1, MSG_OOB);       /* ДА! */
    }
}

int main()
{   int sockfd;
    struct sigaction act;
```

```
    bzero(&act, sizeof(act));
    act.sa_handler = sig_handler;
    sigaction(SIGURG, &act, 0);    /* регистрируем сигнал SIGURG */

    /*** устанавливаем соединение ***/
    /*— запуск обработчика сигналов SIGIO и SIGURG —*/
    if ( fcntl(clientfd, F_SETOWN, getpid()) != 0 )
        perror("Can't claim SIGURG and SIGIO");
    /*** другие действия ***/
}
```

В этом фрагменте сервер отвечает на запросы, посылаемые клиентом. Код клиента будет немного другим:

```
/*****************************************************************/
/***   Обмен срочными сообщениями между клиентом и сервером   ***/
/***    (клиент посылает сигналы).                             ***/
/***    (Взято из файла heartbeat-client.c на Web-узле.)       ***/

int serverfd, got_reply=1;
void sig_handler(int signum)
{
    if ( signum == SIGURG )
    {   char c;
        recv(serverfd, &c, sizeof(c));
        got_reply = ( c == 'Y' )              /* Получен ответ */
    }
    else if (signum == SIGALARM)
        if ( got_reply )
        {
            send(serverfd, "?", 1, MSG_OOB);         /* Ты жив? */
            alarm(DELAY);                    /* Небольшая пауза */
            got reply = 0;
        }
        else
            fprintf(stderr, "Lost connection to server!");
}

int main()
{   struct sigaction act;

    bzero(&act, sizeof(act));
    act.sa_handler = sig_handler;
    sigaction(SIGURG, &act, 0);
    sigaction(SIGALRM, &act, 0);

    /*** устанавливаем соединение ***/
    /*— запуск обработчика сигналов SIGIO и SIGURG —*/
    if ( fcntl(serverfd, F_SETOWN, getpid()) != 0 )
        perrorj"Can't claim SIGURG and SIGIO");
    alarm(DELAY);
    /*** другие действия ***/
}
```

Можно реализовать полностью двустороннюю связь, осуществив несложную проверку на сервере. Если сообщение от клиента не поступило в течение заданного промежутка времени, сервер будет знать о том, что на клиентском конце соединения произошла ошибка. Срочные сообщения позволяют получить больший контроль над соединением, если они поддерживаются на обоих концах соединения.

Резюме

До сих пор в каждой главе рассматривался один из кусочков мозаики, каковой является код высокопроизводительного сервера или клиента. Любая сетевая программа должна управлять информацией, которую она отправляет и принимает. Иногда применение многозадачности может привести к снижению производительности, если проявить невнимательность при написании программы.

Чтобы добиться оптимальной производительности, необходимо придерживаться золотой середины, пользуясь преимуществами как многозадачного режима, так и средств опроса каналов ввода-вывода. Это приводит к усложнению программы, но если тщательно все спланировать и предусмотреть, полученный результат будет стоить потраченных усилий.

Параметры сокета обеспечивают надежный контроль над процедурой создания пакетов. Они существенно расширяют возможности управления сокетами, позволяя, в частности, отправлять сообщения, не дожидаясь заполнения буфера.

При отправке и получении сообщений лучше пользоваться функциями библиотеки Socket API, так как они выполняются быстрее, чем стандартные низкоуровневые и высокоуровневые функции ввода-вывода. Кроме того, они позволяют повысить надежность программы. В то же время, когда речь идет о сетевых программах, надежность является труднодостижимой целью, с которой связаны отдельные методики и приемы программирования. В следующей главе мы остановимся на данном вопросе более подробно.

Создание устойчивых сокетов

Глава 10

В этой главе...

Итак, наша задача — создание клиентских и серверных приложений коммерческого уровня. Это достойная цель, даже если программа будет распространяться бесплатно вместе с исходными текстами на условиях открытой лицензии. Ведь никому не хочется, чтобы его критиковали за ошибки программирования. Так как же сделать хорошую программу безупречной? Хороший вопрос!

В первую очередь следует подумать о том, чего вы стремитесь достичь. Если программа создается для конкретной категорий пользователей, анализируете ли вы программу с точки зрения такого пользователя? Можете ли вы с кем-нибудь из них встретиться и узнать, чего на самом деле они ждут от вашей программы? Насколько надежной должна быть клиентская или серверная программа?

Создание устойчивой сетевой программы не является чем-то особенным, просто сначала необходимо взглянуть на всю картину в целом. Полученное приложение всегда будет выполняться одновременно с какой-нибудь другой программой, поэтому на разработчике лежит дополнительная ответственность. Недостаточно сделать исходный текст программы удобочитаемым.

В этой главе будет дано несколько ценных советов по созданию устойчивых программ. Их нельзя считать исчерпывающими, так как на данную тему можно писать целые книги. Тем не менее приведенный материал поможет читателям избежать несколько типичных ошибок сетевого программирования.

Методы преобразования данных

Первый шаг в обеспечении устойчивости сетевой программы заключается в использовании функций преобразования из библиотеки Socket API. Существует множество функций, преобразующих адреса, имена и двоичные данные из одной кодировки в другую. Они важны, если необходимо гарантировать переносимость, тестируемость и долговечность программы.

Как описывалось в главе 2, "Основы TCP/IP", в сети применяется обратный порядок следования байтов. Это не имеет значения, если работать за компьютером Alpha или 68040, где по умолчанию используется данная кодировка. В таких системах функции заменяются "заглушками", которые не выполняют никаких действий. Но если вы читаете эту книгу, то, скорее всего, ваша программа будет распространяться в среде Linux. В этом случае функции преобразования обеспечивают правильное представление информационных структур.

Тысячи программистов протестировали все библиотеки функций, имеющиеся в Linux. Возможно, некоторые из этих проверок касались только одной конкретной конфигурации, но организация GNU внимательно следит за надежностью распространяемых ею пакетов, тщательно и скрупулезно отлаживая их. Это служит фундаментом разработки устойчивых программ.

Некоторые программисты предпочитают создавать свои собственные интерфейсы и библиотеки. Если последовать данному подходу и отказаться от использования стандартных функций, можно потратить больше времени на изобретательство, чем на собственно разработку. В результате получится более крупная и сложная программа, которую труднее тестировать. Если же все-таки окажется, что в библиотеках нет требуемой функции, при ее разработке старайтесь придерживаться стиля и философии библиотечных вызовов UNIX. Приведем примеры:

- если функция завершается без ошибок, она должна возвращать значение 0;
- если в процессе выполнения функции произошла ошибка, она должна возвращать отрицательное значение и записывать код ошибки в библиотечную переменную errno;
- пользуйтесь стандартными кодами ошибок;
- передавайте структуры по ссылкам;
- старайтесь определять низкоуровневые функции и строить на их основе высокоуровневые;
- определяйте все параметры-указатели, предназначенные только для чтения, со спецификатором const;
- лучше создавать структуры, чем typedef-определения (макротипы);
- предпочтительнее задавать имена переменных и функций в нижнем регистре, а не в верхнем, в то же время константы должны записываться прописными буквами;
- ведите журнальный файл для регистрации событий, ошибок и нестандартных ситуаций.

Это лишь некоторые из правил. Наилучшим решением будет просмотреть текст стандартной функции и взять его за основу.

В целом необходимо отметить, что программу, написанную стандартным и понятным способом, легче использовать, модифицировать и улучшать. Подумайте: сам Линус Торвальдс объявил о том, что не собирается владеть правами на ядро Linux всю свою жизнь. Применение стандартных функций и методик делает ядро долговечным.

Проверка возвращаемых значений

При работе с функциями библиотеки Socket API необходимо проверять результаты их работы. В этом состоит особенность сетевого программирования: ошибка может возникнуть в любое время, причем иногда это не связано с самой программой.

В первую очередь нужно проверять все коды завершения функций. Некоторые из функций являются критическими с точки зрения работы программы.

- bind(). Программа, работающая с конкретным портом, должна его зарезервировать. Если это невозможно, необходимо узнать об этом как можно раньше. Ошибки могут быть связаны с конфликтами портов (порт уже используется другой программой) или с проблемами в самом сокете.
- connect(). Нельзя продолжить работу, если соединение не установлено. Сообщения об ошибках могут иметь вид "host not found" (узел не найден) или "host unreachable" (узел недоступен).
- accept(). Программа не может начать соединение, если данная функция не возвращает положительное число. (Теоретически возможно, что легальный дескриптор сокета равен нулю, но это очень необычная ситуация.) Наиболее распространенный код ошибки в данной ситуации — EINTR (вызов прерван сигналом). Это не критическая ошибка. Следует

либо вызвать функцию sigaction() с флагом SA_RESTART, либо проигнорировать ошибку и повторно вызвать функцию.

- Все функции ввода-вывода (recv(), send() и т.д.). Эти функции определяют, было ли сообщение послано или принято успешно. Возникающие в них ошибки свидетельствуют о разрыве соединения либо о прерывании по сигналу (см. выше). Применять высокоуровневые функции, например fprintf(), не стоит, так как они не позволяют отслеживать ошибки. Соединение может быть разорвано в любой момент, вследствие чего сигнал SIGPIPE приведет к аварийному завершению программы.
- gethostbyname(). Если в процессе работы этой функции произошла ошибка, будет получено значение 0 (или NULL). При последующем извлечении значения пустого указателя возникнет ошибка сегментации памяти, и программа завершится аварийно.
- fork(). Значение, возвращаемое этой функцией, указывает на то, где осуществляется вызов — в предке или потомке. Если оно отрицательно, значит, потомок не был создан или произошла системная ошибка.
- pthread_create(). Подобно функции fork(), необходимо убедиться в том, что дочернее задание было успешно создано.
- setsockopt()/getsockopt(). У сокетов есть множество параметров, с помощью которых можно настраивать работу программы. Как правило, необходимо быть уверенным в успешном завершении этих функций, чтобы программа могла продолжить нормальную работу.

Неуспешное завершение любой из перечисленных функций приводит к одному результату — программа завершается аварийно или же ее работа становится непредсказуемой. Общее правило таково: если от успешного вызова функции зависит работоспособность программы, всегда проверяйте код ее завершения. Когда источник ошибки очевиден, выведите сообщение об этом на консоль. Это даст пользователю возможность как можно раньше узнать о проблеме.

Ниже перечислен ряд менее критичных функций, возвращаемые значения которых можно проигнорировать.

- socket(). Ошибка в данной функции возникает только тогда, когда дескриптор сокета нельзя получить (нет привилегий или ядро не поддерживает эту функцию), указан неправильный параметр или таблица дескрипторов переполнена. В любом случае функции bind(), connect() и др. вернут ошибку вида "not a socket" (дескриптор не относится к сокету).
- listen(). Если перед этим функция bind() завершилась успешно, мало вероятно, чтобы в данной функции произошла ошибка. Правда, следует учитывать, что длина очереди ожидания ограничена.
- close() или shutdown(). Если дескриптор файла неправильный, файл не был открыт. Так или иначе, после вызова любой из этих функций можно считать файл закрытым и продолжать работу.

Как правило, за функцией, не являющейся критической, следует другая, более важная функция, которая сообщает о возникшей ошибке. Кроме того, неудачное завершение одной из перечисленных функций не приводит к катастрофическим последствиям. Конечно, это не означает, что на них можно не обращать внима-

ние. При любых обстоятельствах дополнительная проверка того, успешно ли завершилась функция, только повышает надежность программы.

Можно также перехватывать ошибки, возникающие не в системных или библиотечных функциях. Они связаны с динамичной природой сетей, в которых клиенты и серверы могут периодически "уходить в себя". Сетевая подсистема отслеживает некоторые ошибки в протоколах TCP/IP, постоянно проверяя готовность канала к двунаправленному обмену сообщениями.

Если программа длительное время не посылала никаких сообщений, она должна самостоятельно проверить доступность канала. В противном случае ошибка, связанная с отсутствием соединения, будет представлена как ошибка ввода-вывода, что дезориентирует пользователя. Определить подобного рода ошибку можно, вызвав функцию getsockopt() с аргументом SO_ERROR:

```
int error;
socklen_t size = sizeof(error);
if ( getsockopt(sockfd, SOL_SOCKET, SO_ERROR, &error, &size)
     == 0 )
   if ( error != 0 )
      fprintf(stderr, "socket error: %s(%d)\n", stderror(error),
              error);
```

Придерживаясь такого подхода, можно перехватывать ошибки до того, как они попадут в подсистему ввода-вывода. Это дает возможность исправить их, прежде чем пользователь обнаружит проблему.

Система также информирует программу об ошибках посредством сигналов. Например, сигналы могут посылаться, когда соединение было закрыто на противоположном конце канала. Некоторые из сигналов можно проигнорировать, но лучше перехватывать все сигналы, которые реально могут быть посланы программе, если вы хотите избавить себя от бессонных ночей, проведенных за отладкой программы.

Обработка сигналов

В сетевом приложении задействовано много технологий, и некоторые из них связаны с сигналами. Программа должна уметь правильно их обрабатывать. Из тех сигналов, которые приводят к аварийному завершению программы, чаще всего забывают о сигнале SIGPIPE.

С обработкой сигналов связаны свои проблемы, о ряде из которых упоминалось в главе 7, "Распределение нагрузки: многозадачность". Основная из них связана с тем, что любой процесс одновременно принимает только один сигнал конкретного типа. Если во время выполнения обработчика поступает другой такой же сигнал, программа не узнает об этом событии.

Решить эту проблему можно несколькими способами. Во-первых, в обработчике необходимо стараться выполнять минимальное число действий. Вызов любой функции ввода-вывода может привести к потере последующих сигналов. Хуже того, если произойдет блокирование ввода-вывода, последствия для программы окажутся катастрофическими (она зависнет). Следует также избегать циклов И пытаться сделать алгоритм обработчика линейным. Конечно, из этого правила есть исключения, но в целом чем меньше команд вызывается в обработчике, тем лучше.

Во-вторых, можно разрешить прерывать выполнение обработчика. Применять данный подход следует осторожно, так как обработчик сигналов может помещать свои данные при каждом следующем вызове в специальный аппаратный стек. Глубина этого стека по умолчанию невелика, поэтому легко возникает переполнение стека. ,

В-третьих, можно заставить обработчик помещать сообщения о сигналах в очередь главной программы, которая будет сама их обрабатывать. Это не столь эффективное решение, как кажется на первый взгляд. Сигнал говорит лишь о том, что что-то произошло. Программа знает только тип сигнала (в Linux их 32) и больше ничего. Программе придется самостоятельно определять, относится ли группа однотипных сообщений к одному или нескольким сигналам.

Порядок обработки каждого сигнала зависит от типа сигнала. Из всех 32-х сигналов (информацию о них можно получить в приложении А, "Информационные таблицы", и разделе 7 интерактивной документации) чаще всего обрабатываются такие: SIGPIPE, SIGURG, SIGCHLD, SIGHUP, SIGIO и SIGALRM.

SIGPIPE

В руководстве по UNIX сказано, что лишь простейшие программы игнорируют этот сигнал. Он не возникает в среде, где одновременно работает только одна программа. В ряде случаев он не очень важен для приложения, и получив этот сигнал, оно вполне может завершиться. Однако при написании клиентских или серверных приложений необходимо тщательно следить за тем, чтобы программа корректно восстанавливала свою работу.

Ошибка канала возникает, когда узел-адресат закрывает соединение до окончания сеанса передачи данных (см. файлы sigpipe-client.c и sigpipe-server.c на Web-узле). То же самое произойдет, если направить длинный список файлов программе постраничной разбивки, например less, а затем завершить ее работу, не дойдя до конца списка, — будет выдано сообщение "broken pipe" (разрыв канала). Избежать получения сигнала SIGPIPE можно, задав опцию MSG_NOSIGNAL в функции send(). Но это не лучший подход.

Точный порядок обработки сигнала зависит от программы. В первую очередь необходимо закрыть файл, поскольку канал больше не существует. Если открыто несколько соединений и программа осуществляет их опрос, следует узнать, какое из соединений было закрыто.

С другой стороны, вполне вероятно, что сеанс еще не завершен: остались данные, которые требуется передать либо получить, или действия, которые нужно выполнить. Когда клиент и сервер общаются по известному им протоколу, досрочное закрытие соединения мало вероятно. Скорее всего, либо случилась системная ошибка, либо произошел разрыв на линии. В любом случае, если необходимо завершить сеанс, придется повторно устанавливать соединение. Можно сделать это немедленно либо выдержать небольшую паузу, чтобы дать возможность удаленной системе загрузиться повторно. Если после нескольких попыток не удается восстановить соединение, можно уведомить пользователя и спросить у него, что делать дальше (так поступают некоторые Web-броузеры).

В случае, если сеть продолжает оставаться нестабильной, сохраняйте данные в некоторых контрольных точках, чтобы можно было легко определить, когда именно возникают проблемы.

SIGURG

При передаче данных между клиентом и сервером необходимо учитывать все возможные способы обмена информацией. Программы могут посылать друг другу запросы на прерывание потока данных или инициализирующие сигналы (см. главу 9, "Повышение производительности"), пользуясь механизмом внеполосной передачи. Подобную ситуацию следует планировать заранее, так как по умолчанию сигнал SIGURG игнорируется. Его обработку нужно запрашивать особо.

Если программа получает несколько срочных сообщений подряд и не успевает их все обработать, лишние сообщения будут потеряны. Это объясняется тем, что во входящей очереди сокета зарезервирован только один байт для срочных сообщений.

SIGCHLD

Сигнал SIGCHLD возникает в многозадачной среде, когда дочернее задание (в частности, процесс) завершается. Ядро сохраняет контекст задания, чтобы родительская программа могла проверить, как завершился дочерний процесс. Если программа проигнорирует этот Сигнал, ссылка на контекст останется в таблице процессов в виде процесса-зомби (см. главу 7, "Распределение нагрузки: многозадачность").

Обычно при получении сигнала SIGCHLD программа вызывает функцию wait(). Однако следующий сигнал может поступить быстрее, чем завершится данная функция. Это неприятная проблема, но ее легко решить, обрабатывая все сигналы в цикле.

Поскольку функция wait() блокирует работу программы (а в обработчике сигналов это крайне нежелательно), воспользуйтесь вместо нее функцией waitpid():

```
#include<sys/types.h>
#include<sys/wait.h>
int waitpid(int pid, int *status, int options);
```

Параметр pid может принимать разные значения; если он равен —1, функция будет вести себя так же, как и обычная функция wait(). Параметр status аналогичен одноименному параметру функции wait() и содержит код завершения потомка. Чтобы предотвратить блокирование функции, следует задать параметр options равным WNOHANG. Когда все процессы-зомби, ожидающие обработки, будут обслужены, функция вернет значение 0. Ниже показан типичный пример обработчика сигнала SIGCHLD.

```
/*** Улучшенный пример уничтожения зомби ***/
/*******************************************/
void sig_child(int signum)
i
    while ( waitpid(-1, 0, WNOHANG) > 0 );
}
```

Это единственный случай, когда в обработчике сигналов следует применять цикл. Такова особенность работы функции waitpid(). Предположим, обработчик

вызывается, когда завершается один из прроцессов-потомков. Если бы на месте указанной функции стояла функция wait() и во время ее выполнения пришел новый сигнал, он был бы просто потерян. А вот функция waitpid() на следующей итерации цикла благополучно обнаружит появившийся контекст потомка. Таким образом, одна функция обрабатывает все отложенные команды завершения, а не только одну.

SIGHUP

Что произойдет с дочерним процессом, если завершится родительская программа? Он получит сигнал SIGHUP. По умолчанию процесс прекращает свою работу. Обычно этого вполне достаточно.

Большинство серверов лучше работает в фоновом режиме в виде демонов, с которыми не связан регистрационный командный интерпретатор. В действительности некоторые демоны запускают дочерний процесс, которому делегируются все полномочия. Когда он начинает работу, родительский процесс завершается. Преимущество такого подхода заключается в том, что процесс не отображается в списке заданий.

Стандартный способ повторного запуска демона сострит в передаче ему сигнала SIGHUP. Обычно это приводит к уничтожению текущего выполняемого процесса, и программа init создает новый процесс. Если же у нее не получается это сделать, можно вызвать функцию exec(), чтобы принудительно запустить сервер. Предварительно необходимо вручную уничтожить все дочерние процессы, относящиеся к старому предку. (Данную методику можно применять при обработке всех сигналов, приводящих к завершению работы сервера.)

SIGIO

Ускорить работу сетевой программы можно, поручив обработку событий ввода-вывода ядру. Правильно написанная программа получает сигнал SIGIO всякий раз, когда буфер ввода-вывода становится доступен для очередной операции (обращение к нему не вызовет блокирования программы). В случае записи данных это происходит, когда буфер готов принять хотя бы один байт (пороговое значение устанавливается с помощью параметра SO_SNDLOWAT сокета). В случае чтения данных сигнал поступает, если в буфере есть хотя бы один байт (пороговое значение устанавливается с помощью параметра SO_RCVLOWAT сокета). О реализации обработчика этого сигнала рассказывалось в главе 8, "Механизмы ввода-вывода", а о способах повышения производительности подсистемы ввода-вывода — в главе 9, "Повышение производительности".

SIGALRM

Подобно сигналу SIGIO, программа получает сигнал SIGALRM, только если явно его запрашивает. Обычно он генерируется функцией alarm(), которая "будит" программу после небольшой паузы. Этот сигнал часто используется демонами, которые проверяют, работает ли та или иная программа.

Управление ресурсами

Сигналы — это лишь малая часть ресурсов программы. Они позволяют снизить вероятность повреждения системы и повысить производительность программы. Но необходимо также помнить о файлах, "куче" (динамических областях памяти), статических данных, ресурсах процессора, дочерних процессах и совместно используемой памяти. По-настоящему надежный сервер (да и клиент тоже) должен тщательно заботиться о своих ресурсах.

Управление файлами

При запуске программы автоматически создаются три стандартных файла (потока); stdin, stdout и stderr. Это широко известный факт, как и то, что любой из перечисленных потоков можно переадресовать. Проблем с такой переадресацией не возникает, так как ядро самостоятельно очищает потоки и освобождает их дескрипторы, когда программа завершает работу.

Другое дело — новые файлы и сокеты. Необходимо помнить о них и закрывать каждый файл. Несмотря на то что в случае завершения программы ядро также закрывает все связанные с ней файлы, в процессе работы программы они занимают часть ресурсов ядра и памяти. Если не контролировать ситуацию, внезапно может возникнуть нехватка ресурсов.

Динамическая память ("куча")

Работа с "кучей" (динамически выделяемой памятью) требует тщательного слежения за каждым полученным блоком памяти. Для многих программистов, однако, это настоящая ахиллесова пята. Они часто забывают освобождать память, а возникающие "утечки" очень трудно обнаружить. Существуют специальные библиотеки функций (например, ElectricFence), позволяющих отслеживать выделенные блоки памяти, но лучше всего придерживаться определенных правил работы с памятью.

Первая и наиболее часто встречающаяся ошибка заключается в том, что забывают проверять значения, возвращаемые функциями malloc() и calloc() (в C++ необходимо перехватывать все исключения); Если блок памяти запрашиваемого размера недоступен, функция возвращает NULL (0). В ответ на это, в зависимости от особенностей программы, можно завершить работу, изменить установки и повторно вызвать функцию, уведомить пользователя и т.д. Некоторые программисты любят проверять корректность каждой операции выделения памяти с помощью функции assert(). К сожалению, если она завершается неуспешно, программа *всегда* прекращает работу.

Работая с памятью, будьте последовательны. Вызвав однажды функцию calloc(), вызывайте ее во всех остальных местах программы. Не смешивайте разные методики. В частности, в C++ оператор new не всегда работает корректно, если в программе встречаются вызовы функций malloc() и calloc(). А если выполнить оператор delete по отношению к блоку памяти, выделенному с помощью функции malloc(), результат будет непредсказуем.

Возьмите *за* правило присваивать освобождаемым указателям значение NULL. Это позволит быстро находить *недействительные ссылки* (указатели, ссылающиеся на области памяти после того, как они были освобождены).

При выделении памяти можно запросить ровно столько, сколько нужно или сколько предположительно понадобится. Это два разных подхода к одной проблеме. В первом случае *(точное выделение)* программа запрашивает блок памяти, размер которого в точности соответствует размеру той или иной переменной в данный момент времени. Во втором случае *(выделение с запасом)* программа запрашивает более крупный блок, с тем чтобы впоследствии записывать в него данные разного размера.

В табл. 10.1 перечислены преимущества каждого подхода. Их можно чередовать в одной и той же программе, следует только помнить о том, где и какого размера блоки были выделены.

Таблица 10.1. Сравнение методик выделения памяти

Точное выделение	Выделение с запасом
Выделяется ровно столько памяти, сколько нужно в программе	Выделяется более крупный блок, в котором используется столько памяти, сколько понадобится в тот или иной момент
Выделенная память не тратится впустую	Почти всегда часть памяти не используется
Требуется несколько вызовов функций выделения и освобождения памяти	Требуется только один вызов функции выделения и один — функции освобождения
Высокая вероятность фрагментации "кучи"	Малая вероятность фрагментации "кучи"
Методика эффективна, когда память многократно выделяется в разных местах программы	Методика полезна, когда память выделяется в какой-то одной функции и сразу после этого освобождается
Может вести к неэкономному расходу памяти, так как для каждого выделенного блока создается отдельный описательный заголовок (именно так подсистема динамической памяти выполняет работу с "кучей")	Создается только один заголовок для всего блока
Методика одинаково применима в любой системе	Методика также применима в любой системе, но Linux обеспечивает для нее дополнительные преимущества (с физической памятью связаны только те страницы размером 1 Кбайт, которые реально используются)

Ошибки сегментации в функции malloc()

Если при вызове функции malloc() возникает ошибка сегментации, значит, программа повредила блок памяти, выделенный кем-то другим. Причиной ошибки обычно является неправильное применение строковых указателей или выход за пределы массива с последующим повреждением ячеек памяти, непринадлежащих программе.

В крупной программе может быть несколько независимых модулей, где применяются различные методики выделения памяти. Работая сообща с другими программистами, установите "правила игры". Например, если из одного модуля в другой передается указатель, может ли он модифицироваться или необходимо обязательно делать копию?

Как правило, указатель "принадлежит" тому модулю, в котором он был создан. Это означает, что когда указатель передается в другой модуль, нужно каким-то образом создать копию адресуемого блока памяти. Такой подход называют *детальным копированием,* поскольку все ссылки внутри блока также должны быть раскрыты и скопированы.

Статическая память

Среди всех типов ресурсов меньше всего проблем возникает со статической памятью. Сюда входят инициализированные и неинициализированные переменные, а также стековые переменные. В следующем фрагменте программы показаны различные виды статической памяти:

```
int  Counter;                                  /* неинициализированные
                                                                данные */
char *words[] = {"the", "that", "a", 0};  ;/* инициализированные
                                                                данные */
void fn(int arg1, char *arg2)                  /* параметры (стек) */
{  int i,  index;                 /* автоматические переменные (стек) */
```

Во избежание проблем при работе с ресурсами данного типа необходимо стараться инициализировать переменные, прежде чем использовать их. Задайте в компиляторе опцию -Wall, чтобы он выдавал соответствующие предупреждения.

Ресурсы процессора, совместно используемая память и процессы

Что касается последних трех типов ресурсов, то здесь достаточно сделать лишь несколько замечаний.

- *Совместно используемая память.* Работа с ней напоминает работу с файлами. Необходимо открывать, блокировать и закрывать доступ к общим областям памяти.
- *Ресурсы процессора.* Программа может легко захватить все время процессора. Нужно не забывать периодически освобождать процессор.
- *Процессы.* Когда программа выполняется в многозадачном режиме, система сообщает ей статус завершения каждого дочернего процесса. Необходимо принимать и обрабатывать все подобные уведомления, чтобы не засорять таблицу процессов процессами-зомби.

Критические серверы

Получение информации о внешних и внутренних событиях (сигналах, например) важно для понимания того, как работает система. Создавая клиентские и серверные приложения, необходимо заранее определить, *что* может произойти и *когда*. Это позволит жестко регламентировать работу программы в любых ситуациях. Сложность предварительного анализа связана с тем, что компьютеры могут

иметь самую разную конфигурацию, даже если на них установлена одна и та же операционная система.

Но многообразие конфигураций — это еще не самое страшное. В сетевом программировании вообще трудно заранее делать какие-либо допущения. Тем не менее определенная степень контроля все же имеется. Если вы четко понимаете, что должна делать программа и с кем она будет взаимодействовать, можно составить правила ее поведения в тех или иных ситуациях. А за соблюдение корректности внешних компонентов пусть отвечает пользователь.

Предположим, сервер должен выполняться в виде демона. Если ему необходимо получать доступ к определенным файлам, можно оговорить, что эти файлы обязаны находиться в конкретном каталоге и иметь заданный формат. Системный администратор будет знать, что в случае ошибки необходимо проверить эти файлы.

Серверы требуют особенного подхода к проектированию. Пользователи предполагают, что сервер будет доступен на момент обращения к нему. Они также надеются, что время его реакции будет "разумным". Необходимо выяснить, что значит "разумным". Кроме того, нужно определить, за сколько времени, по мнению пользователей, сервер должен возобновить работу в случае отказа. Все это зависит от того, насколько критическим является сервер.

Что называется критическим сервером

В отличие от клиентов, которые часто подключаются и отключаются, серверы должны функционировать постоянно, как того ожидают пользователи. В случае клиентов можно не слишком заботиться об управлении памятью и файлами. При закрытии профаммы менеджер памяти закрывает открытые файлы и освобождает выделенные блоки памяти (по крайней мере, в Linux).

В противоположность этому серверы потенциально могут работать бесконечно долго. Клиент предполагает, что сервер можно вызвать в любое время и он, словно джин из лампы, способен выполнить любое желание (в пределах разумного). Джин не может сказать: "Подожди, пока я перезагружусь". Сервер должен быть всегда доступен и готов обрабатывать запросы.

Ожидаемая степень доступности определяет критичность сервера. Некоторые серверы более критичны, чем другие. Например, HTTP-сервер просто должен выдать ответ в течение некоторого интервала времени. Есть серверы, которые контролируют выполнение транзакций, чтобы ни одна из сторон не теряла информацию, а любой сеанс выглядел непрерывным; они находят применение, в частности, в системах денежных переводов и электронной коммерции.

Коммуникационные события и прерывания

В процессе покупки товара через Internet соединение неожиданно разрывается. Что могло стать причиной этого? Ведь протокол TCP считается достаточно надежным. Что же вызвало появление ошибки? Объяснений может быть масса.

Разрывы соединений могут приводить к утрате данных, денег и даже жизни. Как бы ни были соединены между собой два компьютера, всегда существует риск потери связи. Анализируя, что *могло бы* произойти, необходимо выяснить, какие типы информационных каналов существуют и как протокол TCP взаимодействует с каждым из них.

Физические прерывания

В сети может существовать столько видов физических соединений и способов *потери несущей* (пропадание электрического или оптического сигнала, передающего пакеты), что перечислить их все было бы трудно и вряд ли целесообразно. Суть в том, что соединение между точками А и Б может быть разорвано.

Протокол TCP достаточно успешно справляется с подобными событиями. Не имеет значения тип физического носителя: кабель, оптоволокно или радиоволны. Протокол разрабатывался в те времена, когда угроза ядерной войны была вполне реальной, поэтому алгоритм работы маршрутизатора подразумевает возможность выхода из строя целых сетей. Если в сети обнаруживается другой маршрут к тому же пункту назначения, сетевая подсистема обнаруживает его и перепрограммирует маршрутизаторы. Проблема заключается лишь в том, что на выявление нового маршрута и конфигурирование маршрутизаторов уходит некоторое время.

Обычно протокол TCP решает проблему самостоятельно, не требуя вмешательства извне. Но если новый сетевой путь не может быть проложен, программа должна сама позаботиться о возобновлении сеанса.

Сбои маршрутизаторов

Физические разрывы вызывают сбои в работе маршрутизаторов, проявляющиеся в виде циклов. Сообщение циркулирует между маршрутизаторами до тех пор, пока не будет выявлено и исправлено. Это может вызывать дублирование и потерю пакетов. Однако в случае одиночного TCP-соединения программа не сталкивается с подобными проблемами, так как протокол TCP оперативно исправляет их.

Пропадание канала между клиентом и сервером

ЕСЛИ нужно восстановить сеанс, следует учесть возможность дублирования. Сеанс требуется возобновлять, когда пропадает клиент или сервер. Сервер исчезает в случае фатального сбоя. При этом все данные о транзакциях теряются. Если исчезает клиент, пропадают только те данные, которые передавались в момент потери связи. Крах сервера имеет более тяжелые последствия, чем сбой клиента.

Особенности возобновления сеанса

Когда пропадает соединение, клиент должен повторно подключиться к серверу. Это связано с целым рядом проблем. Обычно протокол TCP решает их автоматически, но очень редко это проходит незаметно для клиента или сервера.

Первая проблема связана с нарушением транзакции. Если клиент не отслеживает каждое сообщение, а предполагает, что сервер благополучно их принимает, результатом может стать потеря сообщения. Даже несмотря на то что сервер быстро восстанавливает свою работу, потерянного сообщения уже не вернуть.

Вторая проблема носит противоположный характер: дублирование транзакции. Это не то же самое, что потеря пакетов. В особо важных соединениях как клиент, так и сервер отслеживают отправляемые сообщения. В определенный момент может оказаться, что на компьютер пришла копия сообщения, хотя еще не было отправлено подтверждение на получение оригинала.

Предположим, например, что клиент запрашивает перевод 100$ с депозитного счета на текущий. До тех пор пока не будет получено подтверждение от сервера, клиент сохраняет сообщение в очереди транзакций. Затем происходит сбой, клиент аварийно завершает работу, и система сохраняет очередь транзакций. После перезагрузки клиент извлекает транзакции из очереди и повторно их выполняет. Если сервер еще раз выполнит ту же самую транзакцию, будет переведено 200$, а не 100$.

Последняя проблема связана с процедурой возобновления сеанса. Часто соединения не являются безопасными и не требуют прохождения различных уровней контроля. В случае разрыва соединения достаточно заново подключиться. С другой стороны, на многих серверах необходимо пройти процедуру проверки — *аутентификацию,* чтобы получить право подключиться к серверу. Обычно это реализуется в виде регистрационного приглашения, в котором нужно ввести имя пользователя и пароль.

Другой формой проверки является *сертификация,* при которой выясняется, действительно ли клиент тот, за кого себя выдает. В этом алгоритме подразумевается наличие третьей стороны — органа сертификации. Когда сервер принимает запрос на подключение и начинает процесс регистрации, он требует от клиента сертификат подлинности, который затем направляется в орган сертификации. Оттуда поступает подтверждение подлинности.

Все эти и ряд других проблем следует учесть до того, как начнется написание программы. Если вспомнить о безопасности и методиках восстановления соединений после того, как программа реализована, никакими "заплатками" нельзя будет закрыть изъяны исходного алгоритма.

Способы возобновления сеанса

Процедура возобновления сеанса является частью системы безопасности. Необходимо позаботиться о защите критически важных данных и о сведении к минимуму операций взаимодействия с пользователем.

Несмотря на то что сервер первым обнаруживает потерю соединения, он не может начать процесс восстановления, если взаимодействие происходит по протоколу TCP. Клиент сам должен подключиться к серверу. Поэтому клиентское TCP-приложение нужно сделать достаточно "разумным", чтобы оно могло обнаруживать потерю соединения и восстанавливать его.

Обратное цитирование

Заставить клиента обнаружить разрыв соединения не так-то просто, поскольку сетевые ошибки возникают не сразу; а через какое-то время. Можно применять обратное квитирование — устанавливать соединение в обоих направлениях. Обычно клиент подключается к серверу. Но в алгоритме обратного квитирования сервер в ответ на запрос клиента сам подключается к «ему. Через обратный канал можно посылать служебные сообщения (например, о необходимости повторного подключения). Реализовать такой канал можно не по протоколу TCP, а с помощью надежного варианта протокола UDP.

Процесс установления соединения может включать принудительную повторную аутентификацию или сертификацию либо автоматическую регистрацию в системе. Это необходимый шаг при организации безопасных сеансов связи. Если повторное соединение устанавливается в рамках того же самого приложения, можно восстановить предыдущие параметры аутентификации и зарегистрировать-

ся без участия пользователя. Процесс сертификации должен быть проведен заново. К счастью, все это обычно осуществляется незаметно для пользователя.

В ходе сеанса пользователь выполняет транзакции. Одни из них связаны с выборкой данных, а другие — с их модификацией. Чтобы обеспечить достоверность информации, клиент и сервер отслеживают все транзакции, приводящие к изменению данных. Следить за выборкой данных, как правило, не нужно (если только это не является частью системы безопасности).

Избежать потерь транзакций можно, если назначать им идентификационные номера, регистрировать их в журнале и высылать подтверждение по каждой из них. У транзакции должен быть уникальный идентификатор. Он отличается от идентификатора TCP-сообщения, поскольку является уникальным во всех сеансах. (В действительности придерживаться столь жесткого подхода не обязательно. Вполне допустимо повторно использовать идентификатор через определенный промежуток времени. Это упрощает реализацию программ.)

К примеру, клиент посылает запрос на снятие денег с кредитной карточки. Одновременно он регистрирует транзакцию в локальном журнале. Сервер в ответ выполняет два действия. Прежде всего он подтверждает получение сообщения (транзакция *опубликована)*. Когда транзакция будет завершена, сервер посылает еще одно подтверждение (транзакция *зафиксирована)*.

Получив второе подтверждение, клиент списывает отложенную транзакцию. (Критические транзакции не должны просто списываться. Их следует помещать в архив.)

"Тонкие" клиенты

Если риск потери синхронизации с сервером слишком велик, создайте *"тонкий" клиент,* который не хранит информацию локально. Такой клиент по-прежнему отслеживает состояние транзакций, но все изменения регистрируются в виде запросов к серверу.

Когда сеанс внезапно прекращается, клиент и сервер должны его восстановить. После этого обе стороны проверяют отложенные транзакции, сравнивая их идентификаторы. Те, которые оказываются завершенными, списываются. Подобный подход помогает устранить проблему дублирования транзакций. Удаляются только те транзакции, которые завершены. Остальные считаются отложенными и должны быть повторно опубликованы.

Согласованная работа клиента и сервера

При работе с одновременно выполняющимися сетевыми программами можно столкнуться с теми же проблемами взаимоблокировок и зависаний, что и в случае потоков и процессов. Однако в данном случае их труднее обнаружить.

Проблемы конкуренции в сетевом программировании имеют немного другой характер. Причина этого проста: клиенты и серверы не имеют совместного доступа к ресурсам. Они выполняются отдельно и изолированно. Единственный канал между ними — это физическая среда передачи данных по сети. Кроме того, в многозадачном программировании снять взаимоблокировку очень трудно (если вообще возможно). Сетевые взаимоблокировки снимать легче.

Сетевые взаимоблокировки

В большинстве сетевых соединений определяется, какая программа должна начинать диалог первой. Например, одна программа посылает запросы, а другая отвечает на них. В качестве иллюстрации рассмотрим серверы HTTP и Telnet. HTTP-сервер принимает запросы от клиента, предоставляя клиенту право вести диалог. С другой стороны, сервер Telnet выдает пользователю приглашение на ввод имени и пароля. Будучи зарегистрированным в системе, клиент знает о том, что сервер готов отвечать, когда видит строку приглашения. Сервер Telnet может принимать и асинхронные команды, посылаемые посредством прерываний клавиатуры (<Ctrl+C>).

Сетевая взаимоблокировка возникает очень просто: клиент и сервер забывают о том, чья очередь говорить, поэтому переходят в режим бесконечного ожидания. Эту форму тупика трудно обнаружить, так как его симптомы совпадают с признаками перегруженности сети: обе стороны не получают сообщения в течение определенного времени.

Чтобы решить эту проблему, можно задать для соединения период тайм-аута. Это очень хорошая идея, так как ни клиент, ни сервер (что особенно важно) не будут зависать в ожидании сообщений. Но завершение тайм-аута говорит лишь о том, что сообщение не поступало слишком долго.

Другой способ избавления от взаимоблокировок заключается в периодическом переключении ролей. Если сервер руководит диалогом, через некоторое время он может передать эту роль клиенту. Когда возникает взаимоблокировка, клиент и сервер одновременно меняются ролями и по истечении определенного времени разом начинают передачу данных. Это позволяет им быстро упорядочить диалог.

Эффективный способ предотвращения, обнаружения и снятия взаимоблокировок заключается в применении алгоритма "перестукивания", рассмотренного в предыдущей главе. Вместо того чтобы посылать друг другу запрос "Ты жив?", они могут передавать сообщение, указывающее на то, находится ли данная сторона в режиме ожидания.

Сетевое зависание

Другая проблема, с которой часто сталкиваются в сетевом программировании, — это зависание. Предположим, имеется сервер, опрашивающий десять соединений с помощью функции select(), однако выделенной ему доли процессорного времени хватает только на пять из них. Существует вероятность, что какое-то соединение никогда не будет обслужено.

Подобная проблема возникает при подключении к очень загруженным Internet-серверам. Ее симптом примерно такой же, как и в случае взаимоблокировки: программа подключается к серверу и не получает сообщений в течение долгого времени. Соединение в данном случае было благополучно установлено, и программа даже получила несколько байтов, а затем — тишина.

Решение проблемы здесь не такое прямое, как в предыдущем случае. В первую очередь необходимо найти хороший баланс между загруженностью процессора/системы и количеством активных соединений. При определении глубины очереди ожидания помните, что ничего плохого в отложенных соединениях нет, если они будут обслужены в пределах отведенного для этого времени. Необходимо измерить, сколько длится каждый сеанс, и выбрать длину очереди.

Можно использовать другой подход — осуществить *динамическое планирование* соединений или назначить им *приоритеты*. Если, к примеру, выполняются три процесса, только один из них может получить доступ к процессору в конкретный момент времени. Поэтому планировщик повышает эффективный приоритет процесса, проверяемого в настоящий момент. По завершении обслуживания приоритет процесса вновь понижается. Тем самым обеспечивается рациональное распределение ресурсов процессора.

Аналогичная методика применима и в отношении соединений, только алгоритм назначения приоритетов будет другим. Те соединения, в которых имеются готовые данные, получат более высокий приоритет.

Отказ от обслуживания

Сетевые злоумышленники вызывают особые формы взаимоблокировок и зависаний. О них следует знать, хотя это и достаточно старый способ атаки на сервер. Как описывалось выше, сетевая подсистема принимает клиентские запросы на подключение по определенному порту. У данного порта есть очередь ожидания.

Нарушитель подключается к этому же порту. Процесс трехфазового квитирования завершается, и сетевая подсистема помещает запрос в очередь. Сервер извлекает запрос из очереди и принимает его (помните, что соединение не установлено, пока не вызвана функция accept()). Затем сервер создает для него сервлет.

Итак, вот суть проблемы. Если нарушитель ничего не делал и не посылал никаких данных, подсистема TCP/IP завершит соединение по тайм-ауту. Казалось бы, все нормально. Но злоумышленник оказывается умнее. Он посылает несколько байтов — этого недостаточно, чтобы заполнить какой-либо буфер, но достаточно, чтобы блокировать функцию ввода-вывода.

Сервлет получает первые байты и блокируется, ожидая остальных. Нарушитель знает, что по данному соединению сервер уже не функционирует, и создает следующее соединение. В конце концов сервер поглотит все системные ресурсы, не выполняя никаких полезных действий. Даже если задать лимит на число соединений, все ресурсы сервера (в том числе очередь ожидания) будут заняты запросами злоумышленника. Всем остальным пользователям будет отказано в обслуживании.

Предотвратить подобную ситуацию можно в три этапа.

1. Всегда задавайте тайм-ауты для всех функций ввода-вывода. Их легко реализовать, и они помогут не потерять контроль над сервером.
2. Всегда оставляйте один процесс (обычно родительский) свободным, чтобы он мог следить за дочерними процессами. С его помощью можно определить, сколько процессов находится в зависшем состоянии.
3. Ограничьте число соединений от конкретного узла или подсети (идентификатор узла включается в запрос на подключение). Это слабая мера защиты, так как злоумышленники очень изобретательны, и можно наказать совершенно невинных пользователей.

Резюме: несокрушимые серверы

В данной главе рассказывалось о том, как создавать надежные и стабильно работающие серверные и клиентские приложения. Было рассмотрено, как избегать проблем, связанных с сетевыми атаками, взаимоблокировками и зависаниями. Объяснялось, как с помощью обработчиков сигналов усилить надежность программы и не допустить распространенных ошибок программирования.

Соображения, касающиеся надежности программ, как правило, служат проявлением здравого смысла: следует применять функции преобразования и проверять значения, возвращаемые различными функциями. Тем не менее полезно знать, какие именно функции необходимо проверять и что приведение данных к обратному порядку следования байтов не приводит к снижению производительности.

Важно также понимать, как работает сервер и как он взаимодействует с клиентом. Это помогает настраивать работу сервера и делать его устойчивым.

Часть III

Объектно-ориентированные сокеты

В этой части...

Глава

11 Экономия времени за счет объектов

В этой главе...

Батарейка в часах дает им энергию, которая приводит в действие часовой механизм. Когда нужно заменить батарейку, вы просто вынимаете ее и вставляете новую. Представьте, как было бы здорово, если бы то же самое можно было делать с программами, — отключить старую и подключить новую.

Объектно-ориентированная технология является попыткой достичь такого уровня взаимодействия. Она позволяет сосредоточиться на особенностях каждого конкретного компонента и снабдить его четким и неизменным интерфейсом, посредством которого он может взаимодействовать с другими компонентами.

В данной главе рассматриваются основные концепции, лежащие в основе объектно-ориентированной технологии. Сначала мы совершим небольшой исторический экскурс и узнаем, как развивались технологии программирования. В конце главы рассказывается, как применить описанные концепции в Языках, не являющихся объектно-ориентированными.

Примечание

Эта глава предшествует главам, в которых рассказывается о конкретных способах создания объектно-ориентированных сокетов. Чтобы понять особенности реализации сокетов в таких языках, как Java и C++, необходимо получить базовые представления об объектах. В книгах, посвященных объектно-ориентированному программированию, не обойтись без вводного теоретического раздела. Большинство программистов не понимает до конца объектную технологию, поэтому создаваемые ими приложения не всегда соответствуют ее исходным положениям.

Эволюция технологий программирования

Программирование не имеет столь древней истории, как физика, математика или другие точные науки. В то же время, будучи компьютерной наукой, оно в значительной степени опирается на математику, в частности такие ее разделы, как булева алгебра, теория графов и статистика. Во второй половине XX века программирование развивалось стремительными темпами, пройдя целый ряд этапов, на каждом из которых формировалась новая технология программирования, опиравшаяся на предыдущую, но расширявшая ее. Основными этапами были функциональное, модульное, абстрактное и объектно-ориентированное программирование. В следующих разделах мы постараемся раскрыть взаимосвязь между ними.

Функциональное программирование: пошаговый подход

Первая методология разработки программного обеспечения развилась из концепции блок-схем. Идея заключалась в том, что в программе образовывались блоки вызова функций, принятия решений, обработки данных и ввода-вывода. Блок-схема демонстрировала пошаговый алгоритм преобразования осмысленных входных данных в требуемые выходные результаты.

В функциональном моделировании выделялись этапы описания исходных данных, анализа задачи и системного проектирования. Конечным результатом было получение конкретной программной реализации.

На этапе описания исходных данных выяснялись системные требования. Здесь устанавливались границы между данными, вычисляемыми или определяемыми в самой программе, и данными, задаваемыми извне. На этом этапе разработчик должен был описать программное окружение системы и категорию пользователей, взаимодействующих с ней, а также решить, какие функции следует предложить пользователю.

На этапе анализа задачи строились диаграммы потоков данных в системе, разрабатывались архитектура системы и ее основные компоненты. Этот этап был еще достаточно абстрактным, и анализируемые данные представлялись не в конкретном виде, а обобщенно: определялось, что должно быть на входе каждого компонента и что — на выходе.

При переходе от анализа задачи к системному проектированию мог выделяться промежуточный этап. На нем определялись системно-зависимые, унаследованные и сторонние интерфейсы, необходимые для реализации системы. Также требовалось принять решение относительно пользовательского интерфейса и используемых сетевых протоколов.

Последний этап, системное проектирование, был посвящен разделению проблемы на функции и процедуры. В результате формировался либо общий алгоритм программы, либо ее блок-схема.

Любой программист в той или иной степени применяет функциональное моделирование, даже если пишет самую обычную программу. Блок-схемы всегда полезны для понимания общей структуры программы. Кроме того, первый этап — определение системных требований — тоже очень важен.

К сожалению, переход от блок-схемы к собственно программному коду далеко не всегда очевиден. Как можно заметить, даже на последнем этапе еще не создавалась конкретная программа. Несмотря на последовательный переход от постановки проблемы к описанию ее решения, программист, занимавшийся непосредственно реализацией, часто сталкивался с трудностями, воплощая то, что до него существовало только на бумаге.

Модульное программирование: структурный подход

Одной из проблем функционального моделирования является частое нарушение правил видимости программных компонентов. Под областью видимости переменной, функции, процедуры или модуля понимаются границы, в пределах которых к каждому из этих компонентов можно обратиться. Когда такие границы в программе не выделены, программист невольно "сбрасывает все в кучу", чтобы максимально ускорить решение поставленной задачи. В такого рода программах любая функция имеет доступ ко всем остальным функциям и переменным. Создание глобальных переменных служит ярким примером нарушения правил видимости.

Глобальные переменные

Не воспринимайте вышесказанное так, будто никогда нельзя использовать глобальные переменные. В некоторых ситуациях требуется очень экономное расходование памяти. Программы могут для временных расчетов создавать общедоступные переменные, с которыми одновременно работает несколько процессов или потоков. Но только помните, что это усложняет процесс программирования.

Многие программисты избегают глобальных переменных из-за возможности побочных эффектов. Когда с такой переменной связано несколько разделов программы и в одном из них значение переменной изменяется, это отразится на всех остальных разделах (естественно, сказанное не относится к переменным-константам).

Модульное программирование определяет потребность в задании правил видимости, которым должны следовать все программисты. Все переменные снабжаются функциями (интерфейсами), посредством которых осуществляется работа с ними. Суть заключается в том, что внутренние изменения модулей не отражаются на способах их использования.

У модульного программирования есть и недостатки, но в свое время эта концепция помогла написать множество библиотек многократно используемых функций. Создавать подключаемые модули очень удобно, поскольку их интерфейсы редко меняются. Кроме того, программисты не обязаны были знать детали их реализации, а это упрощало модификацию модулей.

Абстрактное программирование: отсутствие ненужной детализации

Иногда в процессе реализации обнаруживается столько вариантов, что программисту приходится прибегать к абстракции. Применяя модульное программирование на абстрактном уровне, можно представить данные как некие информационные блоки, не связанные с конкретной структурой модуля. Классическим примером является очередь, организованная по принципу FIFO (First In, First Out — первым пришел, первым обслужен). Программе не требуется знать, что находится в очереди, нужно лишь принимать и извлекать элементы.

Первые программные абстракции были реализованы для очередей, стеков, деревьев, словарей, массивов и т.п. С каждым из перечисленных объектов связан свой набор методов, с которыми работают все программы. К примеру, в пятой версии операционной системы UNIX, прежде чем она была разделена, существовало 10 реализаций очереди. Применяя абстрактное программирование, можно избежать подобного дублирования усилий и сосредоточиться на главном.

Решая поставленную задачу, необходимо проверить, можно ли представить ее в общем виде. Не исключено, что существуют части программы, которые применяются многократно в различных ситуациях. Для многих задач уже имеются отлаженные обобщенные алгоритмы решения.

Абстрактный подход к программированию оказался существенным шагом вперед. Благодаря ему программист избавляется от необходимости знать о том, с какими данными работает программа, и может сосредоточить усилия на решении основной задачи.

Объектно-ориентированное программирование: естественный способ общения с миром

В настоящее время программирование развивается в объектно-ориентированном направлении. В объектном подходе концепция модульного программирования расширяется двумя вопросами: "Каким знанием обладает программа?" и "Что она делает?" Возможно, правильнее было бы употреблять термин *моделирование обязанностей*.

Объекты приближают программу к реальному миру. Все в природе наделено свойствами (атрибутами) и поведением (функциями или методами), а также внутренними особенностями. Дети наследуют черты своих родителей. Все это находит отражение в объектах.

Рациональные методы программирования

Конечная цель и мечта любого программиста — избежать многократного написания одного и того же кода. Как было бы здорово — написать программу, которую можно использовать снова и снова, лишь незначительно модифицируя для каждого конкретного случая!

Чтобы достичь заветной вершины, программист должен придерживаться двух ориентиров: принципов повторного использования и подключаемое™. Ниже данные понятия раскрываются более подробно.

Повторное использование кода

При правильном применении объектно-ориентированный поход позволяет создавать программы, которые могут повторно использоваться другими программистами. Священная цель всех технологий программирования — "напиши один раз, используй многократно". Благодаря объектам эта цель становится реально достижимой.

Сегодня программисты в основном сталкиваются с теми же проблемами, что и предыдущее поколение программистов. Большинство проблем остались неизменными, и их приходится решать снова и снова. В области сетевого программирования имеется ряд хорошо проработанных решений, например почтовые системы и HTTP-серверы, но, к сожалению, они надежно защищены законами об авторских правах.

Принцип повторного использования заключается в том, чтобы не изобретать колесо, а брать за основу готовые решения. Представьте, сколько строился бы дом, если бы строителям пришлось, кроме всего прочего, отливать металл для гвоздей. И это сегодня, когда в заводских условиях изготавливаются целые дома, доставляемые по частям прямо на участок в течение трех дней! Почему нельзя делать то же самое в компьютерной индустрии? Можно, но для этого требуется дисциплина.

Синдром ПНМ (придумано не мной)

На самой первой своей работе мне посчастливилось иметь дело с наиболее талантливыми и умными инженерами, с которыми я когда-либо сталкивался. Стремление выделиться из толпы требовало от меня постоянных умственных усилий. Я обратил внимание на одну характерную черту, свойственную всем этим людям и ограничивающую их возможности. Их отличала подверженность синдрому ПНМ (придумано не мной), который проявлялся в следующем высказывании: "Если это сделано не нами, значит, зто сделано неправильно .(или не идеально)". Я убедился в том, что подобная ошибочная (если не сказать заносчивая) .точка зрений распространена практически во всех фирмах, занимающихся разработкой программного обеспечения.

Принцип повторного использования можно понимать двояко: как поиск готовых решений и как продвижение своих собственных разработок. Первый случай понятен — это лишь вопрос доверия между вами и сторонним программистом. Важно также наличие канала связи с ним.

Во втором случае вы сами становитесь вершителем чужих судеб. Требуется, чтобы создаваемые библиотеки функций были максимально надежными и чтобы вы имели возможность быстро реагировать на запросы пользователей. Кроме того, если есть проверенный способ распространения обновлений, становится проще выпускать новые версии продукта. Когда программа функционирует хорошо и имеет интуитивно понятный интерфейс, пользователи часто обращаются за новыми версиями.

Важно уметь мыслить обобщенно. Представьте проблему в виде головоломки, состоящей из набора элементов. Чем в более обобщенном виде будут представлены все элементы, тем выше вероятность того, что другие пользователи смогут повторно их использовать.

Создание подключаемых компонентов

Другая важнейшая цель объектно-ориентированного программирования заключается в возможности заменять один модуль другим, улучшенным его вариантом. К примеру, батарейки делаются различными производителями и имеют разные характеристики, но используются одинаковым образом. Они могут быть обычными гальваническими или же новыми, ионно-литиевыми. Несмотря на это, у всех батареек общий интерфейс, т.е. два полюса зарядов: один положительный и один отрицательный. Батарейки первого типа дешевле, а вторые могут перезаряжаться и более долговечны.

Придерживаясь такого подхода, следует обеспечивать постоянство интерфейса модуля. Тогда в любой ситуации его можно будет безболезненно заменить обновленной версией (с улучшенной производительностью, повышенной надежностью, исправленными ошибками и т.д.). Возможность взаимной замены библиотек и модулей называется *подключаемостью.*

При написании подключаемого кода следует придерживаться определенных правил. Прежде всего необходимо определить интерфейс и неукоснительно его придерживаться. Тщательно продуманный интерфейс обеспечивает долговечность модуля. Если интерфейс больше не соответствует требованиям сегодняшнего дня, разработчик переписывает его. Хорошим примером гибкого интерфейса является системный вызов `socket()`. Выглядевшая поначалу неудобной, эта функция на самом деле оказалась столь гибкой, что благополучно "пережила" смену нескольких сетевых технологий.

Вторым правилом является *минимализм.* Интерфейс следует делать как можно более простым, чтобы к нему было проще подстраиваться. *Связность* — это число различных элементов данных, передаваемых в модуль. Чем выше связность, тем сильнее зависимость от модуля и тем сложнее его встраивать. Что касается сокетов, то они представляют собой смесь простых интерфейсов и сложных процедур взаимодействия. Чтобы заставить сокет работать, необходимо выполнить несколько системных вызовов (до семи). Это позволяет адаптироваться к самым разным ситуациям, однако существенно усложняет программирование.

Третье, и последнее, правило — *многослойность.* Это не просто разбиение данных на модули. Имеется в виду многоуровневый подход ко всей технологии. В

качестве примера можно привести модель OSI. На каждом ее уровне свои законы и интерфейсы.

Основы объектно-ориентированного программирования

Как уже упоминалось, объектно-ориентированное программирование (ООП) на сегодняшний день является ведущей технологией программирования. В ней реализован целый ряд новых концепций, позволяющих решать большинство существующих проблем. Основными из них являются абстракция, полиморфизм, наследование и инкапсуляция.

Особенность ООП заключается в том, что его можно применять везде, даже в обычных процедурных языках и языках ассемблера. Все, в конце концов, сводится к пониманию методики и дисциплине программирования.

Инкапсуляция кода

Первой ключевой концепцией ООП является инкапсуляция. Она критически важна для реализации принципа повторного использования, так как предписывает скрывать все детали реализации в защищенной оболочке интерфейсов. В большинстве случаев глобальные переменные совершенно недопустимы в ООП. Все данные неразрывно связаны со своими объектами. Если эта связь нарушается, теряется возможность повторного использования и автономного встраивания модуля.

Глобальные объекты

В хорошей программе не должно быть глобальных данных. Все данные должны модифицироваться только их непосредственными владельцами. Возникает вопрос: как следует интерпретировать глобальные объекты? Правило гласит, что они не считаются глобальными данными. Их вполне можно создавать, и на практике это используется очень часто.

В объекте инкапсулируются данные, о которых никто не должен знать или доступ к которым должен быть ограничен. Есть два различных типа инкапсулируемой информации: внутренние данные и внутренняя реализация. В первом случае от посторонних глаз прячутся сами данные и их структура.

Предположим, есть объект, реализующий функции счетчика. Внешним программам не нужно знать, как именно в объекте хранятся числа. Если какая-нибудь программа напрямую обращается к внутренней переменной объекта, то при модификации объекта потребуется изменить и внешнюю ссылку на него.

Под внутренней реализацией понимаются локальные функции объекта. Они носят служебный характер и предназначены для упрощения программирования. Никакой связи с внешними программами они не имеют.

Все интерфейсы, которые существуют между объектом и внешним миром, связаны с его функциональными характеристиками. Обычно эти функции называются *сервисами*. Все остальные функции инкапсулированы внутри объекта и недоступны извне.

Наследование поведения

Допустим, нужно написать модуль, который делает то же самое, что и другой модуль, но несколько функций в нем отличаются. Добиться этого можно с помощью механизма наследования. Старая поговорка гласит: "Обращайтесь с ребенком как со взрослым, и он будет вести себя правильно".

У объекта есть ряд атрибутов и методов. Можно создать производный от него объект, который унаследует все свойства предка, изменив некоторые из них и добавив ряд своих. На рис. 11.1 изображена иерархия объекта Device (устройство), у которого есть ряд потомков.

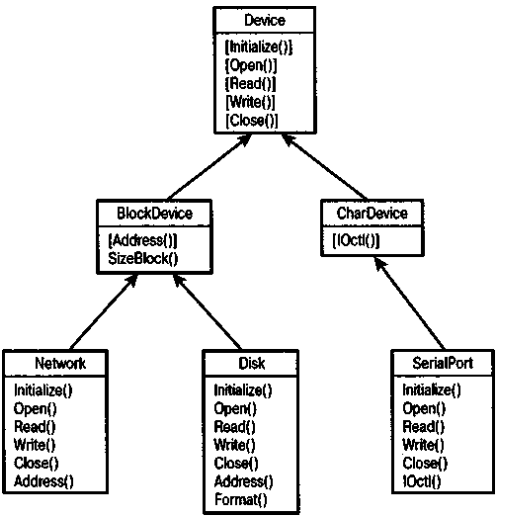

Рис. 11.1. В иерархию наследования объекта Device (устройство) входят два абстрактных объекта — BlockDevice (блок-ориентированное устройство) и CharDevice (байт-ориентированное устройство) — и три обычных: Network (сеть), Disk (диск) и SerialPort (последовательный порт)

Глядя на иерархию, можно сказать, что "последовательный порт (SerialPort) является байт-ориентированным (CharDevice) устройством (Device)". В объекте Device определены пять интерфейсных (абстрактных) функций: initialize(), open(), read(), write() и close(). В объекте CharDevice к ним добавляется функция IOctl(). Наконец, в объекте SerialPort все эти функции реализованы так, как это требуется в случае последовательного порта.

Абстракция данных

Третья базовая концепция, абстракция, схожа с описанной ранее моделью абстрактного программирования, но немного ее расширяет. В абстрактном программировании разработчик сосредоточивает свои усилия на создании алгоритма функции, игнорируя тип данных, с которыми она имеет дело. Таким способом пишутся базовые вычислительные структуры, в частности стеки и словари.

В ООП абстракция означает концентрацию усилий вокруг модулей, которые можно постепенно расширять и совершенствовать. Абстракция тесно связана с наследованием и позволяет создавать объекты, которые на самом деле не существуют в природе. Возьмем такой пример. Чау-чау — это порода собак. Собаки относятся к семейству псовых отряда хищных класса млекопитающих. В природе нет такого животного, как пес, хищник или млекопитающее. Тем не менее их характеристики позволяют отличить собаку от кошки, оленя или рыбы. Абстрактный объект обладает атрибутами, общими для всей иерархии, но их реализация может быть оставлена потомкам.

В отличие от абстрактных объектов, в обычных объектах все методы полностью реализованы. Если снова обратиться к рис. 11.1, то объекты CharDevice и Device являются чистыми абстракциями. Они определяют, какие интерфейсы должны быть реализованы в их потомках. Кроме того, в объекте CharDevice можно задать стандартные реализации интерфейсов, чтобы в последующих объектах иерархии на них можно было опираться.

Благодаря абстракции можно вызвать функцию объекта, не зная, как именно она реализована. Например, у объекта Device есть два потомка: Disk и Network. В следующем фрагменте программы создается переменная dev абстрактного типа Device, а затем в нее помещается ссылка на объект Disk:

```
/* Создаем объект Disk и записываем ссылку на него */
/* в переменную dev абстрактного типа Device       */
/*************************************************/
Device *dev = new Disk();
dev->Initialize();
```

Хотя в объекте Device метод Initialize() не реализован, а лишь объявлен, компилятор понимает, что данный метод относится к объекту Disk. В то же время метод Address() нельзя вызвать по отношению к объекту Device, так как он объявлен ниже в иерархии. Чтобы получить к нему доступ, необходимо немного модифицировать код:

```
BlockDevice *dev = new Disk();
dev->Address();
```

Нужный интерфейс выбирается при объявлении переменной. Конечно, можно просто привести переменную к требуемому типу, но не во всех языках программирования обеспечивается контроль типов во Время операции приведения.

Полиморфизм методов

Четвертая, и последняя, из базовых концепций ООП — *полиморфизм*. Его суть заключается в том, что родственные методы, реализованные на разных уровнях иерархии, могут иметь одинаковые имена. Это упрощает их запоминание.

Принадлежность метода к тому или иному объекту определяется на основании его аргументов. Например, вместо двух методов — PlayVideo() и PlayMIDI() — можно создать один метод Play(), которому в первом случае передается видеоклип, а во втором случае — MIDI-файл.

Поскольку компоновщик не может определить, к какому объекту относится "двусмысленный" метод, это должен сделать компилятор. В процессе уточнения имен компилятор разрешает все неоднозначности, вставляя перед именем метода имя его объекта.

Характеристики объектов

С объектами связан целый ряд терминов и определений. Важно понимать их назначение, чтобы правильно употреблять.

Классы и объекты

Программисты часто путаются в терминах "класс" и "объект". Класс — это описание категории родственных объектов, а объект — это конкретный экземпляр класса. Класс можно сравнить с чертежом, а объект — с домом, построенным на основании этого чертежа.

Читателям наверняка известен тип данных struct языка C, описывающий структуру, которая состоит из набора полей. Класс — это разновидность структуры, в которую кроме переменных могут также входить и функции. Создать объект класса — это то же самое, что создать переменную, в качестве типа которой указан тэг структуры.

Замечание

До сего момента термин *объект* означал как класс, так и собственно объект. Это было сделано умышленно, чтобы уменьшить путаницу.

В иерархии наследования есть два типа классов: *надкласс* (родительский класс) и *подкласс* (производный, или дочерний, класс). Надкласс, находящийся на самом верхнем уровне иерархии, называется *суперклассом* и обычно является абстрактным. Любой класс, наследующий свое поведение от некоторого родительского класса, называется подклассом.

Атрибуты

Отдельные поля структуры становятся атрибутами в классе. Атрибуты могут быть как скрытыми (инкапсулированными), так и *опубликованными* (являющимися частью открытого интерфейса класса). Как правило, все атрибуты класса скрыты от внешнего мира.

Свойства

Опубликованные атрибуты класса называются *свойствами*. Обычно они доступны только для чтения или же представлены в виде связки функций *Getxxx()* и Setxxx(), позволяющих получать и задавать их значения.

Семейства Get() и Set()

Функции семейств Get() и Set() не считаются методами. Они относятся к особому набору функций, предназначенных исключительно для извлечения и установки свойств объектов, и не расширяют их функциональные возможности.

Подклассы наследуют свойства своих родительских классов.

Методы

Методы — это действия, которые выполняет объект. Благодаря полиморфизму дочерние классы могут переопределять методы, унаследованные от своих родительских классов. Обычно при этом вызывается исходный метод, чтобы родительский класс мог проинициализировать свои закрытые атрибуты.

Атрибуты, свойства и методы называются *членами* класса.

Права доступа

Права доступа определяют, кто может обращаться к той или иной части класса. В Java и C++ определены три уровня доступа: private, protected и public.

- private. Доступ к членам класса разрешен только из его собственных методов.
- protected. Доступ к членам класса разрешен из его методов, а также из методов его подклассов.
- public. Доступ к членам класса разрешен отовсюду.

Эти ограничения можно ослаблять, создавая дружественные классы и функции.

Отношения

Объекты взаимодействуют друг с другом тремя различными способами. Первый — наследование — был описан выше. Все три способа идентифицируются ключевыми фразами: *является, содержит, использует.* На этапах анализа и проектирования эти отношения помечаются специальными значками.

- Отношение "является". Наследование: один класс наследует свойства и методы другого.
- Отношение *"содержит"*. Включение: один класс является членом другого.
- Отношение *"использует"*. Использование: один класс объявлен дружественным другому или же вызывает его открытые методы.

Между двумя объектами должно существовать только одно отношение. Например, объект А не может быть потомком объекта Б и в то же время содержать его в качестве встроенного объекта. Подобная двойственность отношений свидетельствует о неправильно выполненном анализе.

Расширение объектов

Когда есть хороший объект, всегда хочется его расширить или еще улучшить. Помните о том, что перечисленные ниже способы расширения поддерживаются не во всех языках.

Шаблоны

Благодаря абстракции и полиморфизму можно создавать в родительских классах обобщенные методы, конкретная реализация которых предоставляется в дочерних классах. А теперь представьте, что существуют обобщенные классы, экземплярами которых являются конкретные классы. В C++ такие классы называются *шаблонами*. В них дано описание методов, не зависящее от конкретного типа данных.

При создании конкретного экземпляра шаблона указывается, с данными какого типа будет работать этот класс. Обычно на базе шаблонов создаются контейнеры объектов. Например, очередь и стек работают независимо от того, данные какого типа в них находятся.

Постоянство

Обычно программисты имеют дело с объектами, которые прекращают свое существование по завершении программы. Но в некоторых программах пользователю разрешается задавать параметры, влияющие на работу объектов. Эти параметры должны оставаться в силе при последующих запусках программы.

Такого рода информация сохраняется в файле. Когда программа запускается на выполнение, она загружает свои параметры из файла и продолжает работу с того момента, с которого ее прекратила. Можно даже сделать так, что в случае системного сбоя программа незаметно для пользователя восстановит разорванное соединение.

Потоковая передача данных

Представьте, что объект пакует сам себя и записывает на диск либо посылает куда-то по сети. По достижении места назначения или же когда программа загружает объект, он автоматически распаковывает себя. Подобная методика используется при обеспечении постоянства объектов и в распределенном программировании. В частности, она реализована в Java. Элементы потоковой обработки можно применять и в C++, но самоидентификация объектов (называемая *интроспективным анализом* в Java) здесь невозможна.

Перегрузка

Перегрузка операторов (поддерживаемая в C++) расширяет концепцию полиморфизма, Некоторые программисты ошибочно полагают, что это тождественные понятия, но на самом деле перегрузка операторов является объектным расширением C++. В Java, например, она не поддерживается, хотя этот язык считается объектно-ориентированным.

В C++ разрешается добавлять новый (не переопределять старый!) смысл к внутренним операторам языка. С ними можно обращаться как с полиморфными методами класса, создавая дополнительные реализации, которые работают с новыми типами данных. Перегрузка операторов подвержена целому ряду ограничений.

- *Расширение, а не переопределение.* Нельзя создавать операторы, чьи операнды относятся к тому же типу, что и раньше; например, нельзя поменять смысл операции (int)+(int).
- *Не все операторы доступны.* Переопределять можно практически все операторы, за исключением нескольких (например, условный оператор ?:).
- *Число параметров должно совпадать.* Новый оператор должен иметь то же число операндов, что и его исходная версия, т.е. он должен быть либо унарным (один операнд), либо бинарным (два операнда).

Нельзя также изменить приоритет оператора и его ассоциативность (порядок вычисления операндов).

Интерфейсы

В C++ существует одна проблема. Если класс А использует класс Б, структура последнего должна быть непосредственно известна программисту или же одним из предков класса Б должен быть класс, являющийся потомком для А. Когда класс Б поддерживает несколько интерфейсов, возникают неприятности с множественным наследованием.

В Java можно просто объявить, что класс поддерживает конкретный абстрактный интерфейс. Это не налагает на сам класс никаких ограничений или дополнительных обязательств и не добавляет к нему лишнего кода.

События и исключения

Последнее расширение связано с восстановлением работоспособности программы в случае ошибок. В языке С это постоянная проблема, так как обработка ошибок, как правило, ведется не там, где они возникают.

С помощью исключений можно задать, когда и как следует обрабатывать конкретные виды ошибок, возникающих в объектах, причем обработчик исключений является членом того же класса, что и объект, в котором возникла ошибка. События — это ситуации, возникающие асинхронно и вызывающие изменение хода выполнения программы. Обычно они применяются в интерактивных программах, которые либо дожидаются наступления нужного события, чтобы начать выполняться, либо выполняются, пока не наступит требуемое событие.

Особые случаи

Объектная технология сталкивается с теми же проблемами, что и другие технологии программирования. Не все задачи можно сформулировать в терминах ООП. Ниже мы рассмотрим особые виды классов и попытаемся разобраться, как с ними работать.

Записи и структуры

Правило гласит, что класс без методов является записью. Часто именно записи служат основным средством представления данных. Например, в базах данных информация хранится в виде записей.

Запись, или структура, — это неупорядоченный набор данных. Единственные методы, присутствующие в нем, — это функции вида Get() и Set(). Объекты подобного рода встречаются редко. Необходимость их существования проверяется следующими вопросами.

- *Существует ли тесная связь между полями записи?* Например, может оказаться, что при модификации одного поля должно измениться другое. Следует выявить связанные таким способом поля и запрограммировать их изменения в функциях Set().
- *Является ли объект частью более крупной системы?* В базах данных на многие отношения, существующие между таблицами, наложены ограничения в виде деловых правил, обеспечивающих целостность данных. Если видеть только одну сторону отношения, то может казаться, что никаких функций не требуется. На самом деле функции связаны со всем отношением в целом.

Наборы функций

Набор функций противоположен записи, т.е. набору данных. Это класс, в котором присутствуют только методы, но нет никаких атрибутов и свойств. В качестве примера можно привести библиотеку математических функций. Такие функции выполняются над числами типа int и float, но они не связаны с ними и не формируют с ними единый класс.

Если в результате проектирования у вас получился класс, представляющий собой набор функций, проверьте его правильность следующими вопросами.

- *Правильно ли распределены обязанности объектов?* Возможно, процесс проектирования зашел дальше, чем нужно, в результате чего нарушилась атомарность объектов — были созданы два объекта вместо одного. Не исключено, что некоторые из них следует объединить.
- *Существует ли тесная взаимосвязь между классами?* Когда один класс активно вызывает методы другого класса, то, возможно, некоторые из них просто принадлежат неправильному классу.
- *Связаны ли между собой методы ?* Любой метод должен явно или неявно влиять на выполнение других методов класса. Если этого не происходит, значит, методы не связаны друг с другом.

Языковая поддержка

Поддержка объектов внедрена во многие современные языки программирования. Существует даже объектная версия Cobol. Классическими объектно-ориентированными языками являются SmallTalk и Eiffel. Объектно-ориентированными можно считать также языки Java и C++, знакомству с которыми будут посвящены две последующие главы. В то же время поддержка объектов во всех этих языках реализована настолько по-разному, что необходимо провести их классификацию.

Классификация объектно-ориентированных языков

В действительности лишь некоторые языки являются *истинно* объектно-ориентированными. Среди распространенных языков к таковым можно отнести, пожалуй, лишь SmallTalk. Любые действия, выполняемые в этом языке, рассматриваются как действия над объектами.

По отношению к таким языкам, как C++ и Java, можно сказать, что они *предназначены* для работы с объектами. В них поддерживаются практически все концепции ООП, и на этих языках пишутся очень эффективные объектно-ориентированные программы. Но ничто не мешает вам, к примеру, в среде C++ написать и скомпилировать обычную C-программу. В ней даже можно создать квазиобъекты с помощью типа данных struct. Даже Java-программа может представлять собой одну большую функцию `main()`. Особенностью таких языков является то, что они не заставляют программиста придерживаться всех принципов объектной технологии.

Наиболее слабой формой объектной ориентации является *поддержка*. В таких языках элементы объектной технологии служат дополнением к базовым возможностям языка, и применять их необязательно. В качестве примера можно привести Perl и Visual Basic. Поскольку объектные возможности этих языков ограничены, в них вводится универсальный тип данных variant, с помощью которого обеспечивается абстракция. Однако появление такого типа нарушает принцип инкапсуляции, так как программа, принимающая данные типа variant, должна знать их внутреннюю структуру.

Работа с объектами в процедурных языках

Объектная технология — замечательная вещь, когда программа пишется на объектно-ориентированном языке (таком как C++ или Java). Но это не всегда возможно. Что делать в подобном случае?

Объектная технология создавалась на базе более ранних технологий (абстрактное и модульное программирование), поэтому многие элементы ООП можно реализовать и в обычных процедурных языках. Но для этого требуется определенная дисциплина программирования.

- *Инкапсуляция.* Необходимо представить все интерфейсы модуля в виде процедур или функций. Им следует присваивать имена вида *<имя_модуля>_<имя_функции>*().

- *Абстракция.* Если язык поддерживает операцию приведения типа, то абстракцию можно обеспечить, записывая в одно из полей структуры идентификатор требуемого типа данных.
- *Классы и объекты.* Можно создать их прообразы, если в языке поддерживаются записи или структуры.
- *Атрибуты.* Это просто поля записи.
- *Свойства.* Для каждого опубликованного свойства необходимо создать связку функций Get( )/Set().
- *Методы.* Если в каком-либо языке не поддерживаются функции (только процедуры), можно эмулировать их, возвращая значение через один из параметров процедуры.
- *Отношения.* Отношения включения и использования доступны в любом языке.
- *Постоянство.* Можно самостоятельно отслеживать параметры программы и загружать их при запуске.
- *Потоковая передача.* Чтобы упаковывать и распаковывать данные, необходимо знать их внутреннюю структуру и иметь возможность выполнять приведение типов.
- *События и исключения.* Можно эмулировать обработку событий и исключений, но для этого потребуется выполнять переходы между функциями (например, с помощью функции setjump() в языке C), а это не очень хорошая идея.

Резюме: объектно-ориентированное мышление

Объектная технология — это основа хороших программных разработок. Правильно применяя принципы ООП (абстракция, полиморфизм, наследование и инкапсуляция), можно создавать эффективные программные модули, которые будут многократно использоваться другими программистами. Кроме того, их легче отлаживать и расширять, чем обычные программы. Наконец, при объектно-ориентированном подходе программист больше занят анализом предметной области, чем собственно программированием.

Глава 12

Сетевое программирование в Java

В этой главе...

До сего момента вопросы сетевого программирования и, в частности, работы с сокетами рассматривались применительно к языку C. Его достоинства очевидны для системного программиста, но в результате получаются программы, которые не всегда переносимы и не всегда допускают многократное использование.

Java — прекрасный пример объектно-ориентированного языка, в котором можно создавать многократно используемые, переносимые компоненты. В Java обеспечиваются два вида переносимости: на уровне исходного текста и на уровне кода. Переносимость первого типа означает, что все программы должны компилироваться на любой платформе, где поддерживается сам язык Java. (Компания Sun Microsystems оставляет за собой право объявлять некоторые интерфейсы, методы и классы устаревшими и не рекомендуемыми для дальнейшего использования.)

Концепция переносимости на уровне кода в действительности не нова. Принцип "скомпилировал однажды — запускай везде" легко реализовать, имея соответствующие средства. Java-программа компилируется в байт-код, который выполняется в рамках виртуальной машины. Виртуальная машина Java (JVM — Java Virtual Machine) интерпретирует каждую команду последовательно, подобно микропроцессору. Конечно, скорость интерпретации байт-кода не сравнится со скоростью выполнения машинных кодов (создаваемых компилятором языка C), но, поскольку современные процессоры обладают очень высоким быстродействием, потеря производительности не столь заметна.

Java — простой и в то же время интересный язык. Обладая навыками программирования в C/C++ и разбираясь в особенностях объектной технологии, можно быстро изучить его. В этом языке имеется очень мощный, исчерпывающий набор стандартных библиотек классов, в котором не так-то легко ориентироваться. Поэтому не помешает всегда держать под рукой интерактивную документацию по JDK (Java Development Kit — комплект средств разработки в среде Java) и несколько хороших справочников.

В предыдущей главе рассказывалось о назначении объектной технологии. В этой главе речь пойдет о том, как программировать сокеты в объектно-ориентированной среде Java. Чтобы не вдаваться в чрезмерные детали, я предполагаю, что читатель знаком с Java и хочет изучать непосредственно сетевое программирование.

Прежде всего мы рассмотрим, какие классы существуют в Java для работы с сокетами, какие имеются средства ввода-вывода, как конфигурировать сокеты и работать с потоками.

Работа с сокетами

Многие программисты считают, что основные преимущества Java — независимость от интерфейса пользователя и встроенные сетевые возможности. Предпочтительным сетевым протоколом в Java является TCP. С ним легче работать, чем с дейтаграммами (протокол UDP), кроме того, это наиболее надежный протокол. В Java можно также посылать дейтаграммы, но напрямую подобная возможность не поддерживается базовыми библиотечными классами ввода-вывода.

Программирование клиентов и серверов

Каналы потоковой передачи (TCP-соединения) лучше всего соответствуют возможностям Java. Java пытается скрывать детали сетевого взаимодействия и упрощает сетевые интерфейсы. Многие операции, связанные с поддержкой протокола TCP, перенесены в библиотечные классы ввода-вывода. В результате в создании сокетов принимает участие лишь несколько объектов и методов.

TCP-клиенты Java

Вот как, например, создается клиентский сокет:

```
Socket s = new Socket(String Hostname, int PortNum);
Socket s = new Socket(InetAddress Addr, int PortNum);
```

Самый распространенный вариант таков:

```
Socket s = new Socket("localhost", 9999);
```

Для подключения к серверу больше ничего не требуется. Когда виртуальная машина (ВМ) создает объект класса Socket, она назначает ему локальный номер порта, выполняет преобразование данных в сетевой порядок следования байтов и подключает сокет к серверу. Если требуется дополнительно указать локальный сетевой интерфейс и порт, то это делается так:

```
Socket s = new Socket(String Hostname, int PortNum,
                      InetAddress localAddr, int localPort);
Socket s = new Socket(InetAddress Addr, int PortNum,
                      InetAddress localAddr, int localPort);
```

Класс InetAddress преобразует имя узла или IP-адрес в двоичный адрес. Чаще всего с объектом этого класса не работают напрямую, так как проще сразу вызвать конструктор Socket(), которому передается имя узла.

Поддержка стандартов IPv4/IPv6

В настоящее время Java поддерживает стандарт IPv4. Согласно проекту Merlin (информацию можно получить на Web-узле java.sun.com), поддержка стандарта IPv6, появится, когда она будет внедрена в операционные системы. Классы наподобие InetAddress, осуществляющие преобразование имен, должны легко адаптироваться к новым протоколам. Но очевидно, что некоторые функции, например InetAddress.getHostAddress(), придется заменить при переходе на новый, расширенный формат адресов.

Прием/отправка сообщений

После создания объекта класса Socket программа еще не может посылать или принимать через него сообщения. Необходимо предварительно связать с ним входной (класс InputStream) и выходной (класс OutputStream) потоки:

```
InputStream i = s.getInputstream();
OutputStream o = s.getOutputStream();
```

Чтение и запись данных осуществляются блоками:

```
byte[] buffer = new byte[1024];
int bytes_read = i.read(buffer); // чтение блока данных из сокета
o.write(buffer); // запись массива байтов в сокет
```

С помощью метода InputStream.available() можно даже определить, поступили данные во внутренние буферы ядра или нет. Этот метод возвращает число байтов, которые программа может прочитать, не рискуя быть заблокированной.

```
if ( i.available() > 100 ) // чтение не производится, если
                           в буфере меньше 100 байтов
    bytes = i.read(buffer);
```

После завершения работы можно закрыть сокет (а также все каналы ввода-вывода) с помощью одного-единственного метода Socket.close():

```
// очистка
s.close();
```

Ручная очистка и автоматическая уборка мусора

Все объекты в Java создаются, с помощью оператора new. Они передаются программе через указатели, но сами указатели скрыты в теле классов, поэтому они не освобождаются явно. Когда виртуальная машина сталкивается с нехваткой ресурсов, она запускает процесс уборки мусора, во время которого неиспользуемые указатели освобождаются и память возвращается системе. Тем не менее при работе с сокетами их нужно самостоятельно закрывать, так как, в конце концов, лимит дескрипторов файлов может оказаться исчерпанным.

В листинге 12.1 показано, как создать простейший эхо-клиент.

Листинг 12.1. Пример простейшего эхо-клиента в Java

```
//*****************************************************************
// Простейший эхо-клиент (из файла SimpleEchoClient.Java)
//*****************************************************************
Socket s = new Socket("127.0.0.1, 9999"); // создаем сокет
InputStream i =    s.getInputstream();    // создаем входной поток
OutputStream o = s.getOutputStream();     // создаём выходной поток
String str;
do
{
   byte[] line = new byte[100];
   System.in.read(line);           // читаем строку с консоли
   o.write(line);                  // посылаем сообщение
   i.read(line);                   // принимаем его обратно
   str = new String(line);         // преобразуем его в строку
   System.out.println(str.trim()); // отображаем сообщение
}
while ( !str.trim().equals("bye") );
s.close();                         // закрываем соединение
```

В этом примере продемонстрирован простой цикл чтения и отправки сообщений. Он еще не доведен до конца, так как при попытке компиляции будет выдано предупреждение о том, что не перехватываются некоторые исключения. Пере-

хват исключений — это важная часть любых сетевых операций. Поэтому к показанному тексту нужно добавить следующий программный код:

```
try
{
   // <- Здесь должен размещаться исходный текст
}
catch (Exception err)
{
   System.err.println(err);
}
```

Блоки try...catch делают пример завершенным. Полученный текст можно вставить непосредственно в метод `main()` основного класса программы.

TCP-серверы Java

Как можно было убедиться, Java упрощает создание и закрытие сокетов, а также чтение и запись данных через них. Работать с серверами еще проще. Серверный сокет создается с помощью одного из трех конструкторов:

```
ServerSocket s = new ServerSocket(int PortNum);
ServerSocket s = new ServerSocket(int PortNum, int Backlog);
ServerSocket s - new ServerSocket(int PortNum, int Backlog,
                                  InetAddress BindAddr);
```

Параметры Backlog и BindAddr заменяют собой вызовы С-функций listen() (создание очереди ожидания) и `bind()` (привязка к конкретному сетевому интерфейсу). Если вы помните, в языке С текст серверной программы занимал 7—10 строк. В Java то же самое можно сделать с помощью двух строк:

```
ServerSocket s = new ServerSocket(9999);
Socket c = s.accept();
```

Назначение объекта ServerSocket состоит лишь в организации очереди ожидания. Когда поступает запрос от клиента, сервер с помощью метода ServerSocket.accept() создает новый объект класса Socket, через который происходит непосредственное взаимодействие с клиентом.

В листинге 12.2 показано, как создать простейший эхо-сервер.

Листинг 12.2. Пример простейшего эхо-сервера в Java

```
// Простейший эхо-сервер (из файла SimpleEchoServer.Java)
//*********************************************************************
try   .
{
   ServerSocket s = new ServerSocket("9999"); // создаем сервер
   while (true)
   {
      Socket c = s.accept(); // ожидаем поступления запросов
      InputStream i = c.getInputstream(); // входной поток
      OutputStream o =c.getOutputStream(); // выходной поток
      do
```

```
            byte[] line = new byte[100]; // создаем временный буфер
            i.read(line);        // принимаем сообщение от клиента
            o.write(line);       // посылаем его обратно
          }
          while ( !str.trim().equals("bye") );
          c.close();                    // закрываем соединение
       }
    }
    catch (Exception err)
    {
       System.err.println(err);
    }
```

Передача UDP-сообщений

Иногда возникает необходимость передавать сообщения в виде дейтаграмм, т.е. по протоколу UDP. В Java есть ряд классов, которые позволяют работать с UDP-сокетами. Основной из них — это класс DatagramSocket.

UDP-сокет создается очень просто:

```
DatagramSocket s = new DatagramSocket();
```

При необходимости можно указать локальный порт и сетевой интерфейс:

```
DatagramSocket s = new DatagramSocket(int localPort);
DatagramSocket s = new DatagramSocket(int localPort,
                                      InetAddress localAddr);
```

Сразу после своего создания UDP-сокет готов к приему и передаче сообщений. Это осуществляется в обход стандартных классов ввода-вывода. UDP-пакет формируется с помощью класса DatagramPacket, которому передается массив байтов:

```
DatagramPacket d = new DatagramPacket(byte[] buf, int len);
DatagramPacket d = new DatagramPacket(byte[] buf, int len,
                                      InetAddress Addr, int port);
```

Первый вариант конструктора предназначен для создания объекта, который принимает сообщение. С помощью второго конструктора создается отправляемый пакет. В нем дополнительно указываются адрес и порт назначения. Параметр buf ссылается на предварительно созданный массив байтов, а параметр len определяет длину массива *или* максимальную длину принимаемого пакета.

Чтобы изучить применение этих классов, рассмотрим пример, в котором два одноранговых компьютера обмениваются дейтаграммами. Схожие примеры приводились в главе 4, "Передача сообщений между одноранговыми компьютерами". В листинге 12.3 показан текст программы-отправителя.

Листинг 12.3. Создание UDP-сокета и отправка дейтаграммы

```
// Простейший отправитель дейтаграмм
//  (из файла SimplePeerSource.java)
```

```
DatagramSocket s = new DatagramSocket(); // создаем сокет
byte[] line = new byte[100J;
System.out.print("Enter text to send: ");
int len = System.in.read(line);
InetAddress dest =   // выполняем преобразование адреса
            InetAddress.getByName("127.0.0.1");
DatagramPacket pkt = // создаем дейтаграмму
            new DatagramPacket(line, len, dest, 9998);
s.send(pkt);          // отправляем сообщение
s.close();            // закрываем соединение
```

В данном примере после отправки дейтаграммы соединение сразу же закрывается. Это вполне допускается делать, даже если сообщение еще не покинуло локальный компьютер. Очередь сообщений будет существовать до тех пор, пока все находящиеся в ней сообщения не будут отправлены.

Текст программы-получателя представлен в листинге 12.4.

Листинг 12.4. Прием дейтаграммы и ее отображение

```
// Простейший получатель дейтаграмм
//  (из файла SimplePeerDestination.Java)

DatagramSocket s = new DatagramSocket(9998); // создаем сокет
byte[] line = new byte[100];
DatagramPacket pkt = // создаем буфер для поступающего сообщения
            new DatagramPacket(line, line.length);
s.receive(pkt);      // принимаем сообщение
String msg = new String(pkt.getData()); // извлекаем данные
System.out.print("Got message: " + msg);
s.close();           // закрываем соединение
```

Групповая передача дейтаграмм

Протокол UDP позволяет отправить одно сообщение нескольким адресатам. Передача сообщения может происходить как в режиме группового вещания, так и в режиме широковещания. Последний не поддерживается в Java.

Чтобы принять участие в групповом вещании, программа подключается к определенному IP-адресу, зарезервированному для групповой рассылки. Все программы, входящие в группу, будут получать сообщения, посылаемые по этому адресу. Групповой сокет представляется в Java объектом класса MulticastSocket, у которого есть два конструктора:

```
MulticastSocket ms = new MulticastSocket();
MulticastSocket ms = new MuiticastSocket(int localPort);
```

Хотя программа может создать групповой сокет без привязки к порту, порт все же должен быть выбран, прежде чем программа сможет принимать сообщения. Причина заключается в том, что все групповые сокеты должны отфильтровывать ненужные сообщения.

Когда групповой сокет создан, он ведет себя так же, как и обычный UDP-сокет (объект класса DatagramSocket). Через него можно непосредственно отправлять и получать сообщения. Чтобы перейти в режим группового вещания, нужно присоединиться к группе. В стандарте IPv4 выделен следующий диапазон адресов для группового вещания: 224.0.0.0-239.255.255.255.

```
MulticastSocketms=newMulticastSocket(16900);
ms.joinGroup(InetAddress.getByName("224.0.0.1"));
ms.joinGroup(InetAddress.getByName("228.58.120.11"));
```

С этого момента сокет будет получать сообщения, посланные по адресам 224.0.0.1:16900 и 228.58.120.11:16900. Программа может отвечать как непосредственно отправителю дейтаграммы, так и всей группе сразу. Во втором случае не обязательно присоединяться к группе. Достаточно указать соответствующий адрес, и сообщение будет разослано всей группе.

Объектно-ориентированное программирование и планирование на будущее

Стандарт IPv6 поддерживает групповую передачу UDP-сообщений и планирует реализовать многоадресную доставку TCP-пакетов. Тем самым проблема ненадежности протокола UDP будет решена. Но дело в том, что класс MulticastSocket порожден от класса DatagramSocket, поэтому не может быть адаптирован к грядущим изменениям, Это хороший пример того, к чему приводит отсутствие планирования. Создавая иерархию объектов, лучше оставить место для последующих изменений или расширений, чем потом переделывать всю иерархию.

В листинге 12.5 показано, как создать и сконфигурировать групповой сокет.

Листинг 12.5. Создание группового сокета, привязка его к порту 16900, присоединение к группе и ожидание сообщений

```
//*************************************************************
// Простейший получатель групповых сообщений
//  (из файла SimpleMulticastDestination.Java)
//*************************************************************
MulticastSocket s = new MulticastSocket{16900); /•/ Создаем сокет
ms.joinGroup(InetAddress.getByName("224.0.0.1")); // присоединение
                                                  к группе
String msg;
do
{
   byte[] line = new byte[100];
   DatagramPacket pkt=new DatagramPacket(line, line.length);
   ms.receive(pkt);
   msg=newString(pkt.getData());
   System.out.println("From "+pkt.getAddress()+'4"tmsg.trim());
}
while ( !msg.trim().equalsf"close") );
ms.close();              // закрываем соединение
```

В этом примере создаваемый групповой сокет связывается с портом 16900, через который будут поступать сообщения. После подключения к адресу 224.0.0.1 программа формирует пакет, предназначенный для приема сообщений.

Ввод-вывод в Java

До сего момента в примерах использовались очень простые интерфейсы ввода-вывода. Сила сокетов в Java заключается еще и в том, что их можно связывать с самыми разными потоками ввода-вывода. В Java имеется целый ряд классов, обеспечивающих различные формы -чтения и записи данных.

Классификация классов ввода-вывода

В Java имеется шесть основных типов информационных потоков. Все связанные с ними классы служат определенным целям и порождаются от базовых классов Reader, Writer, InputStream и OutputStream.

- *Память.* Ввод-вывод, основанный на буферах памяти. Обращения к реальным аппаратным устройствам не происходит. Массивы, расположенные в ОЗУ, служат виртуальными накопителями данных. К данному типу потоков относятся такие классы, как ByteArraylnputStream, ByteArrayOutputStream, CharArrayReader, CharArrayWriter, StringReader и StringWriter.
- *Файл.* Ввод-вывод средствами файловой системы. Сюда относятся классы FilelnputStream, FileOutputStream, FileReader и FileWriter.
- *Фильтр.* Ввод-вывод, связанный с трансляцией или интерпретацией символьных потоков. Например, из входного потока могут выделяться записи, ограниченные символами новой строки, символами табуляции или запятыми. В эту группу входят классы FilterReader, FilterWriter, PrintWriter и PrintStream (устарел).
- *Объект.* Прием и передача целых объектов. Это одна из наиболее впечатляющих возможностей Java. Достаточно присвоить классу метку Serializable и можно передавать и принимать экземпляры его объектов. Данная возможность реализуется с помощью классов ObjectInputStream и ObjectOutputStream.
- *Канал.* Ввод-вывод, напоминающий механизм межзадачного взаимодействия в языке С. В программе создается канал, который связывается с другим каналом, после чего два программных потока могут обмениваться сообщениями. В эту группу входят классы PipedlnputStream, PipedOutputStream, PipedReader и PipedWriter.
- *Поток.* Общие средства буферизованного потокового ввода-вывода. Именно они используются сокетами. Сюда входят абстрактные классы InputStream и OutputStream. Если нужно передавать данные через сокет в каком-то более конкретном виде, необходимо выполнить преобразование потока в другую форму. Базовые функции преобразования реализуются классами InputStreamReader и OutputStreamWriter.

Есть также ряд классов, не попадающих под данную классификацию, например класс SequenceInputStream, позволяющий объединять два потока в один.

Преобразование потоков

Класс Socket может напрямую работать лишь с двумя классами ввода-вывода: InputStream и OutputStream. Они, в свою очередь, позволяют передавать и принимать только массивы байтов. Если нужно обрабатывать данные как-то иначе, необходимо преобразовать потоки сокета в другую форму.

Предположим, например, что требуется читать строки, а не массивы байтов. Работу со строками удобно вести с помощью класса BufferedReader. Соответствующее преобразование нужно выполнить через класс InputStreamReader, служащий посредником при переходе от классов семейства InputStream к классам семейства Reader:

```
Socket s = new Socket(host, port);
InputStream is = s.getInputstream();
InputStreamReader isr = new InputStreamReader(is);
BufferedReader br = new BufferedReader(isr);
String 1 = br.readLine();
```

Строки 2—5 можно записать короче:

```
BufferedReader br = new BufferedReader(new InputStreamReader(
                                        s.getInputstream()));
```

Передача строк осуществляется немного проще:

```
String msg = new String(
   "<html><body>Welcome to my Java website</body></html>");
SocketServer ss = new SocketServer(9999);
Socket s = ss.accept();
PrintWriter pw = new PrintWriter(s.getOutputStream(), true);
pw.println(msg);
s.close();
```

Первый параметр конструктора класса PrintWriter представляет собой ссылку на выходной поток сокета. Второй параметр сообщает объекту о том, что при каждом вызове метода println() необходимо осуществлять *автоматическую очистку буфера*. Обычно методы класса PrintWriter буферизуют все данные до тех пор, пока не будет достигнут баланс между размером пакета и пропускной способностью канала. В данном случае программа явно указывает на то, когда следует посылать пакеты.

Если обмен данными осуществляется между двумя Java-программами, можно воспользоваться классами ObjectInputStream и ObjectOutputStream для приема/передачи объектов, реализующих интерфейс Serializable:

```
String msg = new String("Test");
Socket s = new Socket(hostname, port);
ObjectOutputStream oos = new ObjectOutputStream(
                                    s.getOutputStream());
oos.writeObject(msg);
```

Принимающая сторона получает сообщение в виде объекта и преобразует его к требуемому типу:

```
Socket s = ss.accept();
ObjectInputStream ois = new ObjectInputStreatn(s.getInputStream());
String newMsg = (String) ois.readObject();
```

Если полученный объект не может быть приведен к указанному типу, интерпретатор сгенерирует исключение ClassCastException. В этом случае можно воспользоваться средствами интроспективного анализа классов, имеющимися в Java, чтобы узнать тип класса.

Конфигурирование сокетов

В главе 9, "Повышение производительности", рассказывалось о том, как работать с многочисленными параметрами сокетов. Аналогичные функции имеются и в Java. Следует, однако, учитывать, что если операционная система не поддерживает тот или иной параметр, то и Java его не предоставляет.

Общие методы конфигурирования

У всех сокетов в Java имеются методы, позволяющие их конфигурировать. Например:

```
getSoTimeout()
setSoTimeout(int timeout)
```

С помощью этих методов можно получить или задать значение параметра SO_TIMEOUT, определяющего период ожидания данных. Этот параметр давно устарел, поэтому в Linux вместо него применяются функции fcntl(), poll() и select().

Следующие два метода позволяют определить или задать размер (в байтах) внутреннего выходного буфера. Соответствующее значение хранится в параметре SO_SNDBUF.

```
getSendBufferSize()
setSendBufferSize(int size)
```

А эти два метода связаны со входным буфером (параметр SO_RCVBUF):

```
getReceiveBufferSize()
setReceiveBufferSize(int size)
```

Следующие методы позволяют соответственно определить и задать, должен ли сокет продолжать обработку буферизованных данных после своего закрытия и в течение какого времени (параметр SO_LINGER):

```
getSoLinger()
setSoLinger(boolean on, int linger)
```

Первый из показанных ниже методов определяет, используется ли алгоритм Нейгла, а второй метод включает или отключает его (параметр TCP_NODELAY):

```
getTcpNoDelay()
setTcpNoDelay(boolean on)
```

Конфигурирование групповых сокетов

Перечисленные здесь методы применимы к объекту MulticastSocket. Следующие два метода позволяют соответственно узнать и задать предельно допустимое число переходов, совершаемых пакетом (параметр IP_MULTICAST_TTL; странно, но Java не позволяет задавать значение TTL для других типов сокетов):

```
getTimeToLive()
setTimeToLive(int ttl)
```

Показанные ниже методы позволяют соответственно определить и задать основной сетевой интерфейс для группового вещания (параметр IP_MULTICAST_IF):

```
getInterface()
setInterface(InetAddress inf)
```

Многозадачные программы

Некоторые методики программирования в Java существенно упрощены. Наряду со встроенными средствами сетевого программирования и классами ввода-вывода в Java имеются встроенные средства многопотокового программирования. В результате создавать программные потоки стало очень просто.

Создание потокового класса

Чтобы создать потоковый класс, необходимо либо сделать его потомком класса Thread, либо реализовать в нем интерфейс Runnable. В обоих случаях требуется определить в классе метод run(), включающий код потока:

```
public class TestThread extends Thread
{
   public void run()
   {
      /*** здесь находится код потока ***/
```

Когда метод run() завершается, поток прекращает свою работу.

Если потоковый класс порождается от класса Thread, у него появляется метод start(), предназначенный для запуска потока. Ниже показано, как запустить поток:

```
  public static void main(String[] args)
   {
      start(); // <-- запуск потока и вызов метода run()
      /*** во время работы потока можно
           выполнять другие действия ***/
```

Если нужно создать несколько одновременно выполняющихся потоков, вызовите метод start() требуемое число раз. Но, как уже говорилось, все потоки получают доступ к одним и тем же данным. Чтобы заставить поток выполняться в своем адресном пространстве, создайте для него отдельный объект:

```
Thread t = new TestThread();
t.start();
```

Корректная работа с потоками

Java-потоки могут легко захватить ресурсы центрального процессора (это особенно справедливо в случае Windows, чем Linux/UNIX). Не забудьте прочитать главу 7, "Распределение нагрузки: многозадачность", чтобы узнать, как предоставить другим заданиям возможность выполняться.

Потоки можно создавать и запускать в любой части программы, в том числе в обработчиках событий и исключений. Но помните о необходимости хранить ссылку на созданный объект, иначе уборщик мусора, запущенный, когда поток находится в неактивном состоянии, может его удалить.

Добавление потоков к классу

Не всегда можно объявить класс производным от класса Thread. Обычно это имеет место, когда класс уже является потомком класса Frame. В Java запрещено множественное наследование, поэтому сделать класс производным от двух классов невозможно. В таком случае необходимо объявить, что класс реализует интерфейс Runnable. Результат будет тем же, а программа претерпит лишь незначительные изменения.

```
public class TestThread extends Frame
                        implements Runnable
{
   public void run()
   {
      /*** код потока ***/
   }
   public static void someMethodf)
   {
      Thread t = new Thread(this);
      t.start();
```

Полужирным шрифтом выделены два изменения по сравнению с исходной версией примера. Во-первых, объявляется, что класс порождается от класса Frame и реализует интерфейс Runnable. Это необходимо, чтобы программа работала правильно. Во-вторых, в программе создается отдельный потоковый объект. У класса Thread имеется конструктор, который принимает указатель на объект, реализующий интерфейс Runnable, и делает этот объект потоковым. Вызов метода start() будет иметь тот же эффект, что и прежде.

Синхронизация методов

Основное преимущество потоков заключается в возможности совместного доступа к ресурсам. Это вызывает проблемы взаимоблокировок, которые были описаны в главе 7, "Распределение нагрузки: многозадачность". В Java эта проблема решается с помощью синхронизированных методов.

Синхронизированный метод является аналогом семафора в библиотеке Pthreads. Он объявляется следующим образом:

```
public synchronized changeSomething()
{
   /*** совместный доступ к ресурсу ***/
}
```

Ключевое слово synchronized объявляет метод критической секцией. Таким образом, если метод вызван в одном потоке, другой поток, обращающийся к методу, вынужден будет ждать.

Иногда необходим более гибкий контроль над ресурсами. Предположим, имеются три потока (родительский и два дочерних), которые управляют входным и выходным буферами сокета. Данные не должны быть отправлены, пока не заполнится выходной буфер, поэтому дочерние потоки периодически проверяют его. Синхронизированный метод получает доступ к буферу, но тот еще не заполнен. Метод может вернуть значение, свидетельствующее о неудаче, в результате чего дочерний поток повторно вызовет метод. С другой стороны, метод может захватить ресурс, но заставить поток временно перейти в неактивный режим.

Вернемся немного назад. Когда потоки соперничают за право доступа к ограниченному ресурсу, они помещаются в очередь ожидания. После того как текущий владелец ресурса заканчивает выполнять синхронизированный метод, следующий поток захватывает контроль над ресурсом. Это продолжается до тех пор, пока потоки не завершатся.

Можно построить работу так, что поток, вошедший в критическую секцию, обнаруживает нехватку ресурса (буфер не заполнен) и автоматически переходит в неактивное состояние, перемещаясь в конец очереди ожидания. Вот как это делается:

```
public synchronized changeSomething()
{
   while ( buffer_not_full) // достаточно ли данных в буфере?
   {
      try { wait() } // нет, перемещаемся в конец очереди
      catch (Exception err)
      { System.err.println(err); }
   }

   /* Отправляем сообщение */

   notifyAll();   // уведомляем ожидающие потоки
}
```

Метод wait() помещает текущий поток назад в очередь ожидания. При этом планировщик "будит" следующий поток. Если он обнаруживает, что данные готовы, он выходит из цикла и отправляет сообщение. Метод notifyAll() сообщает всем ожидающим потокам, что ресурс свободен,

Существующие ограничения

В Java имеются очень удобные средства сетевого программирования, позволяющие быстро создавать полнофункциональные программы. В этом языке можно легко создавать сокеты, отправлять сообщения и обрабатывать исключительные ситуации. Тем не менее в нем есть ряд ограничений.

- *Запутанные средства ввода-вывода.* Во всем многообразии классов ввода-вывода не так-то просто разобраться. Их слишком много и они образуют свою собственную иерархию, из-за чего не всегда очевидно, какой именно класс следует применять в данном конкретном случае.
- *Поддержка только стандарта IPv4.* В настоящий момент Java поддерживает только сети TCP/IPv4. На момент написания книги на Web-узле java.sun.com была информация о проекте Merlin, в рамках которого в язык планировалось внедрить поддержку протокола IPX и стандарта IPv6. Но очевидно, что некоторые из существующих методов придется переписать.
- *Отсутствие низкоуровневых сокетов.* В Java не поддерживаются неструктурированные IP-сокеты.
- *Неполный набор параметров сокетов.* Не все параметры сокетов поддерживаются.
- *Отсутствие эквивалента системного вызова fork ().* Из программы можно запустить только внешний модуль, но нельзя создать новый процесс. Можно эмулировать процесс, создав новый потоковый объект, но при этом нет гарантии целостности ресурсов, так как у потоков общее адресное пространство.
- *Отсутствие широковещания.* В Java не поддерживается широковещание. Возможно, это связано с тем, что данный режим применяется все реже.

Даже несмотря на перечисленные ограничения, Java остается удобной программной средой, в которой можно создавать мощные сетевые приложения.

Резюме

В Java имеется большая библиотека сетевых классов. В пакет Network входят классы, предназначенные для создания потоковых (TCP), дейтаграммных (UDP) и групповых (UDP) сокетов. Есть классы, управляющие адресами, а также их преобразованием.

Возможности сокетов расширяются благодаря мощному набору классов ввода-вывода. Хотя этот набор достаточно сложен, но он позволяет существенно упростить сетевое программирование.

Благодаря средствам работы с потоками в Java становится проще проектировать серверы. Потоковый класс создается двумя способами: либо путем наследования класса Thread, либо путем реализации интерфейса Runnable. В обоих случаях метод start() запускает новый поток, вызывая написанный пользователем метод run().

В этой главе на примере Java было показано, как работать с сокетами в объектно-ориентированной среде. В следующей главе будет рассмотрена реализация сокетов в Ct+.

Программирование сокетов в C++

Глава

13

В этой главе...

В предыдущей главе рассматривалось программирование сокетов в Java. Java — очень мощный язык программирования, обладающий множеством преимуществ, среди которых следует отметить независимость от платформы и оперативную компиляцию. Но не всем нравится Java, в основное из-за отсутствия стабильности и недостаточной производительности, поэтому многие программисты предпочитают использовать C++.

Создание сетевой оболочки (или библиотеки классов) требует знания практически всех технологий, рассматриваемых в книге. Фактически в этой главе будут затронуты некоторые технологии, подробно описываемые лишь в части IV, "Сложные сетевые методики".

Чтобы понять материал, изложенный в данной главе, необходимо быть знакомым с C++ и объектно-ориентированным программированием. От читателя предполагается умение написать программу на C++ и откомпилировать ее.

Зачем программировать сокеты в C++?

Работать с библиотекой классов, инкапсулирующей функции работы с сокетами, удобнее, чем вызывать эти функции вручную. Наличие такой библиотеки позволяет унифицировать процесс создания, конфигурирования и использования сокетов.

Упрощение работы с сокетами

Написав однажды класс, реализующий базовые функции работы с сокетами, можно в дальнейшем сосредоточиться на решении алгоритмических задач, а не на сетевом программировании. Взаимодействовать с сокетами несложно, но требуется соблюдать четкую последовательность действий, чтобы сетевые каналы работали согласованно.

C++ значительно превосходит по своим возможностям язык C и является его надмножеством. Для компиляции программ, написанных на C++, можно даже применять C-компилятор cc. (Это не совсем точно. Когда компилятор cc обнаруживает файл с расширением C или cpp, он вызывает утилиту д++, а не дсс.) Наряду с дополнительными возможностями появляются и дополнительные сложности, но преимущества перевешивают недостатки.

Создав библиотеку сокетов, можно скрыть все сложности за набором простых интерфейсов. Чем проще интерфейс, тем легче работать с ним, а также тестировать и отлаживать программу. Кроме того, выше вероятность, что такой интерфейс будет применяться правильно и большинством программистов.

Отсутствие ненужной детализации

От интерфейсов зависит сложность и простота программы. Чем сложнее интерфейс, тем труднее работать с программой. Чтобы добиться простоты, очень важно скрывать ненужные детали реализации. Это особенно важно при создании сетевой оболочки, которую впоследствии будут использовать рядовые программисты.

Конечно, программист захочет узнать, как автор библиотеки классов реализовал некоторые элементы, но степень информации, которую можно ему предоставить, не должна быть слишком высокой. Инкапсуляция — один из инструментов объектно-ориентированного программирования — является средством защиты как автора библиотеки, так и конечного программиста. Скрыв некоторые детали реализации, можно быть уверенным в том, что программист не изменит их впоследствии. Это также позволит в случае необходимости адаптироваться к системным изменениям, не затронув внешние интерфейсы.

Создание многократно используемых компонентов

Внешние интерфейсы создают своего рода мембрану между библиотекой сокетов и программой. Через эту мембрану должны проходить все запросы, посылаемые программой библиотеке. Как обрабатываются эти запросы, программисту не должно быть известно.

Важно, чтобы интерфейсы были как можно проще. Результат очевиден: полученный набор классов будет пригоден для использования в любой программе. Для одного и того же интерфейса можно создать несколько реализаций и выбирать нужную в зависимости от обстоятельств.

Моделирование по необходимости

На рынке существует достаточно средств, позволяющих создавать библиотеки классов. Труднее найти готовые библиотеки сокетов. В этой главе излагается концепция самостоятельного написания библиотеки, которую при необходимости можно модифицировать и доработать.

Согласно распространенной модели объектно-ориентированного анализа и проектирования, процесс разработки программной системы должен проходить по принципу "сверху вниз" (нисходящее программирование). Процесс начинается с анализа исходных требований и путем последовательной детализации приводит к получению конечного продукта. Идея заключается в минимизации числа свойств, присущих системе, — должны быть реализованы только те свойства, которые действительно необходимы.

Создание сетевой оболочки должно проходить иначе. Далеко не всегда можно сказать наперед, какие возможности понадобятся конечному программисту, поэтому нужно реализовать всего по максимуму. Конечно, обилие кода может привести к тому, что часть его в большинстве случаев останется неиспользованной. Но сегодня это уже не проблема. Современные компоновщики отбрасывают код, ненужный в программе.

Создание библиотеки сокетов

Процесс создания объектной оболочки напоминает построение стены: сначала нужно определить ее общие параметры, а затем сложить кирпичик за кирпичиком в нужной последовательности.

Определение общих характеристик

К библиотеке должны предъявляться некие общие требования. Они задают направления, в которых следует вести работу. Часто полученный список требований недостаточно детализирован, поэтому приготовьтесь задавать вопросы. Библиотека должна поддерживать все основные возможности сокетов, описанные в данной книге.

Поддержка различных типов сокетов

Главное требование, предъявляемое к библиотеке сокетов, — поддержка основных протоколов. В первую очередь, это TCP. В TCP-соединениях всегда есть клиент и сервер. Сервер ожидает поступления запроса от клиента, затем создает соединение и начинает сеанс взаимодействия.

Кроме того, друг с другом могут соединяться одноранговые компьютеры. В этом случае есть не клиент и сервер, а запрашивающая и отвечающая стороны. Такое взаимодействие осуществляется по протоколу UDP.

Особыми типами сетевых взаимодействий являются широковещание и групповое вещание. В обоих режимах сообщение одновременно рассылается нескольким адресатам. Данная тема рассматривается в главе 17, "Широковещательная, групповая и магистральная передача сообщений".

Существуют два других типа сокетов, которые являются слишком низкоуровневыми для нашей библиотеки: это неструктурированные сокеты и сокеты, работающие в беспорядочном режиме. Их можно интегрировать в библиотеку, но для управления ими требуется столь высокий уровень детализации, что лучше предоставить программисту возможность напрямую вызывать API-функции.

Отправка простейших сообщений

Второе требование не менее важно: библиотека должна позволять принимать и отправлять сообщения. У сообщения есть адрес, канал, по которому оно передается, и непосредственно тело.

В предыдущих главах описывались различные методы приема и передачи сообщений, в которых требовалось, чтобы сокет находился в том или ином состоянии. Например, при вызове функции `send()` необходимо, чтобы сокет был подключенным. Наша цель заключается в упрощении данного процесса.

Можно сделать так, чтобы сообщение доставлялось автономно. Например, в главе 11, "Экономия времени за счет объектов", упоминалась потоковая передача данных, подразумевающая их автоматическую упаковку и распаковку. В Java это сделать легко: достаточно объявить, что класс реализует интерфейс serializable. Описанный механизм работает потому, что Java-программа представляет собой интерпретируемый байт-код, не содержащий системно-зависимых данных.

В C++ процедуры упаковки и распаковки необходимо реализовать самостоятельно. Кроме того, в C++ нет понятия интерфейса, поэтому нужно создать общий класс, содержащий соответствующие методы, и порождать от него все остальные классы.

Обработка исключений

Третье требование заключается в необходимости контроля над различными исключениями и ошибками, возникающими при работе в сети. В сетевом программировании ошибки могут произойти в любое время: при создании и конфигурировании сокета, приеме и передаче сообщений и т.д.

Источниками ошибок являются сокеты, соединения, маршрутизаторы и другие компьютеры. Требуется иметь возможность быстро локализовать проблему и восстановить нормальную работу программы.

В C++ обработка исключений осуществляется с помощью конструкции try/catch, аналогичной той, что применяется в Java. Перехватывать нужно все исключения, иначе программа может завершиться аварийно. Непредвиденные исключения можно перехватывать на самом верхнем уровне, т.е. в функции `main()`, и возвращать какое-нибудь осмысленное сообщение, чтобы упростить отладку программы.

Конфигурирование соединений

Последнее из основных требований состоит в возможности конфигурирования сокета или соединения. У всех сокетов есть параметры, которые можно настраивать. Эти параметры влияют на поведение как сокета в целом, так и отдельно каналов ввода-вывода.

Группировка основных компонентов

Следующий шаг заключается в определении всех базовых компонентов библиотеки. *Компонент* — это группа взаимосвязанных объектов, совместно решающих конкретную задачу или предоставляющих заданный сервис. У компонентов также есть атрибуты и интерфейсы, поэтому их можно рассматривать как объекты.

В нашей библиотеке сокетов будет четыре основных компонента. С каждым из них связана своя иерархия классов.

Исключения: обработка ошибок

Во всех сетевых приложениях должны обрабатываться исключения. Исключительные ситуации могут возникать в случае ошибок сети и операций ввода-вывода, при выходе за границы массива и т.д. Иерархия классов исключений берет свое начало от класса Exception, реализующего базовые функции (рис. 13.1).

Иерархии классов исключений бывают очень сложными, поэтому нужно внимательно следить, чтобы не возникали рекурсивные исключения. Например, программа может завершиться крахом из-за одного неосторожного вызова оператора new в обработчике исключений, связанных с выделением памяти. Нужно стремиться сделать код обработчиков как можно более простым и по возможности свести его к присваиванию значений и выдаче сообщений об ошибках.

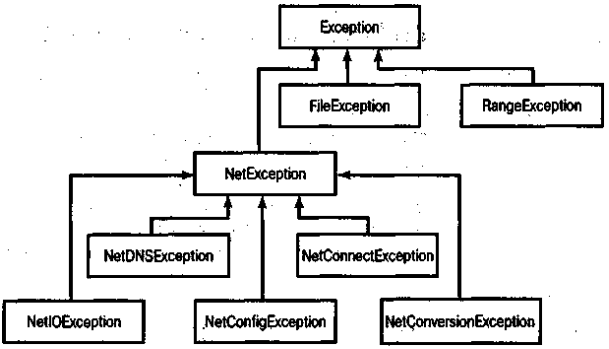

Рис. 13.1. Все классы исключений порождаются от класса Exception, который сам по себе обрабатывает только ошибки работы со строками

Сообщения: упаковка и доставка данных

Для отправки и приема сообщений через подсистему ввода-вывода необходимы блоки данных и буферы. Виртуальный класс Message содержит базовые методы для упаковки (Wrap()) и распаковки (Unwrap()) своих собственных объектов. Это напоминает механизм сериализации в Java.

Другой класс, TextMessage, является примером того, как следует использовать эти методы. Например, метод Wrap() выделяет непрерывный блок памяти и копирует в него текст (ответственность за освобождение блока ложится на того, кто вызывает этот метод). Сообщение в объекте TextMessage является строкой переменной длины.

Метод Unwrap() выполняет противоположное Действие. Он принимает блок данных и восстанавливает его внутреннюю структуру. Иногда сообщение оказывается неполным, например, размер сообщения в объекте TextMessage может превышать 64 Кбайт (максимальный размер буфера), поэтому метод Unwrap() должен в случае необходимости запрашивать недостающие данные. Метод возвращает значение True, когда процесс восстановления завершен.

Адресация: идентификация источника и приемника

Каждое сообщение имеет адрес отправителя и получателя. Знать эти адреса необходимо для того, чтобы обеспечить правильную доставку сообщения. Этой цели служит класс HostAddress. Используя структуру struct sockaddr, он управляет IP-адресами сообщений.

Все адреса, с которыми ведется работа в библиотеке сокетов, должны быть объектами класса HostAddress. Может поддерживаться несколько типов адресов.

Сокеты: создание, конфигурирование и подключение

Последний компонент инкапсулирует все основные функции. В его основе лежит класс Socket, который управляет всеми соединениями и пользуется услугами других компонентов. Конкретные типы соединений реализуются в дочерних классах: SocketServer, SocketClient, Datagram, Broadcast и MessageGroup (групповая доставка). В них инкапсулированы все детали конфигурирования соответствующих протоколов.

В библиотеке предполагается, что программист, использующий эти классы, следует определенным правилам. Например, при вызове метода Send() классы не проверяют, является ли сокет подключенным. Вместо этого просто генерируется исключение.

Построение иерархии классов

Для того чтобы можно было работать с библиотекой сокетов, необходимо создать неабстрактные классы в каждом из перечисленных выше компонентов и сформировать между ними иерархические отношения.

Отношения

На рис. 13.2 изображены базовые компоненты библиотеки и отношения между ними. Как можно было предположить, класс Socket связан со всеми остальными компонентами отношениями "использует" или "генерирует".

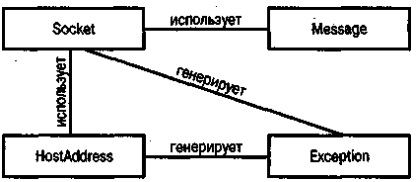

Рис. 13.2. Каждый компонент пользуется услугами какого-нибудь другого компонента

Выявление случаев наследования

Теперь необходимо проанализировать имеющиеся компоненты и выявить внутри них отношения особого вида, называемого *наследованием.* Как описывалось в главе 11, "Экономия времени за счет объектов", концепция наследования подразумевает создание базового класса, от которого порождаются дочерние классы, наследующие его методы и атрибуты. Это позволяет повторно использовать уже написанный код.

В настоящий момент нас интересует класс Socket. На рис. 13.3 изображена схема отношений между его дочерними классами. Классы Broadcast и MessageGroup работают с UDP-сокетами и имеют схожие функции, поэтому являются потомками одного класса — Datagram. Классы SocketClient и SocketServer работают с TCP-сокетами, поэтому в них инкапсулированы другие функции.

Рис. 13.3. Классы Datagram, MessageGroup и Broadcast связаны друг с другом, тогда как классы Socketclient и SocketServer обособлены

Абстрактные элементы

Класс Socket, не представленный на рис. 13.3, должен быть суперклассом, т.е. предком всех остальных классов в своей иерархии. Его назначение заключается в обеспечении дочерних классов стандартным набором атрибутов и методов. В него входят методы Get()/Set() для всех атрибутов, а также методы Send(), Receive() и Close(). В то же время объекты этого класса не должны создаваться напрямую, так как с ним не связан конкретный протокол.

Добавление поддержки неструктурированных сокетов

Можно изменить существующую иерархию таким образом, чтобы класс Socket отвечал за создание неструктурированных сокетов. Однако это может привести к возникновению некоторых трудностей. Дело в том, что функции неструктурированных сокетов ближе к UDP, чем к TCP, поэтому в классах Socketclient и SocketServer их придется *отключать.* Это не самый удачный подход. Можно пойти другим путем — создать дополнительный класс, расположенный в иерархии между классами socket и Datagram. В любом случае структура компонента усложнится. Подобного рода проблемы часто возникают при написании библиотек классов: чем больше возможностей реализуешь, тем сложнее становится иерархия и тем труднее с ней работать.

Классы SocketServer и Socketclient обладают целым рядом общих характеристик, которые не могут быть инкапсулированы в классе Socket. Чтобы разрешить эту проблему, можно создать промежуточный класс SocketStream, который реализует основные свойства TCP-сокета. На рис. 13.4 изображена полная иерархия класса Socket.

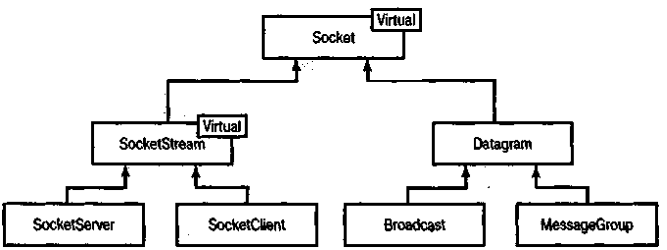

Рис. 13.4. В полную иерархию классов входят два виртуальных класса: Socket и SocketStream

Определение задач каждого класса

Мы достигли этапа, на котором необходимо ответить на вопрос: "Для каких целей можно использовать нашу библиотеку?" В отличие от традиционной методики нисходящего программирования, которая ориентирована на создание проектов, а не библиотек, нельзя заранее предсказать, что конкретно может понадобиться пользователю.

Проекты и библиотеки

Многие программисты пишут библиотеки под конкретные проекты. Их логика понятна: они стараются реализовать только те функции, которые действительно необходимы. Недостаток такого подхода заключается в том, что создаваемым библиотекам зачастую недостает масштабируемости. Они ограничены рамками проекта и могут предложить мало полезного для других проектов. При написании библиотек нужно сначала определить назначение и обязанности самой библиотеки, а затем переходить к описанию конкретных классов.

Атрибуты: что необходимо знать

В большинстве случаев начинают с того, что описывают методы класса. Однако когда речь идет о библиотеке сокетов, нужно в первую очередь определить атрибуты классов.

Все атрибуты в основном связаны с параметрами сокетов. О них рассказывалось в главе 9, "Повышение производительности", а полный их список приведен в приложении А, "Информационные таблицы". Как правило, параметры сокетов не являются переменными-членами класса. Они обрабатываются с помощью методов семейств Get() и Set(), в которых вызываются функции getsockopt() и setsockopt().

На рис. 13.5 изображена иерархия класса Socket с указанием всех методов каждого класса. Сначала перечислены методы, реализующие те или иные функции сокетов, а затем указаны методы, связанные с атрибутами. В ряде случаев происходит лишь установка или сброс атрибута, поэтому с ним не связан метод типа Get(). Не все атрибуты класса являются открытыми. Обычно предоставляется доступ к тем атрибутам, которые меняют поведение класса, а внутренние переменные остаются закрытыми.

Один из атрибутов — параметр IP_TOS — обрабатывается особым образом. С ним связаны сразу четыре метода класса Datagram: MinimizeDelay(), MaximizeThroughput(), MaximizeReliability() и MinimizeCost(). Все они вызывают одну и ту же функцию setsockopt(), но с разными флагами: IPTOS_LOWDELAY (минимальная задержка), IPTOS_THROUGHPUT (максимальная пропускная способность), IPTOS_RELIABILITY (максимальная надежность) и IPTOS_LOWCOST (минимальная стоимость) соответственно. В результате программисту не приходится делать это самостоятельно.

В классах могут присутствовать другие атрибуты, имеющие специальное назначение. Так, в классе SocketServer хранится имя зарегистрированной функции обратного вызова, которая связана с приемом запросов на подключение. В классе MessageGroup хранится список групп, к которым присоединялся сокет.

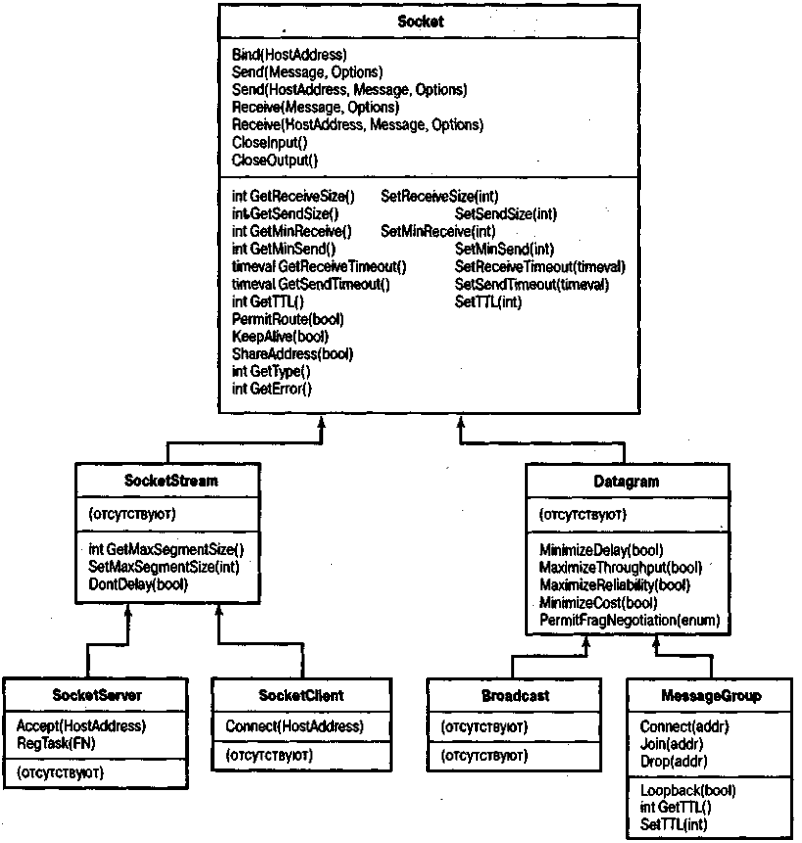

Рис. 13.5. Иерархия класса Socket с указанием методов всех классов

Методы: что необходимо делать

У большинства классов в нашей библиотеке есть не только методы, связанные с атрибутами, но и специализированные методы, реализующие особенности того или иного протокола. В основном все они вызывают свои аналоги из библиотеки Socket API (табл. 13.1).

Таблица 13.1. Специальные методы, реализованные в классах компонента Socket

Класс	Метод	Описание
Socket	Bind()	Вызывает функцию bind()
	Send()	Вызывает функцию send() либо sendto()
	Receive()	Вызывает функцию recv() либо recvfrom()
	CloseInput()	Закрывает входной канал с помощью функции shutdown()
	CloseOutput()	Закрывает выходной канал с помощью функции shutdown()
SocketServer	Accept()	Вызывает функцию accept()
SocketClient	Connect()	Вызывает функцию connect()
MessageGroup	Connect()	Вызывает функцию connect()
	Join()	Подключает сокет к группе многоадресной доставки сообщений
	Drop()	Удаляет сокет из группы многоадресной доставки сообщений

У класса Broadcast нет специальных методов и атрибутов. Все свои методы он наследует от родительских классов, но используется особым образом.

Конструкторы: как создавать объект

У каждого класса есть набор конфигурационных параметров, которые определяют начальное поведение его объектов. Например, у класса Broadcast нет дополнительных методов и атрибутов, но он должен быть сконфигурирован на прием и передачу широковещательных сообщений. Подобные действия выполняются в специальном методе, называемом *конструктором*.

Конструктор вызывается при создании объектов класса. В нем можно выполнять те же действия, что и в обычных методах: создавать объекты, выделять память, выполнять разного рода вычисления и т.д. Например, ниже приведено описание конструктора класса Socket:

```
//*******************************************************
//*** Конструктор класса Socket, вызываемый по умолчанию.
//*** Принимает параметры Network (PF_INET) и Protocol
//*** (SOCK_STREAM), создавая сокет и помещая его
//*** дескриптор в атрибут SD.
Socket::Socket(ENetwork Network, EProtocol Protocol)
{
    .
   SD=socket(Network, Protocol, 0);
   if ( SD < 0 )
     throw NetException("Could not create socket");
}
```

Если в процессе создания сокета произошла какая-то ошибка, нет возможности вернуть в программу сообщение об ошибке. Вместо этого генерируется исключение и вызывается его обработчик. Если же все прошло успешно, конструктор завершается и управление передается вызвавшей его программе.

Не все операции доступны в конструкторе. В частности, в нем нельзя вызывать методы создаваемого им объекта, если только они не бьши объявлены статическими (спецификатор static). Пока конструктор не завершился, компилятор считает, что контекст класса еще не полностью сформирован, поэтому запрещает вызывать обычные его методы.

Статические методы класса не зависят от контекста, и их можно вызывать всегда. Для этого не требуется даже, чтобы существовал объект класса: при вызове статического метода вместо имени объекта можно указывать имя класса, после которого следует оператор ::. По сути, это служебные методы класса. Они редко используются, но в определенных ситуациях оказываются полезными. Некоторые методы, например обработчики сигналов, не могут выполняться в контексте объектов, поэтому они должны быть статическими.

Деструкторы: что необходимо очищать

Подобно тому как конструкторы инициализируют объект и подготавливают его к использованию, *деструкторы* выполняют обратные действия, удаляя объект из памяти. В отличие от Java, где имеется встроенный механизм уборки мусора, в C++ требуется удалять объекты вручную.

Часто объекты не требуют специальных действий по очистке. Но если в них выделялась память, то компилятор не может сообщить о наличии "потерянных" указателей. С помощью конструктора можно задать, что именно следует очищать и когда.

Одно из наиболее часто выполняемых действий в деструкторе — закрытие файлов. Ниже приведено описание деструктора класса Socket:

```
//**************************************
//*** деструктор класса Socket
Socket::~Socket(void)
{
   if ( close(SD) != 0 )
     throw FileException("Can't close socket");
}
```

Все деструкторы следует объявлять виртуальными (спецификатор virtual), чтобы при удалении объекта компилятор мог вызвать все родительские деструкторы в иерархии. Когда деструктор не является виртуальным, это свидетельствует о том, что деструкторы родительских классов не должны вызываться. Если деструктор суперкласса (начального в иерархии) объявлен виртуальным, то деструкторы всех дочерних классов также будут виртуальными. Но для ясности лучше все же явно указывать спецификатор virtual.

Тестирование библиотеки сокетов

Ниже приведено несколько примеров использования создаваемой нами библиотеки сокетов. Все они, а также файлы реализации библиотеки сокетов, имеются на Web-узле.

Эхо-клиент и эхо-сервер

Первый пример (наиболее часто упоминаемый в книге) — это связка эхо-клиента и сервера. С их помощью можно легко проверить, правильно ли устанавливается соединение. Остальное уже будет проще проверять.

Листинг 13.1. Эхо-клиент (echo-client.cpp)

```
// Основное тело клиента
HostAddress addr(strings[l]); // формат <адрес:порт>
try
{
   SocketClient client(addr); // создаем сокет и подключаемся
   TextMessage msg(1024);  //резервируем буфер для сообщения
   do // повторяем цикл до тех пор, пока не будет получено "bye"
   {
      char line[100];
      client.Receive(msg);              // принимаем сообщение
      printf("msg=%s", msg.GetBuffer()); // отображаем его
      fgets(line, sizeof(line), stdin);  // запрашиваем строку у
                                            пользователя
      msg = line;       // помещаем ее в буфер
      client.Send(msg); // и отправляем на сервер
   }
   while ( strcmp(msg.GetBuffer(), "bye\n") != 0 );
}
catch (Exceptions err)
{
   err.PrintException();
}
```

Во всех программах используется одинаковый формат обработчиков try/catch. Необходимо также предусмотреть обработку ошибок, возникающих в самой системе. Следующий фрагмент необходимо разместить на самом верхнем уровне программы:

```
//*****************************************
// Перехват всех непредусмотренных исключений
catch(...)
{
   fprintf(stderr,  "Unknown exception!\n");
```

Он позволяет перехватывать все исключения, которые не были предусмотрены в программе.

Эхо-сервер можно построить на основании встроенного эхо-сервиса, сделать его почтовым или HTTP-сервером. Во всех этих случаях для передачи сообщений подойдет класс TextMessage. В листинге 13.2 показано, как организуется цикл приема запросов на подключение.

Листинг 13.2. Эхо-сервер (echo-server.cpp)

```
// Основное тело сервера
try
{
   SocketServer server(port); // создаем и конфигурируем сокет
   do                         // цикл выполняется бесконечно
     server.Accept(Echoer);   // принимаем запрос от клиента
   while (1);
}
```

Как видите, тело сервера очень простое. Класс SocketServer самостоятельно выполняет все действия по инициализации и конфигурированию сокета. Остальные операции реализованы в функции Echoer() (листинг 13.3).

Листинг 13.3. Обработка сообщений в эхо-сервере (echo-server.cpp, часть 2)

```
// Процедура обслуживания клиентских запросов
try
{
   TextMessage msg(1024); // формируем буфер сообщений
   client.Send(welcome);  // посылаем приветственное сообщение
   do
    {
     client.Receive(msg);  // принимаем сообщение
     client.Send(msg);     // и возвращаем его обратно
    }
   while ( msg.GetSize() > 0 &&
           strcmp(msg.GetBuffer(), "bye\n")  != 0 )
}
```

Опять-таки, алгоритм достаточно прост: сервер принимает сообщение и возвращает его обратно, пока клиент не пришлет строку "bye". Обратите внимание на то, что в этом фрагменте есть свой блок try/catch. Это очень хорошая идея, так как возникающие здесь ошибки связаны непосредственно с соединением и не должны влиять на работу основного тела сервера. Если не указать этот блок, то в случае отключения клиента сервер прекратит работу, так как в блоке try/catch функции main() будет перехвачено исключение.

Размещение обработчиков исключений

Тщательно выбирайте места размещения обработчиков. Не обязательно учитывать все возможные типы исключений. Как правило, отдельный блок try/catch требуется там, где осуществляется прием и передача сообщений, а все остальные исключения можно обрабатывать на самом верхнем уровне программы.

Многозадачная одноранговая передача сообщений

В настоящий момент в библиотеке не реализован многозадачный режим, но это несложно сделать. Ниже приведен классический пример UDP-модуля, который в процессе отправки данных может принимать любое число сообщений (листинг 13.4).

Листинг 13.4. Одноранговая передача сообщений (peer.cpp)

```
// Модуль отправки дейтаграмм
try
{
   HostAddress addr(strings[1]); // определяем собственный адрес
   Socket *channel = new Datagram(addr); // создаем сокет
   if ( !fork() ) //"порождаем новый процесс
     receiver(channel); // вызываем принимающую сторону
   channel->CloseInput();  // закрываем входной канал
   HostAddress peer(strings[2]); // определяем адрес получателя
   TextMessage msg(1024); // резервируем буфер для сообщения
   do
   {
      char line[100];
      fgets(line, sizeof(line), stdin); // запрашиваем строку у
                                           пользователя
      msg = line; // помещаем ее в буфер
      channel->Send(peer, msg); //и отправляем получателю
   }
   while ( !done );
   delete channel;
}
```

Реализуя многозадачный режим, необходимо помнить о следующем.

- *Придерживайтесь правил* — учтите все замечания, которые приводились в главе 7, "Распределение нагрузки: многозадачность". В частности, не забывайте получить от потомка сигнал завершения. Кроме того, нужно помнить, что потоки совместно пользуются каналами ввода-вывода. Если канал закрывается на одном конце, он будет автоматически закрыт во всех потоках.
- *Совместное использование памяти* — это почти не решаемая проблема. Старайтесь как можно меньше использовать общие ресурсы памяти.
- *Ограничения C++* - реализовывать многозадачный режим в C++ опасно. Очень многое делается "за кулисами", поэтому нужно внимательно следить за тем, какой поток какими данными владеет. Если объект создается в стеке, а потом запускается новый поток, последствия могут быть непредсказуемыми.

Существующие ограничения

Описываемая библиотека классов является примером (хоть и незавершенным) того, как можно программировать сокеты в C++. Она должна подсказать читателям, как следует писать свои собственные приложения.

В данный момент средствами библиотеки можно создавать клиентские, серверные, дейтаграммные, а также групповые и широковещательные сокеты. Все их можно конфигурировать, настраивая их параметры. Существует абстрактный класс Message, на основании которого можно создавать пользовательские форматы сообщений. Ниже описано, как можно доработать библиотеку.

Передача сообщений неизвестного/неопределенного типа

Библиотека позволяет порождать новые классы сообщений от класса Message. Это очень удобно, если тип сообщения известен заранее и соответствует общему формату класса Message. Но структура данных не всегда известна, например, если новый клиент взаимодействует с внешним сервером. Решить проблему можно, реализовав абстрактный протокол передачи сообщений. Объект, несущий в себе сообщение, будет содержать не только данные, но и их описание. Это поможет сделать взаимодействие между компьютерами более гибким.

Поддержка многозадачности

Как уже упоминалось, в библиотеке не реализован многозадачный режим. Большинство сетевых приложений выполняет несколько действий одновременно с целью повышения производительности. Если создавать процессы и потоки внутри классов библиотеки, это существенно упростит задачу программиста, пользующегося библиотекой.

В первую очередь при определении класса сокета необходимо решить, что будет создаваться внутри сокета: процессы или потоки? И является ли новое задание отдельным объектом? В этом случае придется порождать новый класс сразу от двух классов: Socket и Process/Thread.

Резюме: библиотека сокетов упрощает программирование

Применяя объектно-ориентированный язык, такой как C++, можно существенно упростить программирование сокетов. Библиотека сокетов координирует взаимодействие компонентов, которые представляют собой наборы сетевых классов. Каждый класс решает свою часть общей задачи.

Описанная в настоящей главе библиотека содержит четыре основных компонента, являющихся абстракциями исключений, сообщений, адресов и сокетов. Компонент Exception отделен от остальных, чтобы уменьшить риск возникновения внутренних ошибок. Компоненты HostAddress и Message являются служебными и применяются в компненте Socket.

Глава

14

Ограничения объектно-ориентированного программирования

В этой главе...

Объектная технология не решает всех проблем. У нее есть своя область применения. Не следует пытаться использовать ее всегда и везде, так как во многих случаях лучше подходят другие технологии программирования.

Если выбор все же сделан в пользу объектов, следует быть готовым к трудностям. Придется заставить себя мыслить объектными категориями и придерживаться совершенно новой парадигмы. Необходимо также помнить об ограничениях, присущих объектной технологии.

Правильное использование объектов

Первая проблема, с которой сталкиваются программисты, разработчики и аналитики, начинающие объектный проект, заключается в выявлении природы самих объектов. На каждой стадии анализа необходимо выяснять, что должна делать система, какую роль в этом играет тот или иной объект и когда задачу можно считать решенной.

Начальный анализ

Создавая объектное приложение, важно правильно начать. Прежде всего необходимо разобраться в потребностях пользователей. Пользователь — это человек или система, которые будут работать с приложением. Следует разработать четкие и понятные интерфейсы взаимодействия с пользователями, чтобы гарантировать предсказуемую работу программы.

На этапе анализа нужно разобрать работу приложения в четырех основных направлениях: контекст, список функций, типичные сценарии использования и общий ход выполнения программы. Контекст — это определение рамок, в которых функционирует программа. Необходимо выявить пользователей программы, построить схему входных и выходных информационных потоков и очертить круг обязанностей пользователей. Это очень важно для создания успешного проекта. Если не указать явно, чего программа *не* делает, пользователь будет подразумевать, что программа реализует соответствующие функции.

Определив контекст, переходим непосредственно к списку функций программы. Каждая функция в данном случае представляет собой пару "глагол — объект", например: "Напечатать отчет". Субъектом действия подразумевается программа.

Список функций логически приводит нас к выявлению типичных сценариев использования программы. Сценарий — это общее описание того, как программа реализует ту или иную функцию. Началом и концом сценария являются внешние данные. Промежуточные этапы определяют последовательность трансформации данных. Из общих сценариев можно выделять конкретные случаи использования. При этом сценарий является функцией, а конкретный случай описывает вариант изменения ее входных данных. Рассмотрим следующий пример.

Сценарий №9: "Отправка сообщения пользователю сети"

[Пользователь вводит сообщение и щелкает на кнопке Send]
Устанавливаем соединение с удаленным узлом
Вызываем абонента
[Абонент подтверждает наличие связи]
Отправляем сообщение
Выдаем подтверждение пользователю
[Пользователь получает код завершения]

Случай №9.1: "Отправка сообщения: в приеме сообщения отказано"

[Пользователь вводит сообщение и щелкает на кнопке Send]
Устанавливаем соединение с удаленным узлом
Вызываем абонента
[Абонент отказывается принять сообщение]
Сообщаем пользователю об отказе
[Пользователь получает код завершения]

Случай №9.2: "Отправка длинного сообщения пользователю сети"

[Пользователь вводит сообщение и щелкает на кнопке Send]
Устанавливаем соединение с удаленным узлом
Вызываем абонента
[Абонент подтверждает наличие связи]
Отправляем сообщение по частям
Выдаем подтверждение пользователю
[Пользователь получает код завершения]

Последний этап — на основании сценариев графически изобразить ход работы программы. Важно, чтобы каждый сценарий был учтен и было показано, как входные данные изменяются на пути к пункту своего назначения.

Именование объектов

После того как потребности пользователей выяснены, необходимо приступить к анализу компонентов программной системы. Поведение каждого компонента определяется тем, что он "знает" (атрибуты) и что он "умеет делать" (методы).

Имя объекта задает его текущее поведение и возможную эволюцию. Хорошее имя всегда является существительным. Например, лучше назвать объект Socket, чем NetIO, поскольку работа в сети подразумевает не только ввод-вывод. Следует избегать употребления глаголов и отглагольных существительных, так как в этом случае осуществляется привязка к тому, что объект делает сейчас, и ограничивается его использование в будущем.

Разграничение этапов анализа и проектирования

На этапе анализа важно не зайти слишком далеко, чтобы не "увязнуть" в ненужных деталях. Например, многие разработчики решают, что для проекта понадобится база данных, задолго до того, как будет проведен анализ задачи. Хорошо это или плохо?

Принятие подобных решений ставит конкретную технологию во главу угла и заставляет корректировать все остальные решения с учетом заранее выбранной технологии. Хороший архитектор не станет определять размер балки, не рассчитав предварительно нагрузку на нее. Точно так же, если системный аналитик заявляет, что программа должна быть написана на Java, не поняв всю проблему в целом, могут возникнуть серьезные трудности при реализации проекта.

На этапе анализа предметную область нужно рассматривать на макроуровне. Если кто-то упоминает конкретную деталь реализации, она должна быть отнесена к этапу проектирования.

Системные ограничения

Иногда в список системных требований входят конкретные аппаратные или программные ограничения. Если их немного и они связаны с финансовым/обеспечением проекта, их можно учесть сразу. Однако в большинстве случаев их можно проигнорировать вплоть до этапа проектирования.

В процессе анализа подразумевается, что имеются все необходимые инструменты реализации. Не беспокойтесь о том, как будет написан тот или иной модуль, а сконцентрируйтесь на рассмотрении основных компонентов системы, их взаимодействии друг с другом и сценариях их использования.

Правильный уровень детализации

Сценарии использования важны для осуществления анализа системы. Но, как и в целом на этапе анализа, важно абстрагироваться от чрезмерных деталей, чтобы гарантировать соответствие структуры программы исходным требованиям. Если, общаясь с заказчиками проекта, погрузиться в детали реализации, рано или поздно заказчики перестанут вас понимать или же запутаются в каком-нибудь сценарии. Если же они захотят подробнее узнать о том, как реализован тот или иной модуль, значит, вы на правильном пути.

Избыточное наследование

На этапе проектирования классы, определенные на этапе анализа, наполняются деталями. Здесь необходимо быть очень внимательным, так как велико искушение связать между собой все классы отношениями наследования. Основной довод здесь таков: "Мы провели тщательный и полноценный анализ. Давайте теперь так же тщательно выполним проектирование". Тем не менее подобная дотошность редко необходима.

В большинстве случаев достаточно построить естественную иерархию классов и добавить к ним пользовательский интерфейс. Порождать один класс от другого нужно только тогда, когда в нем изменяется модель поведения или добавляются новые функции. Если функция не была выявлена на этапе анализа, но появилась в процессе проектирования, проверьте, не является ли это упущением.

Избыточное наследование — это следствие неправильного моделирования обязанностей объектов. Когда все классы связаны между собой отношениями наследования, возникает сложная паутина отношений, и полученный проект трудно сопровождать даже при наличии документации. Пытаясь модифицировать один класс в иерархии, неизбежно затрагиваешь другой, что противоречит сути инкапсуляции.

Неправильное повторное использование

Распространенный миф, прпулярный среди поклонников объектно-ориентированного программирования, гласит, что любой компонент можно использовать многократно. Это очень "абстрактная" вера. Многие классы наилучшим образом используются именно там, где они изначально проектировались.

Если необходимо повторно использовать класс, нужно убедиться, что его назначение и интерфейс хорошо соответствуют новой ситуации. Когда большую часть интерфейса приходится переписывать, это говорит либо о несоответствии класса, либо о том, что он был плохо спроектирован. Вот вопросы, на которые нужно ответить.

- Соответствует ли основное назначение класса поставленной задаче?
- Достаточно ли переписать всего один или два метода?
- Правильную ли роль играют родительские классы?
- Правильно ли обрабатываются существующие данные?

Если на один из этих вопросов дан отрицательный ответ, имеет место неправильное использование класса. Нужно либо найти другой класс, либо отступить от иерархии и создать совершенно новый класс.

Правильное применение спецификатора friend

Наследование — это опасное средство, особенно если наследуется внешний класс или применяется спецификатор friend. Данный спецификатор позволяет одному классу, называемому *дружественным,* получить доступ к закрытым членам другого класса в обход механизма инкапсуляции. Это также дает возможность интегрировать класс с набором внешних функций (в частности, с перегруженными операторами).

Применение дружественных функций и классов позволяет повысить эффективность программы, но только в том случае, если все они неразрывно связаны с основным классом. Большинство специалистов считает наличие дружественных классов дурным тоном и признаком плохого проектирования, как если бы в программе присутствовал оператор goto.

Перегруженные операторы

Даже перегрузка операторов в настоящее время считается нежелательной, так как часто приводит к усложнению библиотеки классов. Если же ее необходимо применять, придерживайтесь следующих правил.

- *Не передавайте данные по значению или через указатель* — вместо этого везде, где возможно, применяйте ссылки (s). В противном случае могут возникнуть потерянные указатели или же произойдет снижение производительности из-за постоянного вызова конструкторов и деструкторов.
- *Всегда делайте перегруженным оператор присваивания* — это необходимо, чтобы правильно осуществлялось преобразование типов.
- *Выбирайте оператор, соответствующий смыслу операции,* — например, запись <строка>+<строка> понятна, а запись *<стек> не обязательно означает выражение <стек>->Pop{).
- *При необходимости делайте перегруженными операторы* new *и* delete — некоторые авторы рекомендуют делать это всегда, хотя это спорный вопрос.
- *Никогда не перегружайте непонятные операторы* — перегруженные операторы вызова функции ("()") и доступа к полям структур ("." и "->") редко будут использоваться правильно.

В целом старайтесь придерживаться разумного подхода к программированию. Если хотите, чтобы созданной программой пользовались другие люди, сделайте ее понятной для них.

Объекты не решают всех проблем

Некоторые люди заявляют, что ООП — единственно верная технология программирования. Конечно, с помощью объектов можно сделать многое, но определенные задачи все же проще решать другими средствами. Нельзя ограничивать себя каким-то одним инструментом.

И снова об избыточном наследовании

Важное место в ООП занимает концепция наследования, которая подразумевает повторное использование существующих классов. К сожалению, некоторые программисты считают это панацеей от всех бед, поэтому применяют наследование везде, где только возможно. Они пытаются объединить все классы в одну иерархию, даже в тех случаях, когда это вовсе не требуется. (Обратите внимание на то, что подобные суждения неприменимы в отношении Java, где все классы порождаются от класса Object.)

Недоступный код

ООП позволяет эффективно решать многие традиционные задачи программирования. Поскольку эти задачи постоянны и неизменны, возможность повторного использования кода очень важна. Как сказал кто-то, не исключено, что большинство программ уже было кем-то написано. Тем не менее не стоит забывать, что не все подвластно объектам. Есть области, в которых проявляются недостатки ООП.

Неполные классы

Первое ограничение возникает при создании неполных классов, которые занимают промежуточное положение в иерархии и содержат часть методов реализованными, а часть — абстрактными. Как правило, создавать экземпляры таких классов нельзя.

Пустые методы

При создании дочернего класса может оказаться, что вызов определенного метода родительского класса должен быть запрещен из соображений безопасности или потому, что этот метод не входит в контекст нового класса. Некоторые программисты находят выход из этой ситуации, создавая пустые методы, не имеющие никакого поведения. Тем не менее метод по-прежнему присутствует в классе и занимает определенные ресурсы.

Мутации объектов

Еще один недостаток заключается в *мутации объектов,* когда один объект может быть преобразован в другой без операции приведения типа. Даже в объектно-ориентированных языках программирования со строгой типизацией, таких как C++ и Java, эта операция потенциально опасна. Мутация — это форма преобразования, при которой объект переходит из одного состояния в другое по определенным правилам. Сначала создается общий объект, который по мере анализа

постепенно раскрывает свои свойства, в результате чего может даже получиться совершенно новый, ранее неизвестный в программе класс. Мутация широко применяется при работе с объектными потоками, где имеется большой двоичный объект (BLOB — binary large object), который нужно "расшифровать". Например, можно создать конструктор трансляции, который в зависимости от внутренней структуры полученного аргумента создает объект того или иного класса.

Мутация выполняется при соблюдении следующих условий.

- *Большой двоичный объект должен оставаться цельным* — модуль преобразования не должен копировать объект или менять его структуру. Должен меняться лишь способ интерпретации объекта.
- *Объект должен мутировать в рамках существующей иерархии наследования* — чтобы один объект можно было преобразовать в другой, они должны иметь общего предка.
- *Модуль преобразования должен уметь создавать экземпляры абстрактного класса* — это может вызывать споры, но в процессе анализа объекта, как правило, приходится начинать с абстрактного класса.
- *В результате всегда должен получаться экземпляр конкретного класса* — причина этого понятна: не должен существовать объект неполного класса.
- *Модуль преобразования должен иметь доступ к закрытым членам класса* — это, казалось бы, противоречит принципу инкапсуляции, но для того чтобы выяснить природу двоичного объекта, необходимо иметь доступ к внутренним частям конечного класса.
- *Модуль преобразования должен поддерживать пустые методы и неполные классы* — иногда методы определяемых классов нужно отключать динамически, чтобы результирующий объект вел себя правильно.

Мутировавшие объекты — это отличное средство работы с классами, о которых на этапе создания программы еще не было известно.

Проблема чрезмерной сложности

Объектная технология имеет и другие ограничения помимо тех, которые определяются природой самих объектов. Ниже рассматривается ряд примеров, когда применение объектов не дает ожидаемого результата.

Игнорирование устоявшихся интерфейсов

За время существования объектно-ориентированного программирования профессионалы усвоили ряд уроков. Основной из них — важность устоявшихся интерфейсов. Интерфейс класса определяет его долговечность. Но у программистов редко хватает времени на разработку наилучшего интерфейса.

Как правило, интерфейсы получаются слишком специализированными или функционально перенасыщенными. Часто встречается такой подход: "Этот метод здесь полезен — почему бы не реализовать его еще здесь и вот здесь?" Интерфейс должен оставаться простым и интуитивно понятным.

Другая проблема возникает при работе с крупными библиотеками, когда для получения доступа к той или иной возможности нужно создать экземпляры нескольких классов, причем в правильном порядке. Это характерно для графических приложений, так как графические интерфейсы обычно очень сложные.

Множественное наследование

Сложность возникает вследствие увлечения наследованием. В C++ можно создавать классы, являющиеся потомками сразу нескольких классов. Управлять такой иерархией достаточно сложно. В главе 11, "Экономия времени за счет объектов", вводилось понятие связности модулей. Когда применяется множественное наследование, возникает чрезмерная связность между классами. В случае изменения родительских классов разработчику часто приходится проверять интерфейс дочернего класса, что неудобно.

С теоретической точки зрения множественное наследование допустимо, но не практично. Как правило, проект, в котором встречается множественное наследование, реализован неоптимальным способом. Обычно имеет место неправильное распределение обязанностей между классами. Не исключено, что можно из двух родительских классов выделить общее ядро и реализовать его в виде суперкласса, а родительские классы объединить в один.

Большинство проблем с множественным наследованием можно решить, выделив общие компоненты родительских классов в абстрактный суперкласс или же воспользовавшись отношениями включения. Обычно новый класс ближе к одному из родительских классов, чем к другому. В этом случае первый класс нужно сделать предком, а второй — встроенным объектом нового класса.

Исключением из этого правила являются потоковые классы. Если класс связан с родительским классом и в то же время представляет собой отдельное задание (процесс или поток), множественного наследования не избежать.

Разрастание кода

Применяя наследование, перегрузку и шаблоны, можно столкнуться с новой проблемой: разрастание исходных текстов библиотеки или класса. Увеличение размеров объектов не очень существенно, если в системе много памяти. Но чем больше строк в программе, тем выше вероятность ошибок.

В зависимости от сложности классов их размер может на 20—50% превышать размер функциональных аналогов, написанных средствами модульного программирования. В основном это связано с усложнением синтаксиса. Однако эта жертва оправдана: несмотря на повышение сложности программы увеличиваются ее возможности и усиливается контроль над типами данных.

Впервые столкнувшись с объектами, программисты были удивлены разрастанием кода. В некоторых случаях размер программы, написанной по объектной технологии на C++, на порядок превышал размер аналогичной модульной C-программы. Это, конечно, крайность, но вполне можно ожидать двух- или пятикратного увеличения программы.

Можно избежать разрастания, придерживаясь следующих правил.

- Старайтесь не перегружать операторы без особой необходимости.
- Не пытайтесь втиснуть все классы в рамки единой иерархии наследования.
- Не применяйте виртуальное наследование.
- Поменьше используйте виртуальные методы.
- Старайтесь избегать множественного наследования.

- Не злоупотребляйте inline-функциями. (Некоторые специалисты утверждают, что следует вообще избегать макроподстановки функций, оставив это на усмотрение компилятора.)

Проблема управления проектами

Объектный проект выдвигает новый и непривычный набор проблем для руководителя, которому приходится согласовывать усилия большого числа людей и проверять, в правильном ли направлении движется работа. Руководитель объектного проекта должен быть универсалом. Умение правильно распределять обязанности, координировать, организовывать и даже быть дипломатом — все это чрезвычайно важно для успешной реализации проекта. Нужно ведь не только уложиться в заданные сроки и не превысить бюджет. Если команда распадается после завершения проекта, или документация оказывается неудовлетворительной, или клиент остается неудовлетворенным, проект можно считать неудачным.

Мы затронем лишь несколько аспектов, касающихся руководства проектами. На самом деле это очень серьезная проблема, которую нельзя осветить на нескольких страницах. Каждый руководитель рано или поздно осознает, что успешность проекта зависит от команды разработчиков, а не от самого продукта.

Нужные люди в нужное время

Причина, по которой конечный продукт не является определяющим фактором успеха, заключается в том, что продукт был затребован людьми, нуждающимися в нем. Его принятие считается само собой разумеющимся, если продукт соответствует требованиям, предъявленным пользователями.

Успех команды разработчиков более важен, а руководитель управляет не только людьми, входящими в группу программистов. Ниже перечислен список членов команды и указана роль каждого из них.

- *Спонсор проекта* — выделяет деньги на его реализацию и определяет основные направления проекта.
- *Представитель заказчика* — взаимодействует с заказчиком и согласовывает с ним промежуточные итоги. Он должен быть знаком с образом мышления заказчика и представлять, как будет применяться разрабатываемая система. Он определяет исходные требования, предъявляемые к продукту, и оказывает консультации по ходу выполнения проекта.
- *Бизнес-аналитики* — собирают и обрабатывают информацию, касающуюся исходных требований. Они должны согласовывать промежуточные итоги с представителем заказчика. Впоследствии оказывают помощь при тестировании продукта.
- *Технические специалисты* — занимаются анализом в процессе проектирования. Они связывают программные модели с объектами реального мира и регулярно согласовывают промежуточные итоги с бизнес-аналитиками.
- *Специалист, отвечающий за контроль качества,* — собирает информацию о проекте и руководит процессом тестирования. После того как бизнес-аналитики утвердят исходные требования к проекту, специалист по контролю начнет разрабатывать тесты, на основании которых будет осуществ-

ляться прием проекта. Когда процедура проектирования будет завершена, этот человек приступит к выполнению граничных и рабочих тестов.

- *Программисты* — осуществляют проектирование системы и программируют ее на выбранном языке.

Это гораздо больше, чем большинство руководителей может себе представить. Если руководитель проекта грамотно управляет процессом, каждый человек исполняет четко отведенную ему роль и вносит свою лепту в повышение качества продукта.

Между двух огней

Несмотря на участие в проекте большого числа людей, руководитель часто сталкивается с вопросом: "Почему программа еще не написана?" Его задает как спонсор проекта, так и представитель заказчика. Первый хочет увидеть результат вложения денег, а второй — побыстрее получить готовый продукт.

Как свидетельствует мировой опыт, лишь 20% времени проекта уходит непосредственно на программирование. Остальные 80% поровну распределяются между этапами анализа/проектирования и тестирования. В американской программной индустрии показатели обычно такие: 33% — анализ и проектирование, 34% — программирование и 33% — тестирование. Итого программирование занимает в лучшем случае 34% времени. Так почему бы не тратить спокойно время на анализ исходных требований и проектирование?

Чтобы успокоить спонсора и заказчика, проинформируйте их о графике работ и постоянно держите в курсе того, как продвигается работа. Показывайте им презентации и документацию.

Если времени все же не хватает, сократите этап проектирования, выполняя проектирование оперативно, в процессе программирования. Конечно, такой подход чреват недостатками, для разрешения которых может потребоваться более тесно взаимодействовать с заказчиком.

Избегайте создания одноразовых прототипов, которые демонстрируются заказчику как свидетельство того, что работа движется. Они редко оказываются полезными. Обычно не хватает времени на то, чтобы впоследствии вернуться к прототипу и "все сделать правильно".

Тестирование системы

Цикл разработки программного обеспечения построен на определении потребностей пользователей и последующей проверке того, удовлетворяются ли эти потребности. Подобного рода проверка называется *тестированием* или *контролем качества*.

В реалиях под проверкой подразумевается не только тестирование. Необходимо собрать продукт в единую систему, проверить все ее компоненты, провести итоговую бизнес-оценку и сравнить продукт с другими продуктами данной серии. Среди всех участников проекта специалисту по контролю качества выпадает самая трудная миссия.

Объектная технология не помогает в данном вопросе, а лишь усложняет все в два-три раза. Раньше достаточно было тестирования по методу прозрачного и черного ящиков (подробно они описаны ниже). С появлением модулей возникла потребность в промежуточном тестировании. А в объектном проекте нужно тес-

тировать наследование, полиморфизм, интерфейсы классов и т.д. Это заставило многих изменить существующие подходы к тестированию.

Основная ошибка, которую делают многие, заключается в том, что разработчику или программисту позволяют самостоятельно тестировать свою работу. Проблема здесь очевидна: те же логические упущения, которые привели к появлению ошибки, могут помешать ее обнаружить. Вероятность такого исхода слишком велика, чтобы не предусмотреть его. Необходимо нанять хорошего инженера-испытателя, который проведет перекрестную проверку системы.

Начиная объектный проект, руководитель и заказчик должны решить, сколько времени займет тестирование. Необходимо помнить, что для разного рода проверок требуются время и ресурсы. Число человеко-часов, которое уйдет на выполнение объектного проекта, будет в два-три раза выше, чем в случае обычного программного проекта.

Промежуточное тестирование

Выше говорилось о различных формах тестирования программного обеспечения. Под *прозрачностью* системы понимается число ее видимых внутренних компонентов. Когда система полностью открыта (прозрачный ящик), испытатель проверяет каждую ее функцию и инструкцию, каждый модуль и условный блок. Обычно это вполне может сделать программист или разработчик.

Тестирование по методу черного ящика позволяет оценить систему с точки зрения конечного пользователя. Это напоминает приемочные испытания, когда правильность системы проверяется на основании типичных сценариев ее использования.

При промежуточном тестировании испытатель больше знает о внутренней реализации системы (вплоть до уровня отдельных модулей или функций, но не дальше), чем в предыдущем случае. Проверка здесь также проводится на основании типичных сценариев использования.

Понятие системной интеграции

С появлением объектов стало размываться понятие системы как набора тесно связанных компонентов. Появились проекты, представляющие собой совокупность программ, которые работают локально или распределены по сети. В действительности истинной системной интеграции уже давно не существует. Это стало очевидным в последние несколько лет господства Internet, когда цикл разработки программных продуктов сократился до 3-х месяцев. Компании, употребляющие термин "системная интеграция", как правило, подразумевают совсем не то, что первоначально означал этот термин. В случае объектов интеграция происходит тогда, когда выпускается финальная версия класса.

Резюме: зыбучие пески ООП

Объектная технология предоставляет множество возможностей, но все имеет свою цену. Приходится сталкиваться со сложностями теории, технологическими трудностями и проблемой обучения персонала. Важно правильно руководить командой разработчиков, чтобы каждый участник процесса четко знал свою роль и работал согласованно с остальными.

Часть IV

Сложные сетевые методики

В этой части...

Глава 15 Удаленные вызовы процедур (RPC)

В этой главе...

Если смотреть с точки зрения прикладного программиста, то знать все детали сетевого программирования достаточно затруднительно. Иногда хочется просто сосредоточиться на разработке конкретного программного алгоритма, предоставив детали сетевого взаимодействия соответствующим библиотекам и функциям. И здесь на арену выходит технология RPC (Remote Procedure Calls — удаленные вызовы процедур).

RFC-модули управляют всеми сетевыми соединениями и регулируют передачу данных в сети. Во многих Приложениях работа В сети — это лишь малая часть общего набора функциональных возможностей. В таких случаях, тратить время на написание и отладку сетевых интерфейсов было бы слишком дорого, и неэффективно. С другой стороны, можно создать специальные компоненты (реализующие вызовы сетевых сервисов), которые будут использоваться в различных программах в готовом виде. Эти компоненты должны взаимодействовать с клиентами, серверами и одноранговыми компьютерами. В зависимости от протокола, такой компонент может быть либо сложным, включающим большое число проверок и ретрансляций (UDP), либо простым (TCP).

В этой главе демонстрируются два способа создания RPC-компонентов: вручную, без применения каких-либо специальных инструментов, и с помощью утилиты rpcgen. Но прежде необходимо вспомнить о том, что такое сетевая модель, и понять, какое место в ней занимает RPC.

Возвращаясь к модели OSI

В главе 5, "Многоуровневая сетевая модель", рассказывалось о том, что в сетевой модели все операдии и функциональные возможности распределены по уровням, что позволяет скрывать информацию как от пользователя, так и от программы. На самом деле в каждую операцию включаются сведения о том, как сетевая подсистема должна передавать данные от одного компьютера к другому. Имеются также средства, позволяющие обеспечить доставку данных. Согласно сетевой теории, пользователь не должен взаимодействовать ни с какими сетевыми интерфейсами, кроме средств прикладного уровни (№7).

Однако в семействе протоколов TCP/IP всего 4 уровня. Последний из них — TCP — располагается далеко от прикладного уровня, хотя и обеспечивает надежную доставку сообщений. Это самый надежный протокол семейства, достаточно мощный и гибкий. Он позволяет создавать сокеты, которые ведут себя подобно файловым потокам. Но ответственность за реализацию верхних сетевых уровней возлагается на приложения.

Это создает определенную проблему, так как написано много приложений, дублирующих работу друг друга. Сравните протоколы FTP и Telnet. В каждом из них есть этап аутентификации - процедура регистрации пользователя в системе. Хотя интерфейс регистрации реализовать несложно, происходит дублирование усилий. Более того, задача программиста от этого только усложняется. Когда *возникает* необходимость встроить в программу процедуру аутентификации, ее приходится писать самостоятельно.

Часть функциональных возможностей можно покрыть своими собственными интерфейсами. Это позволит другим программам использовать готовые функции, как часто происходит в различных сетевых моделях, а сами интерфейсы можно сделать *прозрачными* по отношению к сети. Программы, работающие с "прозрачными" протоколами, избавляют пользователя (или программиста) от не-

обходимости самостоятельно взаимодействовать с сетью. Идея заключается в том, чтобы сделать сетевое соединение по возможности автоматизированным.

Сравнение методик сетевого и процедурного программирования

При написании сетевых приложений сталкиваешься с такими аспектами программирования, с которыми программным инженерам редко приходится иметь дело. Это очень интересный процесс, но он требует специальных знаний. Необходимо заранее продумать, как будут взаимодействовать клиент и сервер (система).

Сетевое программирование отличается от обычного процедурного программирования. В последнем гораздо больше "свободы", так как ошибки не столь критичны и обязанностей меньше. Когда создается программа, используемая на разных узлах сети (клиенте и сервере), пользователи ожидают повышенный уровень качества и надежное восстановление после сбоев.

Чтобы создать библиотеку сетевых компонентов, нужно преодолеть множество преград. Две из них — производительность и надежность, но есть также ограничения самой технологии.

Границы языков

Первая проблема, с которой можно столкнуться, — это сам язык сетевого взаимодействия. Например, в языке C можно делать много такого, что не разрешается сетевыми протоколами. В первую очередь нужно помнить о том, что передача данных в сети осуществляется в виде *значений* этих данных. Указатели недопустимы по очевидной причине: адрес памяти на одном компьютере не равен адресу на другом. Это противоречит некоторым распространенным принципам программирования. Никому не нравится передавать структуру или массив по значению, так как копии данных "съедают" память и ресурсы процессора.

Копирование данных в языке C

В языке C предполагается, что программист не хочет передавать большие блоки данных по значению, так как это ведет к снижению производительности из-за ненужного копирования данных. Поэтому по возможности стараются задавать параметры функций в виде указателей или ссылок. Во всех современных языках поддерживаются скалярные (int и char) и векторные (массивы и структуры) типы данных. По умолчанию в C-программах скалярные величины передаются по значению. В то же время некоторые компиляторы выдают предупреждение, если аналогичным образом передается структура. Чтобы передать массив по значению, нужно создать новый тип данных или записать массив в структуру. В любом случае программа использует аппаратный стек для временного хранения данных, и необходимо убедиться, что размер этого стека достаточен для помещения в него больших информационных блоков.

В результате возникает проблема возвращения данных обратно из процедуры или функции. Можно передавать все данные по значению, но иногда без указателей не обойтись. Ниже показан прототип функции getphoto(), реализующей запрос к серверу на получение фотографии. Типы данных image_t и host_t являются поль-

зовательскими. Без применения указателя невозможно было бы обнаружить ошибки, так как ни структура, ни массив не могут принимать значение NULL.

```
/*******************************************************/
/*** Пример, передачи аргумента по значению -- запрос? ***/
/*** на получение фотографий                           ***/
/*******************************************************/
image_t *getphoto(host_t host);
```

Возвращение векторов по значению

В языке С разрешается передавать вектор по значению.Но в конкретной реализации компилятора могут быть свои особенности. В зависимости от того, как в процессорах реализован аппаратный стек, компилятор может на самом деле не копировать результаты S стек. Вместо этого он создает локальную или статическуюдаременную и возвравщает ссылку на нее. Получив ее, вызывающая функция копирует значение переменной в нужную область памяти. В многопотоковых программах это может вызывать различные побочные эффекты. Кроме того, некоторые компиляторы настолько "умны", чго фиксируют адреса памяти перед вызовом функции. Когда функция заполняет вектор, это изменение в действительности отражается на исходной области памяти.

Параметры функций сами по себе представляют отдельную проблему. Большинство профаммистов не обозначает, какие параметры являются входными, какие — выходными, а какие — смешанными. Но сетевая подсистема должна различать, какие параметры отправляются по сети, а какие — заполняются данными. Как правило, соответствующие указания даются только в документации. Рассмотрим следующий пример:

```
/*******************************************************/
/***                Пример сетевой функции           ***/
```

```
/*                    ***ЗАМЕЧАНИЯ***                  */
/* getuserinfo() — получение информации о пользователе */
/*                    от сервера                       */
/*    user — (входной) идентификатор пользователя      */
/*    host — (входной) внешнее имя сервера             */
/*'    data-- (выходной) результаты запроса            */
/* ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ: успех йЯи неудача            */
/*                          (проверьте переменную errno) */
int getuserinfo(char* user, char* host, userinfo_t *data);
```

Параметры задаются в следующем порядке: сначала входные, затем смешанные, в конце — выходные. Все параметры передаются по ссылке, и разработчик вызова подразумевает, что пользователь ознакомился с текстом замечания.

Сохранение сеанса в активном состоянии

Взаимодействие клиента и сервера на уровне вызова сервисов предполагает наличие "прозрачного" канала. Не нужно требовать от пользователя зарегистрироваться больше одного раза и проверять, сколько соединений устанавливается в профамме. Все это является частью концепции сетевых сеансов.

Сетевой вызов в своей простейшей форме не хранит информацию о состоянии, он лишь требует от сервера выполнить четко определенное действие. До в случае организации сеанса необходимо осуществить ряд подготовительных процедур как на стороне клиента, так и на стороне сервера.

Сетевой сеанс подразумевает определенные гарантии сохранности информации о состоянии, которые ожидаются большинством пользователей. В случае RPC подобные гарантии нужно реализовать самостоятельно.

Организация диалога

Первый шаг в реализации функциональных возможностей сеансового уровня заключается в определении порядка взаимодействия и строгом его соблюдении. В большинстве RFC-соединений требуется определенная форма аутентификации, но сам диалог инициируется клиентом.

Начальное соединение между клиентом и сервером обычно требует регистрации/Клиент должен предоставить определенную информацию о себе (например, имя пользователя и пароль). Для соблюдения безопасности необходимо обеспечить конфиденциальность этой процедуры, воспользовавшись какой-нибудь разновидностью шифрования, чтобы другой пользователь подсети не смог перехватить информацию. После того как регистрация будет завершена, клиент и сервер решат, кто должен начинать диалог.

Иногда потеря сообщения может привести к разрыву соединения: и клиент, и сервер начинают ждать друг друга, а в результате возникает тайм-аут. Как описывалось в главе 10, "Создание устойчивых сокетов", есть способы следить за тем, чья сейчас очередь передавать данные. Проще всего периодически посылать внеполосные сообщения, информируя противоположную сторону о своем состоянии.

Контрольные точки

В некоторых соединениях, ориентированных на использование сетевых сервисов, могут выполняться критические транзакции, которые не допускают потерю данных (например, банковские транзакции). В этом случае, после того как клиент и сервер начали диалог, необходимо периодически сохранять промежуточную информацию о состоянии (создавать *контрольные точки)*.

Создание контрольной точки напоминает вызов SQL-инструкции COMMIT. Во время сеанса может выполняться несколько транзакций, каждая из которых изменяет состояние сервера (например, снимает деньги со счета или переводит их на другой счет). Как клиент, так и сервер должны следить за выполнением транзакций и проверять их результаты. Контрольная точка — это как бы промежуточная проверка баланса после нескольких трансферных операций.

С контрольными точками связан один дополнительный нюанс. Иногда клиент в целях аудита ведет журнал транзакций, хранящийся в файле на локальном диске. В этом случае нужно шифровать данные в каждой транзакции.

Возобновляемые соединения

Для сеанса требуется одно соединение, которое будет постоянно оставаться открытым. Это требование не всегда осуществимо, так как сетевое соединение может в любой момент прерваться. Кроме того, библиотека Socket API не сообщает явным образом о том, когда происходит разрыв соединения.

Возобновление соединения означает не только автоматическое повторное подключение, но и то, что программа должна проверять *живучесть* канала (он активен и в нем нет помех). Способы проверки каналов описывались в главе 10, "Создание устойчивых сокетов".

Восстановить соединение легко, если подобная возможность предусмотрена в программе. Одна из проблем при потере Связи заключается в необходимости определить, в какой точке находилась программа до этого. Эту точку следует пометить чем-то наподобие закладки. Такие закладки называются *идентификаторами сеансов* и поддерживаются как клиентами, так и серверами. Не имея данного идентификатора, возобновить сеанс вряд ли возможно.

Когда программа обнаруживает разрыв соединения, она просто повторно подключается к серверу. Важно, чтобы этот процесс был максимально автоматизирован и требовал минимального вмешательства пользователя.

Автоматические подключения

Цель заключается в том, чтобы снабдить пользователя удобным интерфейсом, позволяющим ему с наименьшими усилиями возобновлять сеанс. Некоторые приложения не имеют этого и требуют от пользователя повторно вводить пароль или устанавливать повторное соединение вручную.

Необходимо предоставить пользователю возможность вводить столько информации, сколько он сам захочет. Это означает, что может потребоваться хранить регистрационную информацию на протяжении сеанса. Следует, однако, помнить о безопасности. Запись паролей в файл на случай автоматической повторной регистрации создает совершенно ненужную брешь в системе защиты. (Удивительно, но некоторые профессиональные приложения реализуют данную возможность, причем по умолчанию!) Кроме того, можно закончить сеанс и забыть, что регистрационная информация где-то хранится. В самых защищенных соединениях необходимо заставлять пользователя проходить процедуру аутентификации при каждом повторном подключении.

Связующие программные средства

Можно написать набор сервисных функций, взаимодействующих с сервером, которые будут скрывать детали сетевых соединений. Как правило, подобные программные средства промежуточного уровня выполняют *пассивную* работу, т.е. в них не производится никаких вычислений, а лишь осуществляется копирование и трансляция данных из одного места в другое.

Промежуточное ПО — неотъемлемая часть современных программных продуктов. Большинство приложений взаимодействует с этим ПО на различных уровнях. Чрезвычайно важно, чтобы связующие компоненты были надежными.

Первый шаг в создании хорошей связующей сервисной функции заключается в определении долгосрочного интерфейса взаимодействия. Вряд ли другим программистам понравится, если вы объявите им, что написанная /функция будеу модифицирована через полгода. Краткосрочные интерфейсы для такой фундаментальной технологии, как сетевое программирование, никогда не завоюют популярности.

Таким образом, сначала нужно сформулировать общий замысел: определить цели и намерения, подвести теоретическую базу. Затем можно переходить к созданию сетевых заглушек.

Сетевые заглушки

Когда взаимодействие клиентской программы с сетевыми функциями носит пассивный характер, интерфейс сводится к вызовам простейших процедур. В Основном эти процедуры осуществляют лишь упаковку и распаковку параметров сетевых вызовов.

Функции getphoto() и getuserinfo(), упоминавшиеся выше, занимаются лиШь тем, что запрашивают данные. Эти функции могут Использоваться в системе идентификации пользователей. Когда требуется получить информацию о ком-то вместе с его фотографией, вызывается программа, которая Запрашивает и отображает эту информацию.

Сетевые заглушки — это специальные промежуточные процедуры, которые определяют, какая информация нужна сетевой функции и каким образом сервер может выполнить этот запрос. Их интерфейс также должен быть максимально простым, так как чем больше информации предоставляет пользователь, тем труднее выполнить запрос.

Реализация сервисных функций

После того как сервисная функция приняла запрос на стороне клиента, она должна упаковать данные и отправить их на сервер. Сервер, в свою очередь, реализует свой набор сервисных функций, которые распаковывают данные и обслуживают запрос. Подобный процесс составляет суть промежуточного ПО.

Определение клиентских сервисных функций

Клиентские сервисные функции запускаются только по запросу, чтобы сетевое соединение активизировалось лишь на время выполнения запроса. Им необходимо знать адресата, запрос и данные. Большинство запросов выполняется по принципу единой транзакции, поэтому связующим функциям достаточно просто упаковать данные и послать их на сервер. В промежутках между запросами модуль сервисных функций неактивен. Такая схема работы отлично подходит для протокола UDP, в котором от клиента к серверу передается единственное сообщение.

В других ситуациях требуется постоянно открытое соединение, т.е. ведется полноценный диалог, а не просто выполняется одиночный запрос. Это область протокола TCP. В данном случае роль клиентских сервисных функций меняется, так как канал следует держать открытым, к тому же клиент может вести несколько диалогов с разными серверами. Помимо обслуживания традиционных запросов, модуль сервисных функций должен хранить информацию о сеансах и соединениях. Как правило, для этого в модуль добавляются две дополнительные функции: для открытия, и закрытия сеанса.

Реакция на стороне сервера

Серверные рвязующие функции ведут себя так же, как и на стороне клиента. Сервисные модули можно создавать не только на клиентском компьютере, но и на сервере, однако структура самих модулей различна.

Первое, что следует учитывать при создании серверных модулей, — это возможность одновременного поступления запросов. Можно принимать и обрабаты-

вать их последовательно (помните, что ядро помещает все сообщения в очередь, пока функция recv() не извлечет их). Но нельзя заранее сказать, как долго запрос может находиться в очереди. На обработку предыдущих запросов может уйти очень много времени.

Реализация многозадачного режима на сервере позволяет сократить задержки, возникающие в системах обработки транзакций. Linux работает гораздо эффективнее, когда программа является многозадачной (разделена на процессы и потоки). Для каждого нового запроса будет создано отдельное задание, связанное с его обработкой. Схема работы, профаммы будет немного иной, чем описанная в главе 6, "Пример сервера".

1. Клиентская программа вызывает клиентский сетевой модуль.
2. Клиентский сетевой модуль упаковывает запрос и данные и посылает их в сеть, после чего переходит в режим ожидания ответа.
3. Серверный сетевой модуль принимает запрос и переадресует его серверной профамме.
4. Серверная программа обрабатывает запрос и возвращает результат.
5. Серверный сетевой модуль принимает результат, упаковывает его и отправляет обратно в сеть.
6. Клиентский сетевой модуль принимает сообщение, содержащее результаты запроса, распаковывает их и возвращает клиентской профамме.
7. Клиентская программа принимает результаты выполнения функции.

Читатели, должно быть, обратили внимание на то, что клиентский и серверный модули занимают разное положение в своей иерархии. Клиентский модуль находится на самом нижнем уровне, тогда как серверный модуль расположен выше связанной с ним профаммы. Подобная двойственность представляет собой проблему с точки зрения реализации повторно используемых компонентов.

Решить проблему можно двумя способами. Первый заключается в написании внешней процедуры, которую будет вызывать серверный сетевой модуль. Серверная профамма должна определить эту процедуру как точку входа в блок обработки запросов. Данное решение неудобно тем, что приходится использовать предопределенные имена процедур.

Другое решение состоит в предоставлении сервиса регистрации. В процессе инициализации, еще до того как сервер будет готов обслуживать соединения, серверная профамма должна зарегистрировать все свои подпрофаммы обработки запросов. Это решение выглядит наиболее удачным и гарантирует долговечность интерфейсов.

Реализация представительского уровня

В процессе передачи данных часто достаточно лишь скопировать их в тело сообщения и отправить его в сеть. Однако нет уверенности в том, что все клиенты понимают формат данных сервера. Очевидный пример — порядок следования байтов. Можно также упомянуть о различиях между кодировками ASCII и EBCDIC.

Представительский уровень модели OSI обеспечивает средства преобразования данных из одного формата в другой. Проектируя интерфейсы, необходимо преду-

смотреть, на каких клиентских и серверных платформах будет работать программа. Важно добиться, по крайней мере, независимости от порядка следования байтов.

Создание RPC-компонентов с помощью утилиты rpcgen

В состав Linux часто входит утилита rpcgen, позволяющая создавать свои собственные RFC-модули. Она существенно упрощает RFC-программирование и помогает создавать полноценные сетевые приложения. Ниже описаны ее синтаксис и особенности применения.

Язык утилиты rpcgen

Утилита rpcgen — это еще один языковый транслятор, распространенный в среде UNIX. Язык, поддерживаемый этой утилитой, позволяет определять интерфейс сетевой программы и данные, передаваемые ею по сети. По существующему соглашению утилита работает с файлами, имеющими расширение .x. Формат файла напоминает формат программы, написанной на языке C, за несколькими исключениями.

Технология RPC находит применение в разных областях. В частности, она служит неотъемлемой частью технологии COM+, которая является расширением более ранней технологии DCOM. В файлы .IDL

Создание простейшего интерфейса

Ниже показано, как определить интерфейс, в котором на сервере запрашивается время в формате UTC (число секунд, прошедших с 1-го января 1970 г.):

```
/*** Определение интерфейса, запрашивающего время на сервере ***/
/*********************** ****************************************/
program RPCTIME
{
    version RPCTIMEVERSION
    {
        long GETTIME() =1
    } = 1;
} = 2000001;
```

В первой строке объявляется имя программы — RPCTIME. Объявление имен в верхнем регистре является общепринятым соглашением, хотя сама утилита rpcgen преобразует все имена в нижний регистр. У одного и того же интерфейса может быть несколько версий, поэтому в следующем блоке определяется первая версия. Внутри него размещен собственно вызов процедуры.

Каждому блоку в разделе объявлений присваивается числовой идентификатор. Он важен для организации взаимодействия клиента с сервером. Имени версии, а

также имени процедуры можно присвоить произвольный идентификатор, а вот некоторые программные идентификаторы являются зарезервированными. Можно смело работать с числами в диапазоне 2000000-3000000.

Как можно заметить, в данном примере функция возвращает значение типа long, а не time_t. Утилита rpcgen позволяет указывать любой тип (в действительности она просто игнорирует его), оставляя семантическую Проверку компилятору языка С. Тем не менее допускается давать утилите указания относительно преобразования типов. Как и в языке С, можно создать макроопределение, чтобы дальше использовались стандартные обозначения:

```
typedef long time_t;
```

Подсистема XDR утилиты rpcgen автоматически будет выполнять все необходимые преобразования.

Запуск утилиты должен осуществляться так:

```
rpcgen -a rpctime.x
```

Опция -а указывает на необходимость создания всех файлов, которые могут потребоваться для работы клиента и сервера. Без этой опции будут созданы только клиентский (rpctime_clnt.c) и серверный (rpctime_svc.c) интерфейсные файлы. Если определялись дополнительные типы данных, будет также сгенерирован XDR-файл (rpctime_xdr.c). Не редактируйте ни один из этих файлов — они создаются динамически на основании исходного X-файла.

При наличии опции -а утилита rpcgen дополнительно сгенерирует тестовые клиентский (rpctime_client.c) и серверный (rpctime_server.c) файлы, а также файл Makefile.rpctime, предназначенный для последующей компиляции всего проекта. Это существенно упрощает процесс разработки.

Будьте осторожны, выполняя команду make clean

Если вы планируете включить тестовые файлы в проект, не запускайте утилиту pake со стандартной опцией clean. Установки файла Makefile приводят к их ошибочному удалению.

Просмотрев полученные файлы, можно заметить, что вызов процедуры переименован и теперь называется gettime_1(). Суффикс _1 обозначает номер версии. Возможно, было бы проще просто вызвать функцию gettime() и предоставить системе возможность самой решить, какую из версий следует вызвать в зависимости от числа и типа параметров. Но утилита rpcgen так не работает.

После запуска команды rpcgen -а и получения тестовых файлов можно приступить к написанию кода, связанного с обработкой информации о времени. Откройте файл rpctime_client.c и добавьте в него указанный ниже фрагмент:

```
/*************************************************/
/***       Фрагмент RFC-клиента                ***/
/*************************************************/

/* — Генерируется автоматически —*/
result_1 = gettime_1((void*)&gettime_1_arg, clnt);
if (resultlt_1 == (long *) NULL) {
   clnt_perror(clnt, "call failed");
}
/* — Добавьте следующий код —*/
```

```
else
   printf("%d|%s", *result_l, ctime(result_l));
```

Добавленная инструкция просто осуществляет вывод результатов. Серверный код будет таким:

```
/*********************************************/
/***          Фрагмент RFC-сервера         ***/
/*********************************************/
static long result;
time(&result);    /*— Пользовательский код —*/
return&result;
```

Утилита rpcgen добавляет раздел комментариев, указывающий на то, что серверный код следует вставлять между блоком объявления переменных и оператором return. Отредактировав файлы, скомпилируйте и протестируйте проект:

```
make -f Makefile.rpctime
./rpctime_seryer &
./rpctime_client127.0.0.1
```

В результате на экран будет выдано текущее Число секунд в стандартном ASCII-формате.

Использование более сложных X-файлов

Разобравшись с работой утилиты rpcgen, можно попробовать воспользоваться ее встроенными типами данных string и bool_t. Последний представляет собой отсутствующие в языке С булевы значений. Переменная типа bool_t может содержать либо 0, Либо 1, хотя для ее хранения отводится больше одного бита.

Тип string требует более подробных пояснений. В языке С тип char* обозначает указатель на char, массив элементов типа char или строку, завершающуюся символом NULL. Возникает Определенная неоднозначность, разрешить которую и предназначен тип string. Определения его значений выглядят так:

```
string filename<100>; /*— длина до 100 символов —*/
string myname<>;      /*— произвольная длина —*/
```

Все строки имеют произвольную длину. Если же известна предельная" длина, то ее можно задать. Это позволит гарантировать, что подсистема XDR передаст строго указанное число байтов.

В X-файл можно включать определения объединений (union), которые содержат конструкцию switch:

```
/***          Определение RFC-объединения          ***/

union proc res switch (int Err)
{
   case 0:
        string Data<>;  /*— Если параметр Err равен 0,
                значением объединения будет строка Data —*/
   default:
```

```
        void; /*— Если параметр Err не равен 0,
                   у объединения нет значения ---*/
};
```

В данном примере объединение будет либо строкой, либо пустым множеством, в зависимости от значения параметра Err. Утилита rpcgen преобразует это определение в следующую структуру:

```
struct proc_res
{
   int Err;
   union
   {
      char *Data;
   } proc_res_u;
};
```

Объединения полезны, когда необходимо вернуть конкретный код ошибки, не задействуя подсистему XDR. Серверные функции всегда возвращают указатели на результирующие значения, а не сами значения. Если вызов функции завершился неудачно, возвращается NULL. Но, во-первых, это не слишком содержательная информация, а во-вторых, данное значение может перехватываться функциями более низкого уровня.

В следующем фрагменте программы объединение используется для того, чтобы вернуть либо информацию о файле, либо код ошибки:

```
/*************************************************************/
/***              Объявление программы RPCProc             ***/

union proc_res switch (int Err)
{
   case 0:   /*— При отсутствии ошибок возвращается строка —*/
        string Data<>;              /*— Длина не ограничена —*/
   default:  /*— При наличии ошибок ничего не возвращается —*/
        void;


program RPCPROC
{
   version RPCPROCVERSION
   {
      proc_res READPROC(string) = 1; /*— имя файла в каталоге
                                          /proc  —  */
   } = 1;
} = 2000025;
```

Сервер принимает имя файла, указывающее на элемент виртуальной файловой системы /proc, и открывает файл. Если не возникает ошибок, сервер читает содержимое файла, закрывает его и возвращает результат. Здесь можно столкнуться с небольшой проблемой. Дело в том, что сервер возвращает ссылку на результирующие данные, поэтому они не должны помещаться в стек. Все возвращаемые

значения являются статическими, чтобы, не возникало проблем со стеком. Тем не менее строки всегда приводятся к типу char*.

При заполнении возвращаемого сообщения необходимо выделить память для вставки строк. Это означает также, что необходимо как-то освобождать ату память, иначе произойдет потеря ресурсов. Имеется специальная XDR-процедура, осуществляющая очистку после предыдущего вызова:

```
xdr_free((xdrproc_t)xdr_proc_res, (void*)&result);
```

Единственный параметр, который изменяется, — это xdr_<*возвращаемый_тип*> (в данном случае xdr_proc_res). Всегда нужно вызывать эту функцию, прежде чем переходить к другим действиям. При первом вызове она ничего не предпринимает. (Полный текст примера имеется на Web-узле.)

Добавление записей и указателей

Следующий тип данных должен быть знаком читателям. Тип struct соответствует структуре в языке C и служит в основном тем же целям. Есть и одно новшество, связанное с ограничением утилиты rpcgen: сервисным функциям можно передавать лишь один параметр. Если параметров несколько, необходимо применять структуры.

Определение структуры выглядит так:

```
/*********************************/
/*** Фрагмент программы RPCList ***/

typedef struct NodeStruct *TNode;
struct NodeStruct
{
   string Name<>;
   TNode Next;
}
```

Обратите внимание на то, что в теле структуры присутствует указатель (TNode). Если помните, раньше говорилось о том, что по сети нельзя передавать указатели: адресные пространства не совпадают, поэтому ссылки на конкретные адреса бессмысленны. Но подсистема XDR достаточно "умна": она копирует все адресные ссылки и преобразует их в специальную структуру, которая "расшифровывается" на стороне клиента или сервера. Благодаря этому по сети можно передавать двоичные деревья, связанные списки и другие динамические конструкции.

Единственная странность заключается в синтаксисе X-файла. Тип указателя должен обязательно быть задан с помощью ключевого слова typedef::

```
typedef struct NodeStruct TNode;
struct NodeStruct
{
   string Name<>;
   TNode *Next;  /*— Неправильно! —*/
```

Учет состояния сеанса в открытых соединениях

В процесс передачи информации от клиента к серверу вовлечены не только сами данные. Программа часто отслеживает также состояние системы, чтобы правильно реагировать на запрос. Выше рассматривались программы, в которых ничего не было известно о результатах работы других сервисных функций. Все вызовы не зависели друг от друга *(соединение, не имеющее состояния)*.

В соединениях противоположного типа необходимо хранить некоторую информацию о состоянии. В качестве примера можно привести сведения об аутентификации клиента (пользователь должен зарегистрироваться в системе, прежде чем получить доступ к данным) и запросы к базам данных (сервер может не передавать всю таблицу целиком).

Выявление проблем с состоянием сеанса

Соединения, не имеющие состояния, просты. Клиент посылает информацию, а сервер возвращает ответ. Если сообщение было потеряно, клиент повторяет свой запрос.

В соединениях, имеющих состояние, клиент и сервер должны убедиться в том, что протокол установления соединения обеспечивает идентификацию и восстановление сеанса в случае сбоя. В таких соединениях приходится решать следующие проблемы.

- *Текущее состояние* — в сеансе должно храниться описание уже выполненных действий. Никому не хочется отвечать на одни и те же вопросы или повторно выполнять те же самые действия.
- *Направление сеанса* — сеанс должен протекать в определенном направлении. Все действия в ту или другую сторону должны быть четко определены.
- *Восстановление состояния* — при разрыве соединения системе может понадобиться вернуться в предыдущее согласованное состояние. С этого момента клиент и сервер должны благополучно возобновить работу. В случае необходимости можно полностью отменить транзакцию.

В некоторых случаях корректное восстановление затруднительно. Но в целом клиент и сервер должны уникальным образом идентифицировать текущий сеанс работы.

Хранение идентификаторов

В простейшем случае в соединении хранится идентификатор сеанса. Этот идентификатор должен быть зашифрован, чтобы только клиент и сервер понимали его значение. Рассмотрим такой пример. Клиент запрашивает у серверной базы данных первую строку из таблицы результатов запроса. Чтобы извлечь следующую строку, сервер должен иметь возможность связать очередной вызов с первоначальным запросом.

Можно решить эту проблему, разделив все вызовы на три этапа: инициализация, транзакция и завершение. Назначение первого этапа заключается в организации сеанса и запуске процесса обработки данных. На втором этапе функции меняют состояние сеанса. На последнем этапе происходит завершение всех отложенных вызовов и закрытие соединения. С этого момента все полученные данные считаются окончательными.

После того как сеанс окончен, его идентификатор считается недействительным.

Следование заданному маршруту

Три основных состояния — инициализация, транзакция и завершение — это ключевые вехи, очерчивающие направление сеанса. Имея информацию о том, в каком состоянии находится сеанс, клиент и сервер точно знают, что они могут делать и как перейти в следующее состояние.

Все состояния должны быть уникальными, как и операции перехода между ними. Один и тот же переход не может привести к двум разным состояниям. В частности, не должно быть так, чтобы в результате транзакции и клиент, и сервер перешли в режим ожидания сообщения. Без подобного детерминизма нельзя корректно восстановить сеанс в случае сбоя.

Восстановление после ошибок

Среди имеющихся проблем восстановить сеанс после сбоя труднее всего. В сети существуют разные причины сбоев. Они могут возникать на аппаратном уровне, при маршрутизации и наличии помех в канале связи, из-за ошибок в программах и т.д.

Сбой сеанса — неприятная проблема, если не быть к ней готовым. В первую очередь программа должна проверять каждый идентификатор сеанса. Необходимо также обеспечить безопасность идентификатора, истечение его срока действия и идентификацию конкретного состояния. Можно включать в него признак последнего состояния и даже возможного следующего состояния.

Восстановление состояния связано с определенными трудностями. Прежде всего нужно определить, что же делать дальше. Вот несколько идей.

- *Сброс соединения* — клиента заставляют начать сеанс заново. Сервер отменяет все транзакции, имевшие место в ходе сеанса.
- *Возврат к предыдущему состоянию* — клиент и сервер переходят в заранее известное состояние. Незаконченные транзакции отменяются. Если новые состояния клиента и сервера противоречат друг другу, выбираются другие состояния либо происходит сброс соединения.
- *Принудительный переход в заданное состояние* — сервер указывает клиенту, какое состояние является правильным. Если клиент не может в него перейти, происходит сброс соединения.

Выбор того или иного варианта делается на основании того, насколько критичными являются транзакции.

Резюме: создание набора RPC-компонентов

Используя свой опыт программирования сокетов, можно помогать другим разрабтчикам создавать сетевые приложения, не заботясь о деталях реализации. Удаленные вызовы процедур можно реализовывать двумя способами: самостоятельно создавая сетевые интерфейсы или применяя стандартные средства, в частности утилиту rpcgen.

6 первом случае необходимо задать, как будут передаваться данные. Здесь нет никаких ограничений; можно передавать столько параметров, сколько нужно для функции, причем в том порядке, который считается наиболее удобным. Ио нельзя использовать указатели, и копировать все данные приходится вручную.

Утилита rpcgen позволяет сосредоточиться на программе, а не на деталях передачи данных. В ней имеется множество средств автоматического преобразования и поддерживается передача указателей. С этой утилитой очень легко работать, и она существенно упрощает программирование.

Большинство RFC-соединений не имеет состояния. Они функционируют по схеме "запрос — ответ". Но в некоторых соединениях требуется более сложное взаимодействие, когда текущий вызов функции зависит от предыдущих вызовов. Чтобы достичь этого уровня взаимодействия, необходимо самостоятельно организовывать сеансы, осуществлять обработку информации о состоянии и восстановление после сбоев.

На технологии RPC основан протокол SSL (Secure Socket Layer — протокол защищенных сокётов), в котором поддерживается множество состояний. Особенности этого протокола, а также вопросы сетевой безопасности рассматриваются в следующей главе.

Глава

16 Безопасность сетевых приложений

В этой главе...

Безопасная система предоставляет не больше функций, чем незащищенная, просто последняя подвержена атакам и ее данные становятся ненадежными. Уровень надежности определяет качество обслуживания и напрямую связан с репутацией компании.

В большинстве случаев безопасность создаваемого программного продукта считается само собой разумеющейся. В исходных требованиях редко говорится: "Сервер должен быть невосприимчив к сетевым атакам". Безопасность — это базовое требование, и клиент воспринимает ее как неотъемлемую часть продукта. Поэтому в программы нужно встраивать необходимые средства защиты. Самый сложный вопрос: "Какой уровень безопасности необходим?"

В этой главе рассматриваются многие аспекты безопасности сетевых приложений: основные концепции и термины, особенности среды Internet, методы защиты и их реализация, а также протокол SSL.

Потребность в защите данных

Сетевое программирование основано на идее совместного доступа к данным и распределения вычислений. Предоставляя доступ к своим данным, необходимо точно знать, кто к ним обращается: друг или враг. Увеличивающееся число атак на современные компьютерные системы свидетельствует о том, что врагов становится все больше, поэтому зачастую безопасность продукта более важна, чем его функциональные возможности.

С процессами сетевого взаимодействия неразрывно связаны понятия идентификации и доверия. Для обеспечения безопасности требуется проводить идентификацию на стороне как клиента, так и сервера. Только после того как противоположная сторона будет опознана, можно начинать доставку или получение информации:

Уровни идентификации

Определение уровней идентификации пользователя — важная часть процесса создания безопасной системы. Простейшая форма защиты — *аутентификация* — подразумевает обычную проверку регистрационной информации пользователя. Это первая "дверь", в которую проходит пользователь, попадая в систему. В качестве примера можно вспомнить сеанс удаленной регистрации, когда у пользователя запрашиваются имя и пароль.

Следующий уровень защиты — *авторизация* — предусматривает ограничение или запрет доступа к тем или иным ресурсам системы. Утилита удаленной регистрации в Linux позволяет объединить аутентификацию с авторизацией, назначая пользователям групповые права доступа. В некоторых операционных системах средства авторизации весьма ограничены. Например, в Windows 95 и 98 для получения полного доступа к совместно используемым файлам достаточно ввести пароль.

Предыдущие два уровня защиты имеют тот недостаток, что клиент не может определить, является ли сервер "троянским конем". *Сертификация* обеспечивает наивысший уровень безопасности. Она требует, чтобы доверенное третье лицо (сервер сертификатов) гарантировало подлинность как клиента, так и сервера.

Формы взаимообмена

Познакомившись с формами идентификации, рассмотрим, как происходит передача информации между компьютерами. Во время обмена данными могут иметь место два негативных процесса: *подсматривание* и *вторжение*. Подсматривание представляет собой проблему только в том случае, если канал обмена является закрытым. Вторжение, возникающее при регистрации пользователя в системе под чужим именем и паролем, может иметь более тяжелые последствия, так как пользователь способен модифицировать не принадлежащие ему данные.

Очень часто обмен данными осуществляется в открытую, поскольку передаваемые данные не компрометируют ни одну из сторон и не носят конфиденциальный характер. Подобным образом функционируют Web-серверы и многие простейшие сетевые службы.

Открытый обмен данными не означает, что кто угодно может просматривать передаваемую информацию. Имеется в виду, что две взаимодействующие программы не предпринимают никаких мер предосторожности для обеспечения конфиденциальности. Любой компьютер, находящийся в той же локальной сети, может перехватывать открытые сообщения. Открытые каналы в самой большой степени подвержены подсматриванию и вторжению.

Другой формой обмена является групповая передача сообщений, осуществляемая в рамках протоколов TCP/IP. Распространяемые сообщения доступны всем компьютерам данного сегмента сети. Маршрутизаторы не пропускают в сеть посторонние широковещательные сообщения и не выпускают локальные сообщения из сети. Такая форма общения более защищена, чем открытая передача данных, но все равно подвержена подсматриванию (в рамках данного сетевого сегмента). Вторжение здесь мало вероятно, так как в подсеть обычно объединяются "дружественные" компьютеры.

Частные сообщения ограничивают доступ к информации двумя лицами: отправителем и получателем. В них обычно передаются конфиденциальные данные, разглашение которых может привести к серьезным неприятностям. Для обеспечения безопасности и целостности таких сообщений требуются дополнительные меры как на физическом, так и на логическом уровне.

Следует также упомянуть о такой форме сетевого взаимодействия, как выполнение программ на других сетевых узлах. Это может осуществляться средствами RPC или же путем прямой передачи команд (через такие утилиты, как telnet, rlogin или rsh). Данные, передаваемые традиционным способом, называются *пассивными*. В случае повреждения они меняют свой смысл или интерпретацию. Распределенные команды считаются *активными* данными. Будучи искаженными, они меняют порядок своего выполнения и могут нанести вред удаленному компьютеру. Примером такого "оборотня" является вирус Love Bug в Microsoft Outlook.

Активные данные представляют собой самую серьезную угрозу для безопасности системы. Давая разрешение на выполнение удаленных команд, следует убедиться в надежности удаленного клиента.

Проблема безопасности в Internet

В Internet возникают свои, особые проблемы, вызываемые природой самой сети. Можно много рассказывать о том, как создавалась глобальная сеть, но важнее всего то, что она предназначалась для функционирования в условиях ядерной ка-

тастрофы. Это объясняет, почему пути распространения информации от компьютера к компьютеру определяются динамически.

Ни один канал передачи данных в Internet не является постоянным. Нельзя сказать сетевой подсистеме: "Возьми этот пакет и доставь его получателю через такие-то маршрутизаторы". Каждый пакет может распространяться по-своему, периодически проходя через "вражеские" территории;

Все является видимым

Создавая сетевые программы, необходимо постоянно помнить следующее правило: Internet не гарантирует конфиденциальность канала передачи. Когда в зале нью-йоркской фондовой биржи брокер кричит "Продаю!", его слышат все, кто находятся в зале. Но в большинстве случаев они не обращают на него внимания: там это в порядке вещей. Однако иногда находятся люди, для которых данный конкретный крик означает возможность заработать деньги.

Аналогичным образом сообщение может распространяться внутри и вне пределов надежных зон. *Надежная зона* — это сеть, безопасность которой проверена. В качестве примера можно привести магистральные сети AT&T и US Sprint. Другие крупные корпорации покупают к ним доступ через выделенные линии. Они также являются надежными, так как провайдер зоны предоставляет ограниченную гарантию безопасности.

Корпорации создают надежные зоны как часть своих внутренних сетей, которые также строятся на базе протоколов Internet. Иногда две компании объединяют свои внутренние сети через Internet (часто по выделенной линии), и полученная сеть называется *экстрасетью.* Таким образом, границы между внутренними сетями, экстрасетями и глобальной сетью Internet начинают размываться.

В Internet существуют не только надежные зоны, но и "ничейные земли", где сообщения подвержены перехвату. В этих зонах рядовым пользователям могут быть доступны средства, обычно имеющиеся только в распоряжении администратора или пользователя root. Обладая такими возможностями, .нарушители способны просматривать и даже модифицировать сообщения, В этом случае сила бесплатной и мошной операционной системы, каковой является Linux, становится ее недостатком.

Лучше всего стараться избегать риска и защищать передаваемые данные. Придерживайтесь общих рекомендаций, например, всегда надежно прячьте регистра^ ционную информацию и никогда не передавайте номер кредитной карточки в незашифрованном виде.

Виды атак

Защита клиента и сервера начинается с определения того, каким формам атак они могут подвергнуться. Ниже перечислены основные из них.

- *Прослушивание линии* — подслушивающая программа пропускает через себя поток сообщений, ожидая полезных для себя данных. В эту категорию попадают сетевые анализаторы.
- *Разделение линии* — программа-вредитель ограничивает доступ к серверу или клиенту. Она делает что-то с сетью или самим компьютером, что

замедляет время его реакции или вообще делает его недоступным. Это атаки вида "нападение команды ping" b "отказ в обслуживании".

- *Захват линии* — вместо того чтобы затруднять соединение, программа может принимать на себя роль одной из сторон. В некоторых стеках протоколов, например в том, что реализован в Linux, подобная форма атаки затруднена, так как используется система порядковых номеров пакетов.

Могут появляться новые способы вторжений, но, как правило, они попадают в одну из перечисленных категорий. Вирусы представляют собой совершенно другую проблему. Они связаны с недостатками в системе защиты, но не являются разновидностью атак на защищенные данные.

В сетевом программировании защищать нужно все, что подключено к сети. Вопросы безопасности следует рассматривать независимо от типа операционной системы. В Linux средства защиты встроены в систему, и это было задумано изначально. Эффект очевиден: мы имеем очень надежную и защищенную, систему. Те операционные системы, которые реализованы по такому же принципу, обеспечивают достаточный уровень безопасности. Остальные никогда его не достигнут. Тем не менее безопасность сетевых программ не должна зависеть от операционной системы, поэтому степень защищенности программ определяется прежде всего разработчиком.

Незащищенность TCP/IP

Большинство компьютеров сегодня использует протоколы Internet. Даже компании Novell и Microsoft начали отходить от своих патентованных протоколов, чтобы их продукты могли легко работать в Internet. Но насколько безопасно семейство протоколов TCP/IP?

Вообще-то, протокол IP довольно легко "взломать". Сетевые анализаторы, описанные в этой книге, служат хорошим примером того, насколько легко наблюдать за сетевыми сообщениями, Используя неструктурированные сокеты (помните, что протокол IP доступен только пользователю root), программа-взломщик может смоделировать любой протокол. Проще всего поддаются фальсификации протоколы ICMP, UDP и RDP (Reliable Datagram Protocol — протокол надежной доставки дейтаграмм; еще не реализован в большинстве операционных систем).

Каждый протокол в стеке TCP/IP имеет свои слабые стороны. Некоторые специалисты по вопросам безопасности заявляют, что протокол TCP взломать труднее всего. Они заявляют, что порядковый номер пакета служит своего рода гарантией — если нельзя угадать следующий порядковый номер, то нельзя и захватить канал.

С другой стороны, протокол TCP не определяет алгоритм вычисления порядковых номеров. Требуется лишь, чтобы в течение времени своего существования пакет имел уникальный номер. Таким образом, если алгоритм вычисления следующего номера можно предугадать, канал становится ненадежным (обратите внимание: слабым местом является не сам номер, а алгоритм его вычисления), т.е. протокол TCP действительно в определенной степени защищен, но не настолько, чтобы стали невозможными подсматривание и захват канала.

Защита сетевого компьютера

Как же защитить систему от атаки и вторжения? Сетевая безопасность — это очень сложная смесь различных инструментов и правил. Ниже будет дан ряд практических советов, касающихся защиты компьютеров. Некоторые из них не имеют прямого отношения к сетевой безопасности, но все же важны. Не стоит рассматривать приведенный материал как исчерпывающее описание предмета. Чтобы стать специалистом в области защиты данных, нужно читать много разной литературы.

Ограничение доступа

Повысить безопасность серверных и клиентских программ можно следующими средствами.

- *Права доступа к файлам* — в первую очередь следует убедиться, что у файлов верные права доступа и владельцы. Некоторые программы должны выполняться с правами пользователя root (например, для доступа к неструктурированным IP-пакетам). Внимательно проверяйте действия, выполняемые в этих программах. Как правило, программы должны отказываться от привилегий суперпользователя после успешного открытия IP-сокета.
- *Ограничение привилегий* — необходима определить, какие действия имеет право выполнять удаленная программа в ходе сеанса. Некоторые команды должны быть запрещены без должной авторизации.
- *Уменьшение доступных портов* — нужно ограничивать число портов, которые можно открывать. Каждый доступный порт на порядок увеличивает риск взлома системы.
- *Распределение обязанностей сетевых плат* — если имеется маршрутизатор с несколькими Ethernet-платами (или сетевыми интерфейсами), необходимо конфигурировать сервисы только для одной из них. Большинство сервисов пр умолчанию работает со всеми доступными устройствами, но потребность в этом возникает редко. Например, если есть две платы, то программу Telnet, можно сконфигурировать на работу только с той из них, которая предоставляет доступ во внутреннюю сеть, а не с той, которая подключена к Internet.
- *Запрещение ненужных сервисов* — если потребность в конкретном сервисе не возникает (или возникает очень редко), не запускайте его. Если он вдруг понадобится, активизируйте его, но до этого держите выключенным.

Сервер будет работать эффективнее, если отключить все ненужные возможности и сервисы. Просматривая журнальные файлы, можно быстро определить, какие сервисы важные, а какие — лишние.

Брандмауэры

Брандмауэры формируют специальный защитный слой между внешними клиентами и внутренними серверами. Их услуги очень важны с точки зрения обеспече-

ния безопасности внутренней сети. Как правило, брандмауэры функционируют, незаметно для клиента и образуют совершенно прозрачный канал связи с сервером.

Основная роль брандмауэра заключается в фильтрации, которая бывает двух видов: пассивная и активная. При *пассивной фильтрации* проверяется лишь адрес каждого сообщения. Хороший брандмауэр скрывает адреса всех внутренних серверов и осуществляет принудительную трансляцию всех адресов. Кроме того, он может выполнять переназначение портов. Пассивная фильтрация — это первый барьер между клиентом и сервером.

Активная фильтрация связана с более глубоким анализом пакета. Здесь делается попытка определить, не содержит ли пакет опасных или недопустимых команд. При подобной форме фильтрации необходимо знать, какие сервисы доступны на сервере. Например, в FTP-соединении требуются иные средства безопасности, чем в HTTP-соединении.

Серверы, обслуживаемые брандмауэром, могут использовать незарегистрированные IP-адреса. Например, это происходит, когда в сети применяется протокол DHCP. Брандмауэр выполняет трансляцию таких адресов (пассивную фильтрацию), поскольку клиент не сможет начать соединение, не получив реальных адресов.

Некоторые сетевые администраторы полагают, что клиент не может взаимодействовать с сервером, имеющим фиктивный IP-адрес. Это не совсем правильно и ведет к неверному пониманию принципов безопасности. Например, в протоколе FTP используется механизм обратного вызова: клиент делает запрос к серверу на получение файла, но сервер создает канал связи с клиентом. Клиент может находиться в сети, защищенной брандмауэром. В этом случае внешний сервер способен установить соединение через брандмауэр. Это происходит даже в том случае, когда брандмауэр транслирует все адреса.

Демилитаризованные зоны

Брандмауэры увеличивают сетевую безопасность, формируя надежный канал между клиентом и сервером. В некоторых организациях концепция брандмауэров продвинута еще на один шаг вперед: в них создаются так называемые *демилитаризованные зоны* (сокращенно ДМЗ). Нечто подобное происходило в средние века, когда вокруг городов и замков в Европе и Средней Азии возводили двойные или даже тройные стены, демонстрировавшие их могущество.

ДМЗ служат надежным барьером на пути незаконного проникновения. Нарушитель взламывает одну стену и сразу же обнаруживает еще одну. За последним рубежом может находиться внутренняя сеть компании, содержащая информацию, которая составляет корпоративную или коммерческую тайну. Как показывает опыт предков, многоуровневая оборона затрудняет вторжение, позйоляет эффективнее выявлять диверсантов и защищать себя от нападений.

В простейших конфигурациях информационные серверы размещены за пределами брандмауэров и не имеют доступа к внутренней сети (рис. 1-6.1). Это совершенно безопасная система, так как в ней нет канала утечки важной информации. Но возникают две очевидные проблемы: как сервер получает обновления и каким образом осуществляется синхронизация внутренней и внешней информации? Одно из решений — перенос данных вручную. Такого рода серверные "островки" лишь затрудняют доступ к базам данных, инструментальным средствам и динамическим данным, располагаемым во внутренней сети. В демилитаризованной

зоне объединяются в единое целое канал доступа к Internet, каналы обслуживания внешних клиентов и внутренняя сеть компании (рис. 16.2).

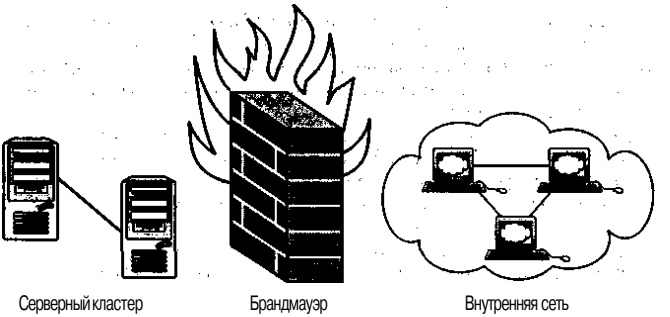

Рис. 16.1. В простейшем случае серверы Internet размещаются за пределами брандмауэра

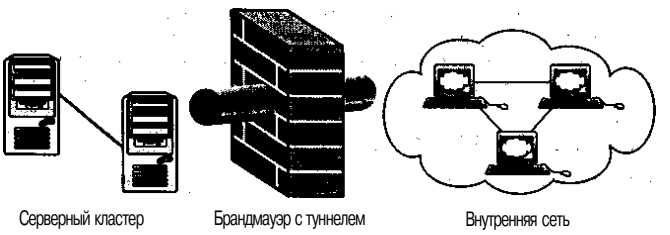

Рис. 16.2. В демилитаризованной зоне имеется канал передачи информации через брандмауэр

Чтобы демилитаризованная зона обеспечивала максимальную эффективность, работа в ней должна быть организована по определенным правилам. Сами брандмауэры редко предоставляют много сервисов: это делают серверы внутри них. Но если в пределах ДМЗ реализуется слишком много информационных услуг, то о ДМЗ лучше забыть и не тратить понапрасну деньги. Необходимо учитывать, что чем ближе к внутренней сети, тем слабее должны становиться меры безопасности. Ниже перечислен ряд общепринятых правил.

- *Минимизируйте число портов* (предоставляемых брандмауэрами), Большинство брандмауэров делает доступным внешнему миру лишь небольшое число портов. Через эти порты клиенты получают только ту информацию, которая предлагается компанией. Чем больше будет открытых портов, тем больше будет у нарушителей способов проникнуть в сеть.

Брешь в RPC

Многие сетевые программисты признают, что с увеличением числа портов возрастает риск незаконного проникновения. Но неопытные программисты не всегда понимают, что программы, которыми они пользуются, могут создавать порты динамически. Например, программа portmap в RPC создает порты. А на технологии RPC основаны такие технологии, как Java RMI и Microsoft COM/COM+. Поэтому Web-мастера, которые пишут серверные сценарии, иногда не догадываются, что с каждым новым объектом сценария появляется новая брешь в системе защиты сервера.

- *Минимизируйте число сервисов* (предоставляемых серверами). Самые близкие к Internet серверы должны предлагать абсолютный минимум сервисов. Обычно эти сервисы тесно связаны с портами, по которым брандмауэр осуществляет пассивную фильтрацию. В качестве примера можно привести серверы HTTP, HTTPS и FTP (только чтение).
- *Ограничивайте число баз данных* (внутри ДМЗ). Базы данных часто содержат самую ценную информацию компании. Ни в одной из демилитаризованных зон не должно быть баз данных с закрытой информацией. В некоторых компаниях в одной из ДМЗ организуется служба каталогов, а не внутренняя TCP-сеть, но такой поход опасен, так как база каталогов часто содержит незашифрованные имена пользователей и пароли. Если серверам внутри ДМЗ требуется доступ к базе данных, Можно легко создать модули запросов, реализующие доступ к внешнему серверу и обратно во внутреннюю сеть. Эти модули должны быть скомпилированными, что уменьшит вероятность их взлома.
- *Используйте скомпилированные сервисные программы* (предоставляемые серверами). Конечно, сценарии проще писать на Perl, чем на C/C++, но они имеют понятную структуру, что делает их потенциально небезопасными. Старайтесь создавать такие программы, которые не дадут возможности нарушителю изменить поведение Сервера, если вторжение все • же произойдет.
- *Разграничивайте области трансляции адресов.* В каждой ДМЗ должны использоваться собственное пространство фиктивных IP-адресов и свои средства трансляции. Смена формы адресации при переходе от одной зоны к другой делает проникновение во внутреннюю сеть более затруднительным, хотя и усложняет управление сетью.

Сегодняшнее положение дел в Internet напоминает ситуацию со средневековыми городами и замками, которые могли ожидать нападений отовсюду. В Internet нет правил, и в сети часто царит анархия. Обеспечение должных мер безопасности позволяет защитить не только сервер, но и его реальных клиентов.

Защита канала

В сети есть два различных компонента, которые требуют пользовательской настройки: физический канал и сообщение. Защита каждого из них от нападений очень важна:

Шпионаж на аппаратном уровне

Очевидным физическим ограничителем является выделенный канал. К сожалению, он далеко не так безопасен, как некоторые думают. Даже для выделенных

линий не сложно реализовать методику непроникающего подслушивания. Тем не менее это хорошая стартовая площадка. Существуют разные способы физической передачи данных. Мы рассмотрим четыре основных: электрический/оптический и проводниковый/непроводниковый.

Электрическую проводниковую линию (витую пару или коаксиальный кабель) прослушать проще всего. Гибкие проводники не защищены от подключения пробника, а особенности электромагнетизма не позволяют обнаружить прослушивающее устройство. Негибкие проводники более защищены и менее подвержены потерям сигнала, но они дороже и с ними труднее работать.

Непроводниковые электрические (радио- и микроволны) и оптические (инфракрасное и направленное лазерное излучение) каналы вообще не защищены — достаточно поставить антенну-перехватчик в районе или на пути распространения сигнала. При низкой частоте сигнал передается широковещательно, и меры безопасности здесь минимальны.

Наиболее защищены оптические проводниковые линии. Однако несколько лет назад исследователи обнаружили, что при сгибе проводника под определенным (критическим) углом прриходит утечка светового сигнала. Тем самым было доказано, что даже оптоволокно подвержено прослушиванию. Чтобы решить эту проблему, используйте жесткие оптические кабели и не позволяйте сгибать их под критическим углом.

Подводя итог, можно отметить, что негибкие проводники меньше подвержены прослушиванию. Среди них оптические проводники самые надежные, кроме того, у них очень широкая полоса пропускания.

Перехват сообщений

Потребность в защите сообщений не столь очевидна, ведь брандмауэры фильтруют всю информацию, поступающую из сети и уходящую в нее. Но они используют конкретные порты и сервисы, заранее .известные клиенту. Изменение стандартных установок и интерфейсов позволяет повысить безопасность.

Если вы не являетесь приверженцем открытого программного обеспечения, то лучше использовать закрытые или патентованные протоколы, так как это затрудняет вторжение (хотя и привлекает внимание). Например, можно вместо традиционного протокола HTTP придумать новую, специализированную его разновидность. Но у такого подхода есть существенный недостаток: нужно написать пользовательский интерфейс и распространить его среди своих клиентов.

Другой подход заключается в лимитировании промежутков времени, когда может происходить взаимодействие с сервером и передача информации. Например, во время второй мировой войны союзники наполняли эфир разговорами североамериканских индейцев, а в заданные интервалы передавали сообщения тактического или стратегического характера. Аналогичным образом можно создать систему денежных переводов, которая функционирует постоянно, передавая несущественную информацию и лишь в заданный момент включая трансферный механизм.

Высокий процент случайности — вот ключ к безопасности сообщений. Предположим, имеется банкомат, подключенный.к Internet (ужасная мысль!). Он должен посылать главному серверу запросы на осуществление транзакций и принимать от него ответы. Эффекта случайности можно достичь, если постоянно генерировать ложные транзакции. А в нужный момент сервер и банкомат распознают,

что транзакция настоящая. (Конечно, это слишком примитивный алгоритм, чтобы его можно было реализовать.)

Самый, действенный способ скрыть реальное сообщение в потоке случайных данных — применить шифрование. Чем сильнее алгоритм шифрования, тем выше процент случайных данных и тем труднее дешифровать сообщение. Диапазон возможных вариантов определяется *ключом* шифрования. Идеальный ключ имеет бесконечный размер. Но с увеличением длины ключа сложность вычислений возрастает экспоненциально. Поэтому хороший ключ — тот, который достаточно длинен, чтобы свести к минимуму вероятность нахождения исходного смысла сообщения путем последовательного перебора вариантов. Как правило, применяются 128-разрядные ключи.

Шифрование и фильтрующие брандмауэры

Некоторые технояогии нельзя свести вместе, чтобы обсспечть более мощную защиту. Например, применение шифрования делает бесполезной акттивную фильтрацию, осуществляемую, брандмауэром. Напомню, что при активной фильтрации брендмауэр просматривает каждый пакет, проверяя, нет ли в нем подозрительных команд. Но если данные зашифрованы, то лишь клиент, и сервер знают об их назначении. Выполнять активную филтрацию в такои случае нет необходимости.

Ключ длиной 128 битов настолько велик, что если бы компьютер пытался каждую наносекунду (10^{-9}) применить один из вариантов ключа, на полный перебор ему понадобилось бы 10^{20} лет. В настоящее время этого вполне достаточно, чтобы обеспечить стойкость алгоритма.

Шифрование бывает двух типов: одностороннее (с потерей данных) и двустороннее (без потерь). В первом случае часть информации об исходных данных теряется. Например, в UNIX пароли шифруются односторонне. Такого рода шифрование не предусматривает восстановления данных. Когда пользователь вводит пароль, важно, чтобы с образцом совпала его зашифрованная форма.

Запросы, посылаемые серверу, предполагают получение результатов. Возвращаемые данные засекречиваются таким образом, чтобы восстановить их Можно было только с помощью ключа дешифрования. Клиент и сервер должны знать алгоритмы шифрования, применяемые противоположной стороной, а также пару ключей шифрования/дешифрования.

Шифрование сообщений

Концепция шифрования не нова. Самым знаменитым шифровальным устройством была немецкая машина Enigma ("загадка"). От нее ведут свое начало многие современные криптографические алгоритмы. Идея шифрования заключается в том, чтобы только истинный получатель сообщения мог понять его смысл.

При сетевом взаимодействии подразумевается, что на противоположной стороне сообщения перехватываются. Подслушивающее устройство способно распознать и проанализировать пакет каждого протокола, рассмотренного в настоящей книге. Конечно, чтобы перехватывать все сообщения, шпион должен физически находиться в той же самой сети. Кроме того, ему придется отсеять много "мусора", чтобы добраться до действительно ценных сообщений.

Виды шифрования

В сетях применяются два различных алгоритма шифрования: с открытым и симметричным ключом. В первом алгоритме используются два ключа: один — для шифрования (открытый), а другой — для дешифрования (секретный). Во втором алгоритме для обеих целей применяется один и тот же ключ. При дешифровании выполняется та же последовательность действий, что и в случае шифрования, но в обратном порядке. Естественно, обе стороны должны хранить ключ в тайне. Сам алгоритм обычно достаточно прост и быстр, а в качестве ключа может использоваться случайное число.

В алгоритме шифрования с открытым ключом, как уже говорилось, ключ шифрования является открытым, а дешифрования — секретным. Серверы передают открытый ключ любому клиенту по сети. Секретный ключ хранится на сервере и используется для расшифровки поступающих сообщений. (Открытый ключ не представляет особого интереса для шпиона, так как данные можно расшифровать только с помощью секретного ключа.)

Шифрование с открытым ключом имеет ряд ограничений. Во-первых, ключи должны быть связаны друг с другом. Сервер не может случайным образом сгенерировать ключ шифрования, не получив также ключ дешифрования. Это резко ограничивает число возможных ключей. Например, в случае 128-разрядного шифра имеется лишь 2^{32} пар ключей. Во-вторых, из-за парности ключей стойкость шифра снижается. Скажем, 128-разрядный алгоритм шифрования с открытым ключом имеет такую же стойкость, что и 64- или 32-разрядный симметричный шифр.

Опубликованные алгоритмы шифрования

В Internet имеются различные системы симметричного шифрования и шифрования с открытым ключом. Среди них наиболее известны RSA (названа по фамилиям авторов: Ривест (Rivest), Шамир (Shamir) и Адельман (Adelman); относится к алгоритмам с открытым ключом), DES (Data Encryption Standard — стандарт шифрования данных) и RC2/RC4 (шифры Ривеста), Некоторые алгоритмы до сих пор являются патентованными и за их использование надо платить (патент на RSA истек в октябре 2000 года).

В большинство дистрибутивов Linux либо входят пакеты шифрования, либо предоставляется ссылка на Web-сервер компании-разработчика.

Проблемы с шифрованием

С алгоритмами шифрования связан интересный круг проблем. Во-первых, до недавнего времени в США существовал запрет на экспорт "сильных" алгоритмов шифрования. Это одна ИЗ) причин, почему в некоторые дистрибутивы не входят криптографические пакеты. В процессе инсталляции система пытается запросить пакеты у зарубежного сервера. В 2000 г.. правительство сняло подобные ограничения, поэтому получать шифры стало проще.

Другая проблема заключается в скорости вычислений. Хотя симметричные алгоритмы достаточно быстры, начальные данные в процессе сетевого взаимодействия шифруются по более медленному алгоритму с открытым ключом. Применять его для крупных блоков данных нецелесообразно.

Последняя проблема может показаться необычной. Шифрование в действительности вызывает то, что оно по своей сути должно устранять: взломы и шпионаж. Очевидно, в Internet есть лига хакеров, которым приятно быть героями новостей.

ПротоколSSL

Как уже упоминалось, два компьютера должны договориться о том, какие шифры они будут использовать для безопасного общения. Этой цели служит SSL (Secure Sockets Layer — протокол защищенных сокетов). Он позволяет существенно снизить вероятность взлома. В SSL применяются как алгоритмы с открытым ключом, так и симметричные шифры. Протокол регулирует процесс установления соединения, обмена ключами и собственно передачи данных.

Вопросы применения SSL для установления соединения и обмена ключами выходят за рамки нашей книги. Ниже речь пойдет о том, как написать SSL-клиент и сервер средствами библиотеки OpenSSL. Возможности этой библиотеки также раскрываются не в полной мере, так как на это ушла бы целая книга.

Библиотека OpenSSL

Полнофункциональная, хотя и недостаточно хорошо документированная версия библиотеки SSL-функций называется OpenSSL. Она доступна по адресу www.openssl.org и ориентирована на несколько платформ, включая, конечно же, Linux. Прежде чем начать работать с библиотекой, необходимо предпринять ряд действий по ее конфигурированию, компилированию и инсталляции (поскольку библиотека не поставляется в скомпилированном виде, некоторые из перечисленных ниже этапов могут не работать в Mandrake и Red Hat Linux; схожие проблемы, могут возникнуть и в других дистрибутивах).

1. Загрузите tar-архив, откройте его в безопасном (не корневом) каталоге и перейдите в созданный каталог.
2. Запустите сценарий config (./config). Если он выдает сообщения об ошибках, необходимо явно указать тип операционной системы, например ./config linux-elf.
3. Запустите утилиту make, чтобы скомпилировать исходные файлы.
4. Выполните команду make test, чтобы проверить результат.
5. Войдите в систему как пользователь root. Выполните команду make install, чтобы переместить файлы в нужные каталоги (/usr/local/ssl).
6. Создайте символические ссылки на библиотеки:

```
ln -s /usr/local/ssl/lib/libssl.a/usr/lib/
ln -s /usr/local/ssl/lib/libcrypto.a/usr/lib/
```

7. Создайте ссылку на включаемые файлы:

```
ln -s /usr/local/ssl/include/openssl/usr/include
```

8. В файле /etc/man.config добавьте в переменную среды MANPATH путевое имя /usr/local/ssl/man, ссылающееся на каталог документации к библиотеке (возможно, потребуется запустить утилиту makewhatis).

9. Добавьте в переменную среды PATH путевое имя /usr/local/ssl/bin.

В процессе компиляции будут использоваться статические библиотеки, поэтому не удивляйтесь, если размер исполняемых файлов превысит 600 Кбайт. Чтобы сделать библиотеки совместно используемыми, следует поменять у файлов libssl.a и libcrypto.a расширение: вместо .a — .so.

После завершения всех этапов можно приступать к созданию защищенных сокетов. На этапе компоновки нужно подключать библиотеки в определенном порядке (компоновщик не сможет разрешить внешние ссылки, если библиотеки поменять местами):

```
cc test.c -lssl -lcrypto
```

Инсталлировав пакет, можно заметить, что некоторые демонстрационные файлы написаны на C++. Это крайность. Все вызовы библиотечных функций успешно реализуются средствами C. (Вообще-то, если заглянуть в сами файлы, то окажется, что в них на самом деле содержится C-код!)

Создание SSL-клиента

Итак, давайте попробуем написать связку клиент/сервер. Создать SSL-клиент и SSL-сервер так же просто, как и в случае обычных сокетов, так как библиотека OpenSSL основана на вызовах рассматривавшихся в предыдущих главах функций сокетов.

Первый шаг заключается в инициализации библиотеки OpenSSL:

```
/*************************************************************/
/***                  Инициализация клиента                ***/
/*************************************************************/
SSL_METHOD «method;
SSL_CTX *ctx;
OpenSSL_add_all_algoritms();     /* загружаем модули шифрования */
SSL_load_error_strings(); /* загружаем и регистрируем таблицы
                                  сообщений tfb ошибках */
method = SSLv2_client_method();    /* создаем новый клиентский
                                                  метод */
ctx = SSL_CTX_new(method);            ./* создаем новый контекст */
```

Если функции возвращают значение NULL или 0, следует вывести сообщение об ошибке:

```
ERR_print_errors_fp(stderr);       /* записываем сообщения об
                                     ошибках в поток stderr */
```

Далее создается традиционный сокет:

```
/***      Подключение клиентского сокета к SSL-серверу        ***/

struct sockaddr_in addr;
```

```
struct hostent *host = gethostbyname(hostname);
int sd = socket(PF_INET, SOCK_STREAM, 0);       /* создаем сокет */
bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(port);                    /* серверный порт */
addr.sin_addr.s_addr = *(long*)(host->h_addr); /* адрес сервера */
connect(sd, &addr, sizeof(addr));       /* подключаемся к серверу */
```

После установления соединения между клиентом и сервером необходимо создать экземпляр объекта SSL и связать его с соединением:

```
y*********************************************************/
/***        Инициализируем протокол SSL и создаем          ***/
/***          зашифрованный канал                          ***/
/*********************************************************/
SSL *ssl = SSL_new(ctx);  /* создаем новое SSL-соединение */
SSL_set_fd(ssl,  sd);       /* подключаем дескриптор сокета */
if ( SSL_connect(ssl) == -1 ) /* устанавливаем соединение */
    ERR_print_errors_fp(stderr);     /* сообщаем об ошибках */
```

С этого момента имеется полностью зашифрованное SSL-соединение. Получить набор шифров можно следующим образом:

```
char* cipher_name = SSL_get_cipher(ssl);
```

Можно также получить цифровые сертификаты:

```
/*************************************************************/
/**                    Получение сертификатов                 ***/
/*************************************************************/
char line[1024];
X509 *x509 = X509_get_subject_name(cert);     /* имя владельца */
X509_NAME_oneline(x509, line, sizeof(line)); /* преобразование */
printf("Subject: %s\n", line);
x509 = X509_get_issuer_name(cert);               /* кем выдан */
X509_NAME_oneline(x509, line, sizeof(line)); /* преобразование */
printf("Issuer: %s\n", line);
```

Наконец, программы могут начать обмениваться данными с помощью функций, напоминающих вызовы send() и recv(). Но есть несколько отличий. Во-первых, параметр flags отсутствует; во-вторых, в случае ошибки возвращается —1. В определении функций send() и recv() сказано, что при неудачном завершении они возвращают отрицательное значение.

```
/*************************************************************/
/***                  Прием и отправка сообщений

int bytes;
bytes = SSL_read(ssl, buf, sizeof(buf));  /* получаем/дешифруем */
bytes = SSL_write(ssl, msg, strlen(msg)); /* шифруем/отправляем */
```

В библиотеке есть множество других функций для управления потоком, изменения состояния соединения и конфигурирования сокетов.

Создание SSL-сервера

Код клиента и сервера отличается лишь незначительно. В обоих случаях инициализируется библиотека, устанавливается соединение и создается объект SSL. Но на сервере нужно выполнить несколько дополнительных действий. Прежде всего, немного отличается процедура инициализации:

```
/***                    Инициализация сервера                    ***/
/*************************************************************/
SSL_METHOD «method;
SSL_CTX  *ctx;
OpenSSL_add_all_algoritms();    /* загружаем модули шифрования */
SSL_load_error_strings();  /*  загружаем  и  регистрируем  таблицы
                                          сообщений об ошибках */
method = SSLv2_server_method();    /* создаем новый серверный
                                                      метод */
ctx = SSL_CTX_new{method);            /* создаем новый контекст */
```

Читатели, должно быть, обратили внимание на то, что используется протокол SSL2, а не SSL3. Это необходимо, если подключение к серверу осуществляется через броузер Netscape.

В отличие от клиента, сервер должен получить свой сертификат. Это осуществляется в два этапа: сначала загружается файл сертификата, а затем — файл секретного ключа. Оба должны быть загружены в процессе инициализации. Как правило, и сертификат, и секретный ключ размещаются в одном и том же файле:

```
/***     Загружаем файлы сертификата и секретного ключа     ***/
/*************************************************************/
/* загружаем сертификат из файла */
SSL_CTX_use_certificate_file(ctx, CertFile, SSL_FILETYPE_PEM);
/* загружаем секретный ключ из файла */
SSL_CTX_use_PrivateKey_file(ctx, KeyFile, SSL_FILETYPE_PEM);
/* проверяем секретный ключ */
if  (  !SSL_CTX_check_private_key(ctx)  )
    fprintf(stderr, "Key & certificate don't match");
```

Клиент также может загрузить файл сертификатов, но это не обязательно и зависит от используемой схемы безопасности.

Примечание

Как правило, чтобы иметь возможность распространять программное обеспечение в Internet, нужно купить сертификат у одного из центров сертификации, например фирмы VeriSign. Если же сертификаты нужны для целей отладки, их можно создать средствами библиотеки OpenSSL (не используйте сертификаты, поставляемые с библиотекой). Написанный на Perl сценарий CA.pl находится в каталоге /usr/local/ssl/misc, создаваемом при инсталляции библиотеки. Если сценарий не задает серию вопросов, значит, в переменной среды PATH не записан путь к каталогу /usr/local/ssl/bin.

Серверный сокет создается так же, как и клиентский:

```
/***          Конфигурируем серверный порт сокета          ***/

struct sockaddr_in addr;
int sd, client;
sd = socket(PP_INET, SOCK_STREAM, 0);       /* создаем сокет */
bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(port);
addr.sin_addr.s_addr = INADDR_ANY /* разрешаем любое сетевое
                                                семейство */
bind(sd, &addr, sizeof(addr));       /* подключаемся к порту */
listen(sd, 10);              /* переходим в режим ожидания */
client - accept(server, &addr, &len); /* принимаем запрос */
```

Как и в случае клиента, сервер должен создать объект SSL и связать его с клиентским соединением:

```
/***                  Создание SSL-сеанса                  ***/

ssl = SSL_new(ctx);             /* создаем новое SSL-соединение */
SSL_set_fd(ssl, client);       /* связываем сокет с соединением */
if ( SSL_accept(ssl) == FAIL ) /* принимаем запрос по протоколу
                                                        SSL */
    ERR_print_errors_fp(stderr);
else
{  int bytes;
   bytes = SSL_read(ssl, buf, sizeof(buf));       /* принимаем
                                                   сообщение */
   SSL_write(ssl, reply, strlen(reply));       /* посылаем ответ */
}
```

Описанные этапы действительно просты. Загрузите программу с нашего Web-узла и попытайтесь подключиться к броузеру Netscape. Поскольку он не сможет распознать сделанный вами сертификат, появится предупреждающее сообщение и будет выдано несколько диалоговых окон для временной регистрации сертификата.

Резюме: безопасный сервер

Сокеты позволяют подключаться к самым разным компьютерам и создавать распределенные алгоритмы. Однако информация, которая передается между компьютерами, не скрыта от других компьютеров, подключенных к той же сети. Большинство компаний вынуждено представлять себя в Internet, так как это диктуется требованиями бизнеса. Но размещение конфиденциальной информации в Internet ведет к шпионажу, краже данных и их повреждению. Ответственность за это лежит на самой сети Internet.

Потери можно свести к минимуму, если заранее планировать обеспечение безопасности своих программных продуктов. Брандмауэры, системные политики

и шифрование — вот те средства, которые позволяют защитить Web-сервер от нападений.

Когда два или несколько компьютеров взаимодействуют в открытой среде, такой как Internet, необходимы определенные протоколы и алгоритмы, уменьшающие вероятность шпионажа. Шифрование бывает различных видов и обеспечивает разный уровень безопасности. Два стандартных варианта шифрования — с открытым и с симметричным ключом. Оба этих метода применяются в современных коммуникационных протоколах, в частности в SSL.

Существуют разные реализации функций SSL. На момент написания книги не было стандартной библиотеки таких функций. Но для языка C имеется открытая библиотека OpenSSL, которая обладает множеством возможностей и содержит более 200 функций. Применяя тщательное планирование и стандартные средства наподобие OpenSSL, можно создавать эффективные, надежные и безопасные программы, работающие в среде Internet.

В следующей главе мы познакомимся с другой стороной совместного использования данных: режимами широковещания и группового вещания. Технологии безопасности призваны обеспечить конфиденциальность данных, а режимы широковещания и группового вещания, наоборот, предназначены для доставки данных как можно большему числу заинтересованных лиц.

Глава 17

Широковещательная, групповая и магистральная передача сообщений

В этой главе...

Сеть — это среда передачи сообщений из одного места в другое. В сети применяется множество различных технологий и имеются каналы с разной пропускной способностью. Но обычно недооценивают тот факт, что все компьютеры подключены к магистральной, или опорной, сети определенного типа. Как извлечь из этого преимущество?

Некоторые сообщения предназначаются нескольким адресатам одновременно. Вместо того чтобы отправлять одно и то же сообщение многократно, можно послать одно сообщение, но заставить все заинтересованные компьютеры принять его. Тем самым сетевой трафик не будет "засоряться".

В протоколе IP предусмотрены два способа распределенной доставки сообщений: широковещание и групповое вещание. В этой главе мы рассмотрим, как реализуются оба способа.

Широковещание в пределах домена

Первый режим распределенной доставки сообщений — широковещание [RFC919, RFC922] — основан на особенностях подсетей и применяющихся в них масок. Это принудительная форма доставки: все компьютеры в подсети должны получить сообщение. (На самом деле, если компьютер не поддерживает широковещательный режим, сетевая плата откажется принять сообщение. Тем не менее сообщение занимает канал, и любой узел при желании сможет его прочитать.)

Пересмотр структуры IP-адреса

Наличие подсети очень важно с точки зрения отправки и приема широковещательных сообщений. Как описывалось в главе 2, "Основы TCP/IP", когда с помощью команды ifconfig конфигурируется сетевой интерфейс, задается также сетевая маска и широковещательный адрес. Последний — это специальный адрес, по которому компьютер ожидает поступления сообщений.

Границы подсети определяются сетевой маской и широковещательным адресом. Например, если есть подсеть 198.2.56.*XXX,* в которую входят 250 узлов, то маской будет адрес 198.2.56.0, а широковещательный адрес будет таким: 198.2.56.255. (В главе 2, "Основы TCP/IP", упоминалось о том, что маска подсети, в принципе, может быть произвольной. Просто помните: последний значащий бит задает границу маски.)

Фактическая реализация широковещательного режима происходит на низком уровне: в аппаратной части и в ядре. Чтобы понять весь процесс, необходимо начать с организации физического соединения.

Послать широковещательное сообщение легче, чем принять его. Чтобы отправить сообщение, достаточно указать IP-адрес получателя (сетевая подсистема впоследствии преобразует его в MAC-адрес Ethernet-платы принимающей стороны). А чтобы получить сообщение, сетевая плата должна прослушивать сеть, выявляя сообщения с заданным MAC-адресом. Проблема заключается в том, что у широковещательного сообщения один адрес, а в подсети все компьютеры имеют разные MAC-адреса.

В результате было решено, что, когда программа посылает широковещательное сообщение, ядро автоматически назначает ему MAC-адрес, состоящий из всех единиц (FF:FF:FF:FF:FF:FF). Он служит сигналом для всех сетевых плат принять сообщение, даже если на конкретном компьютере нет программ, ожидающих широковещательных сообщений.

Сетевая плата реагирует на сообщение, помещая его во внутренние буферы. По завершении операции плата уведомляет ядро, посылая ему запрос на прерывание. Ядро считывает пакет и проверяет IP-адрес получателя. Если .он оказывается широковещательным адресом, ядро записывает пакет в очередь сетевой подсистемы.

Сетевая подсистема (обычно это UDP) анализирует сообщение и, если находит широковещательный сокет с совпадающим номером порта, перемещает пакет в канал ввода-вывода этого сокета. В противном случае пакет удаляется. Лучше указывать номер порта, чтобы ядро автоматически отфильтровывало ненужные сообщения. Иначе можно "захлебнуться" в потоке ненужных ответных сообщений.

Работа в широковещательном режиме

Включить широковещательный режим можно с помощью параметра сокета SO_BROADCAST. В остальном все остается прежним: программа создает обычный дейтаграммный сокет.

```
/***    Создание широковещательного дейтаграммного сокета    ***/

const int on=1;
sd = socket(PF_INET, SOCK_DGRAM, 0);
if ( setsockopt(sd, SOL SOCKET, SO_BROADCAST, Son, sizeof(on))
     1= 0 )
     panic("set broadcast failed");
bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin~port  =  htons(port);
addr.sin~addr.s_addr  =  INADDR_ANY;
if ( bind(sd, Saddr, sizeof(addr)) != 0 )
     panic)"bind  failed");
addr.sin_port = htons(atoin(strings[2]));
if ( inet_aton(strings[1], Saddr.sin_addr) == 0 )
     panic("inet_aton failed (%s)", strings[l]);
```

Активизировав широковещательный режим, можно отправлять сообщения по широковещательному адресу. Обычно создается отдельный процесс или поток для посылающей и принимающей сторон. В отличие от других соединений, где на одно сообщение приходит один ответ, в широковещании на каждое посланное сообщение может прийти несколько ответов. Поэтому в одном из каналов необходимо отключить входную очередь:

```
/*** Разделение обязанностей при отправке и приеме сообщений. ***/
/*** На отправляющей стороне входной канал закрывается.       ***/

if ( fork() )
     Receiver(sd);
else
{
     shutdown(sd, SHUT RD); /* закрываем входной канал */
     Sender(sd);
}
wait(0);
```

Ограничения широковещательного режима

Широковещательные сообщения имеют свои ограничения. В первую очередь следует отметить, что сообщение доставляется тольЦо компьютерам, входящим в подсеть. Правда, можно увеличить диапазон доставки, изменив широковещательный адрес. Но широковещание в глобальной сети невозможно, так как адрес 255.255.255.255 больше недопустим в Internet. (Кроме того, большинство маршрутизаторов не позволяет широковещательным сообщениям проходить через них. Поэтому, даже если сообщению присвоен правильный широковещательный адрес, к примеру 198.2.255.255, маршрутизатор, находящийся по адресу 198.2.1.0, может проигнорировать сообщение.)

Широковещательное сообщение доставляется всем компьютерам подсети. На аппаратном уровне широковещание реализовано так, что все сетевые платы будут получать сообщения. Это создает дополнительные неудобства тем компьютерам, которым такие сообщения в действительности не нужны.

Еще одной проблемой является протокол. В IPv4 широковещательная передача поддерживается только для дейтаграмм. Если необходима надежная потоковая доставка широковещательных сообщений, ее придется реализовать самостоятельно.

Передача сообщения группе адресатов

Многоадресный режим [RFC1112] решает многие проблемы широковещательного режима. Идея широковещания заключается в доставке сообщения всем компьютерам в текущем диапазоне адресов. В многоадресном режиме устанавливаются определенные групповые адреса, к которым подключаются компьютеры, желающие получать групповые сообщения.

Многоадресный режим имеет следующие преимущества перед широковещанием.

- *Поддержка всех необходимых протоколов.* Можно осуществлять групповую доставку как дейтаграмм, так и TCP-пакетов. Что касается TCP, то такая поддержка еще не встроена в Linux, но можно найти свободно распространяемые библиотеки соответствующих функций.
- *Возможность группового вещания в глобальной сети.* Можно подключаться к глобальным адресным группам. Правда, еще существуют зоны Internet, где не допускается групповая доставка сообщений.
- *Ограниченное число слушателей.* Групповое вещание не обязывает все сетевые платы принимать сообщения (подробнее об этом — в разделе "Как реализуется многоадресный режим в сети").
- *Поддержка IPv6.* В стандарте IPv6 широковещание не поддерживается вообще, а функции многоадресной доставки приняты и даже расширены.

Включение поддержки многоадресного режима

В большинстве дистрибутивов Linux имеется ядро, в котором активизирован многоадресный режим (загляните в каталог /proc/net/dev_mcast, чтобы узнать, какое ядро выполняется в настоящий момент), но может быть не включена многоадресная маршрутизация либо, если ядро загружалось по сети, есть вероятность, что многоадресный режим отключен по умолчанию, В подобных случаях нужно переконфигурировать и повторно скомпилировать ядро.

Подключение к группе адресатов

Работать в многоадресном режиме так же просто, как подключаться к списку рассылки. Программа, присоединившаяся к адресной группе, будет получать все сообщения, публикуемые ее членами. Поэтому следует убедиться, что существующий сетевой канал сможет справиться с потоком сообщений.

В главе 2, "Основы TCP/IP", говорилось о том, что в пространстве возможных IP-адресов выделен диапазон адресов, зарезервированных для группового вещания: 224.0.0.0-239.255.255.255. Этот диапазон, в свою очередь, подразделяется на более мелкие диапазоны, определяющие *область видимости адреса,* т.е. как далеко сможет дойти сообщение, прежде чем маршрутизатор блокирует его (табл. 17.1).

Таблица 17.1. Диапазоны и области видимости групповых адресов

Область видимости	Число переходов (TTL)	Диапазон адресов
Кластер	0	224.0.0.0-224.0.0.255
Сервер	< 32	239.255.0.0-239.255.255.255
Организация	< 128	239.192.0.0-239.195.255.255
Глобальная сеть	<= 255	224.0.1.0-238.255.25.255

Чтобы подключиться к группе, необходимо вызвать функцию setsockopt(), передав ей в качестве параметра структуру ipjnreq:

```
/************************************************************/
/***     Структура ipjnreq для задания группового адреса

struct ip_mreq
{
    struct in_addr imrjnultiaddr; /* известная адресная группа */
    struct in_addr imr_interface; /* сетевой интерфейс */
};
```

Поле imrjnultiaddr задает адресную группу, к которой необходимо присоединиться. Формат его такой же, как и у поля sin_addr структуры sockaddr_in. Поле imr_interface позволяет выбрать конкретный сетевой интерфейс узла. Это напоминает функцию bind(), которая делает то же самое, а если в качестве адреса указать константу INADDR_ANY, то сообщения будут приниматься через любой доступный интерфейс. Однако в многоадресном режиме эта константа имеет несколько иной смысл.

Стандартный интерфейс группового вещания

Если в поле imr_interface присутствует константа INADDR_ANY, ядро самостоятельно выберет сетевой интерфейс. По крайней мере, в ядре Linux версий ***2.2.xx*** эта константа не означает "прослушивать все сетевые интерфейсы". Поэтому, если в системе есть несколько сетевых устройств, подключаться к группе необходимо по каждому устройству в отдельности.

Следующий фрагмент программы иллюстрирует, как использовать структуру ipjnreq для подключения к группе. Поле imr_interface задано равным INADD_ANY исключительно в демонстрационных целях. Так можно делать только в том слу-

чае, когда в системе имеется один-единственный сетевой интерфейс, иначе результаты будут непредсказуемыми.

```
/*************************************************/
/*'**          Подключение к адресной группе          ***/
/*************************************************/
const char *GroupID = "224.0.0.10";
struct ip_mreq mreq;
if ( inet_aton(GroupID,&mreq.imr_multiaddr)== 0 )
    panic("address (%s) bad", GroupID);
mreq.imr interface.s_addr = INADDR_ANY;
if ( setsockopt(sd, SOL_IP, IP_ADD_MEMBERSHIP, &mreq,
                sizeof(mreq)) != 0 )
    panic("Join multicast failed");
```

Если предполагается подключение другой программы по тому же самому адресу, необходимо установить параметр сокета SO_REUSEADDR. В настоящее время в Linux не поддерживается совместное использование портов (SO_REUSEPORT).

```
/*************************************************************/
/***  Включение режима совместного использования адресов   ***/
/*************************************************************/
if ( setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on))
    != 0 )
    panic("Can't reuse address/ports");
```

Число групп, к которым можно подключаться, зависит от аппаратных ограничений и настроек ядра. Во многих версиях UNIX допускается 16 подключений от одного компьютера. Да, именно от компьютера, а не от программы. Дополнительная информация по этому поводу приведена ниже, в разделе "Как реализуется многоадресный режим в сети".

Когда прослушивание порта закончено, нужно выйти из состава группы:

```
/*************************************************************/
/***              Выход из состава адресной группы           ***/
/*************************************************************/
if ( setsockopt(sd, SOL_IP, IP_DROP_MEMBERSHIP, &mreq,
                sizeof(mreq)) != 0 )
    panic("Drop multicast failed");
```

Можно просто выйти из программы, и система автоматически разорвет все соединения.

Очистка групповых соединений

Многоадресный режим — один из тех немногих случаев, когда лучше вручную производить необходимую очистку по окончании работы. Если выйти из программы, а ядро столкнется с какими-то проблемами, сообщений об ошибках не будет. Кроме того, даже после завершения программы можно продолжать получать сообщения. Это сделает порт недоступным для новых экземпляров программы или же они будут получать посторонние сообщения.

Подключиться к многоадресной группе достаточно легко. По сути, для создания локальной группы достаточно выбрать неиспользуемый IP-адрес. Иное дело глобальные адреса. Чтобы опубликовать сервис группового вещания в Internet, необходимо запросить адрес и порт у организации IAB (Internet Architecture Board — архитектурный совет сети Internet).

Отправка многоадресного сообщения

Послать сообщение группе адресатов так же легко, как и отправить дейтаграмму. Для UDP-подсистемы достаточно указать адрес, а все остальное осуществляется автоматически.

Принадлежность групповых адресов

Хотя для того или иного сетевого сервиса можно назначить адрес, его нельзя закрепить за данным сервисом. Другие программы тоже могут передавать сообщения по этому адресу. Клиент будет получать все сообщения, посылаемые ему по сети. Ответственность за сортировку сообщений ложится на ядро и саму программу.

У многоадресного режима есть интересный "побочный эффект": не нужно быть членом группы, чтобы посылать ей сообщения. Достаточно отправить сообщение по определенному адресу/порту. Сервер, осуществляющий потоковую многоадресную доставку пакетов, вообще никогда не подключается к группе. Но для приема многоадресных сообщений нужно быть членом группы.

Возможность отправлять многоадресные сообщения, не будучи членом группы, очень полезна. Тем самым сервер избавляется от ненужного ему трафика, связанного с потоком групповых пакетов.

Как реализуется многоадресный режим в сети

В операционных системах (и в частности, в Linux) многоадресный режим реализован таким образом, что программа, подключающаяся к группе, может легко "захлебнуться" в потоке сообщений. Важно знать, как организуется работа в этом режиме, чтобы обеспечить должную производительность и быстро фильтровать сообщения.

Многие сетевые платы поддерживают замещение MAC-адресов (принятие дополнительного идентификатора на короткое время). MAC-адреса есть у всех сетевых плат. В отличие от широковещательного режима, где используется фиксированный широковещательный адрес (FF:FF:FF:FF:FF:FF), при групповом вещании создается временный MAC-адрес, имеющий стандартный префикс 01:00:5E, а последние три байта (в действительности 23 бита) берутся из группового адреса. Например, групповой адрес 224.138.63.10 (EO:8A:3F:OA) преобразуется в MAC-адрес 01:00:5E:OA:3F:OA (обратите внимание на то, что цифра 8 во фрагменте 8A исчезла).

Операционная система программирует сетевую плату таким образом, чтобы она воспринимала данный адрес. Это можно сделать тремя способами.

- *Таблица адресов.* Сетевая плата хранит все адреса в полном виде. Каждое сообщение проверяется на предмет наличия собственного MAC-адреса платы, а также всех адресов в таблице. Это стопроцентно надежный способ.
- *Хеш-коды (неполная фильтрация).* Сетевая плата хранит хеш-коды MAC-адресов в 64- или 512-битовом массиве. Когда сообщение поступает в сеть, плата преобразует его MAC-адрес в хеш-код. Если соответствующий бит в массиве установлен, плата принимает сообщение. При таком способе могут быть приняты сообщения, не предназначенные для данного компьютера. За фильтрацию нежелательных сообщений отвечают программные средства более высокого уровня.

- *Беспорядочный режим*. Некоторые сетевые платы не поддерживают многоадресный режим. Выход заключается в том, чтобы перевести плату в беспорядочный режим, в котором она принимает все сообщения. Нежелательные сообщения фильтруются программным путем. Это наименее удобный способ.

Сетевые платы не знают о существовании портов, поэтому на верхних уровнях стека TCP/IP сообщения должны помещаться в соответствующие очереди или удаляться. Помните, что, даже если программа не прослушивает ни один из портов, компьютер все равно будет принимать все сообщения, адресованные группе, к которой подключилась программа. Если работа ведется по протоколу UDP или TCP, ядро будет автоматически осуществлять фильтрацию портов. Если используются неструктурированные сокеты, фильтрацию должна выполнять программа.

Глобальная многоадресная передача сообщений

Первый этап отправки многоадресных сообщений во внешний мир заключается в написании программы и включении режима группового вещания на аппаратном уровне и уровне ядра. Далее необходимо правильно сконфигурировать маршрутизатор. Но это еще не все. Многоадресное сообщение сталкивается с целым рядом преград, прежде чем попасть в глобальную сеть.

Конфигурирование маршрутизатора

В Internet сообщение доставляется от отправителя получателю с использованием двух адресов: IP- и MAC-адреса. Для первого имеются встроенные средства маршрутизации, а второй редко бывает известен, пока сообщение не отправлено. Маршрутизатор предполагает наличие только одного получателя для заданного сообщения; если с одним адресом связаны два получателя, это обычно считается ошибкой.

Многоадресный режим требует особой обработки. Необходимо специальным образом конфигурировать маршрутизаторы. В первую очередь они должны понимать сгенерированные MAC-адреса. Это существенно отличается от того, что обычно делают маршрутизаторы.

Когда был создан протокол группового вещания, разработчики Internet-стандартов предложили и IGMP (Internet Group Management Protocol — межсетевой протокол управления группами) [RFC1112, RFC2236], предназначенный для передачи служебных сообщений группам адресатов. IGMP-сообщение несет в себе MAC-адреса и идентификаторы групп, к которым хотят подключиться или от которых хотят отключиться компьютеры в подсети. Дело в том, что сообщение о подключении к группе распространяется только в пределах своей области видимости. Поэтому маршрутизатор всегда должен вести список доступных групп. (Если в маршрутизаторе происходит сбой, информация не теряется. Когда компьютер в подсети изъявляет желание присоединиться к группе, маршрутизатор перестраивает список групп.)

Режим группового вещания уникален тем, что, хотя все члены группы видят каждое сообщение, физически они могут получать разные копии сообщения. Маршрутизатор должен передавать все многоадресные (и широковещательные)

сообщения в каждую подсеть в пределах заданной области видимости. Это требует явного или неявного копирования сообщения.

Маршрутизатор часто подключен к нескольким сетям и управляет трафиком между ними. При явном копировании сообщение передается из одной сети в другую. Неявное копирование связано с особенностями функционирования сети. Любой компьютер в пределах подсети видит все сообщения, так как физически он подключен к тому же кабелю, что и остальные. Неявное копирование происходит, когда сообщение читается более чем одним компьютером.

Туннелирование через брандмауэры (многоадресная магистраль)

Некоторые маршрутизаторы и брандмауэры не имеют необходимой поддержки адресных групп, поэтому компьютер в пределах подсети не сможет получать сообщения, передаваемые из внешнего мира. Маршрутизатор, поддерживающий только одноадресные сообщения, называется *однонаправленным*.

Потребность в широком использовании возможностей группового вещания возникла еще в 1992 году. Но, к сожалению, в сети тогда присутствовали в основном однонаправленные маршрутизаторы. Решение заключалось в том, чтобы инкапсулировать новый протокол в базовом протоколе IP и передавать сообщение непосредственно принимающей стороне, которая будет распаковывать его и отправлять дальше. Эта технология получила название *многоадресной магистрали (multicast backbone,* или *Mbone)* и может также применяться в брандмауэрах.

Как она работает? В магистрали имеется многоадресный маршрутизатор (демон mrouted), который принимает сообщения и передает их во внешний домен другому серверу mrouted. Демон mrouted получает каждый полный пакет сообщения и вставляет его в другой пакет со своим заголовком. В результате образуется сообщение с двумя IP-заголовками. На другом конце сервер mrouted распаковывает сообщение и помещает результирующий пакет в свою подсеть.

Ограничения многоадресного режима

Многоадресный режим позволяет справиться со многими проблемами широковещательного режима, но у него есть свои собственные проблемы. Они связаны с аппаратной поддержкой, производительностью, отсутствием должной идентификации членов группы и уникальностью сообщений.

Аппаратная поддержка

Выше описывалось, как реализуется многоадресный режим в операционной системе и на аппаратном уровне. К сожалению, некоторые сетевые платы вообще не поддерживают многоадресный и беспорядочный режимы. А для получения группового сообщения необходимо, чтобы плата, по крайней мере, поддерживала беспорядочный режим.

Недостаточная производительность

При получении широковещательных или многоадресных сообщений, особенно если сетевая плата работает в беспорядочном режиме, программе приходится отфильтровывать очень много пакетов. При групповом вещании эта проблема НС столь остра, но она все же присутствует.

Кроме того, канал, по которому клиент подключается к сети, может быть не в состоянии обрабатывать все сообщения, распространяемые в рамках группы (например, "живое" видео при наличии модема, работающего со скоростью 56 Кбит/с). Что происходит в таком случае? Маршрутизатор хранит сообщения в очереди до тех пор, пока срок их действия не истечет. Потенциальное решение этой проблемы заключается в усилении поддержки со стороны маршрутизаторов.

Ответственность за каждое сообщение

Еще одна проблема связана с тем, что любой пользователь сети способен послать групповое сообщение, даже не будучи членом группы. Это может приводить к разного рода злоупотреблениям.

Уникальность сообщений

Как известно, при передаче дейтаграмм может возникнуть дублирование сообщений. Похожая проблема существует и в групповом вещании, но она отчетливее выражена, так как заранее известно, что сообщение должно копироваться в момент перехода из одной подсети в другую.

В качестве примера представим, что компьютер подключен к Internet через несколько маршрутизаторов. Этого не должно быть, так как в правильно спроектированной сети на каждую подсеть приходится один маршрутизатор, но такое все же происходит. Когда компьютер подключается к группе, оба маршрутизатора получают запрос и передают его родительскому маршрутизатору. Позднее на родительский маршрутизатор поступает многоадресное сообщение. Он проверяет его адрес и обнаруживает, что этот адрес зарегистрирован дочерними маршрутизаторами посредством протокола IGMP. В результате сообщение будет направлено обоим маршрутизаторам, и каждый из них передаст его одному и тому же компьютеру. Чтобы избежать этой проблемы, необходимо, во-первых, помечать каждое сообщение, а во-вторых, стараться не усложнять сетевую топологию.

Резюме: совместное чтение сообщений

Когда необходимо распространять сообщения между несколькими компьютерами, можно использовать широковещательный и групповой режимы доставки. В широковещании применяется схема адресации, основанная на масках подсетей. Каждый компьютер, входящий в подсеть, обязан принять сообщение. В глобальной сети широковещание недопустимо.

Многоадресный режим (групповое вещание) устраняет многие ограничения широковещания и обладает расширенными возможностями. Это более предпочтительный способ совместного чтения сообщений в локальной и глобальной сетях, даже несмотря на то, что маршрутизаторы могут посылать несколько копий оного и того же сообщения. В отличие от широковещания, чтение групповых сообщений не является обязательной процедурой.

Групповое вещание — это развивающаяся технология, и все больше старых однонаправленных маршрутизаторов заменяется многоадресными маршрутизаторами. Тем не менее широковещание не исчезло, так как оно находит применение в некоторых низкоуровневых протоколах.

Глава

18 Неструктурированные сокеты

В этой главе...

Иногда для реализации проекта недостаточно имеющихся функций. Приводится начинать с нуля. Протоколы TCP и UDP не позволяют получить доступ к внутренним элементам IP-пакета. Этой цели служат такие низкоуровневые элементы, как неструктурированные сокеты. В настоящей главе мы рассмотрим, для чего могут понадобиться неструктурированные сокеты и как самостоятельно создавать сетевые пакеты.

Когда необходимы неструктурированные сокеты

Согласно сетевой модели, неструктурированный IP-пакет находится лишь на один уровень выше, чем реальный сетевой кадр. Он не содержит дополнительных компонентов, свойственных высокоуровневым протоколам. Программы, работающие с такими пакетами, обладают более высокой производительностью, но меньшей функциональностью. Поэтому необходимо четко представлять, когда имеет смысл работать с неструктурированными сокетами,

Протокол ICMP

На уровне неструктурированных сокетов появляется доступ к ICMP (Internet Control Message Protocol — протокол управляющих сообщений в сети Internet), который используется для передачи управляющих сообщений и сообщений об ошибках. Протоколы более высокого уровня, в частности TCP и UDP, не дают возможности посылать ICMP-пакеты. Кроме того, в библиотеке Sockets API нет такого типа сокетов, как SOCK_MSG. Необходимо создать неструктурированный сокет (тип SOCK_RAW), сформировать и заполнить ICMP-заголовок и лишь затем отправить сообщение.

В протоколе ICMP имеется целый ряд сервисов, перечисленных в приложении А, "Информационные таблицы". Среди них есть сервисы эха (используется командой ping), меток времени, запросов маршрутизатора и адресных масок. Естественно, все они реализованы на очень низком уровне, так как протокол ICMP предназначен для осуществления лишь самых простых задач.

Заполнение IP-заголовка

При самостоятельном заполнении IP-заголовка необходимо установить параметр сокета IP_HDRINCL. Поля заголовка и их назначение описывались в главе 3, "Различные типы Internet-пакетов". Часть полей задается с помощью функции setsockopt(), но есть поля, значения которых задать невозможно.

Необходимость в работе с отдельными полями IP-пакета возникает, например, когда нужно проверить фрагментацию пакета. Кроме того, это требуется при программировании альтернативных версий протокола IP, перечисленных в табл. 3.1.

Ускоренная передача пакетов

Как уже упоминалось, протоколы низкого уровня обладают повышенным быстродействием. По сути, неструктурированные пакеты распространяются по сети с той же скоростью, с которой передаются физические кадры.

Большинство сетевых кадров, или фреймов, имеет размер 1—2 Кбайт при пропускной способности сети 10 Мбит/с. В более быстродействующих сетях размер кадра может быть больше. Если необходимо добиться высокой скорости, следует стремиться к тому, чтобы размер пакета соответствовал размеру кадра. Это устранит потребность во фрагментации пакета. Однако подобного соответствия добиться труднее, чем кажется, так как напрямую размер кадра узнать нельзя. Информацию можно запросить у пользователя или извлечь из конфигурационных настроек оборудования, но вряд ли подобная сложность оправдана.

Решением проблемы является использование неструктурированных сокетов. И все же большинство программ работает с протоколами высокого уровня, обеспечивающими надежность и контроль над соединением.

Ограничения неструктурированных сокетов

Неструктурированные сокеты обладают большой гибкостью, но они не являются панацеей от всех бед. Необходимо знать и об их ограничениях.

- *Потеря надежности.* Теряется множество возможностей высокоуровневых протоколов. Например, в TCP гарантируется доставка: отправленное сообщение обязательно будет получено. Неструктурированные сокеты, как и протокол UDP, не гарантируют доставку пакетов.
- *Отсутствие портов.* На столь низком уровне не поддерживается понятие портов (виртуальных сетевых соединений). Это представляет собой серьезную проблему, поскольку при отсутствии портов ядро передает неструктурированный пакет всем низкоуровневым сокетам, работающим по одинаковому протоколу. Другими словами, ядро не может определить адресата пакета, если имеется несколько неструктурированных сокетов с одним и тем же номером протокола. Данная проблема иллюстрируется в программе MyPing, рассматриваемой ниже.
- *Нестандартные схемы сетевого взаимодействия.* Отправитель и получатель должны четко представлять, что они делают и как это будет воспринято на противоположной стороне. Нельзя написать программу передачи неструктурированных пакетов, не создав также программу, принимающую эти пакеты. У ICMP-сообщений имеется свой собственный получатель в стеке TCP/IP.
- *Отсутствие встроенной поддержки ICMP-сообщений.* Как упоминалось выше, для протокола ICMP нет средств автоматического создания сообщений, как в протоколах TCP (SOCK_STREAM) и UDP (SOCK_DGRAM). Необходимо сначала создать неструктурированный сокет, затем сформировать структуру заголовка, подобную приведенной в листинге 3.2, заполнить ее, вычислить контрольную сумму, добавить данные и отправить сообщение.

- *Недоступность TCP- и UPD-пакетов.* К пакетам протоколов TCP и UDP нельзя получить доступ на низком уровне. Можно задать любой номер протокола, даже 6 (TCP) или 17 (UDP), но не все операционные системы смогут правильно обработать полученное сообщение. Иначе говоря, на основании неструктурированного сокета .можно создать UDP-пакет, построить и инициализировать UDP-заголовок, а затем послать сообщение, но ответ на него получен не будет.
- *Защита на уровне суперпользователя.* В отличие от других протоколов, программа, работающая с неструктурированными сокетами, должна иметь привилегии пользователя root.

Если все эти ограничения несущественны, то неструктурированные сокеты позволят существенно повысить скорость работы программы.

Работа с неструктурированными сокетами

Если выбор сделан в пользу неструктурированных сокетов, необходимо узнать, как с ними работать. Ниже описываются соответствующие алгоритмы и рассматривается новый системный вызов.

Выбор правильного протокола

Первый шаг в создании неструктурированного сокета заключается в выборе правильного протокола. Список номеров, имен и синонимов протоколов приведен в файле /etc/protocols (описан в приложении А, "Информационные таблицы"). Получить номер протокола на основании его символического имени позволяет функция getprotobyname():

```
#include <netdb.h>
struct protoent* getprotobyname(const char* name);
```

В случае успешного завершения функция возвращает указатель на структуру protoent. В поле p_proto этой структуры и хранится интересующий нас номер протокола:

```
struct protoent* proto;
int sd;
proto=getprotobyname("ICMP");
sd = socket(PF_INET, SOCK_RAW, proto->p_proto);
```

Как уже говорилось, для протокола ICMP не предусмотрена собственная константа, определяющая тип сокета. Вместо этого во втором параметре функции socket() следует указывать константу SOCK_RAW.

Создание ICMP-пакета

После создания неструктурированного сокета необходимо сформировать пакет, У каждого пакета есть какой-то заголовок. В случае протокола IP заголо-

вок — это единственное содержимое пакета, функция sendtsof) М
у пакета заголовок или нет, поэтому им нужно управлять программно.

У ICMP-пакета есть свой заголовок и блок данных. С целью упрощения программирования (и для обеспечения архитектурной независимости) в Linux включен библиотечный файл, содержащий определение ICMP-заголовка:

```
#include <netinet/ip_icmp.h>
struct,                    icmphdr                    ,*icmp_header;
```

Для всего пакета необходимо определить отдельного структуру:

```
#define PACKETSIZE   64   /* байта '*/
struct packet_struct
{
    struct icmphdr header;
    char message[PACKETSIZE-sizeof( struct icmphdr)];
} packet;
```

Эту структуру можно использовать для приема и отправки сообщений.

Вычисление контрольной суммы

Следующий шаг, по крайней мере в отношении ICMP-пакета, заключается в вычислении контрольной суммы. Она представляет собой обратный код числа, получаемого путем суммирования всех байтов пакета. К сожалению, в Linux нет функции, которая осуществляла бы подсчет контрольной суммы, несмотря на то, что самому ядру приходится делать это неоднократно. Возможный вариант функции представлен в листинге 18.1.

Листинг 18.1. Вычисление контрольной суммы ICMP-пакета

```
unsigned short checksum(void *b, int len)
{   unsigned short *buf = b, result;
    unsigned int sum=0;                                    .

    for ( sum=0; len>1;  len-=2)   /* Складываем все
                                      двухбайтовое блова */

    if ( len == 1 )                /* если остался лишний байт, */
        sum += *(unsigned char*) buf; /* прибавляем его к сумме */
    sum = (sum >> 16) +• (sum & OxFFFF);    /* добавляем перенос */
    sum += (sum >> 16);                                  /* еще раз */
    result = -sum;                    /* инвертируе,м результат */
    return result;      /*      возвращаем двухбайтовое значейие */
}
```

Принимающая сторона выполняет противоположную проверку: все байты пакета суммируются и к ним прибавляется контрольная сумма. В результате должен получиться нуль, так как контрольная сумма вычисляется в обратном коде. Это очень удобный способ проверки, но у него есть свои недостатки. Основной из них заключается в том, что данный алгоритм позволяет выявлять одиночные ошибки. Если же две ошибки компенсируют друг друга, они не могут быть выявлены.

Управление IP-заголовком

В IP-заголовке имеется собственное поле контрольной суммы, которая вычисляется по такому же алгоритму, но охватывает не весь пакет, а лишь непосредственно заголовок. В отличие от протокола ICMP, IP-подсистема сама, осуществляг ет подсчет контрольной суммы.

IP-подсистема оставляет без изменений все поля неструктурированного пакета (см. листинг 3.1), за исключением поля версии протокола и контрольной суммы. Если номер версии задать равным нулю, функция sendto() подставит вместо него значение, соответствующее тому протоколу, который используется в сети в настоящий момент (4 — PF_INET, 6 — PF_INET6). Если же указать другое значение, оно будет оставлено без изменений. Большинство прочих полей заполняется с помощью функции setsockopt().

Сторонний трафик

Непосредственное управление IP-заголовком открывает ряд необычных возможностей. Одна из них — это низкоуровневое управление сторонним трафиком. Предположим, в сети есть три компьютера: клиент, сервер и промежуточный сервер. Сервер-посредник принимает сообщение от клиента, выполняет предварительную обработку и посылает сообщение дальше на сервер. Это и называется сторонним трафиком.

А теперь допустим, что на сервере установлена старая система, которая умеет возвращать ответ только исходному отправителю сообщения. Почти во всех протоколах, кроме TCP, ответ направляется тому компьютеру, чей IP-адрес указан в поле автора IP-пакета. Манипулируя IP-заголовком, можно подставить в это поле адрес сервера-посредника.

Сторонний трафик и вопросы этики

Возможность, подмены IP-адреса отправителя сама по себе очень интересна, но она очень редко применяется в практических целях. Обычно программисты, использующие ее, пытаются обман ным путем проникнуть в сеть, выдавая себя за кого-то другогр. При этом они, как правило, рискуют нарушить национальные или международные законы. Internet-провайдеры способны легко обнаружить "замаскированные" пакеты, отслеживая сообщения и проверяя исходные IP-адреса. К тому же, в Linux их вообще нельзя создать: система сама заполняет поле исхрдного адреса.

Как работает команда ping

При создании программы, работающей с неструктурированными сокетами, обычно начинают с имитации команды ping, которая позволяет проверить,.есть ли абонент в сети. На подобный запрос обычно отвечает сама IP-подсистема: нет приложения или сервера, ожидающего запросов от команды ping.

Поскольку пользователь обычно хочет знать время, за которое запрос возвращается обратно, в программе создаются два процесса: один отправляет сообщения, а другой их принимает. Модуль отправки, как правило, выдерживает секундную паузу между сообщениями, а принимающий модуль регистрирует все сообщения и время, когда они поступили.

Принимающий модуль программы MyPing

.Текст принимающего модуля прост. После создания неструктурированного сокета программа начинает вызывать в цикле функцию recvfrom() (листинг 18.2).

Листинг 18.2. Цикл приема сообщений в программе MyPing

```
/***          Принимающий модуль программы MyPing.          ***/
/***          Взято из файла MyPing.c на Web-узле.          ***/
/*****************************************************************/
for (;;)
{  int bytes, len=sizeof(addr);

   bzero(buf, sizeof(buf)); /* очищаем буфер и читаем сообщение */
   bytes = recvfrom(sd, buf, sizeof(buf), 0, &addr, &len);
   if ( bytes > 0 )                    '   /* если ошибок нет, */
      display(buf, bytes);          /* проверяем идентификатор и
                                          отображаем сообщение */
   else
      perror("recvfrom");
}
```

Функция display() преобразует аргумент buf в структуру packet_struct и проверяет идентификатор пакета. Если идентификатор совпадает, значит, пакет адресован данному процессу.

Необходимость идентификатора объясняется тем, как ядро обрабатывает неструктурированные сокеты. Выше уже говорилось о том, что ядро не может определить, кому адресован неструктурированный пакет, так как в нем не указан порт. Поэтому ядро доставляет пакет всем неструктурированным сокетам, работающим по соответствующему протоколу.

Протокол — это ключ для ядра. Если программа регистрирует сокет для протокола ICMP, в программу будут поступать все ICMP-сообщения (даже те, которые предназначаются другим процессам). Как различить их? ICMP-пакет включает поле идентификатора, в которое большинство программ записывает свой идентификатор процесса (PID — process ID). Опрашиваемый узел возвращает пакет обратно вместе с идентификатором. Если он не соответствует идентификатору процесса, принимающая сторона удаляет пакет.

Отправляющий модуль программы MyPing

Модуль отправки сообщений выполняет больше работы. Наряду с подготовкой и собственно передачей сообщения он также должен принимать все побочные сообщения, которые ядро помещает в его очередь (листинг 18.3).

Листинг 18.3. Цикл отправки сообщений в программе MyPing

```
/*********************.*************************************/
/***        Модуль отправки сообщений программы MyPing.    ***/
/***        Взято из файла MyPing.c на Web-узле.           ***/
/***
for (;;)
{  int len=sizeof(r_addr);
```

```
    /*—Приникаем   все, сообщения, досылаемые ядром  --*/
    if ( recvfrom(sd, &pckt, sizeof(pckt), ft/ &r addr, slen) > 0 )
        print("***Got  message!***\n");

    /*— Инициализируем отправляемый пакет—-*/
    bzero(&pckt, sizeof(pckt));           /* обнуляем содержимое */
    pckt.hdr.type = ICMP_ECHO;          /^запрашиваем эхо-сервис */
    pckt.hdr.un.echo.id = pid;           /* задаем идентификатор */
    for ( i = 0; i < sizeof(pckt.msg)-1; i++ )
        pckt.msg[i] = i+'0';            /*          заполняем буфер */
    pckt.msg[i] = 0;        /* завершаем строку нулевым символом */
    pckt.hdr.un.echo.sequence = cnt++;  /* устанавливаем счетчик */
    pckt.hdr.checksum =           /* вычисляем контрольную сумму */
         checksum(&pckt, sizeof(pckt));
    if ( sendto(sd, &pckt, sizeof(pckt), 0, addr, /* отправляем! */
              sizeof(*addr))<= 0 )
        perror("sendto");
    sleep(l);                   /* пауза в течение одной секунды */
}
```

Для того чтобы программа работала правильно, она должна изменить некоторые стандартные настройки сокета. В первую очередь все программы семейства ping устанавливают параметр TTL (число переходов пакета) равным максимально возможному значению: 255. Кроме того, сокет модуля отправки должен быть переведен в неблокируемый режим, чтобы вызов функции recvfrom() не приводил к зависанию.

Как работают программы трассировки

Другое применение протокол ICMP находит в программах трассировки маршрутов. Читателям наверняка доводилось использовать команду traceroute для определения пути к тому или иному компьютеру в сети. Программу трассировки можно создать на основе программы MyPing, так как в ней тоже используются ICMP-сообщения.

Базовый алгоритм заключается в том, что программа посылает адресату эхо-запрос со слишком малым значением TTL. Ближайший маршрутизатор помечает пакет как устаревший и возвращает сообщение об ошибке (ICMP). Программа увеличивает значение TTL и повторяет запрос. Так продолжается до тех пор, пока сообщение не достигнет адресата. Замысел очевиден: каждый маршрутизатор указывает в сообщении об ошибке свой адрес. Здесь подойдет только неструктурированный сокет, в противном случае сообщения от маршрутизаторов не будут получены.

Цикл трассировки организуется примерно так же, как и в модуле отправки сообщений в программе MyPing. Дополнительный код связан с обработкой ответов:

```
TTL = 0;
do
{  int len=sizeof(r_addr);
   struct iphdr *ip;
```

```
    TTL++;
    if ( setsockopt(sd, SOL_IP, IP_TTL, &TTL, /* задаем значение
                                                        TTL */
                    sizeof(TTL)) != 0 )
        perror("Set TTL option");

    /*** Инициализируем сообщение
        (см. модуль отправки программы MyPing) ***/

    if ( sendto(sd, &pckt, sizeof(pckt), 0, addr, /* отправляем! */
                sizeof(*addr)) <= 0 )
        perror("sendto");
    if ( recvfrom(sd, buf, sizeof(buf), 0,     /* получаем ответ */
                  &r_addr, &len) > 0 )
    {   struct hostent *hname;
        ip = (void*) buf;
        printf("Host #%d: %s \n", cnt-1,   /* отображаем IP-адрес
                                                маршрутизатора */
               inet_ntoa(ip->saddr));
        hname = gethostbyaddr(           /* пытаемся узнать имя */
           (void*)&r_addr.s_addr,  sizeof(r_addr.s_addr),
           r  addr.sin_family);
        if ("hname !=NULL )
            printf("(%s)\n", hname->h_name);
        else
            perror("Name");
    }
    else
        perror(recvfrom);
}
/*— Цикл повторяется до тех пор, пока адрес получателя
     не совпадет с адресом отвечающей стороны —*/
while ( r_addr.sin_addr.s_addr != addr->sin_addr.s_addr );
```

Если запустить программу несколько раз, то можно заметить, что маршрут к одному и тому же компьютеру немного меняется. Это не ошибка, а нормальное поведение. Изменчивость сетей рассматривалась в главе 2, "Основы TCP/IP".

Резюме: выбор неструктурированных сокетов

Неструктурированные сокеты позволяют работать с протоколом ICMP, используемым IP-подсистемой для передачи сообщений об ошибках. Благодаря этим сокетам можно также создавать свои собственные протоколы. Основное преимущество неструктурированных сокетов — высокая скорость работы, так как по своей структуре низкоуровневые IP-пакеты близки к физическим сетевым кадрам. Неструктурированные сокеты находят широкое применение в программах семейств ping и traceroute.

IPv6: следующее поколение протокола IP

Глава

19

В этой главе...

Internet сегодня — это сеть с огромным объемом трафика. Вся сила и гибкость глобальной сети заключается в IP-пакете, посредством которого сообщения разносятся по всему миру. Не сразу стали очевидными ограничения, присущие структуре IP-пакета, ограничения, влияющие на дальнейший рост сети.

В этой главе мы рассмотрим следующее поколение протокола IP: стандарт IPv6. Приводимые выше примеры основывались исключительно на протоколе IPv4. Большинство полученных знаний применимо и в отношении нового протокола, но необходимо учесть некоторые изменения, чтобы обеспечить создаваемым программным проектам долгую жизнь.

Существующие проблемы адресации

Протокол IPv4 в основном ориентирован на адресацию компьютеров в пределах подсети. Как описывалось в главе 2, "Основы TCP/IP", IP-адрес представляет собой 32-разрядное значение (4 байта).. Все адреса группируются по классам. Когда различные организации покупают блоки адресов, они делают это в рамках классов, причем размер блока в разных классах неодинаков: от нескольких сотен до нескольких миллионов адресов.

Когда процесс освоения Internet принял глобальный масштаб, организация IAB, руководящий орган сети, не на шутку забеспокоилась. Исследования показали, что меньше 1% выделенных адресов были связаны с реальным компьютером или сетью. Как результат — адресное пространство Internet стало вырождаться.

Специалисты предполагают, что в ближайшие несколько лет число компьютеров, подключенных к Internet, стократно увеличится. Протокол IPv4 просто не рассчитан на адресное пространство такого размера.

Решение проблемы вырождающегося адресного пространства

Организация IAB предприняла ряд мер по разрешению возникшей проблемы. Стоимость владения реальными IP-адресами возросла, поэтому компании стали применять протокол DHCP и системы IP-маскирования для сокращения числа выделяемых адресов.

Кроме того, возникла потребность в расширении существующей 32-разрядной схемы адресации. Решением стала спецификация IPv6 [RFC2460]. В протоколе IPv6 используются 128-разрядные адреса (16 байтов). Это позволит выделять адреса по крайней мере в течение следующего столетия. Эффективное число адресов в IPv4 —- около двух миллиардов (исключая специальные адреса). В IPv6 адресное пространство имеет размер порядка 1x1038!

Особенности стандарта IPv6

Адрес в протоколе IPv6 претерпел существенные изменения в сравнении с адресом IPv4, поскольку его размер увеличился. Каждый адрес теперь состоит из восьми шестнадцатеричных чисел, разделенных двоеточиями, например 2FFF:80:0:0:0:0:94:1. Для краткости повторяющиеся нули можно заменять двумя двоеточиями: 2FFF:80::94:1.

К адресу по-прежнему можно добавлять номер порта, который представляет собой десятичное число, записываемое через точку. Например, добавление порта 80 к приведенному выше адресу будет выглядеть так: 2FFF:80::94:1.80.

Адресное пространство IPv6 уже разделено на области, или группы [RFC 1897, RFC2471]. В этих группах делается попытка объединить разнородные адреса (как в IPX). На момент написания книги группы еще не были четко определены и постоянно менялись.

Среди всевозможных групп диапазон адресов, начинающихся с битов 001, заменяет адреса IPv4. На рис. 19.1 изображено содержимое такого адреса.

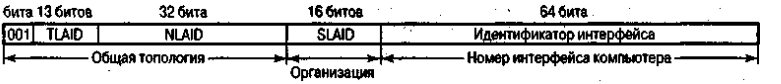

Рис. 19.1. Адрес IPv6 разделен на пять основных частей, определяющих тип адреса, порядок маршрутизации и собственно узел

На рис. 19.1 представлено пять компонентов адреса, назначение которых описано в табл. 19.1.

Таблица 19.1. Компоненты адреса IPv6

Поле	Описание
Идентификатор выделения	Трехбитовый флаг (001), определяющий открытый Internet-адрес (3 бита)
TLAID	Идентификатор агрегации верхнего уровня (Top-Level Aggregation ID); представляет провайдера одной из магистральных сетей Internet (13 битов)
NLAID	Идентификатор агрегации следующего уровня (Next-Level Aggregation ID); представляет провайдера более низкого уровня (32 бита)
SLAID	Идентификатор агрегации уровня организации (Site-Level Aggregation ID); может представлять крупную корпорацию или подсеть (16 битов)
Идентификатор интерфейса	Идентификатор конкретного компьютера (64 бита)

В главе 2, "Основы TCP/IP", рассказывалось о проблеме "каждый знает каждого", которую приходилось решать при работе в сети. Эта проблема была связана с тем, что все маршрутизаторы должны были знать каждый существующий адрес для правильной доставки сообщений. Вот почему компьютеры не могли идентифицироваться своими MAC-адресами.

В IPv4 проблема решалась путем интеграции классов в структуру IP-адреса. В IPv6 используется более простой способ. Все адреса разбиты на три области (TLA, NLA и SLA), но они распознаются не одновременно, а последовательно. Маршрутизаторы перемещают сообщения из одной области в другую, основываясь на той части адреса, которая за ними закреплена.

Как ллргут совместно работать IPv4 и IPv6?

Обещанные механизмы маршрутизации делают предложенные изменения достаточно привлекательными. Но сети Internet только предстоит перейти на стандарт IPv6. Еще очень долгое время будет использоваться старый протокол IPv4. Каким образом серверы и клиенты смогут работать сразу с двумя протоколами?

Почти все системы, которые поддерживают стек протоколов IPv6, поддерживают и стек IPv4. Такие *двухстековые* системы могут существовать до тех пор, пока большинство приложений не перейдет полностью на протокол IPv6. Основная проблема заключается в координации обоих протоколов.

На практике увеличение Длины адреса оказывает небольшое влияние на стек протоколов. В Internet в основном используются протбколы UDP и TCP, а их пакеты в любом случае просто вставляются в IP-пакет. Когда клиент IPv4 отправляет сообщение двухстековому серверу, оно попадает в стек IPv4. После того как IP-подсистема распакует сообщение, окажется, что оно имеет формат TCP или UDP. Если придерживаться подобного стиля программирования, то переход от IPv4 к IPv6 окажется безболезненным.

Протокол IPv6 включает протокол IPv4, сохраняя все его полезные особенности, кроме тех, которые больше не нужны. Чтобы выполнить преобразование адреса из формата IPv4 в формат IPv6, необходимо старшие 80 битов адреса IPv6 оставить равными нулю, а оставшаяся часть должна представлять собой число 0xFFFF плюс адрес IPv4. Например, адрес 128.10.48.6 примет вид ::FFFF:128.10.48.6, т.е. ::FFFF:800A:3006 (повторюсь, что запись :: означает все нули).

Естественно, обратное преобразование выполнить нельзя, поэтому приложение, работающее по стандарту IPv4, не может напрямую получать сообщения IPv6. (Некоторые системы допускают *искажение адресов*. В них 128-разрядный адрес преобразуется во временный 32-разрядный. Когда приложение IPv4 принимает "расширенное" сообщение, оНо видит только искаженный адрес. Если посылается ответное сообщение, ядро осуществляет обратное преобразование.)

Работа по протоколу IPv6

Для использования протокола IPv6 необходимо не только слегка модифицировать имеющиеся программы, но также соответствующим образом сконфигурировать ядро и сетевую подсистему. В некоторые дистрибутивы Linux поддержка IPv6 не включена с целью упрощения процедуры инсталляции и устранения лишних проблем, связанных с безопасностью. Ниже описывается, как включить в системе поддержку нового протокола и адаптировать к нему программы.

Конфигурирование ядра

Можно достаточно быстра узнать, поддерживает ли текущая версия ядра протокол IPv6. Если имеется файловая система /proc, загляните в каталог /proc/net. В нем должны быть файлы igmp6 и if_inet6. Если их нет, необходимо загрузить модуль ipv6.o.

Ошибка утилиты ifconfig

Программа ifconfig, являющаяся частью пакета net_tools, имеет ошибку, которая не позволяет ей автоматически загружать модуль ipv6.o. В этом случае необходимо перекомпилировать исходный файл ipv6 в немодульном режиме.

Если нужно сконфигурировать ядро, перейдите *в* каталог его исходных текстов и запустите программу конфигурации. (В некоторых дистрибутивах исходные файлы ядра по каким-то причинам не инсталлируются автоматически, В таком случае нужно сделать это самостоятельно. Получить исходные файлы ядра можно на Web-узле www.kernel.org.) Проверьте правильность всех установок, после чего перейдите в меню Experimental Drivers. Затем необходимо выбрать элемент IPv6 в группе It Is Safe to Leave These Untouched меню Network Settings. Некоторое ядра позволяют установить эксклюзивный режим IPv6. Не делайте этого! Кроме того, повторю, что можно включить файл ipv6 в состав ядра, не компилируя его как модуль. Это позволит сэкономить время на отладке ядра.

Не забудьте создать резервную копию старого ядра, чтобы в случае возникновения проблем можно было вернуться к исходному состоянию.

Конфигурирование программных средств

Далее необходимо настроить системные программные средства. Если в ядре предусмотрена поддержка протокола IPv6, запустите программу ifconfig без аргументов. Она отобразит текущие настройки всех сетевых интерфейсов. Когда поддержка IPv6 включена, во второй и третьей строках для каждого интерфейса будет отображаться адрес IPv6.

Если эти адреса отсутствуют, запустите программу с опцией --help. Программа выведет список всех поддерживаемых протоколов. Очевидно, протокол IPv6 будет отсутствовать в списке. В этом случае необходимо установить пакет net_tools, переконфигурировать и заново скомпилировать его. После инсталляции измените сценарий configure.sh, включив поддержку протокола IPv6. Затем скомпилируйте пакет и скопируйте все исполняемые файлы туда, где они должны находиться (обычно это каталог /sbin).

После того как ядро и программные пакеты скомпилированы, необходимо перезагрузить компьютер. Когда система будет готова, вызовите программу ifconfig, которая покажет привязку существующих адресов IPv4 к новым адресам. Можно даже добавить псевдоним IPv6 с помощью такой команды:

```
ifconfig eth() add <адрес IPv6>
```

Например:

```
ifconfig eth() add 2FFF::80:453A:2348
```

Теперь можно писать новые программы.

Преобразование вызовов IPv4 в вызовы IPv6

Для того чтобы привести существующие программы к новому формату, достаточно сделать лишь несколько изменений. По сути, все будет работать как и

раньше, если при написании программ придерживаться правил, описанных в начальных главах.

Первое изменение произошло в структуре сокета. Теперь ее тип не sockaddr_in, a sockaddr_in6:

```
struct sockaddr_in6 addr;
bzero(&addr, sizeof(addr));
```

Эта структура будет передаваться в функции bind(), connect() и accept() и ряд других. Второе изменение заключено в типе сокета. Один и тот же сокет не может принимать пакеты различных протоколов. Поэтому при создании сокета нужно указывать домен PF_INET6:

```
sd = socket(PF_INET6, SOCK_STREAM, 0); /* TCP6 */
/*— ИЛИ —*/                          .      .,      . .
sd = socket(PF_INET6, SOCK_DGRAM, 0);   /* UDP6 */
/*— ИЛИ —*/
sd = socket(PF_INET6, SOCK_RAW, 0);     /* ICMP6 или
                                  неструктурированные сокеты */
```

Как видите, все остается прежним, за исключением семейства протоколов. Таков был изначальный замысел функции socket(); Далее нужно указать другое семейство адресов:

```
addr.sin6_family = AF_INET6;
addr.sin6_port = htons(MY_PORT);
if ( inet_pton(AF_INET6, "2FFF::80:9ACO:351",   &addr.sin6_addr)   == 0 )
    perror("inet_pton failed");
```

Структура sockaddr_in6 имеет ряд дополнительных полей, которые можно проигнорировать. Просто обнулите их (с помощью функции bzero() или memset()), и все будет работать правильно. Больше ничего от программы не требуется.

В предыдущем фрагменте программы появилась новая функция inet_pton(). Она и ее двойник inet_ntop() очень полезны для преобразования различных форм адресов. Они поддерживают протоколы IPv4, IPv6, Rose и IPX. Правда, нынешние GNU-версии недокументированы и поддерживают только семейства адресов AF_INET и AF_INET6.

Прототипы функций таковы:

```
tinclude <arpa/inet.h>
int inet pton(int domain, const char* prsnt, void* buf);
char *inet_ntop(int domain, void* buf, char* prsnt, int len);
```

Функция inet_pton() преобразует адрес из символьного представления в двоичную форму с сетевым порядком следования байтов, помещая результат в буфер buf. Функция inet_ntop{) выполняет обратное преобразование. Параметр domain определяет семейство адресов, а параметр len — длину массива prsnt.

Преобразование неструктурированных сокетов в сокеты IPv6

При использовании сокетов UDP или TCP больше ничего не нужно менять. Если же в программе создается неструктурированный сокет или применяется протокол ICMP, необходимо создать две дополнительные структуры, обеспечивающие доступ к расширенным возможностям стандарта IPv6.

Заголовок пакета IPv6 имеет более простую структуру, чем в стандарте IPv4, но в два раза больший размер (40 байтов, а не 20). Этот размер фиксирован, так как никакие опциональные параметры не указываются. Вот каким будет определение заголовка (порядок следования байтов — сетевой):

```
/***          Определение заголовка пакета IPv6          ***/

union IPv6_Address
{
    unsigned char u8[16];
    unsigned short int u16[8];
    unsigned long int u32[4];
    unsigned long long int u64[2];
};

struct IPv6_Header
{
    unsigned int version:4;       /* версия протокола IP (6) */
    unsigned int priority:4;
    unsigned int flow_label:24;
    unsigned int payload_len:16; /* число байтов после
                                    заголовка */
    unsigned int next_header:8;  /* протокол (6 для TCP) */
    unsigned int hop_limit:8;    /* то же, что TTL */
    union IPv6_Address source;
    union IPv6_Address dest;
};
```

Только три поля требуют пояснений. Поле priority является экспериментальным и задает приоритет пакета. Если в сети возникает затор, маршрутизатор может задерживать или удалять низкоприоритетные пакеты. По умолчанию все пакеты имеют нулевой приоритет.

Поле flow_label также экспериментальное и связано с полем priority. В данном случае поток (flow) — это последовательность пакетов, которые перемещаются от отправителя к получателю через серию маршрутизаторов. Метка потока помогает маршрутизаторам определять, какая дополнительная обработка пакетов требуется. У сообщения есть только одна метка. После того как сообщение создано, метка будет одинаковой во всех пакетах. Задавать поля priority и flow_label можно, когда устанавливается необязательное поле sin6_flowinfo структуры sockaddr_in6.

Наконец, поле payload_len может содержать 0, что означает огромный пакет. Когда в сети с пропускной способностью 1 Гбит/с (пусть даже 100 Мбит/с) передаются маленькие пакеты (менее 100 Кбайт), канал большей частью работает

вхолостую. Если задать это поле равным ,0, между заголовком и телом пакета будет вставлена дополнительная запись. Ниже дано ее определение с сетевым порядком следования байтов:

```
/***    Дополнительный заголорок сверхбольшого пакета (Jumbo)    ***/

struct  Jumbo_Payload
{
    unsigned char option;     /* равно 194 */
    unsigned char length;     /* равно 4  (байта)  */
    unsigned long bytes; '    /* длина сообщения */
};
```

Благодаря этой записи появляется возможность отправлять пакеты размером до 4 Гбайт.

Протокол ICMPv6

Физическая структура заголовка протокола ICMPv6 [RFC2463] такая же, как и в ICMPv4, но содержание полей type и code существенно изменилось. К примеру, эхо-запрос и эхо-ответ теперь имеют другие номера (128 и 129 соответственно). Некоторые коды больше не поддерживаются. Полный список кодов приведен в приложении А, "Информационные таблицы".

Новый протокол группового вещания

Другое архитектурное изменение заключается в реализации многоадресного режима. Изменения коснулись трех аспектов: аппаратной поддержки, адресации и маршрутизации.

Как описывалось в главе 17, "Широковещательная, групповая и магистральная передача сообщений", режим группового вещания реализуется на аппаратном уровне: сетевая плата должна принимать пакеты, в которых указан не ее собственный MAC-адрес. Измененный MAC-адрес формируется на основе запрашиваемого группового адреса: подсистема IPv6 берет четыре последних байта адреса и добавляет к ним префикс 33:33. Например, если запрашиваемый групповой адрес FF02::725:6832:D012, то полученный MAC-адрес будет иметь вид 33:33:68:32:00:12 (725 игнорируется).

Структура группового адреса в IPv6 иная, чем в IPv4. Первый байт, FF, указывает на то, что это групповой адрес. Следующий байт разделен на два 4-битовых поля, содержащих дополнительную информацию о типе группы и деталях маршрутизации. Первое из этих полей содержит четыре флага, каждый из которых имеет определенный смысл. Самый младший бит равен нулю, если групповой адрес является конкретным, и единице, если это переходный адрес. Остальные три флага в настоящее время зарезервированы.

Второе поле определяет область видимости адреса (локальный он или глобальный). Чем выше номер, тем шире диапазон. В IPv4 область видимости определялась на основании значения TTL (чем оно меньше, тем скорее пакет устареет). Кроме того, весь диапазон адресов был поделен на четыре области видимости. В IPv6 распределение областей иное (табл. 19.2).

Таблица 19.2. Поле области видимости в IPv6

Значение	Область	Описание
1	Узел	Локальная область в пределах того же компьютера (как 127.0.0.1)
2	Канал	Сообщения остаются в пределах группы, определенной маршрутизатором; маршрутизатор не позволяет таким сообщениям пройти дальше
5	Сервер	Сообщения остаются в пределах сервера
8	Организация	Сообщения остаются в пределах организации
14	Глобальная	Все маршрутизаторы пропускают сообщения в глобальную сеть (пока срок их действия не истечет)

Наконец, правила маршрутизации (см. главу 17, "Широковещательная, групповая и магистральная передача сообщений") остаются теми же. В IPv6 применяется тот же протокол IGMP, что и в IPv4, для передачи запросов на подключение к группе и отключение от нее. Поскольку распространение сообщений осуществляется на основании MAC-адресов, дополнительное поведение не требуется.

Процедура подключения к группе в IPv6 почти такая же, что и в IPv4, но структура выбора адреса будет иной:

```
/***        Структура для задания группового адреса в IPv6        ***/
/*****************************************************************/
struct ipv6_mreq

    struct in6_addr ipv6mr_multiaddr; /* групповой адрес IPv6 */
    unsigned  int  ipv6mr_interface;        /* номер интерфейса */
```

Первое поле, ipv6mr_multiaddr, содержит групповой адрес сообщения в формате IPv6 (например, FF02::10). В следующем поле задается номер интерфейса: О означает все интерфейсы, 1 — первый интерфейс (ethO) и т.д.

Подключение к группе осуществляется следующим образом:

```
/*****************************************************************/
/***          Подключение к адресной группе в IPv6           ***/
/*****************************************************************/
const char *GroupID = "FF02::4590:3AO";
struct ipv6_mreq mreq;
if ( inet_pton(GroupID, &mreq.ipv6mr_multiaddr) == 0 )
    panic("address (%s) bad", GroupID);
mreq.ipv6mr_interface =0;                  /* любой интерфейс */
if ( setsockopt(sd, SOL_IPV6, IPV6_ADD_MEMBERSHIP, &mreq,
                sizeof(mreq)) != 0 )
    panic("Join multicast failed");
if ( setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on))
     != 0 )
    panic("Join multicast failed");
```

Достоинства и недостатки IPv6

Протокол IPv6 избавился от ряда устаревших особенностей протокола IPv4. Во-первых, что самое очевидное, благодаря 128-разрядной адресации существенно расширился диапазон адресов. Сети в настоящее время стали гораздо более сложными, чем раньше, и протокол IPv4 уже не в состоянии с ними справиться:, С новыми адресами гораздо проще работать маршрутизаторам.

Другое полезное изменение — увеличение размера пакета, что позволяет полнее использовать возможности гигабитных сетей. Ранее, в IPv4, размер пакета ограничивался значением 64 Кбайт. Теперь полезная нагрузка значительно возросла, так как размер пакета Jumbo может достигать 4 Гбайт.

Еще одним преимуществом протокола является улучшенная поддержка группового вещания. Правда, основные изменения коснулись структуры адреса, поэтому работа в многоадресном режиме ведется так же, как и прежде.

Можно выделить три ограничения IPv6. Во-первых, объем служебных данных пакета удваивается. Заголовок типичного сообщения IPv4 занимает 20 байтов. В IPv6 это значение составляет 40 байтов. Но в отличие от IPv4 длина пакета не учитывается при вычислении размера заголовка. Таким образом, в IPv4 максимальный размер пакета составляет 65535 байтов, а в IPv6 (не в режиме Jumbo) — 65535+40 байтов.

В IPv6 больше не поддерживается широковещание. На самом деле это не проблема. Многоадресный режим предоставляет большую гибкость и больше возможностей для контроля, поэтому широковещание больше не применяется в современных проектах.

Наконец, в IPv6 в заголовок пакета не включена контрольная сумма. Это упрощает заголовок, но затрудняет работу со старыми интерфейсными устройствами, которые не помещают контрольную сумму или CRC-код в физический кадр. Новые устройства выполняют все проверки целостности данных для операционной системы.

Ожидаемая поддержка со стороны Linux

Сообщество разработчиков Linux уделило очень большое внимание протоколу IPv6. Во все версии Linux, начиная с 2.2.0, встроена полная поддержка текущего варианта протокола. Единственным недостатком IPv6 является то, что он еще не окончательно принят в качестве стандарта и некоторые его элементы не до конца определены. Ни одна из операционных систем не совместима с ним на 100%.

Технология 6bone

Подобно тому как в IPv4 существовали проблемы с поддержкой многоадресного режима, которые привели к появлению технологии Mbone (многоадресная магистраль), многие маршрутизаторы не поддерживают протокол IPv6. В результате энтузиасты IPv6 создали технологию 6bone, в которой пакет IPv6 помещается внутрь пакета IPv4, передаваемого от одного маршрутизатора к другому. Эта технология находит все большее применение в Европе и на Дальнем Востоке.

И последнее замечание: в связи со своей экспериментальностью протокол IPv6 может вызывать появление брешей в системе защиты компьютера. Лучше не применять его там, где внутренняя сеть соединена с Internet.

Резюме: подготовка программ к будущим изменениям

Протокол IPv6 устраняет многие ограничения, присущие схеме адресации IPv4. Расширяя диапазон адресов на несколько порядков, IPv6 становится стандартом ближайшего будущего для Internet. Он уже принят в сетях IPX, используется в Европе и на Дальнем Востоке.

Адаптация программ, написанных для IPv4, к требованиям стандарта IPv6 не составит особого труда, если при написании программ следовать правилам и примерам, представленным в этой книге. Функция socket() действительно упрощает модификацию программ при переносе их в сети другого типа.

Протокол IPv6 включает как составную часть и протокол IPv4, поэтому адреса IPv4 будут по-прежнему распознаваться новыми системами. В большинстве операционных систем, скорее всего, будут два стека протоколов, чтобы можно было осуществлять трансляцию адресов обоих типов.

Ядро Linux, по крайней мере версия 2.2.0, поддерживает IPv6, но не во всех дистрибутивах это ядро соответствующим образом скомпилировано. В этой главе описывалось, что нужно делать в подобной ситуации.

Кроме того, в данной главе рассказывалось, как преобразовать существующие программы с учетом нового стандарта. Рассматривались две новые функции, выполняющие преобразование адресов. Важно учитывать наличие протокола IPv6 в приложениях, чтобы они могли эффективно использоваться в будущем.

Часть V

Приложения

В этой части...

Приложение

A

Информационные таблицы

В этом приложении...

В данном приложении сведены справочные таблицы, имеющие отношение к программированию сокетов.

Домены: первый параметр функции socket()

В табл. АЛ перечислены значения первого параметра функции socket(). Эти же константы можно использовать в функции bind(), хотя считается, что в ней все константы должны иметь префикс AF_. В настоящее время оба семейства констант равнозначны. Определения структур находятся в файле sys/socket.h.

Таблица А.1. Семейства протоколов для первого параметра функции socket()

Семейство	Описание и пример	Адресная структура

PF_UNSPEC	Неопределенное семейство	`struct sockaddr {`
		`  unsigned short int sa_family;`
		`  unsigned char  sa_data[14];`
		`};`

PF_LOCAL
PF_UNIX
PF_FILE
	BSD-метод доступа к локальным именованным каналам
	`#include <linux/un.h>`
	`struct sockaddr_un addr;`
	`addr.sun_family = AF_UNIX;`
	`strcpy(addr.sun_path,`
	`  "/tmp/mysocket");`
		`#define UNIX_PATH_MAX 108`
		
		`struct sockaddr_un {`
		`  sa_family_t sun family;`
		`  char sun_path[UNIX_PATH_MAX];`
		`};`

PF_INET	Семейство протоколов IPv4
	`#include <linux/in.h>`
	`struct sockaddr_in addr;`
	`bzero(&addr,  sizeof(addr));`
	`addr.sin_family = AF_INET;`
	`addr.sin_port = htons(9999);`
	`if ( inet_aton("127.0.0.1",`
	`  &addr.sin_addr) == 0 )`
	`  perror("Addr conversion");`
		`struct sockaddr in {`
		`  sa_family_t  sin_family;`
		`  unsigned short int  sin_port;`
		`  struct in_addr  sin_addr;`
		`  unsigned char  pad[];`
		`};`

PF_AX25	Семейство протоколов радиолюбительской связи AX.25
	`#include <linux/AX25.h>`
		`typedef struct {`
		`  char ax25_call[7];`
		`} ax25_address;`
		
		`struct sockaddr_ ax25 {`
		`  sa_family_t  sax25 family;`
		`  ax25_address  sax25_call;`
		`  int  sax25_ndigis;`
		`};`

Семейство	Описание и пример	Адресная структура

PF_IPX	Семейство протоколов Novell
	#include <linux/ipx.h>
		struct sockaddr ipx
		{
		sa_family_t sipx_family;
		__u16 sipx_port;
		__u32 sipx_network;
		unsigned char
		sipx_node[IPX_NODE_LEN];
		__u8 sipx_type;
		/* выравнивание */
		unsigned char sipx_zero;
		}

PF_APPLETALK	Семейство протоколов AppleTalk
	#include <linux/atalk.h>
		struct sockaddr_at {
		sa_family_t sat family;
		u8 sat_port;
		struct at_addr {
		u16 s_net;
		u8 s_node;
		} sat_addr;
		char sat_zero[];
		};

PF_NETROM	Семейство протоколов радиолюбительской связи NetROM	
PF_BRIDGE	Мультипротокольный мост	
PF_ATMPVC	Постоянные виртуальные каналы ATM	

PF_X25	(Зарезервировано для проекта X.25)
	#include <linux/x25.h>
		typedef struct {
		char x25_addr[16];
		} x25 address;
		
		struct sockaddr_x25 {
		sa_family_t sx25 family;
		/* Адрес X.121 */
		x25_address sx25_addr;
		};

Продолжение табл. А. 1

Семейство	Описание и пример	Адресная структура

PF_INET6	Семейство протоколов IPv6
	#include <linux/in6.h>
		struct in6_addr
		{
		union {
		_u8 u6_addr8[16];
		_u16 u6_addr16[8];
		_u32 u6 addr32[4];
		#if ("OUL) >~ox£ffffffff
		#ifndef _RELAX_IN6_ADDR_ALIGNMENT
		/* 64-разрядное выравнивание
		не поддерживается.
		Но лучше выполнять
		принудительное выравнивание,
		когда это возможно */
		_u64 u6_addr64[2];
		#endif
		#endif
		} in6_u;
		#define s6_addr in6_u.u6_addr8
		#define s6_addr16 in6_u.u6_addr16
		#define s6_addr32 in6_u.u6_addr32
		#define s6_addr64 in6u.u6_addr64
		
		struct sockaddr_in6 {
		unsigned short int sin6_family;
		_ u16 sin6_port;
		_ u32 sin6_flowinfo;
		struct in6_addr sin6_addr;

PF_ROSE	Семейство протоколов радиолюбительской связи Rose
	#include <linux/rose.h>
		typedef struct {
		char rose addr[5];
		} rose address;
		
		struct sockaddr_rose {
		sa_family_t srose_family;
		rose_address srose_addr;
		ax25_address srose_call;
		int srose_ndigis;
		ax25_address srose_digi;
		
		struct full_sockaddr_rose {
		sa family t srose_family;
		rose_address srose_addr;
		ax25_address srose_call;
		unsigned int srose ndigis;
		ax25 address
		srose_digis[ROSE_MAX_DIGIS];

Семейство	Описание и пример	Адресная структура
PF_DECnet	(Зарезервировано для проекта DECnet)	
PF_ NETBEUI	(Зарезервировано для проекта 802.2LLC) #include <linux/netbeui.h>	#define NB_NAME_LEN 20 struct sockaddr_netbeui { sa_family snb_family; char snb _name[NB_NAME_LEN]; char snb_devhint[IFNAMSIZ];
PF_SECURITY	Протоколы безопасности	
PF_KEY	Набор функций управления шифрами	
PF_NETLINK PF_ROUTE	Эмуляция 4.4 BSD #include <linux/netlink.h>	struct sockaddr_nl { sa_family_t nl_family; unsigned short nl_pad; __u32 nl_pid; __u32 nl_groups;
PF_PACKET	Семейство пакетных протоколов #include <linux/if packet.h>	struct sockaddr_pkt { unsigned short spkt_family; unsigned char spkt_device[14]; unsigned short spkt_protocol ; struct sockaddr_ll { unsigned short sll_family; unsigned short sll_protocol; int sll_ifindex; unsigned short sll_hatype; unsigned char sll_pkttype; unsigned char sll_halen; unsigned char sll_addr[8];
PF_ASH	Семейство Ash	

Окончание табл. А. 1

Семейство	Описание и пример	Адресная структура

PF_ECONET	Семейство Acorn Econet
	# include <linux/if ec.h>
		struct ec_addr
		{
		/* Номер станции */
		unsigned char station
		/* Номер сети */
		unsigned char net;
		};
		
		struct sockaddr_ec
		{
		unsigned short sec_family;
		unsigned char port;
		/* Байт флагов */
		unsigned char cb;
		/* Тип сообщения */
		unsigned char type;
		struct ec_addr addr;
		unsigned long cookie;
		};

PF_ATMSVC	Коммутируемые виртуальные каналы ATM	
PF_SNA	Проект Linux SNA	

PF_IRDA	Сокеты IRDA
	#include <linux/irda.h>
		struct sockaddr_irda {
		sa_family_t sir_family
		/* Селектор LSAP/TSAP */
		unsigned char sir_lsap_sel;
		/* Адрес устройства */
		unsigned int sir_addr;
		/* Обычно <сервис:IrDA:TinyTP */
		char sir_name[25];
		};

Типы: второй параметр функции socket()

Во втором параметре функции socket() выбирается тип протокола. Некоторые из констант, перечисленных в табл. А.2, лишь указывают на то, что протоколы данного типа станут поддерживаться в будущем.

Таблица А.2. Типы протоколов для второго параметра функции socket()

Тип протокола	Описание
SOCK_STREAM	(TCP) Надежное двустороннее потоковое соединение. Сокеты данного вида можно использовать при вызове высокоуровневых функций, которым передаются аргументы типа FILE*. Потоковый протокол позволяет устанавливать виртуальные соединения с сетью через порты и выделенные клиентские каналы. После установления соединения функция accept() возвращает дескриптор сокета, связанного с новым клиентом
SOCK_DGRAM	(UDP) Ненадежное взаимодействие без установления соединения. Все сообщения передаются независимо друг от друга, и каждое из них может быть потеряно. Этот протокол также поддерживает понятие порта
SOCK_RAW	(IP) Доступ к внутренним полям сетевых пакетов. С помощью сокетов данного типа создаются ICMP-сообщения. Доступ разрешен только пользователю root
SOCK_RDM	(RDM) Надежная доставка сообщений. Гарантируется доставка каждого пакета, но их порядок может быть неправильным. (Еще не реализован в Linux и других версиях UNIX.)
SOCK_SEQPACKET	Последовательная; надежная доставка дейтаграмм фиксированного размера. (Еще не реализован в Linux.)
SOCK_PACKET	(Физический уровень). Сокет переводится в беспорядочный режим (если таковой поддерживается), в котором получает все сетевые пакеты. Этот тип сокетов специфичен для Linux. Доступ разрешен только пользователю root. (Создавать такого рода сокеты не рекомендуется— лучше использовать семейство протоколов PF_PACKET.)

Определения протоколов

В листинге А.1 приведен фрагмент файла /etc/protocols [RFC2292], в котором идентифицируются наиболее распространенные сетевые протоколы. Редактировать этот файл *не* рекомендуется.

Листинг А.1. Файл /etc/protocols

```
ip          0 IP        протокол IP (Internet Protocol)
icrop       1 ICMP      протокол ICMP (Internet Control Message
                                Protocol)
igmp        2 IGMP      протокол IGMP (Internet Group Management
                                Protocol)
ggp         3 GGP       протокол GGP (Gateway-Gateway Protocol)
ipencap     4 IP-ENCAP  протокол инкапсуляции IP-пакетов
st          5 ST        режим потоковой передачи дейтаграмм
tcp         6 TCP       протокол TCP (Transmission Control
                                Protocol)
egp         8 EGP       протокол EGP (Exterior Control Protocol)
pup        12 PUP       протокол PUP (PARC Universal Protocol)
udp        17 UDP       протокол UDP (User Datagram Protocol)
hmp        20 HMP       протокол HMP (Host Monitoring Protocol)
xns-idp    22 XNS-IDP   протокол XNS/IDP (Xerox Network
```

```
                                Standard — Internet Datagram Protocol)
rdp         27 RDP              протокол RDP (Reliable Datagram Protocol)
iso-tp4     29 ISO-TP4          протокол ISO Transport Protocol (класс 4)
xtp         36 XTP              протокол XTP (Xpress Transfer Protocol)
ddp         37 DDP              протокол DDP (Datagram Delivery Protocol)
idpr-cmtp   39 IDPR-CMTP        протокол IDPR-CMTP (Inter-Domain Policy
                                Routing -- Control Message Transport
                                Protocol)
rspf        73 RSPF             протокол RSPF (Radio Shortest Path First)
vmtp        81 VMTP             протокол VMTP (Versatile Message
                                      Transaction Protocol)
ospf        89 OSPFIGP          протокол OSPF IGP (Open Shortest Path
                                First — Interior Gateway Protocol)
ipip        94 IPIP             еще один протокол инкапсуляции IP-пакетов
encap       98 ENCAP            еще один протокол инкапсуляции IP-пакетов
```

Стандартные назначения Internet-портов (первые 100 портов)

В листинге А.2 приведены описания портов (вплоть до порта с номером 100) из файла /etc/services. Привязку портов можно менять, но об этом нужно заранее уведомлять клиентов.

Листинг А.2. Файл /etc/services

```
tcpmux          1/tcp                   мультиплексор портов TCP
rtmp            1/ddp                   протокол RTMP (Routing
                                        Table Maintenance Protocol)
nbp             2/ddp                   протокол NBP (Name Binding
                                                Protocol )
echo            4/ddp                   протокол ATEP (AppleTalk
                                        Echo Protocol)
zip             6/tcp                   протокол ZIP (Zone
                                        Information Protocol)
echo            7/tcp
echo            7/udp
discard         9/tcp    sink null
discard         9/udp    sink null
systat          11/tcp   users
daytime         13/tcp
daytime         13/udp
netstat         15/tcp
qotd            17/tcp   quote
msp             18/tcp                  протокол MSP (Message Send
                                                Protocol)
msp             18/udp                  протокол MSP (Message Send
                                                Protocol)
chargen         19/top   ttytst source
```

```
chargen      19/udp  ttytst source
ftp-data     20/tcp
ftp          21/tcp
fsp          21/udp  fspd
ssh          22/tcp                    система SSH (Secure Shell)
ssh          22/udp                    система SSH (Secure Shell)
telnet       23/tcp
#24 — закрытый
smtp         25/tcp  mail
1 26 — не назначен
time         37/tcp  timeserver
time         37/udp  timeserver
rip          39/udp  resource          протокол RLP (Resource
                                       Location Protocol)
nameserver   42/tcp  name              документ IEN 116
whois        43/tcp  nicname
re-mail-ck   50/tcp                    протокол RMCP (Remote Mail
                                       Checking Protocol)
re-mail-ck   50/udp                    протокол RMCP (Remote Mail
                                       Checking Protocol)
domain       53/tcp  nameserver        сервер доменных имен (DNS)
domain       53/udp  nameserver
mtp          57/tcp                    устарел
bootps       67/tcp                    сервер BOOTP
bootps       67/udp
bootpc       68/tcp                    клиент BOOTP
bootpc       68/udp
tftp         69/udp
gopher       70/tcp                    сервер Gopher
gopher       70/udp
rje          77/tcp  netrjs
finger       79/tcp
www          80/tcp  http              протокол HTTP (HyperText
                                       Transfer Protocol)
www          80/udp                    протокол HTTP (HyperText
                                       Transfer Protocol)
link         87/tcp  ttylink
kerberos     88/top  kerberos5 krb5    Kerberos v5
kerberos     88/udp  kerberos5 krb5    Kerberos v5
supdup       95/tcp
linuxconf    98/tcp
  100 — зарезервирован
```

Коды состояния HTTP 1.1

В табл. А.3 перечислены стандартные коды состояния протокола HTTP 1.1 [RFC2616, RFC2817].

Таблица А.3. Коды состояния HTTP

Класс	Имя класса	Конкретный код и описание
	Информационные сообщения	100 Continue
		101 Switching Protocols
2xx	Успешное завершение	200 OK
		201 Created
		202 Accepted
		203 Non-Authoritative Information
		204 No content
		205 Reset Content
		206 Partial Content
3xx	Перенаправление	300 Multiple Choices
		301 Moved Permanently
		302 Moved Temporarily
		303 See Other
		304 Not Modified
		305 Use Proxy
4xx	Клиентская ошибка	400 Bad Request
		401 Unauthorized
		402 Payment Required
		403 Forbidden
		404 Not Found
		405 Method Not Allowed
		406 Not Acceptable
		407 Proxy Authentication Required
		408 Request Timeout
		409 Conflict
		410 Gone
		411 Length Required
		412 Precondition Failed
		413 Request Entity Too Large
		414 Request-URI Too Long
		415 Unsupported Media Type
5xx	•Серверная ошибка	500 Internal Server Error
		50% Not Implemented
		502 Bad Gateway
		503 Service Unavailable
		504 Gateway Timeout
		505 HTTP Version Not Supported

Параметры сокетов (функции get/setsockopt())

В табл. А.4—А.7 перечислены всевозможные параметры сокетов. В разных версиях UNIX размерность некоторых параметров может меняться. Например, параметр IP_TTL в Linux имеет тип int, но заполняется только первый байт. В AIX тот же самый параметр имеет тип char. (Колонка со звездочкой "*" означает поддержку в Linux, колонка с буквой "Ч" — возможность чтения, колонка с буквой "З" — возможность записи.)

Таблица А.4. Общие параметры сокетов

Уровень	Параметр	Описание			З	Значение	Тип
SOL_SOCKET	SO_ATTACH_FILTER	Подключение фильтра	?	?	?	Целое	int
SOL_SOCKET	SO_BINDTODEVICE	Привязка к устройству	?	?	?	Строка	char*
SOL_SOCKET	SO_ BROADCAST	Разрешение широковещательного режима	Да	Да	Да	Булево	int
SOL_SOCKET	SO_BSDCOMPAT	Включение режима совместимости с BSD	Да	Да	Да	Булево	int
SOL_SOCKET	SO_DEBUG	Разрешение отладки сокета	Да	Да	Да	Булево	int
SOL_SOCKET	SO_DETACH_FILTER	Отключение фильтра	?	?	?	Целое	int
SOL_SOCKET	SO_DONTROUTE	Запрет маршрутизации	Да	Да	Да	Булево	int
SOL_SOCKET	SO_ERROR	Код последней ошибки	Да	Да	Да	Целое	int
SOL_SOCKET	SO_KEEPALIVE	Поддержание соединения в активном состоянии	Да	Да	Да	Булево	int
SOL_SOCKET	SO_LINGER	Запрет закрытия сокета на период обработки данных	Да	Да	Да	Задержка	struct linger
SOL_SOCKET	SO_NO_CHECK	Отсутствие проверки	Да	Да	Да	Булево	int
SOL_SOCKET	SO_OOBINLINE	Помещение внеполосных данных в обычную очередь	Да	Да	Да	Булево	int
SOL_SOCKET	SO_PASSCRED	Разрешение передачи пользовательских идентификаторов	Да	Да	Да	Булево	int
SOL_SOCKET	SO_PEERCRED	Идентификаторы передающей стороны	Да	Да	Да	Идентификаторы	struct ucred
SOL_SOCKET	SO_PRIORITY	Задание приоритета очереди	Да	Да	Да	Целое	int
SOL_SOCKET	SO_RCVBUF	Размер входного буфера	Да	Да	Да	Целое	int
SOL_SOCKET	SO_RCVLOWAT	Нижний порог входного буфера	Да	Да	Нет	Целое	int

Продолжение табл.А.4

Уровень	Параметр	Описание		Ч	З	Значение	Тип
SOL_SOCKET	SO_RCVTIMEO	Период тайм-аута входного буфера	Да	Да	Да	Время	struct timeval
SOL_SOCKET	SO_REUSEADDR	Повторное использование адреса	Да	Да	Да	Булево	int
SOL_SOCKET	SO_REUSEPORT	Повторное использование адреса (групповое вещание)	Нет	—	—	Булево	int
SOL_SOCKET	SO_SECURITY AUTHENTICATION	Параметры аутентификации	Нет	—	—	Целое	int
SOL_SOCKET	SO_SECURITY_ENCRYPTION_NETWORK	Параметры шифрования соединения	Нет	—	—	Целое	int
SOL_SOCKET	SO_SECURITY ENCRYPTIONJTRANSPORT	Параметры шифрования передаваемых данных	Нет	—	—	Целое	int
SOL_SOCKET	SO SNDBUF	Размер выходного буфера	Да	Да	Да	Целое	int
SOL_SOCKET	SO_SNDLOWAT	Нижний порог выходного буфера	Да	Да	Да	Целое	int
SOL_SOCKET	SO_SNDTIMEO	Период тайм-аута выходного буфера	Да	Да	Да	Время	struct timeval
SOL_SOCKET	SO_TYPE	Тип сокета	Да	Да	Да	Целое	int

Таблица А.5. Параметры IP-сокетов

Уровень	Параметр	Описание		Ч	З	Значение	Тип
SOL_IP	IP_ADD_MEMBERSHIP	Подключение к адресной группе	Да	Да	Да	Групповой адрес Ipv4	struct ip_mreq
SOL_IP	IP_DROP_MEMBERSHIP	Отключение от адресной группы	Да	Да	Да	Групповой адрес Ipv4	struct ip_mreq
SOL_IP	IP_HDRINCL	Разрешение ручного создания IP-заголовка	Да	Да	Да	Булево	int
SOL_IP	IP_MTU_DISCOVER	Определение максимального размера передаваемого блока	Да	Да	Да	Целое	int
SOL_IP	IP_MULTICAST_IF	Исходящий интерфейс группового вещания	Да	Да	Да	Адрес Ipv4	struct in_addr
SOL_IP	IP_MULTICAST_LOOP	Разрешение обратной групповой связи	Да	Да	Да	Булево	int
SOL_IP	IP_MULTICAST_TTL	Значение TTL для многоадресного режима	Да	Да	Да	Целое	int
SOL_IP	IP_OPTIONS	Опции протокола IP	Да	Да	Да	Опции	int[]

Уровень	Параметр	Описание	*	Ч	З	Значение	Тип
SOL_IP	IP_PKTINFO	Разрешение сбора информации о пакете	Да	Да	Да	Булево	int
SOL_IP	IP_PKTOPTIONS	Опции пакета	Нет	—	—	Опции	int[]
SOL_IP	IP_RECVERR	Разрешение на получение пакетов с описанием ошибок	Да	Да	Да	Булево	int
SOL_IP	IP_RECVOPTS	Разрешение опций поступающих пакетов	Да	Да	Да	Булево	int
SOL_IP	IP_RECVTOS	Определение типа обслуживания поступающих пакетов	Да	Да	Да	Целое	int
SOL_IP	IP RECVTTL	Определение значения TTL поступающих пакетов	Да	Да	Да	Целое	int
SOL_IP	IP_RETOPTS	Разрешение опций возвращаемых пакетов	Да	Да	Да	Булево	int
SOL_IP	IP_ROUTER_ALERT	Разрешение на получение сообщений от маршрутизатора	Нет	—	—	Булево	int
SOL_IP	IPJTOS	Тип обслуживания	Да	Да	Да	Целое	int
SOL_IP	IP_TTL	Значение TTL	Да	Да	Да	Целое	int

Таблица А.6. Параметры сокетов IPv6

Уровень	Параметр	Описание	*	Ч	З	Значение	Тип
SOL_IPV6	IPV6 ADD MEMBERSHIP	Подключение к адресной группе	?	?	?	Групповой адрес IPv6	struct ipv6 mreq
SOL_IPV6	IPV6_ADDFORM	Изменение адреса сокета	?	?	?	Целое	int
SOL_IPV6	IPV6_AUT8HDR	Параметры аутентификации	?	?	?	Целое	int
SOL_IPV6	IPV6_CHECKSUM	Смещение контрольной суммы в неструктурированном пакете	?	?	?	Целое	int
SOL_IPV6	IPV6 DROP MEMBERSHIP	Отключение от адресной группы	?	?	?	Групповой адрес IPv6	struct ipv6_mreq
SOL_IPV6	IPV6_DSTOPTS	Разрешение на получение опций адресата	?	?	?	Булево	int
SOL_IPV6	IPV6_HOPLIMIT	Разрешение на определение предельного числа переходов	?	?	?	Булево	int
SOL_IPV6	IPV6_HOPOPTS	Разрешение на получение опций перехода	?	?	?	Булево	int

Продолжение табл. А.6

Уровень	Параметр	Описание	*	Ч	З	Значение	Тип
SOL_IPV6	IPV6_MULTICAST_HOPS	Число переходов для многоадресного режима	?	?	?	Целое	int
SOL_IPV6	IPV6_MULTICAST_IF	Исходящий интерфейс группового вещания	?	?	?	Целое	int
SOL_IPV6	IPV6_MULTICAST_LOOP	Разрешение групповой обратной связи	?	?	?	Булево	int
SOL_IPV6	IPV6_NEXTHOP	Разрешение на определение следующего перехода	?	?	7	Булево	int
SOL_IPV6	IPV6_PKTINFO	Получение информации о пакете	?	?	?	Булево	int
SOL_IPV6	IPV6_PKTOPTIONS	Опции пакета	?	?	?	Опции	int[]
SOL_IPV6	IPV6_ROUTER_ALERT	Разрешение на получение сообщений от маршрутизатора	?	?	?	Булево	int
SOL_IPV6	IPV6_RXSRCRT	Получение исходного маршрута	?	?	?	Булево	int
SOL_IPV6	IPV6_UNICAST_HOPS	Предельное число переходов	?	?	?	Целое	int

Таблица А.7. Параметры TCP-сокетов

Уровень	Параметр	Описание	*	Ч	З	Значение	Тип
SOL_TCP	TCP_KEEPALIVE	Задержка повторного соединения (заменяется функцией sysctl())	Нет			Целое	int
SOL_TCP	TCP_MAXRT	Максимальное время ретрансляции	Нет	—	—	Целое	int
SOL_TCP	TCP_MAXSEG	Максимальный размер сегмента (буфера передачи)	Да	Да	Да	Целое	int
SOL_TCP	TCP_NODELAY	Активизация алгоритма Нейгла	Да	Да	Да	Булево	int
SOL_TCP	TCP_STDURG	Задание местоположения байта срочного сообщения (заменяется функцией sysctl())	Нет			Булево	int
SOL_TCP	TCP_CORK	Запрет отправки частично заполненных сегментов	Да	Да	Да	Булево	int
SOL_TCP	TCP_KEEPIDLE	Включение режима поддержания активности после указанного периода	Да	Да	Да		int
SOL_TCP	TCP_KEEPINTVL	Интервал между повторными подключениями	Да	Да	Да		int

Окончание табл. А. 7

Уровень	Параметр	Описание	*	Ч	З	Значение	Тип
SOL_TCP	TCP_KEEPCNT	Предельное число повторных подключений	Да	Да	Да		int
SOL_TCP	TCP_SYNCNT	Число передаваемых символов синхронизации	Да	Да	Да		int
SOL_TCP	TCP_LINGER2	Время жизни в состоянии FINWAIT-2	Да	Да	Да		int
SOL_TCP	TCP_DEFER_ACCEPT	Активизировать модуль прослушивания, только когда приходят данные	Да	Да	Да		int
SOL_TCP	TCP_WINDOW_CLAMP	Границы окна сообщений	Да	Да	Да		int

Определения сигналов

В табл. А.8 перечислены стандартные сигналы и приведено их описание. Если во второй колонке указаны три разных номера, то первый из них соответствует BSD, второй — Linux, а третий — System V.

Таблица А.8. Стандартные коды сигналов Linux

Сигнал	Номер	Действие	Описание
SIGHUP	1	A	Сигнал отбоя, полученный от управляющего терминала или процесса
SIGINT	2	A	Сигнал прерывания, полученный с клавиатуры
SIGQUIT	3	A	Сигнал выхода, полученный с клавиатуры
SIGILL	4	A	Неправильная инструкция
SIGTRAP	5	D	Останов при трассировке
SIGABRT	6	B	Сигнал завершения, полученный от функции abort( )
SIGFPE	8	B	Ошибка вычислений с плавающей запятой
SIGKILL	9	АДЕ	Сигнал безусловного уничтожения, полученный от команды **kill**
SIGSEGV	11	B	Неправильная ссылка на память
SIGPIPE	13	A	Разрыв канала: запись в канал, с которым не связан процесс-получатель
SIGALRM	14	A	Сигнал таймера, полученный от функции alarm( )
SIGTERM	15	A	Сигнал завершения, получнный от команды **kill**
SIGUSR1	30, 10, 16	A	Сигнал, определяемый пользователем
SIGUSR2	31, 12, 17	A	Сигнал, определяемый пользователем
SIGCHLD	20, 17, 18	Б	Дочерний процесс остановился или завершился
SIGCONT	19, 18,25		Продолжение работы в случае останова

Окончание табл. А. 8

Сигнал	Номер	Дейртвие	Описание
SIGSTOP	17, 19, 23	ГДЕ .	Останов процесса
SIGTSTP	18, 20, 24	Г	Сигнал останова, полученный от терминала
SIGTTIN	21,21,26	г	Ввод данных с терминала для фонового процесса
SIGTTOU	22, 22, 27	г	Вывод данных на терминал от фонового процесса
SIGIOT	6	ВЖ	Синоним сигнала SIGABRT
SIGEMT	7,-, 7	Ж	Аппаратная ошибка
SIGBUS	10,7,10	АЖ	Ошибка на шине
SIGSYS	12,-, 12	Ж	Неправильный аргумент системного вызова (SVID)
SIGSTKFLT	-, 16, -	АЖ	Ошибка стека в сопроцессоре
SIGURG	16, 23, 21	БЖ	Сигнал о срочном сообщении в сокете (4.2 BSD)
SIGIO	23, 29, 22	АЖ	Ввод-вывод теперь возможен (4.2 BSD)
SIGPOLL		АЖ	Синоним сигнала SIGIO (System V)
SIGCLD	-,-,8	Ж	Синоним сигнала SIGCHLD
SIGXCPU	24, 24, 30	АЖ	Исчерпан лимит времени на доступ к процессору (4.2 BSD)
SIGXFSZ	25, 25, 31	АЖ	Исчерпан лимит на размер файла (4.2 BSD)
SIGVTALRM	26, 26, 28	АЖ	Сигнал виртуального таймера (4.2 BSD)
SIGPROF	27, 27, 29	АЖ	Таймер профилировщика
SIGPWR	29, 30, 19	АЖ	Сбой питания (System V)
SIGINFO	29, -, -	Ж	Синоним сигнала SIGPWR
SIGLOST		АЖ	Потеря ресурса
SIGWINCH	28, 28, 20	БЖ	Сигнал изменения размеров окна (4.3 BSD, Sun)
SIGUNUSED	-,31,-	АЖ	Неиспользуемый сигнал

- А — стандартным действием является завершение процесса.
- Б — стандартным действием является игнорирование сигнала.
- В — стандартным действием является создание дампа оперативной памяти.
- Г — стандартным действием является останов процесса.
- Д — сигнал не может быть перехвачен.
- Е — сигнал не может быть проигнорирован.
- Ж — не совместим со стандартом POSIX.1.

Коды ICMP

В табл. А.9 перечислены различные типы пакетов ICMP [RFC792] и указано, что они означают.

Таблица А.9. Коды ICMP

Тип	Код	Описание
0	0	Эхо-ответ
3		Адресат недоступен
	0	Сеть недоступна
	1	Узел недоступен
	2	Протокол недоступен
	3	Порт недоступен
	4	Необходима фрагментация, но установлен флаг DF
	5	Сбой исходного маршрута
	6	Указанная сеть неизвестна
	7	Указанный узел неизвестен
	8	Исходный узел изолирован (этот код больше не используется)
	9	Доступ к указанной сети запрещен
	10	Доступ к указанному узлу запрещен
	11	Сеть недоступна для заданного типа обслуживания
	12	Узел недоступен для заданного типа обслуживания
	13	Связь не разрешена
	14	Нарушение очередности узлов
	15	Определение очередности невозможно
4	0	Маршрутизатор просит исходный узел снизить скорость передачи
5		Перенаправление
	0	Перенаправление в другую сеть
	1	Перенаправление к другому узлу
	2	Перенаправление в другую сеть для иного типа обслуживания
	3	Перенаправление к другому узлу для иного типа обслуживания
8	0	Эхо-запрос
9	0	Уведомление от маршрутизатора
10	0	Запрос от маршрутизатора
11		Превышение времени
	0	В процессе передачи параметр TTL стал равен 0
	1	В процессе сборки фрагментов пакета параметр TTL стал равен 0
12		Ошибка параметра

Тип	Код	Описание
	0	Неправильный IP-заголовок
	1	Отсутствует требуемая опция
13	0	Запрос на получение метки времени
14	0	Ответ на запрос о получении метки времени
15	0	Информационный запрос
16	0	Ответ на информационный запрос
17	0	Запрос адресной маски
18	0	Ответ на запрос адресной маски

Диапазоны групповых адресов IPv4

В табл. АЛО указаны текущие диапазоны групповых адресов [RFC2365] в порядке возрастания начального адреса.

Таблица А. 10. Диапазоны групповых адресов

Диапазон адресов	Область видимости	Типичное значение TTL
224.0.0.0-224.0.0.255	Кластер	0
224.0.1.0-238.255.25.255	Глобальная сеть	<= 255
239.0.0.0-239.191.255.255	(не определена)	
239.192.0.0-239.195.255.255	Организация	<128
239.196.0.0-239.254.255.255	(не определена)	
239.255.0.0-239.255.255.255	Сервер	<32

Предложенное распределение адресов IPv6

В табл. А.11 перечислены предложенные в стандарте IPv6 диапазоны адресов с указанием битовых префиксов адреса.

Таблица А.11. Предложенное распределение адресов IPV6

Диапазон	Префикс адреса
(зарезервирован)	0000 0000
(не назначен)	0000 0001
NSAP	0000 001
IPX	0000 010

Окончание табл. А. 11

Диапазон	Префикс адреса
(не назначен)	0000 01 1
(не назначен)	0000 1
(не назначен)	0001
Агрегированный глобальный однонаправленный адрес	001
(не назначен)	010
(не назначен)	01 1
(не назначен)	100
(не назначен)	101
(не назначен)	110
(не назначен)	1110
(не назначен)	11110
(не назначен)	1111 10
(не назначен)	1111 110
(не назначен)	1111 11100
Локальный однонаправленный адрес уровня рабочей группы	11111110 10
Локальный однонаправленный адрес уровня сервера	1111 111011
Групповой адрес	1111 1111

Коды ICMPv6

В табл. А. 12 представлены новые ICMP-коды для стандарта IPv6.

Таблица А. 12. Коды ICMPv6

Тип	Код	Описание
1		Адресат недоступен
	0	Нет маршрута к адресату
	1	Доступ запрещен (фильтр брандмауэра)
	2	Адресат не является соседом (неправильное указание исходного маршрута)
	3	Адрес недоступен
	4	Порт недоступен
2	0	Пакет слишком велик
3		Превышение времени
	0	В процессе передачи превышено допустимое число переходов
	1	Превышено допустимое время сборки фрагментов пакета
4		Ошибка параметра
	0	Неправильное поле заголовка

Тип	Код	Описание
	1	Нераспознанный следующий заголовок
	2	Нераспознанная опция
128	0	Эхо-запрос (ping)
129	0	Эхо-ответ (ping)
130	0	Запрос на членство в группе
131	0	Сообщение о членстве в группе
132	0	Прекращение членства в группе
133	0	Запрос от маршрутизатора
134	0	Уведомление от маршрутизатора
135	0	Запрос от соседа
136	0	Уведомление от соседа
137	0	Перенаправление

Поле области видимости в групповом адресе IPv6

В табл. А. 13 приведены различные значения поля области видимости в групповом адресе IPv6.

Таблица А. 13. Поле области видимости в групповом адресе IPv6

Значение	Область	Описание
0	(не определена)	
1	Узел	Локальная область в пределах того же компьютера (как 127.0.0.1)
2	Канал	Сообщения остаются в пределах группы, определенной маршрутизатором; маршрутизатор не позволяет таким сообщениям пройти дальше
3-4	(не определена)	
5	Сервер	Сообщения остаются в пределах сервера
6-7	(не определена)	
8	Организация	Сообщения остаются в пределах организации
9-13	(зарезервированы)	
14	Глобальная	Все маршрутизаторы пропускают сообщения в глобальную сеть (пока срок их действия не истечет)
15	(зарезервировано)	

Поле флагов в групповом адресе IPv6

В табл. А.14 дана принятая в настоящий момент интерпретация поля флагов в групповом адресе IPv6.

Таблица А.14. Поле флагов в групповом адресе IPv6

Номер бита	Описание
0	Флаг переходности: 0 — конкретный адрес; 1 — переходный адрес
1	(зарезервирован)
2	(зарезервирован)
3	(зарезервирован)

Приложение Б

Сетевые функции

В этом приложении...

В этом приложении описаны все библиотечные и системные сетевые функции.

Подключение к сети

В библиотеке Socket API имеется целый ряд функций, предназначенных для создания сокетов и подключения к другим компьютерам.

socket()

Функция socket() формирует двунаправленный канал связи, как правило, с другой сетью. Дескриптор этого канала можно передавать как специализированным сетевым функциям, так и обычным функциям файлового ввода-вывода.

Прототип

```
#include <resolv.h>
#include  <sys/socket.h>
#include <sys/types.h>
int socket(int domain, int type, int protocol);
```

Возвращаемое значение

При успешном завершении функция возвращает корректный дескриптор сокета, в противном случае результат меньше нуля. Дополнительную информацию об ошибке можно узнать с помощью переменной errno.

Параметры

domain	Задает семейство (домен) сетевых протоколов (см. приложение А, "Информационные таблицы")
type	Задает сетевой уровень работы сокета (см. приложение А, "Информационные таблицы")
protocol	Задает конкретный протокол и обычно равен 0 (см. приложение А, "Информационные таблицы")

Возможные ошибки

EPROTONOSUPPORT	Тип протокола или указанный протокол не поддерживается в данном домене
ENFILE	Недостаточно памяти ядра, чтобы создать новую структуру сокета
EMFILE	Переполнение в таблице дескрипторов файлов
EACCES и ENOBUFS	Отсутствует разрешение на создание сокета указанного типа или протокола
ENOMEM	Недостаточно памяти; сокет не может быть создан, пока не будет освобождено достаточное количество ресурсов
EINVAL	Неизвестный протокол, либо семейство протоколов недоступно

Примеры

```
/*** Создание TCP-сокета ***/
int sd;
sd = socket(PF_INET, SOCK_STREAM, 0);

/*** Создание ICMP-сокета ***/
int sd;
sd = socket(PF_INET, SOCK_RAW, htons(IPPROTO_ICMP));
```

bind()

Функция bind() задает порт или имя файла для привязки сокета. В большинстве случаев ядро автоматически вызывает данную функцию, если это не было сделано явно, причем номер порта при каждом последующем запуске программы может быть другим.

Прототип

```
#include <sys/socket.h>
#include <resolv.h>
int bind(int sockfd, struct sockaddr *addr, int addrlen);
```

Возвращаемое значение

В случае успешного завершения возвращается 0. Если возникла ошибка, ее код можно узнать в переменной errno.

Параметры

sockfd	Дескриптор сокета
addr	Номер порта или имя файла
addrlen	Длина структуры addr

Возможные ошибки

EBADF	Указан неверный дескриптор сокета
EINVAL	Сокет уже связан с определенным адресом
EACCES	Запрашиваемый адрес доступен только пользователю root
ENOTSOCK	Указан дескриптор файла, а не сокета

Пример

```
/*** Привязка порта ft 9999 к сокету с любым IP-адресом ***/
struct sockfd;
struct sockaddr_in addr;
sockfd = socket(PF_INET, SOCK_STREAM, 0);
```

```
bzero(Saddr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(9999);          /* любой порт по желанию */
/* привязка к любому сетевому интерфейсу */
addr.sin_addr.s_addr = INADDR_ANY;
/* если выбирается конкретный интерфейс, следует поступить так: */
/* inet_aton("128.1.1.1", &addr.sin_addr); */
if ( bind(sockfd, saddr, sizeof(addr)) != 0 )
    perror("bind");
```

listen()

Функция listen() переводит сокет в режим ожидания запросов на подключение. Это возможно только для сокетов типа SOCK_STREAM. Функция также создает очередь запросов.

Прототип

```
#include<sys/socket.h>
 #include <resolv.h>
int listen(int sockfd, int queue_len);
```

Возвращаемое значение

В случае успешного завершения возвращается 0. Если возникла ошибка, ее код можно узнать в переменной errno.

Параметры

sockfd	Дескриптор сокета типа SOCK_STREAM, который связан с портом
queue_len	Максимальное число запросов в очереди

Возможные ошибки

EBADF	Указан неверный дескриптор сокета
ENOTSOCK	Указан дескриптор файла, а не сокета
EOPNOTSUPP	Для сокета данного типа не поддерживается функция listen(); эта ошибка возникает, если указан дескриптор сокета, не относящегося к типу SOCK_STREAM

Пример

```
/*** перевод сокета в режим прослушивания; ***/,
/*** длина очереди - 10 позиций ***/
int sockfd;
sockfd = socket(PF_INET, SOCK_STREAM, 0);
/*— Привязка к порту с помощью функции bind() —*/
listen(sockfd, 10); /* создание очереди с 10-ю позициями */
```

acceptQ

Функция accept() ожидает поступление запроса на подключение. Когда приходит запрос, функция возвращает дескриптор нового сокета (не зависящий от дескриптора sockfd), ответственного за обслуживание данного конкретного запроса. Это возможно только для сокетов типа SOCK_STREAM.

Прототип

```
#include <sys/socket.h>
#include <resolv.h>
int accept(int sockfd, struct sockaddr *addr, int *addr_len);
```

Возвращаемое значение

Если возвращается неотрицательное значение, то это дескриптор нового сокета, в противном случае это признак ошибки, код которой содержится в переменной errno.

Параметры

sockfd	Дескриптор сокета, который связан с портом и переведен в режим прослушивания
addr	Если этот параметр не равен нулю, функция помещает в него адрес клиента
addr_len	Ссылка на переменную, содержащую размер адресной структуры; в эту же переменную функция записывает реальный размер адреса

Возможные ошибки

EBADF	Указан неверный дескриптор сокета
ENOTSOCK	Указан дескриптор файла, а не сокета
EOPNOTSUPP	Сокет не относится к типу SOCK_STREAM
EFAULT	Параметр addr недоступен для записи
EAGAIN	Сокет находится в режиме неблокируемого ввода-вывода, а очередь ожидания пуста
EPERM	Брандмауэр не разрешил установить соединение
ENOBUFS, ENOMEM	Недостаточно памяти

Примеры

```
/*** Принятие запроса на подключение, ***/
/*** игнорирование адреса клиента ***/
int sockfd = socket(PF_INET, SOCK_STREAM, 0);
/*———Привязка сокета к порту с помощью функции bind() */
/*———Перевод сокета в режим прослушивания с помощью функции
     listen()  -*/
for (;;)
```

```
{  int client;
   client = accept(sockfd, 0, 0);
   /* --- взаимодействие с клиентом  --*/
  close(client);


/*** Принятие запроса на подключение, ***/
/*** отображение адреса клиента на экране ***/
int sockfd = socket(PF_INET, SOCK_STREAM, 0);
/* --- Привязка сокета к порту с помощью функции bind() --- */
/* --- Перевод сокета в режим прослушивания с помощью функции
       listen() --- */
for (;;)
{  struct sockaddr_in addr;
   int client, addr_len = addr;
   clientsd = accept(sockfd, &addr, &addr_len);
   printf("Connected: %s:%d\n", inet_ntoa(addr.sin_addr),
          ntohs(addr.sin_port));
   /* --- взаимодействие с клиентом --- */
  close(client);
```

connect()

Функция connect() подключает сокет к одноранговому узлу или серверу. Данную функцию можно вызывать для сокетов типа SOCK_DGRAM и SOCK_STREAM. В первом случае (протокол UDP) функция просто запоминает номер порта, по которому произведено подключение. Это позволяет впоследствии вызывать функции send() и recv(). Во втором случае (протокол TCP) функция инициирует процедуру трехфазового квитирования для организации потокового взаимодействия.

Прототип

```
#include <sys/socket.h>
#include <resolv.h>
int connect(int sockfd, struct sockaddr *addr, int addr_len);
```

Возвращаемое значение

В случае успешного завершения возвращается 0. Если возникла ошибка, ее код можно узнать в переменной errno.

Параметры

sockfd	Дескриптор только что созданного сокета; можно предварительно вызвать функцию bind() для привязки сокета к порту (если этого не сделать, ядро автоматически назначит сокету ближайший доступный порт)
addr	Адрес и порт узла, с которым устанавливается соединение
addr_len	Длина параметра addr

Возможные ошибки

EBADF	Указан неверный дескриптор сокета
EFAULT	Структура адреса не находится в пользовательском адресном пространстве; это вызвано неправильной ссылкой на параметр addr
ENOTSOCK	Указан дескриптор файла, а не сокета
EISCONN	Сокет уже подключен к узлу; чтобы установить другое соединение, нужно закрыть существующий сокет и создать новый
ECONNREFUSED	Сервер отказался устанавливать соединение
ETIMEDOUT	При попытке установить соединение превышен период тайм-аута
ENETUNREACH	Сеть недоступна
EADDRINUSE	Указанный адрес уже используется
EINPROGRESS	Сокет находится в режиме неблокируемого ввода-вывода, а завершить установку соединения в данный момент невозможно. Нужно периодически проверять доступность сокета для записи с помощью функции select() или poll(), и когда будет получен сигнал о готовности — вызвать функцию getsockopt() для проверки параметра so_ERROR (уровень SOL_SOCKET), в котором будет содержаться 0, если соединение установлено успешно, или один из вышеперечисленных кодов ошибок
EALREADY	Сокет находится в режиме неблокируемого ввода-вывода, а предыдущая попытка установить соединение еще не была завершена
EAFNOSUPPORT	В поле sa_family переданной адресной структуры указано неверное семейство адресов
EACCES	Пользователь пытается подключиться к широковещательному адресу, но флаг широковещания не установлен

Пример

```
/*** Подключение к TCP-серверу ***/
int sockfd;
struct sockaddr_in addr;
sockfd = socket(PF_INET, SOCK_STREAM, 0);
bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = 13; /* сервис текущего времени */
inet_atoi("127.0.0.1", &addr.sin_addr);
if ( connect(sockfd, saddr, sizeof(addr)) != 0 )
    perror("connect");
```

socketpair()

Функция socketpair() создает пару сокетов, которые связаны друг с другом механизмом, напоминающим UNIX-канал. Это подобно вызову функции pipe(), но в распоряжении сокетов оказываются все средства библиотеки Socket API. Функция socketpair() поддерживает только домены PF_UNIX И PF_LOCAL. Созданные сокеты не нужно связывать с файлом с помощью функции bind().

Прототип

```
#include <sys/socket.h>
#include <resolv.h>
int socketpair(int domain, int type, int protocol,
               int sockfds[2]);
```

Возвращаемое значение

В случае успешного завершения возвращается 0. Если возникла ошибка, ее код можно узнать в переменной errno.

Параметры

domain	Должен содержать значение PF_LOCAL или PF_UNIX
type	Должен содержать значение SOCK_STREAM; благодаря этому сокеты будут функционировать подобно каналу, который в некоторых версиях UNIX является двунаправленным, хотя стандарт POSIX.1 этого не требует
protocol	Должен содержать 0
sockfds [2]	Массив целых чисел, в котором функция сохраняет дескрипторы созданных сокетов

Возможные ошибки

EMFILE	Слишком много дескрипторов файлов используется этим процессом
EAFNOSUPPORT	Указанное семейство адресов не поддерживается на данном компьютере
EPROTONOSUPPORT	Указанный протокол не поддерживается на данном компьютере
EOPNOSUPPORT	Указанный протокол не поддерживает создание связанной пары сокетов
EFAULT	Адрес массива sockfds не является корректным

Пример

```
/*** Создание связанной пары сокетов ***/
int sockfd[2];
struct sockaddr_ux addr;
if ( socketpair(PF_LOCAL, SOCK_STREAM, 0, sockfd) != 0 )
    perror("socketpair");
```

Взаимодействие по каналу

После создания сокета и установления соединения можно использовать описанные ниже функции для приема и передачи сообщений.

send()

Функция send() посылает сообщение подключенному одноранговому компьютеру, клиенту или серверу. Она напоминает системный вызов write(), но дополнительно позволяет управлять работой канала посредством ряда опций.

Прототип

```
#include <sys/socket.h>
#include <resolv.h>
int send(int sockfd, void *buffer, int msg_len, int options);
```

Возвращаемое значение

Подобно функции write(), функция send() возвращает число записанных байтов. Это число может быть меньше, чем значение параметра msg_len. В этом случае функцию нужно вызывать до тех пор, пока требуемые данные не будут полностью отправлены. Если возвращаемое значение меньше нуля, произошла ошибка, код которой хранится в переменной errno.

Параметры

sockfd	Дескриптор подключенного сокета, имеющего тип SOCK_DGRAM или SOCK_STREAM
buffer	Отправляемые данные
msg_len	Число посылаемых байтов
options	Набор флагов, указывающих на особые режимы обработки сообщений: * MSG_OOB. Режим внеполосной передачи (срочное сообщение); * MSG_DONTROUTE Запрет маршрутизации сообщения, т.е. оно доставляется адресату напрямую; если адресат недостижим, будет получено сообщение об ошибке; * MSG_DONTWAIT. Не допускать блокирования программы, т.е. не ждать завершения функции send(); если запись невозможна, в переменную errno будет записано значение EAGAIN; * MSG_NOSIGNAL Не посылать сигнал SIGPIPE локальному компьютеру, если по какой-то причине соединение разрывается досрочно

Возможные ошибки

EBADF	Указан неверный дескриптор
ENOTSOCK	Указанный дескриптор связан с файлом, а не с сокетом
EFAULT	Для аргумента buffer указан неправильный адрес
EMSGSIZE	Функция не смогла завершить работу, так как сокет попросил ядро послать сообщение единым блоком, но размер сообщения оказался слишком велик
EAGAIN	Сокет установлен в режим неблокируемой передачи, а запрашиваемая операция приведет к блокировке
ENOBUFS	Система не сумела выделить блок памяти; операция сможет продолжиться, когда освободятся буферы
EINTR	Получен сигнал
ENOMEM	Не хватает памяти

EINVAL	Получен неправильный аргумент
EPIPE	Противоположный конец локального сокета был закрыт; программа также получит сигнал SIGPIPE, если только не был установлен флаг MSG_NOSIGNAL

Пример

```
/*** Отправка сообщения (TCP, UDP) подключенному узлу ***/
int  sockfd;
int bytes, bytes_wrote = 0;
/*——Создание сокета, подключение к серверу/узлу——*/
while  (  (bytes = send(sockfd, buffer, msg_len, 0)) > 0  )
    if  (  (bytes_wrote += bytes)  >= msg_len  )
          break;
if  ( bytes < 0  )
     perror("send");

/*** Передача срочного сообщения (TCP) подключенному узлу ***/
int sockfd;
int bytes, bytes_wrote = 0;
/*——Создание сокета, подключение к серверу——*/
if  (  send(sockfd, buffer, 1, MSG_OOB)) != 1  )
     perror("Urgent message");
```

sendto()

Функция sendto() посылает сообщение указанному адресату, не подключаясь к нему. Обычно эта функция используется для отправки дейтафамм и неструктурированных пакетов, а также в протоколе T/TCP (Transaction TCP), который, впрочем, еще не реализован в Linux.

Прототип

```
#include <sys/socket.h>
#include<resolv.h>
int sendto(int sockfd, void* msg, int len, int options,
           struct sockaddr *addr, int addr_len);
```

Возвращаемое значение

Возвращается число отправленных байтов или —1, если произошла ошибка.

Параметры

sockfd	Дескриптор сокета
msg	Отправляемые данные
len	Число посылаемых байтов
options	Флаги управления (такие же, как и в функции send())
addr	Адрес узла
addr_len	Размер адресной структуры

Возможные ошибки

(Те же, что и в функции send())

Пример

```
/*** Отправка сообщения (TCP, UDP) неподключенному узлу ***/
int sockfd;
struct sockaddr_in addr;
if ( (sockfd = socket(PF_INET, SOCK_DGRAM, 0)) < 0 )
    perror("socket");
bzero(Saddr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(DEST_PORT);
inet_aton(DEST_ADDR, &addr.sin_addr);
if ( sendto(sockfd, buffer, msg_len, 0, &addr, sizeof(addr))
     < 0 )
    perror("sendto");
```

sendmsg()

Функция sendmsg() собирает сообщение из нескольких блоков данных. Если поле msg_name структуры сообщения указывает на структуру sockaddr, функция посылает сообщение, не устанавливая соединение. Если же поле равно NULL, функция предполагает, что сокет подключен к узлу.

Прототип

```
#include <sys/socket.h>
#include <resolv.h>
#include <sys/uio.h>
int sendmsg(int sockfd, const struct msghdr *msg,
            unsigned int options);
```

Возвращаемое значение

Возвращается общее число отправленных байтов или —1, если произошла ошибка (ее код записывается в переменную errno).

Параметры

sockfd	Дескриптор сокета
msg	Указатель на структуру msghdr, в которой находятся адрес получателя, флаги и блоки сообщения. Определение структуры таково:

```
struct iovec
{
  void *iov_base;      /* начало буфера */
 _kernel_size_t iov_len; /* длина буфера */
};
```

```
struct msghdr
{
   _ptr_t msg name; /* Адрес получателя */
   socklen_t msg_namelen; /* Длина адреса */
   struct iovec *msg_iov; /* Массив буферов */
   size_t msg_iovlen; /* Длина массива */
   _ ptr_t msg_control; /* Служебные данные */
   size_t msg_controllen; /* Длина служебных данных */
   int msg_flags; /* Флаги полученного сообщения */
};
```

Через блок служебных данных программа может передавать, к примеру, дескрипторы файлов

options — Флаги управления (такие же, как и в функции send())

Возможные ошибки

(Те же, что и в функции send())

Пример

```
int i, sd, len, bytes;
char  buffer[MSGS][100];
struct iovec ip[MSGS];
struct msghdr msg;
struct sockaddr_in addr;

sd = socket(PF_INET, SOCK_DGRAM, 0);
bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(8080);
inet_aton  (&addr.sin_addr,"127.0.0.1");
bzero(&msg, sizeof (msg));
msg.msg_name = &addr;
msg.msg_namelen = sizeof (addr);
for ( i = 0; i < MSGS; i++ )
{
   io[i].iov_base = buffer[i];
   sprintf(buffer[i], "Buffer #%d: this is a test\n", i);
        io[i].iov_len=strlen(buffer[i]);
}
msg.msg_iov = io;
msg.msg_iovlen = MSGS;
if ( (bytes = sendmsg(sd, &msg, 0)) < 0 )
    perror("sendmsg");
```

sendfile()

Функция sendfile() реализует быстрый способ передачи файла через сокет. Она извлекает данные из источника с дескриптором in_fd и записывает их в приемник с дескриптором out_fd. Указатель текущей позиции исходного файла не

меняется, а файла-получателя — меняется. Функция читает указанное число байтов (параметр count) начиная с заданной позиции (параметр offset). По завершении функции указатель *offset ссылается на байт, идущий за последним прочитанным байтом.

Прототип

```
#include <unistd.h>
int sendfile(int out_fd, int in_fd, off_t *offset, size_t count);
```

Возвращаемое значение

При успешном завершении функция возвращает общее число скопированных байтов. В случае ошибки возвращается —1, а в переменную errno записывается код ошибки.

Параметры

out_f d	Дескриптор получателя (указатель текущей позиции файла меняется)
in_f d	Дескриптор источника (указатель текущей позиции файла не меняется)
offset	Указатель на переменную, в которой содержится начальное смещение
count	Число отправляемых байтов

Возможные ошибки

EBADF	Входной файл не был открыт для чтения или выходной файл не был открыт для записи
EINVAL	Дескриптор некорректен или заблокирован
ENOMEM	Недостаточно памяти для чтения из исходного файла
EIO	Неопределенная ошибка при чтении из исходного файла

Пример

```
#include <unistd.h>

struct stat fdstat;
int client = accept(sd,0,0);
int fd = open("filename.gif", O_RDONLY);
fstat(fd, &fdstat);
sendfile(client, fd, 0, fdstat.st_size);
close(fd);
close(client);
```

recv()

Функция recv() принимает сообщение от подключенного однорангового компьютера, клиента или сервера. Она работает подобно низкоуровневой функции read(), но дополнительно позволяет устанавливать управляющие флаги.

Прототип

```
#include <sys/socket.h>
#include <resolv.h>
int recv(int sockfd, void *buf, int maxbuf, int options);
```

Возвращаемое значение

Функция возвращает число прочитанных байтов или -1 в случае ошибки.

Параметры

sockfd	Дескриптор подключенного сокета
buf	Буфер, в который будет помещено поступившее сообщение
maxbuf	Размер буфера
options	Набор флагов, которые можно объединять с помощью операции побитового сложения:

* MSG_OOB. Запрашивает получение внеполосных данных, которые не передаются в обычном потоке данных. В некоторых протоколах срочные данные помещаются в начало обычной очереди, поэтому рассматриваемый флаг не может применяться при работе с такими протоколами;
* MSG_PEEK. Заставляет функцию читать данные, не удаляя их из очереди. Таким образом, при следующей операции чтения будут прочитаны те же самые данные;
* MSG_WAITALL. Запрашивает блокировку программы до тех пор, пока не будет получено все сообщение целиком. Тем не менее функция может завершиться досрочно, если получен сигнал, произошла ошибка или соединение было разорвано противоположной стороной;
* MSG_ERRQUEUE. Запрашивает прием пакета из очереди ошибок сокета. Информация об ошибке передается в служебном сообщении, тип которого зависит от протокола (для IP-сокета нужно установить параметр IP_RECVERR). В теле сообщения находится структура sock_extended_error;
* MSG_NOSIGNAL. Отключает выдачу сигнала SIGPIPE в потоковом сокете при разрыве соединения на противоположной стороне

Возможные ошибки

EBADF	Указан неверный дескриптор файла
ENOTCONN	Сокет не был подключен (см. функции connect() и accept())
ENOTSOCK	Указанный дескриптор связан с файлом, а не с сокетом
EAGAIN	Задан режим неблокируемого ввода-вывода, а данные недоступны, либо был установлен период тайм-аута, который превышен до того, как были получены данные
EINTR	Сигнал прервал выполнение операции чтения, прежде чем данные стали доступны
EFAULT	Указатель буфера чтения ссылается за пределы адресного пространства процесса
EINVAL	Передан неверный аргумент

Пример

```
/*** Получение сообщения (TCP, UDP) от подключенного узла ***/
int sockfd;
int bytes, bytes_wrote=0;
/*— Создание сокета, подключение к серверу/узлу —*/
if ( (bytes = recv(sockfd, buffer, msg_len, 0)) < 0 )
```

```
    perror("send");

/*** Получение срочного сообщения (TCP) от подключенного узла ***/
/*** Этот код обычно находится в обработчике сигнала SIGURG ***/
int sockfd;
int bytes, bytes_wrote=0;
/*- Создание сокета, подключение к серверу -*/
if ( (bytes = recv(sockfd, buffer, msg_len, MSG_OOB)) < 0 )
    perror("Urgent message");
```

recvfrom()

Функция recvfrom() принимает сообщение от неподключенного однорангового компьютера (UDP и неструктурированные сокеты). В протоколе T/TCP эта функция никогда не используется. Вместо нее вызывается функция accept().

Прототип

```
linclude <sys/socket.h>
linclude <resolv.h>
int recvfromfint sockfd, void* buf, int buf_len, int options,
            struct sockaddr *addr, int *addr_len);
```

Возвращаемое значение

При успешном завершении возвращается число прочитанных байтов. В случае ошибки возвращается — 1, а в переменную errno записывается код ошибки.

Параметры

sockfd	Дескриптор сокета
buf	Буфер для приема сообщения
buf _len	Максимальный размер буфера (сообщение усекается, если буфер слишком мал)
options	Параметры управления каналом (такие же, как и в функции recv())
addr	Адрес и порт отправителя
addr_len	Максимальный размер адресной структуры (адрес усекается, если буфер слишком мал); по завершении функции в этом параметре будет содержаться реальная длина адреса

Возможные ошибки

(Те же, что и в функции recv())

Пример

```
struct sockaddr_in addr;
int addr_len=sizeof(addr), bytes_read;
char buf[1024];
```

```
int sockfd = socket(PF_INET, SOCK_DGRAM, 0);
/*— привязка к конкретному порту —*/
bytes_read = recvfrom(sockfd, buf, sizeof(buf), 0, &addr,
                      &addr_len);
if(bytes_read<0)
    perror("recvfrom failed");
```

recvmsg()

Функция recvmsg() принимает сразу несколько сообщений от одного источника. Она может использоваться с сокетами типа SOCK_DGRAM (аналогично тому, как это происходит в случае функции sendmsg()).

Прототип

```
#include <sys/socket.h>
#include <resolv.h>
#include <sys/uio.h>
int recvmsg(int sockfd, struct msghdr *msg, unsigned int options);
```

Возвращаемое значение

При успешном завершении возвращается общее число полученных байтов, в противном случае возвращается —1.

Параметры

sockfd	Дескриптор сокета
msg	Буфер для принимаемых данных
options	Параметры управления каналом (такие же, как и в функции recv( ))

Возможные ошибки

(Те же, что и в функции recv())

Пример

```
char  buffer[MSGS][1000];
struct sockaddr_in addr;
struct iovec io[MSGS];
struct msghdr msg;

bzero(&addr, sizeof(addr));
msg.msg_name = &addr;
msg.msg_namelen = sizeof(addr);
for ( i = 0; i < MSGS; i++ )
{
   io[i].iov_base = buffer(i);
   io[i].iov_len  = sizeof(buffer[i]);
}
```

```
msg.msg_iov = io;
msg.msg_iovlen = MSGS;
if ( (bytes = recvmsg(sd, &msg, 0)) < 0 )
    perror("recvmsg");
```

Разрыв соединения

После того как программа закончила сеанс связи с внешним узлом, она должна закрыть соединение. Ниже описывается функция, ответственная за разрыв соединения.

shutdown()

Функция shutdown() закрывает указанные части канала передачи данных. Соединение, устанавливаемое посредством сокета, по умолчанию является двунаправленным. Если нужно сделать его доступным только для чтения или только для записи, то следует с помощью функции shutdown() закрыть один из концов канала.

Прототип

```
#include <sys/socket.h>
int shutdown(int sockfd, int how);
```

Возвращаемое значение

Если все прошло успешно, возвращается 0. В случае ошибки ее код можно найти в переменной errno.

Параметры

sockfd	Дескриптор сокета
how	флаг, указывающий на то, какую часть канала следует закрыть: * SHUT_RD (0) — сделать канал доступным только для записи; * SHUT _WR(1) — сделать канал доступным только для чтения; * SHUT_RDWR (2) — закрыть обе половины канала (эквивалентно вызову функции close()). Выполнять эти действия можно только в отношении подключенных сокетов

Возможные ошибки

EBADF	Указан неверный дескриптор сокета
ENOTSOCK	Указанный дескриптор связан с файлом, а не с сокетом
ENOTCONN	Указанный сокет не является подключенным

Пример

```
int sockfd;
struct sockaddr_in addr;
sockfd = socket(PF_INET, SOCK_STREAM, 0);
bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(DEST_PORT);
inet_aton (DEST_ADDR, &addr. sin_addr);
connect(sockfd, &addr, sizeof(addr));
if ( shutdown(sockfd, SHUT_WR) != 0 )
    PANIC("Can't make socket input-only");
```

Преобразование данных в сети

Работая с данными по сети, необходимо учитывать порядок следования байтов, преобразовывать адреса и т.д. В библиотеку Socket API входит достаточно большое число функций, помогающих получать информацию в нужном формате. Ниже описаны функции, которые использовались в книге.

htons(), htonl()

Функции htons() и htonl() преобразуют двоичные данные из серверного порядка следования байтов в сетевой. В случае процессора с прямым порядком хранения данных происходит перестановка байтов. Если в компьютере установлен процессор с обратным порядком байтов, функция возвращает свой аргумент в неизменном виде.

Прототип

```
#include <netinet/in.h>
unsigned short int htons(unsigned short int host_short);
unsigned long int htonl(unsigned long int host_long);
```

Возвращаемое значение

Преобразованный аргумент (16- или 32-разрядный).

Параметры

host_short	16-разрядное значение с серверным порядком следования байтов
host_long	32-разрядное значение с серверным порядком следования байтов

Возможные ошибки

(отсутствуют)

Пример

```
/*** Связываем сокет с портом 1023 ***/
struct sockaddr_in addr;
addr.sin_port = htons(1023);

/*** Связываем сокет с адресом 128.1.32.10 ***/
struct sockaddr_in addr;
addr.sin_addr.s_addr = htonl(0x8001200A);
```

ntohs(), ntohl()

Функции ntohs() и ntohl() преобразуют двоичные данные из сетевого порядка следования байтов в серверный.

Прототип

```
#include <netinet/in.h>
unsigned short int ntohs(unsigned short int network_short);
unsigned long int ntohl(unsigned long int network_long);
```

Возвращаемое значение

Преобразованный аргумент (16- или 32-разрядный).

Параметры

network_short	16-разрядное значение с сетевым порядком следования байтов
network_long	32-разрядное значение с сетевым порядком следования байтов

Возможные ошибки

(отсутствуют)

Пример

```
struct sockaddr_in addr;
int client, addrlen=sizeof(addr);
client = accept(sockfd, &addr, &addrlen);
if ( client > 0 )
    printf("Connected %lX:%d\n", ntohl(addr.sin_addr),
                                 ntohs(addr.sin_port));
```

inet_addr()

Функция inet_addr() считается устаревшей. Она преобразует IP-адрес из точечной нотации в двоичную форму с сетевым порядком следования байтов. В случае неудачи возвращается значение —1, которое также является корректным IP-адресом (255.255.255.255). Лучше пользоваться функцией inet_aton().

Прототип

```
#include<netinet/in.h>
unsigned long int inet_addr(const char *ip_address);
```

Возвращаемое значение

Не нуль	Если все прошло успешно, возвращается преобразованный IP-адрес
INADDR_NONE (-1)	Аргумент является неправильным. (Это ошибка функции. Она не возвращает отрицательное значение, а значение 255.255.255.255 обозначает обычный широковещательный адрес.)

Параметры

ip_address	IP-адрес в традиционной, точечной нотации (например, 128.187.34.2)

Возможные ошибки

(переменная errno не устанавливается)

Пример

```
if ( (addr.sin_addr.s_addr = inet_addr("182.187.34.2")) == -1 )
    perror("Couldn't convert address");
```

inet_aton()

Функция inet_aton() преобразует IP-адрес из точечной нотации в двоичную форму с сетевым порядком следования байтов. Она заменяет функцию inet_addr().

Прототип

```
#include <netinet/in.h>
int inet_aton(const char *ip_addr, struct in_addr *addr);
```

Возвращаемое значение

Если все прошло успешно, возвращается ненулевое значение. В противном случае возвращается нуль.

Параметры

ip_addr	ASCII-строка с IP-адресом (например, 187.34.2.1)
addr	Переменная, куда записывается адрес; обычно заполняется поле sin_addr структуры sockaddr_in

Возможные ошибки

(переменная errno не устанавливается)

Пример

```
struct sockaddr_in addr;
if ( inet_aton("187.43.32.1", &addr.sin_addr) == 0
    perror("inet_aton() failed");
```

inet_ntoa()

Функция inet_ntoa() преобразует IP-адрес из двоичной формы с сетевым порядком следования байтов в точечную нотацию.

Прототип

```
iinclude <netinet/in.h>
int inet_ntoa(struct in_addr *addr);
```

Возвращаемое значение

Функция возвращает строку с адресом.

Параметры

addr	Двоичный адрес (обычно это адресное поле структуры sockaddr_in)

Возможные ошибки

(переменная errno не устанавливается)

Пример

```
clientfd = accept(serverfd, &addr, &addr_size);
if ( clientfd > 0 )
    printf("Connected %s:%d\n", inet_ntoa(addr.sin_addr),
                                ntohs(addr.sin_port));
```

inet_pton()

Функция inet_pton() преобразует адрес IPv4 или IPv6 из символьного представления в двоичную форму с сетевым порядком следования байтов.

Прототип

```
Iinclude <arpa/inet.h>
int inet_pton(int domain, const char* prsnt, void* addr);
```

Возвращаемое значение

В случае успешного завершения возвращается ненулевое значение. Если возникает ошибка, возвращается нуль.

Параметры

domain	Семейство адресов (AF_INET или AF_INET6)
prsnt	ASCII-строка с IP-адресом (например, 187.34.2.1 или FFFF:8090:AO3:3245)
addr	Переменная, куда записывается адрес; обычно заполняется поле sin_addr структуры sockaddr_in или поле sin6_addr структуры sockaddr_in6

Возможные ошибки

(переменная errno не устанавливается)

Пример

```
struct sockaddr_in addr;
if ( inet_pton(AF_INET, "187.43.32.1", &addr.sin_addr) == 0 )
    perror("inet pton() failed");
```

inet_ntop()

Функция inet_pton() преобразует адрес из двоичного представления с сетевым порядком следования байтов в символьную форму. Функция поддерживает семейства адресов AF_INET и AF_INET6.

Прототип

```
#include <arpa/inet.h>
int inet_ntop(int domain, struct in_addr *addr, char* str,
              int len);
```

Возвращаемое значение

Функция возвращает строку str.

Параметры

domain	Семейство адресов (AF_INET или AF_INET6)
addr	Двоичный адрес (обычно это адресное поле структуры sockaddr in)
str	Буфер для строки адреса
len	Размер буфера

Возможные ошибки

(переменная errno не устанавливается)

Пример

```
char str[100];
clientfd = accept(serverfd, &addr, &addr_size);
if ( clientfd > 0 )
    printf("Connected %s:%d\n",
           inet_ntop(AF_INET, addr.sin_addr, str, sizeof(str)),
           ntohs(addr.sin_port));
```

Работа с сетевыми адресами

В библиотеку Socket API входят также функции, позволяющие получить доступ к различным службам имен или самостоятельно выполняющие преобразования имен.

getpeername()

Функция getpeername() определяет адрес или имя компьютера, подключенного к противоположному концу сокета с дескриптором sockfd. Результат помещается в буфер addr. Параметр addr_len определяет размер адресного буфера. Это та же самая информация, которую можно получить с помощью функции accept().

Прототип

```
#include <sys/socket.h>
int getpeername(int sockfd, struct sockaddr *addr,
                socklen_t *addr_len);
```

Возвращаемое значение

В случае успешного завершения возвращается нуль. Если произошла ошибка, ее код помещается в переменную errno.

Параметры

sockfd	Дескриптор подключенного сокета
addr	Буфер, в который помещается адресная структура
addr_len	Размер буфера; этот параметр передается по ссылке, так как функция записывает сюда реальный размер адреса

Возможные ошибки

EBADF	Указан неверный дескриптор

ENOTSOCK	Указанный дескриптор относится к файлу, а не к сокету
ENOTCONN	Сокет не подключен
ENOBUFS	В системе недостаточно ресурсов для выполнения операции
EFAULT	Указатель addr ссылается за пределы адресного пространства процесса

Пример

```
struct sockaddr_in addr;
int addr_len = sizeof(addr);
if ( getpeername(client, &addr, &addr_len) != 0 )
    perror("getpeername() failed");
printf("Peer: %s:%d\n", inet_ntoa(addr.sin_addr),
                        ntohs(addr.sin_port));
```

gethostname()

Функция gethostname() возвращает имя локального узла. Результат помещается в буфер name, размер которого определяется параметром len.

Прототип

```
#include <unistd.h>
int gethostname(char *name, size_t len);
```

Возвращаемое значение

В случае успешного завершения возвращается нуль. Если произошла ошибка, ее код помещается в переменную errno.

Параметры

name	Буфер, в который записывается имя узла
len	Размер буфера

Возможные ошибки

EINVAL	Параметр len является отрицательным либо, если функция выполняется на платформе Linux/i386, значение параметра len оказалось меньше, чем реальный размер имени
EFAULT	Параметр name содержит неправильный адрес

Пример

```
char name[50];
if ( gethostname(name, sizeof(name)) != 0
    perror("gethostname() failed");
printf("My host is: %s\n", name);
```

gethostbyname()

Функция gethostbyname() ищет имя узла в базе данных DNS-сервера и преобразует его в IP-адрес. Функции может передаваться как имя, так и сам адрес. Во втором случае поиск не осуществляется; вместо этого адрес возвращается в полях h_name и h_addr_list[0] структуры hostent.

Прототип

```
#include <netdb.h>
struct hostent *gethostbyname(const char *name);
```

Возвращаемое значение

Функция возвращает указатель на структуру hostent. В случае неудачи возвращается NULL. В структуре содержится список всех имен и адресов, связанных с данным узлом. Макроконстанта h_addr существует для совместимости со старыми приложениями.

```
#define h_addr h_addr_list[0]
struct hostent {
    char *h_name;        /* официальное имя узла */
    char **h_aliases;    /* список псевдонимов */
    int  h_addrtype;     /* тип адреса */
    int  h_length;       /* длина адреса */
    char **h_addr_list;  /* список адресов; нулевой - главный */
};
```

Параметры

name	Имя узла для поиска или IP-адрес

Возможные ошибки

ENOTFOUND	Указанный узел не найден
NO_ADDRESS, NO_DATA	Указанное имя является корректным, но с ним не связан IP-адрес
NO_RECOVERY	Произошла фатальная ошибка сервера имен
EAGAIN	Произошла временная ошибка сервера имен, повторите попытку позднее

Пример

```
int i;
struct hostent *host;
host=gethostbyname("sunsite.unc.edu");
if ( host != NULL )
{
    printf("Official name: %s\n", host->h_name);
    for ( i = 0; host->h_aliases[i] != 0; i++)
        printf("    alias[%d]: %s\n", i+1, host->h_aliases[i]);
```

```
    printf("Address type=%d\n", host->h_addrtype);
    for ( i = 0; i < host->h_length; i++)
        printf("Addr[%d]: %s\n", i+1,
               inet_ntoa(host->h_addr_list[i]));
}
else
    perror("sunsite.unc.edu");
```

getprotobyname()

Функция getprotobyname() просматривает файл /etc/protocols в поиске протокола с указанным именем. Эту функцию можно использовать для трансляции имен протоколов, таких как HTTP, FTP или Telnet, в соответствующие им номера портов.

Прототип

```
#include<netdb.h>
struct protoent *getprotobyname(const char* pname);
```

Возвращаемое значение

В случае успешного завершения функция возвращает указатель на структуру protoent. В противном случае возвращается NULL.

```
struct protoent {
    char     *p_name;     /* официальное имя протокола */
    char     **p_aliases; /* список псевдонимов */
    int      p_proto;     /* номер порта */
};
```

Параметры

pname	Имя протокола; это может быть любое известное имя протокола или псевдоним

Возможные ошибки

(переменная errno не устанавливается)

Пример

```
#include <netdb.h>

int i;
struct protoent *proto = getprotobyname("http");
if ( proto != NULL )
{
    printf("Official name: %s\n", proto->name);
    printf("Port!: %d\n", proto->p proto);
    for ( i = 0; proto->p aliases[I] != 0;
```

```
        printf("Alias[%d]: %s\n", i+1, proto->p_aliases[i]);

else
    perror("http");
```

Управление сокетами

Когда сокет открыт, можно менять самые разные его параметры. Для этой цели предназначены описанные ниже функции.

setsockopt()

Функция setsockopt() изменяет поведение сокета. У каждого параметра сокета есть значение (некоторые доступны только для чтения). Полный список параметров приведен в приложении А, "Информационные таблицы".

Прототип

```
#include <sys/types.h>
#include <sys/socket.h>
int setsockopt(int sd, int level, int optname,
               const void *optval, socklen_t optlen);
```

Возвращаемое значение

В случае успешного завершения возвращается нуль. Если произошла ошибка, ее код помещается в переменную errno.

Параметры

sd	Дескриптор сокета
level	Уровень параметра сокета (SOL_SOCKET, SOL_IP, SOL_IPV6, SOL_TCP)
optname	Имя параметра сокета
optval	Указатель на новое значение параметра сокета
optlen	Длина параметра сокета в байтах

Возможные ошибки

EBADF	Указан неверный дескриптор сокета
ENOTSOCK	Указанный дескриптор относится к файлу, а не к сокету
ENOPROTOOPT	Указанный параметр сокета не известен на данном уровне
EFAULT	Указатель optval ссылается за пределы адресного пространства процесса

Примеры

```
const int TTL=128;
/*— Задание предельного числа переходов равным 128 —*/
```

```
if ( setsockopt(sd, SOL_IP, SO_TTL, &TTL, sizeof(TTL)) != 0 )
    perror("setsockopt()  failed");
```

getsockopt()

Функция getsockopt() возвращает значение указанного параметра сокета.

Прототип

```
#include  <sys/types.h>
iinclude<sys/socket.h>
int getsockopt(int sd, int level, int optname, void *optval,
               socklen_t *optlen);
```

Возвращаемое значение

В случае успешного завершения возвращается нуль. Если произошла ошибка, ее код помещается в переменную errno.

Параметры

sd	Дескриптор сокета
level	Уровень параметра сокета (SOL_SOCKET, SOL_IP, SOL_IPV6, SOL_TCP)
optname	Имя параметра сокета
optval	Буфер для значения параметра сокета
optlen	Длина буфера; это значение передается по ссылке, так как функция записывает сюда реальную длину буфера

Возможные ошибки

EBADF	Указан неверный дескриптор сокета
ENOTSOCK	Указанный дескриптор относится к файлу, а не к сокету
ENOPROTOOPT	Указанный параметр сокета не известен на данном уровне
EFAULT	Указатель optval или optlen ссылается за пределы адресного пространства процесса

Пример

```
int error, size = sizeof(error);
if ( getsockopt(sd, SOL_SOCKET, SO_ERROR, &error, &size)) != 0 )
    perror("getsockopt() failed");
printf("socket error=%d\n", error);
```

API-функции ядра

Приложение

B

В этом приложении...

В этом приложении приведена справочная информация, касающаяся библиотечных функций ядра и системных вызовов, которые не связаны с сокетами напрямую, но обычно используются совместно с ними.

Задания

К заданиям относятся процессы и потоки. Функции библиотеки Pthreads, предназначенные для работы с потоками, будут рассматриваться в следующем разделе. Здесь же речь пойдет о функциях, применяемых к процессам.

fork()

Функция fork() создает новый процесс (независимое задание). Дочерний процесс выполняет тот же программный файл, что и его предок. Необходимо предусмотреть ветвление алгоритма в программе, иначе произойдет дублирование исполняемого кода.

Прототип

```
#include <unistd.h>
pid_t fork(void);
```

Возвращаемое значение

0	Текущее задание является дочерним
>0	Текущее задание является родительским
<0	Дочернее задание не удалось создать; код ошибки содержится в переменной errno

Параметры

(отсутствуют)

Возможные ошибки

EAGAIN	Недостаточно памяти для копирования таблицы страниц родительского задания и создания информационной структуры дочернего задания
ENOMEM	Недостаточно памяти для создания необходимых структур ядра

Пример

```
int PID;
if ( (PID = fork()) == 0 )
{       /*– ПОТОМОК –*/
    /*** выполняем соответствующие действия ***/
    exit(status);
}
else if ( PID > 0 )
{       /*– ПРЕДОК –*/
```

```
    int status;
    /*** выполняем соответствующие действия ***/
    wait(Sstatus); /* эта функция может вызываться
                      в обработчике сигнала SIGCHLD */
}
else  /*- ОШИБКА -*/
    perror("fork() failed");
```

_clone()

Это низкоуровневый системный вызов, предназначенный для создания заданий. Можно непосредственно указывать, какие данные будут совместно использоваться родительским и дочерним заданиями. Эта функция предназначена для настоящих профессионалов, так как любая ошибка может привести к непредсказуемой работе программы.

Прототип

```
iinclude <sched.h>
int  _clone(int  (*fn)(void* arg), void* stacktop,
            int flags, void* arg);
```

Возвращаемое значение

Функция возвращает идентификатор созданного задания. В случае неудачи код ошибки записывается в переменную errno.

Параметры

fn	Указатель на функцию потока, принимающую аргумент типа void*; когда она завершается, операционная система останавливает поток
stacktop	Указатель на вершину стека дочернего задания (самый старший адрес блока данных); этот стек имеет фиксированный размер и не может увеличиваться подобно стеку обычного задания
flags	Набор флагов, определяющих, какие области памяти совместно используются и какой сигнал посылать в случае завершения дочернего задания. Поддерживаются все виды сигналов, и при завершении задания операционная система сгенерирует любой указанный сигнал. Следующие флаги определяют, какие из областей памяти задания будут доступны для совместного использования: * CLONE_VM. Совместное использование области данных между заданиями. Если флаг указан, будут доступны все статические и предварительно инициализированные-переменные, а также блоки, выделенные в куче. В противном случае в дочернем задании будет создана копия области данных; * CLONE_FS. Совместное использование информации о файловой системе: о текущем каталоге, корневом каталоге и стандартном режиме доступа к файлам (значение umask). Если флаг не указан, задания будут вести себя независимо друг от друга; * CLONE FILES. Совместное использование открытых файлов. Когда в одном задании перемещается указатель текущей позиции файла, в другом задании отразится это изменение, и если закрыть файл в одном задании, то и в другом он станет недоступным. Если флаг не

указан, в дочернем задании создаются новые ссылки на открытые индексные дескрипторы;

* CLONE_SIGHAND. Совместное использование таблиц сигналов. Каждое задание может запретить обработку того или иного сигнала с помощью функции sigprocmask( ) , и это не отразится на других заданиях. Если флаг не указан, в дочернем задании создается копия таблицы сигналов;

* CLONE_PID. Совместное использование идентификатора процесса. Применять данный флаг следует осторожно, так как он не всегда поддерживается (как это имеет место в случае библиотеки Pthreads). Если флаг не указан, в дочернем задании создается новый идентификатор процесса

arg	Указатель на блок данных, передаваемых в качестве параметра потоковой функции fn. Эти данные должны находиться в совместно используемой области памяти

Возможные ошибки

EAGAIN	Недостаточно памяти для копирования таблицы страниц родительского задания и создания информационной структуры дочернего задания
ENOMEM	Недостаточно памяти для создания необходимых структур ядра

Пример

```
fdefine STACKSIZE   1024

void Child(void *arg)
{
   /* код потомка */
   exit(0);
}


int main(void)
{  int cchild;
   char *stack = malloc( STACKSIZE);

   if ( (cchild = _clone(&Child, stack+STACKSIZE-1,
                         SIGCHLD, 0)) == 0 )
   { /*** секция дочернего задания — недоступна ***/
    }
   else if ( cchild > 0 )
       wait();
   else
       perror("Can't clone task");
}
```

exec()

Функции данного семейства предназначены для запуска внешних программ (это могут быть либо двоичные исполняемые файлы, либо сценарии, начинающиеся со строки вида #! <*интерпретатор*> [аргументы]). Функция замещает контекст текущего выполняемого задания контекстом внешней программы, которая сохраняет идентификатор запустившего ее процесса и список открытых файлов.

В семейство входят функции execl(), execlp(), execle(), execv() и execvp(), которые являются надстройками к основной функции execve().

Прототип

```
#include <unistd.h>
int execve(const char* path, char* const argv[],
           char* const envp[]);
int execl(const char* path, const char* argv, ...);
int execlp(const char* file, const char* argv, ...);
int execle(const char* path, const char* argv, ...,
           char* const envp[]);
int execv(const char* path, char* const argv[]);
int execvp(const char* file, char* const argv[]);
```

Возвращаемое значение

В случае успешного завершения ни одна из функций семейства не возвращается в программу. Если же произошла ошибка, возвращается значение —1.

Параметры

file	Имя исполняемого файла; функция ищет программу, просматривая список каталогов, указанный в переменной среды PATH
path	Полное путевое имя исполняемого файла
argv	Массив аргументов командной строки исполняемой программы; первым аргументом всегда является имя программы, а последним — значение NULL (0)
arg	Аргумент командной строки исполняемой программы; если за ним следует многоточие **(...)**. то есть и другие аргументы; первым всегда указывается имя программы, а последним — значение NULL (0)
envp	Массив с описанием переменных среды; каждый элемент массива имеет формат *<имя>=<значение>* (например, TERM=vt100); последний элемент массива всегда имеет значение NULL (0)

Возможные ошибки

EACCES	Отсутствуют права на запуск программы или интерпретатора, нет доступа к одному из каталогов в путевом имени либо файловая система смонтирована с указанием параметра noexec
EPERM	Для файла установлен бит SUID или SGID, а файловая система смонтирована с указанием параметра nosuid либо нет прав суперпользователя
E2BIG	Список аргументов слишком велик
ENOEXEC	Исполняемый файл имеет незнакомый формат, предназначен для другой архитектуры или не может быть выполнен по какой-то иной причине
EFAULT	Имя файла находится в недоступном адресном пространстве
ENAMETOOLONG	Имя файла слишком велико
ENOENT	Файл, сценарий или интерпретатор отсутствует

ENOMEM	Недостаточно памяти ядра для завершения операции
ENOTDIR	Компонент путевого имени файла, сценария или интерпретатора не является каталогом
ELOOP	При разрешении путевого имени обнаружено слишком много символических ссылок
ETXTBUSY	Исполняемый файл открыт для записи несколькими процессами
EIO	Ошибка ввода-вывода
ENFILE	Достигнут лимит числа открытых файлов в системе
EMFILE	Достигнут лимит числа открытых файлов в процессе
EINVAL	В исполняемом файле имеется несколько сегментов PT_INTERP
EISDIR	Указанное имя интерпретатора относится к каталогу
ELIBBAD	Интерпретатор имеет незнакомый формат

Пример

```
execl("/bin/ls", "/bin/ls", "-al", "/home", "/boot", 0);
perror("execl() failed"); /* проверка IF не нужна: при успешном
            завершении функция не передает управление программе */

char *args[] = {"ls", "-al", "/home", "/boot", 0};
execvp(args[0], args);
perror("execvp() failed");
```

sched_yield()

Функция sched_yield() отдает контроль над процессором без блокирования. Она сообщает планировщику о том, что текущее выполняемое задание хочет отказаться от оставшейся части выделенного ему процессорного времени. Функция завершится, когда программе будет предоставлен следующий временной интервал.

Прототип

```
#include <sched.h>
int sched_yield(void);
```

Возвращаемое значение

При успешном завершении функция возвращает 0, в противном случае — -1.

Параметры

(отсутствуют)

Возможные ошибки

(не определены)

Пример

```
#include <sched.h>
sched_yield();
```

wait(), waitpid()

Функции wait() и waitpid() дожидаются завершения дочернего процесса и уведомляют об этом родительскую программу. Их назначение заключается в том, чтобы предотвратить проникновение процессов-зомби в таблицу процессов и освободить ценные системные ресурсы. Функция wait() ожидает завершения любого процесса. В функции waitpid() можно указать конкретный процесс или группу процессов. Узнать, как завершился потомок, можно с помощью одного из следующих макросов.

- WIFEXITED(статус) — возвращает ненулевое значение, если дочернее задание завершилось успешно.
- WEXITSTATUS(статус) — возвращает младший байт кода завершения дочернего задания, который мог быть задан в функции exit() или операторе return. Этот макрос может быть вызван только в том случае, если макрос WIFEXITED вернул ненулевое значение.
- WIFSIGNALED(*статус*) — возвращает true, если дочернее задание завершилось из-за не перехваченного сигнала.
- WTERMSIG(статус) — возвращает номер сигнала, вызвавшего завершение потомка. Этот макрос может быть вызван только в том случае, если макрос WIFSIGNALED вернул ненулевое значение.
- WIFSTOPPED(статус) — возвращает true, если дочернее задание приостановлено. Это возможно только в том случае, если функция была вызвана с параметром options, равным WUNTRACED.
- WSTOPSIG(статус) — возвращает номер сигнала, вызвавшего останов дочернего задания. Этот макрос может быть вызван только в том случае, если макрос WIFSTOPPED вернул ненулевое значение.

Прототип

```
linclude <sys/types.h>
linclude <sys/wait.h>
PID_t wait(int *status);
PID_t waitpid(PID_t pid, int *status, int options);
```

Возвращаемое значение

Обе функции возвращают идентификатор завершившегося дочернего задания.

Параметры

PID	Указывает, завершения какого процесса следует дождаться: * < -1 — дождаться любого дочернего процесса, чей идентификатор группы равен идентификатору родительского процесса; * == -1 — дождаться любого дочернего процесса (такое поведение соответствует работе функции wait()); * == 0 — дождаться любого дочернего процесса, чей идентификатор группы равен идентификатору группы родительского процесса; * > 0 — дождаться дочернего процесса, чей идентификатор равен заданному
status	В этом параметре возвращается код завершения потомка; если он не равен 0 или NULL, то содержит значение, указанное в функции exit()
options	* WNOHANG. Указывает на то, что функция должна немедленно вернуть значение, если дочернее задание еще не завершилось; * WUNTRACED. Указывает на то, что функция должна вернуть значение, если дочернее задание приостановлено, а его статус не был сообщен

Возможные ошибки

ECHILD	Процесс с указанным идентификатором не существует или не является потомком вызывающего процесса (это может произойти, если в обработчике сигнала SIGCHLD установлен атрибут SIG_IGN)
EINVAL	Указано неправильное значение параметра options
EINTR	Режим WNOHANG не задан, но был получен неблокируемый сигнал или сигнал SIGCHLD; следует повторно вызвать функцию

Пример

```
void sig_child(int signum)  /* этот обработчик дожидается
                       завершения одного дочернего задания */
{  int status;
   wait(&status);
   if ( WIFEXITED(status) )
       printf("Child exited with the value of %d\n",
              WEXITSTATUS(status));
   if ( WIFSIGNALED(status) )
       printf("Child aborted due to signal 1%d\n",
              WTERMSIG(status));
   if ( WIFSTOPPED(status) )
       printf("Child stopped on signal #%d\n"  WSTOPSIG(status));


void sig_child(int signum)  /* этот обработчик удаляет из
                               таблицы процессов всех зомби */

   while ( waitpid(-1, 0, WNOHANG) > 0 );
}
```

Потоки

Ниже описан ряд функций библиотеки Pthreads.

pthread_create()

Функция pthread_create() создает так называемый облегченный процесс ядра — поток. Он начинает выполняться в функции, на которую ссылается указатель start_fn (ей передается аргумент arg). Потоковая функция должна возвращать значение типа void*, но даже если она этого не делает, поток все равно успешно завершается, а его код завершения устанавливается равным NULL.

Прототип

```
#include <pthread.h>
int pthread_create(pthread_t *tchild, pthread_attr_t *attr,
                   void*  (*start_fn)(void *), void *arg);
```

Возвращаемое значение

В случае успешного завершения возвращается положительное значение. Если при создании потока возникли какие-то ошибки, функция возвращает отрицательное число, а код ошибки записывается в переменную errno.

Параметры

tchild	Дескриптор нового потока (передается по ссылке); при успешном завершении функция возвращает здесь дескриптор созданного потока
attr	Набор атрибутов, описывающих поведение нового потока и его взаимодействие с родительской программой
start_fn	Указатель на функцию, содержащую код потока; функция должна возвращать значение типа void*
arg	Параметр, передаваемый потоковой функции; он должен представлять собой ссылку на переменную, не являющуюся стековой и недоступную для совместного использования (если только не планируется блокировать ее)

Возможные ошибки

EAGAIN	Недостаточно системных ресурсов для создания нового потока (число активных потоков превысило значение PTHREAD_THREADS_MAX)

Пример

```
void* child(void *arg)
{
   /* код потомка */
   pthread_exit(arg);  /* завершение потока и возврат параметра */
}
```

```
int main(void)
{  pthread_t tchild;

   if ( pthread_create(&tchild, 0, child, 0) < 0 )
       perror("Can't create thread!");
   /* выполняем соответствующие действия */
   if ( pthread_join(tchild, 0) != 0 )
       perror("Join failed");
```

pthread_join()

Функция pthread_join() напоминает системный вызов wait(), но дожидается завершения дочернего потока.

Прототип

```
#include <pthread.h>
int pthread_join(pthread_t tchild, void **retval);
```

Возвращаемое значение

В случае успешного завершения возвращается положительное значение. Если возникли какие-то ошибки, функция возвращает отрицательное число, а код ошибки записывается в переменную errno.

Параметры

tchild	Дескриптор потока, завершения которого необходимо дождаться
retval	Указатель на код завершения потока (передается по ссылке)

Возможные ошибки

ESRCH	Не найден поток, соответствующий указанному дескриптору
EINVAL	Поток был отсоединен
EDEADLK	Аргумент tchild идентифицирует вызывающий поток

Пример

(см. пример для функции pthread_create())

pthread_exit()

Функция pthread_exit() завершает выполнение текущего потока, возвращая вызывающему потоку аргумент retval. Можно также воспользоваться обычным оператором return.

Прототип

```
#include <pthread.h>
void pthread exit(void *retval);
```

Возвращаемое значение

(отсутствует)

Параметр

retval	Значение, передаваемое вызывающему потоку; необходимо убедиться, что это не стековая переменная

Возможные ошибки

(отсутствуют)

Пример

(см. пример для функции pthread created)

pthread_detach()

Функция pthread_detach() отключает указанный поток от родительского задания. Обычно нужно ждать завершения каждого дочернего процесса и потока. С помощью этой функции можно дать некоторым потокам возможность выполняться независимо.

Прототип

```
#include <pthread.h>
int pthread_detach(pthread_t tchild);
```

Возвращаемое значение

В случае успешного завершения возвращается положительное значение. Если возникли какие-то ошибки, функция возвращает отрицательное число, а код ошибки записывается в переменную errno.

Параметр

tchild	Дескриптор дочернего потока

Возможные ошибки

ESRCH	Не найден поток, соответствующий указанному дескриптору
EINVAL	Поток уже был отсоединен
EDEADLK	Аргумент tchild идентифицирует вызывающий поток

Пример

```
void* child(void *arg)

   /* код потомка */
   pthread exit(arg); /* завершение потока и возврат параметра */


int main(void)
{   pthread_t tchild;

   if ( pthread_create(&tchild, 0, child, 0) < 0 )
       perror("Can't create thread!");
   else
       pthread_detach(tchild);
   /* выполняем требуемые действия */
}
```

Блокировка

Основное преимущество потоков заключается в возможности совместного использования данных. Поскольку потоки могут одновременно обращаться к одной и той же области памяти, иногда необходимо блокировать память для обеспечения монопольного доступа. Ниже описываются функции, используемые при создании блокировок.

pthread_mutex_init(), pthread_mutex_destroy()

Функции pthread_mutex_init() и pthread_rautex_destroy() создают и уничтожают объекты исключающих семафоров. Необходимость в инициализирующей функции возникает редко, так как можно работать с готовыми семафорами. Функция pthread_mutex_destroy() уничтожает любые ресурсы, связанные с исключающим семафором, но, поскольку в Linux таковые отсутствуют, функция всего лишь проверяет, разблокирован ли ресурс.

Прототип

```
#include <pthread.h>

/*— Готовые семафоры —*/
pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t recmutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
pthread_mutex_t errchkmutex =
                        PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;

int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *mutexattr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Возвращаемое значение

Всегда нуль.

Параметры

mutex	Дескриптор создаваемого или уничтожаемого семафора
mutexattr	Атрибуты семафора; если этот параметр равен NULL, используются стандартные установки (как у семафора PTHREAD_MUTEX_INITIALIZER)

Возможные ошибки

(отсутствуют)

pthread_mutex_lock(), pthread_mutex_trylock()

Функции pthread_mutex_lock() и pthread_mutex_trylock() соответственно блокируют и пытаются блокировать исключающий семафор, защищающий вход в критическую секцию. Семафор в данном случае выступает в роли своего рода билета на указанное место. Если другой поток пытается занять это место, он блокируется до тех пор, пока предыдущий поток не освободит исключающий семафор.

Прототип

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
intpthread_mutex_trylock(pthread_mutex_t *mutex);
```

Возвращаемое значение

В случае успешного завершения возвращается нуль, при возникновении ошибки — ненулевое значение. Точный код ошибки записывается в переменную errno.

Параметр

mutex	Объект исключающего семафора

Возможные ошибки

EINVAL	Исключающий семафор не был инициализирован должным образом
EDEADLK	(функция pthread_mutex_trylock()) Вызывающий поток уже заблокировал исключающий семафор (данная ошибка выдается только для семафоров, в которых включен режим контроля ошибок)
EBUSY	(функция pthread_mutex_lock()) Вызывающий поток в настоящее время заблокирован

Пример

```
pthread mutex t mutex = fastmutex;
```

```
if ( pthread_mutex_lock(&mutex) == 0 )
{
   /*** работа с критическими данными ***/
   pthread_mutex_unlock(&mutex);
```

```
pthread_mutex_t mutex = fastmutex;

/*— Выполнение других действий в ожидании семафора —*/
while ( pthread_mutex_trylock(&mutex) != 0 && errno == EBUSY )
{
   /*** ожидание семафора ***/
}
/*— Семафор получен! Есть доступ к критической секции —*/
if ( errno != ENOERROR)
{
   /*** работа с критическими данными ***/
   pthread_mutex_unlock(&mutex);
}
```

pthread_mutex_unlock()

Функция pthread_mutex_unlock() разблокирует исключающий семафор.

Прототип

```
#include <pthread.h>
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Возвращаемое значение

В случае успешного завершения возвращается нуль, при возникновении ошибки — ненулевое значение. Точный код ошибки записывается в переменную errno.

Параметр

mutex	Объект исключающего семафора

Возможные ошибки

EINVAL	Исключающий семафор не был инициализирован должным образом
EPERM	Вызывающий поток не является владельцем исключающего семафора (данная ошибка выдается только для семафоров, в которых включен режим контроля ошибок)

Пример

(см. пример для функции pthread_mutex_lock())

Сигналы

При работе с процессами и потоками программа может получать сигналы (или асинхронные уведомления). Ниже описаны системные вызовы, позволяющие перехватывать и обрабатывать сигналы.

signal()

Функция signal() регистрирует функцию sig_fn в качестве обработчика сигнала signum. По умолчанию сигнал посылается однократно (после получения первого сигнала все последующие сигналы направляются стандартному системному обработчику).

Прототип

```
#include <signal.h>
void (*signal(int signum,
              void (*sig_fn)(int signum)))(int signum);
-или-
typedef void (*TSigFn)(int signum);
TSigFn signal(int signum, TSigFn sig_fn);
```

Возвращаемое значение

При успешном завершении возвращается указатель на функцию обработки сигнала, в противном случае возвращается нуль.

Параметры

signum	Номер перехватываемого сигнала
sig_fn	Функция-обработчик, вызываемая планировщиком

Возможные ошибки

(переменная errno не устанавливается)

Пример

```
void sig_handler(int signum)
{
   switch ( signum )
   {
       case SIGFPE:

   }
}

if ( signal(SIGFPE, sig_handler) == 0 )
     perrorf("signal()  failed");
```

sigaction()

Подобно функции signal(), функция sigaction() регистрирует процедуру обработки указанного сигнала. В то же время она предоставляет гораздо больше возможностей в управлении сигналами.

Прототип

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *sigact,
              struct sigaction *oldsigact);
```

Возвращаемое значение

При успешном завершении возвращается нуль, иначе — ненулевое значение.

Параметры

signum — Номер перехватываемого сигнала

sigact — Требуемое действие и функция обработки сигнала; объявление структуры sigaction таково:

```
struct sigaction
{
   void(*sahandler)(int);
   sigset_t sa_mask;
   int sa_flags;
   void (*sa_restorer)(void);
};
```

sa_handler — Указатель на функцию обработки сигнала

sajnask — Список сигналов, которые будут блокироваться во время выполнения обработчика сигналов

sa_flags — Флаги, задающие порядок обработки сигнала. Можно использовать такие константы:

* SA_NOCLDSTOP. Если обрабатывается сигнал SIGCHLD, игнорировать случаи, когда дочернее задание приостанавливает свою работу;
* SA_ONESHOT или SA_RESETHAND. Возвращаться к стандартному системному обработчику после получения первого сигнала;
* SA_RESTART. Пытаться восстанавливать прерванный системный вызов во избежание ошибок типа EINTR;
* SA_NOMASK или SA_NODEFER. Позволять аналогичным сигналам прерывать выполнение обработчика во избежание потерь сигналов

sa_restorer — Устарел и больше не используется

oldsigact — Структура, в которой хранится описание предыдущего выполненного действия

Возможные ошибки

EINVAL	Указан неверный сигнал; эта ошибка генерируется также в том случае, когда делается попытка изменить стандартную процедуру обработки сигналов SIGKILL и SIGSTOP, которые не могут быть перехвачены
EFAULT	Указатель sigact или oldsigact ссылается на область памяти, выходящую за пределы адресного пространства процесса
EINTR	Был прерван системный вызов

Пример

```
void sig_handler(int signum)

   switch ( signum )

       case SIGCHLD:

   }
}
```

```
struct sigaction sigact;
bzero(&sigact,  sizeof(sigact));
sigact.sa_handler = sig_handler; /* задание обработчика */
sigact.sa_flags = SA_NOCLDSTOP | SA_RESTART; /* установки */
if ( sigaction(SIGCHLD, &sigact, 0) == 0 )
    perror("sigaction()   failed");
```

sigprocmask()

Функция sigprocmask() задает, какие сигналы разрешено прерывать при обслуживании текущего сигнала.

Прототип

```
#tinclude<signal.h>
int sigprocmask(int how, const sigset_t *sigset,
                sigset_t *oldsigset);
```

Возвращаемое значение

При возникновении ошибки возвращается ненулевое значение, в противном случае — нуль.

Параметры

how	Определяет, как следует интерпретировать параметр sigset:

* SIG_BLOCK. Набор блокируемых сигналов определяется суммой текущего набора сигналов и элементов, перечисленных в параметре sigset;

* SIG_UNBLOCK. Сигналы, перечисленные в параметре sigset, удаляются из текущего на-

бора блокируемых сигналов, причем разрешается разблокировать сигнал, который не был блокирован;

* SIG_SETMASK. Набор блокируемых сигналов задается равным списку, содержащемуся в параметре sigset

sigset	Искомый набор сигналов
oldsigset	Если этот параметр не равен NULL, в него помещается предыдущий набор сигналов

Возможные ошибки

EFAULT	Указатель sigset или oldsigset ссылается на область памяти, выходящую за пределы адресного пространства процесса
EINTR	Был прерван системный вызов

Работа с файлами

Ниже описаны библиотечные и системные функции, связанные с управлением файлами.

bzero(), memset()

Функция bzero() обнуляет указанную область памяти. Она считается устаревшей, и вместо нее рекомендуется использовать функцию memset(), которая заполняет область памяти заданными значениями.

Прототип

```
linclude <string.h>
void bzero(void *mem, int bytes);
void* memset(void *mem, int val, size_t bytes);
```

Возвращаемое значение

Функция bzero() не возвращает значений. Функция memset() возвращает ссылку на измененную область памяти.

Параметры

mem	Инициализируемая область памяти
val	Значение-заполнитель
bytes	Число записываемых байтов (размер области памяти)

Возможные ошибки

(отсутствуют)

Пример

```
bzero(&addr, sizeof(addr));

memset(&addr, 0, sizeof(addr));
```

fcntl()

Функция fcntl() манипулирует дескриптором файла или сокета.

Прототип

```
linclude <unistd.h>
linclude <fcntl.h>
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *flock);
```

Возвращаемое значение

При наличии ошибки возвращается —1 и устанавливается переменная errno. В случае успешного завершения возвращаемое значение зависит от типа операции.

F_DUPFD	Новый дескриптор
F_GETFD	Значение флага
F_GETFL	Значения флагов
F_GETOWN	Идентификатор владельца дескриптора
F_GETSIG	Номер сигнала, посылаемого, когда становится возможным чтение или запись, либо 0 в случае традиционного обработчика сигнала SIGIO

Для всех остальных команд возвращается нуль.

Параметры

fd	Искомый дескриптор
cmd	Выполняемая операция. Некоторые операции дублируют существующие функции, а для некоторых требуется аргумент (arg или flock). Все операции группируются по назначению: * *Дублирование дескриптора* (F_DUPFD). То же, что и функция dup2 (arg, fd). В этой операции дескриптор fd заменяется копией дескриптора arg; * *Закрытие дескриптора при завершении функции* exec() (F_GETFD, F_SETFD). Ядро не передает все дескрипторы файлов дочернему процессу, созданному с помощью функции exec(). Посредством данных операций можно определить текущий режим работы и задать требуемый режим; * *Манипулирование флагами дескриптора* (F_GETFL, F_SETFL). С помощью данных операций можно узнать флаги дескриптора, заданные с помощью функции open(), а также задать флаги O_APPEND, O_NONBLOCK и O_ASYNC; * *Манипулирование блокировками файла* (F_GETLK, F_SETLK, F_SELKW). В операции F_GETLK возвращается структура блокировки, наложенной на файл.

Если файл не блокирован:

* *Определение владельца сигналов ввода-вывода* (F_GETOWN, F SETOWN). Определение или задание идентификатора текущего процесса-владельца сигнала SIGIO;
* *Определение типа посылаемого сигнала* (F GETSIG, F SETSIG). Определение или задание типа сигнала, если могут быть выполнены дополнительные операции ввода-вывода. По умолчанию это сигнал SIGIO

arg	Устанавливаемое значение
flock	Ключ блокировки

Возможные ошибки

EACCES	Операция запрещена из-за наличия блокировки, удерживаемой другим процессом
EAGAIN	Операция запрещена из-за того, что файл отображается в памяти другим процессом
EBADF	Параметр fd не является дескриптором открытого файла
EDEADLK	Обнаружено, что операция F_SETLKW приведет к взаимоблокировке
EFAULT	Указатель flock ссылается за пределы адресного пространства процесса
EINTR	Если выполнялась операция F_SETLKW, то она была прервана сигналом. Если выполнялась операция F_GETLK или F_SETLK, то она была прервана сигналом до того, как была проверена или получена блокировка. Обычно это происходит при дистанционной блокировке файлов (через NFS)
EINVAL	В случае операции F_DUPFD параметр arg является отрицательным или превышает максимально допустимое значение. В случае операции F_SETSIG параметр arg содержит недопустимый номер сигнала
EMFILE	В случае операции F_DUPFD превышено максимально число файлов, открытых в одном процессе
ENOLCK	Таблица блокировок переполнена или произошла ошибка при создании блокировки по сети
EPERM	Попытка сбросить флаг о APPEND для файла, у которого установлен атрибут "только добавление"

Пример

```
#include <unistd.h>
linclude <fcntl.h>

printf("PID which owns SIGIO: Id", fcntl(fd, F_GETOWN));

if ( fcntl(fd, F_SETSIG, SIGKILL) != 0 )
    perror("Can't set signal");

if ( (fd_copy = fcntl(fd, F_DUPFD)) < 0 )
    perror("Can't dup fd");
```

pipe()

Функция pipe() создает канал, который связан с самим собой: входной дескриптор (fd[0]) совпадает с выходным (fd[1]). При записи данных в канал из него будут прочитаны эти же самые данные.

Прототип

```
#include <unistd.h>
int pipe(int fd[2]);
```

Возвращаемое значение

При успешном завершении возвращается 0, в случае ошибки — -1.

Параметр

fd	Массив из двух целых чисел, куда будут записаны дескрипторы созданного канала

Возможные ошибки

EMFILE	В текущем процессе открыто слишком много файлов
ENFILE	Системная таблица файлов переполнена
EFAULT	Неправильная ссылка на память в указателе fd

Пример

```
int  fd[2];
pipe(fd); /* создание канала */
```

poll()

Функция poll(), как и функция select(), дожидается изменения состояния указанных каналов ввода-вывода. Управление списком дескрипторов осуществляется не посредством макросов, а с помощью специальных структур.

Прототип

```
#include <sys/poll.h>
int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

Возвращаемое значение

Если возвращаемое значение меньше нуля, значит, произошла ошибка. Нулевое значение свидетельствует о том, что истек период ожидания. В случае успешного завершения возвращается число каналов, в которых произошли изменения.

Параметры

ufds — Массив структур **pollfd**. В каждой структуре описывается отдельный дескриптор:

```
struct pollfd
{
    int fd;          /* дескриптор файла */
    short events;  /* запрашиваемые события */
    short revents; /* события, которые
                      произошли в канале */
};
```

В поле fd содержится проверяемый дескриптор файла. Поля events и revents обозначают соответственно проверяемые и произошедшие события. События задаются с помощью следующих констант:

* POLLIN — поступили данные;
* POLLPRI — поступили срочные (внеполосные) данные;
* POLLOUT — канал готов для записи;
* POLLERR — произошла ошибка;
* POLLHUP — отбой на другом конце канала;
* POLLNVAL — неправильный запрос, канал fd не был открыт;
* POLLRDNORM — обычное чтение (только в Linux);
* POLLRDBAND — чтение внеполосных данных (только в Linux);
* POLLWRNORM — обычная запись (только в Linux);
* POLLWRBAND — запись внеполосных данных (только в Linux)

nfds — Число каналов, которые следует проверить

timeout — Период ожидания в миллисекундах; если указано отрицательное значение, функция будет ждать бесконечно долго

Возможные ошибки

ENOMEM — Недостаточно памяти для создания записей в таблице дескрипторов файлов

EFAULT — Указатель ссылается на область памяти, выходящую за пределы адресного пространства процесса

EINTR — Поступил сигнал до того, как произошло одно из запрашиваемых событий

Пример

```
int fd_count=0;
struct pollfd fds[MAXFDs];
fds[fd_count].fd = socket(PF_INET, SOCK_STREAM, 0);
/*** вызовы функций bind() и listen)) ***/
fds[fd_count++].events = POLLIN;
for (;;)
{
   if ( poll(fds, fd_count, TIMEOUT_MS) > 0 )
   {   int i;
       if ( (fds[0].revents & POLLIN) != 0 )
       {
```

```
            fds[fd_count].events = POLLIN | POLLHUP;
            fds[fd_count++].fd = accept(fds[0].fd, 0, 0);
        }
        for ( i = 1; i < fd_count; i++ )
        {
            if ( (fds[i].revents & POLLHUP) != 0 )
            {
                close(fds[i].fd);
                /*** перемещаем дескрипторы для
                     заполнения пустых позиций ***/
                fd_count--;
            }
            else if ( (fds[i].revents & POLLIN) != 0 )
                /*** читаем и обрабатываем данные ***/
        }
    }
```

read()

Функция read() читает указанное число байтов из файла с дескриптором fd и помещает результат в буфер. Эту функцию можно использовать как с сокетами, так и с файлами, но она не обеспечивает такого контроля, как функция recv().

Прототип

```
#include <unistd.h>
int read(int fd, char *buffer, size_t buf_len);
```

Возвращаемое значение

Функция возвращает число прочитанных байтов.

Параметры

fd	Дескриптор файла или сокета
buffer	Буфер, в который будут записываться извлекаемые данные
buf_len	Число прочитанных байтов, а также размер буфера

Возможные ошибки

EINTR	Работа функции была прервана сигналом до того, как началось чтение данных
EAGAIN	Задан режим неблокируемого ввода-вывода (с помощью флага O_NONBLOCK), а данные недоступны
EIO	Ошибка ввода-вывода; она может произойти, если фоновый процесс игнорирует или блокирует сигнал SIGTTIN либо лишился управляющего терминала; возможна также низкоуровневая ошибка чтения с диска или магнитной ленты
EISDIR	Указанный дескриптор связан с каталогом

EBADF	Указан неверный дескриптор файла либо файл не был открыт для чтения
EINVAL	Указанный дескриптор связан с объектом, чтение из которого невозможно
EFAULT	Указатель buffer ссылается на область памяти, находящуюся за пределами адресного пространства процесса

Пример

```
int sockfd;
int bytes_read;
char buffer[1024];
/*— создание сокета и подключение к серверу —*/
if ( (bytes_read = read(sockfd, buffer, sizeof(buffer))) < 0 )
    perror("read");
```

select()

Функция select() ожидает изменения статуса одного из заданных каналов ввода-вывода. Когда в одном из каналов происходит изменение, функция завершается. Имеются четыре макроса, предназначенных для управления набором дескрипторов.

- FD_CLR — удаляет дескриптор из списка.
- FD_SET — добавляет дескриптор в список.
- FD_ISSET — проверяет готовность канала к выполнению операции ввода-вывода.
- FD_ZERO — инициализирует список дескрипторов.

Прототип

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
int select(int hi_fd, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

Возвращаемое значение

При успешном завершении функция возвращает число каналов, в которых произошли изменения. В случае ошибки возвращается отрицательное значение. Если превышен допустимый период ожидания, возвращается нуль.

Параметры

hi_f d	Число, на единицу большее номера самого старшего дескриптора в списке
readfds	Список дескрипторов каналов, из которых осуществляется чтение данных
writefds	Список дескрипторов каналов, в которые осуществляется запись данных
exceptfds	Список дескрипторов каналов, предназначенных для чтения внеполосных сообщений
timeout	Число микросекунд, в течение которых необходимо ждать изменений; параметр представляет собой указатель на число; если это число (не указатель) равно нулю, функция немедленно завершается после проверки всех дескрипторов, если же сам указатель равен NULL (0), то период ожидания отсутствует (функция ждет бесконечно долго)
f d	Дескриптор канала, добавляемого, удаляемого или проверяемого
set	Список дескрипторов

Возможные ошибки

EBADF	В один из списков входит неправильный дескриптор
EINTR	Получен неблокируемый сигнал
EINVAL	Указан отрицательный размер списка
ENOMEM	Не хватает памяти для создания внутренних таблиц

Пример

```
int i, ports[]={9001, 9002, 9004, -1};
int sockfd, max=0;
fd_set set;
struct sockaddr_in addr;
struct timeval timeout={2,500000}; /* 2,5 секунды */

FD_ZERO(&set);
bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY;
for ( i = 0; ports[i] > 0; i

     sockfd = socket(PF_INET, SOCK_STREAM, 0);
     addr.sin_port = htons(ports[i] );
     if ( bind(sockfd, &addr, sizeof(addr)) != 0 )
         perror("bind() failed");
     else
     {
         FD_SET(sockfd,&set);
         if ( max < sockfd )
             max = sockfd;
     }
}
if ( select(max+1, &set, 0, &set, &timeout) > 0 )
{
    for ( i = 0; i <= max; i++)
```

```
        if ( FD_ISSET(i, &set) )
        {   int client = accept(i, 0, 0);
            /*** обрабатываем клиентский запрос ***/
        }
}
```

write()

Функция write() записывает указанное число байтов из буфера в файл с дескриптором fd. Можно работать как с файлами, так и с сокетами, но во втором случае предпочтительнее использовать функцию send().

Прототип

```
#include <unistd.h>
int write(int fd, const void *buffer, size_t msg_len);
```

Возвращаемое значение

Функция возвращает число записанных байтов. Это число может быть меньше, чем значение параметра msg_len. Если функции не удалось за один заход записать требуемое число байтов, ее можно вызывать в цикле до тех пор, пока операция не будет завершена. В случае ошибки возвращается отрицательное значение, а в переменную errno помещается код ошибки.

Параметры

f d	Дескриптор файла или сокета
buffer	Буфер, содержащий сообщение
msg_len	Длина сообщения

Возможные ошибки

EBADF	Указан неверный дескриптор файла либо файл не открыт для записи
EINVAL	Указанный дескриптор связан с объектом, запись в который невозможна
EFAULT	Указатель buffer содержит неправильный адрес
EPIPE	Дескриптор связан с каналом или сокетом, который закрыт на противоположной стороне; в этом случае процессу, осуществляющему запись, посылается сигнал SIGPIPE, и если он его принимает, функция write() генерирует ошибку EPIPE
EAGAIN	Задан режим неблокируемого ввода-вывода (с помощью флага 0_NONBLOCK), а в буфере канала или сокета нет места для немедленной записи данных
EINTR	Функция была прервана сигналом до того, как началась запись данных
ENOSPC	В буфере устройства, содержащего файл с указанным дескриптором, нет места для записи данных
EIO	При модификации индексного дескриптора произошла низкоуровневая ошибка ввода-вывода

Пример

```
/*** Запись сообщения (TCP, UDP, неструктурированное) ***/
int sockfd;
int bytes, bytes_wrote=0;
/*— Создание сокета, подключение к серверу —*/
while ((bytes = write(sockfd, buffer, msg_len)) > 0 )
   if ((bytes_wrote += bytes) >= msg_len)
       break;
if ( bytes < 0 )
    perror("write");
```

close()

Функция close() закрывает любой дескриптор (как файла, так и сокета). Если сокет подключен к серверу или клиенту, канал остается активным после закрытия до тех пор, пока не произойдет очистка буферов или не будет превышен период ожидания. Для каждого процесса устанавливается лимит числа открытых дескрипторов. В Linux 2.2.14 функция getdtablesize() возвращает значение 1024, а в файле /usr/include/linux/limits.h этот параметр определен в константе NR_OPEN.

Прототип

```
#include <unistd.h>
int close(int fd);
```

Возвращаемое значение

В случае успешного завершения возвращается нуль. Если произошла ошибка, ее код записывается в переменную errno.

Параметр

fd	Дескриптор файла или сокета

Возможная ошибка

EBADF	Указан неверный дескриптор файла

Пример

```
int sockfd;
sockfd = socket(PF_INET, SOCK_RAW, htons(99));
if ( sockfd < 0 )
    PANIC("Raw socket create failed");

if ( close(sockfd) != 0 )
    PANIC("Raw socket close failed");
```

Приложение Г

Вспомогательные классы

В этом приложении.

В этом приложении описаны классы, определенные в Java API, а также в пользовательской библиотеке сокетов C++, которую мы создавали в главе 13, "Программирование сокетов в C++" (полное ее описание имеется на Web-узле). Каждый класс сопровождается одним из трех обозначений; *класс* (обычный класс, экземпляры которого можно создавать), *абстрактный класс* (класс, определяющий лишь интерфейс для производных классов) и *подкласс* (родительский класс верхнего уровня, экземпляры которого можно создавать).

Исключения C++

Ниже описаны классы объектов-исключений, определенные в пользовательской библиотеке сокетов.

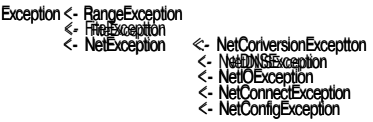

Рис. Г.1. Иерархия классов исключений в C++

Exception (надкласс)

Конструктор:

```
Exception(SimpleStrings);
```

Общее описание: исключение общего характера.
Метод:

```
const char* Getstring();
```
Возвращает строку сообщения

Дочерние исключения:

RangeException	Любое исключение, связанное с выходом за пределы массива; генерируется классом MessageGroup
FileException	Любое исключение, связанное с обработкой файла; генерируется классом Socket

NetException (класс)

Конструктор:

```
NetException(SimpleString s);
```

Общее описание: исключение общего характера при работе в сети.
Родительский класс: Exception
Дочерние исключения:

NetConversionException	Исключение, связанное с преобразованием адреса узла (функции inet_ntop()/inet_pton()); генерируется классом HostAddress
NetDNSException	Исключение, связанное с невозможностью определения имени узла; генерируется классом HostAddress
NetIOException	Исключение при выполнении функций send()/recv(); генерируется классом Socket
NetConnectException	Исключение при выполнении функции bind(), connect(), listen() или accept(); генерируется классами SocketServer, SocketClient и MessageGroup
NetConfigException	Исключение при попытке задания/получения параметров сокета; генерируется всеми классами семейства Socket

Служебные классы C++

Ниже описаны некоторые классы пользовательской библиотеки сокетов, носящие служебный характер. При необходимости их можно заменить стандартными классами C++.

SimpleString (класс)

Конструктор:

```
SimpleString(const char* s);
SimpleString(const SimpleString& s);
```

Общее описание: очень простой строковый тип данных.
Методы:

+ (char *);	Добавляет строку к текущему экземпляру класса
+(SimpleString&);	
const char* GetString();	Возвращает строку сообщения

Генерируемые исключения: (отсутствуют)

HostAddress (класс)

Конструктор:

```
HostAddress(const char* Name=0, ENetwork Network=eIPv4);
HostAddress(HostAddressS Address);
```

Общее описание: класс, предназначенный для управления адресами узлов.
Методы:

void SetPort(int Port);	Задает номер порта
int GetPort(void) const;	Возвращает номер порта,
ENetwork GetNetwork(void) const;	Возвращает тип сети

struct sockaddr* GetAddress(void)const;	Возвращает реальный адрес сокета
int Getsize(void) const;	Возвращает размер адреса сокета
int ==(HostAddresss Address) const;	Проверка на равенство
int !=(HostAddresss Address) const;	Проверка на неравенство
const char* GetHost(bool byName=l);	Возвращает имя узла

Генерируемые исключения:

Exception

NetConversionException

NetDNSException

Классы сообщений C++

Благодаря описанным ,ниже классам можно создавать классы, которые самостоятельно упаковывают и распаковывают собственные данные.

Message (абстрактный класс)

Конструктор: (отсутствует).

Общее описание: шаблон для создания отправляемого или принимаемого сообщения.

Методы:

virtual char* Wrap(int& Bytes) const;	Интерфейс упаковки объектов
bool Unwrap (char* package, int Bytes, int MsgNum);	Интерфейс распаковки объектов

Генерируемые исключения: (отсутствуют).

TextMessage (класс)

Конструктор:

TextMessage(unsignedshortBytes);
TextMessage(const char* Msg, unsigned short Len);

Общее описание: шаблон для создания текстового сообщения.

Родительский класс: Message

Методы:

=(const char* str); =(const TextMessage* s);	Записывает новую строку в объект
+=(const char* str); +=(const TextMessages s);	Добавляет новую строку к объекту
const char* GetBuffer(void) const;	Возвращает строку сообщения
char* Wrapfints Bytes) const;	Упаковывает объект для отправки
bool Unwrap(char* package, int Bytes, int MsgNum);	Распаковывает полученный объект
GetSize(void) const;	Возвращает длину строки
void SetSize(int Bytes);	Задает длину строки
int GetAvailable(void) const;	Возвращает число байтов, доступных в буфере

Генерируемые исключения: (отсутствуют).

Классы сокетов C++

Ниже описаны классы, формирующие интерфейс сокетов. Для непосредственного создания объектов предназначены пять классов: SocketServer, SocketClient, Datagram, Broadcast и MessageGroup. Можно легко расширить иерархию за счет классов библиотеки OpenSSL (SSLServer и SSLClient).

Рис. Г.2. Иерархия класса Socket в C++

Socket (надкласс)

Конструктор:

```
Socket(void);
Socket(int sd);
Socket(ENetwork Network, EProtocol Protocol);
Socket(Sockets sock);
```

Общее описание: класс, содержащий базовые функции работы с сокетами и не предназначенный для прямого создания объектов.

Методы:

void Bind(HostAddress& Addr);	Связывает сокет с портом/интерфейсом
void CloseInput(void) const;	Закрывает входной поток
void CloseOutput(void) const;	Закрывает выходной поток

`int Send(Message& Msg, int Options=0) const;`	Посылает сообщение подключенному узяу
`int Send(HostAddress& Addr, Messages Msg, int Options=0) const;`	Посылает направленное сообщение
`int Receive(Message& Msg, int Options=0) const;`	Принимает сообщение от подключенного узла
`int Receive(HostAddress& Addr, Messages Msg, int Options=0) const;`	Принимает направленное сообщение
`void PermitRoute(bool Setting);`	Разрешает маршрутизацию пакетов
`void KeepAlive(bool Setting);`	Удерживает соединение активным
`void ShareAddress(bool Setting);`	Задает режим совместного использования адреса порта/интерфейса
`int GetReceiveSize(void);` `void SetReceiveSizefint Bytes);`	Возвращает/задает размер входного буфера
`int GetSendSize(void);` `void SetSendSize(int Bytes);`	Возвращает/задает размер выходного буфера
`int GetMinReceive(void);` `void SetMinReceive(int Bytes);`	Возвращает/задает пороговый размер входного буфера для получения сигнала SIGIO
`int GetMinSend(void);` `void SetMinSend(int Bytes);`	Возвращает/задает пороговый размер выходного буфера для получения сигнала SIGIO
`struct timeval GetReceiveTimeout(void);` `void SetReceiveTimeout(struct timevals val);`	Возвращает/задает период ожидания, по истечении которого прием данных будет прерван
`struct timeval GetSendTimeout(void);` `void SetSendTimeout(struct timeval& val);`	Возвращает/задает период ожидания, по истечении которого отправка данных будет прервана
`ENetwork GetType(void);`	Возвращает тип сокета (сети)
`virtual int GetTTL(void);` `virtual void SetTTL(int Hops);`	Возвращает/задает предельное число переходов
`int GetError(void);`	Возвращает сообщение об ошибке, находящееся в очереди

Генерируемые исключения:

NetException
FileException
NetConnectException
NetIOException
NetConf igException

SocketStream (класс)

Конструктор:

```
SocketStream(void);
SocketStrearn(int sd);
SocketStream(ENetwork Network);
SocketStream(SocketStreamssock);
```

Общее описание: класс, определяющий интерфейс потоковых сокетов (SOCK_STREAM).

Родительский класс: Socket

Методы:

int GetMaxSegmentSize(void); void SetMaxSegmentSize(short Bytes);	Возвращает/задает максимальный размер сегмента
void DontDelay(bool Setting);	Включает/отключает алгоритм Нейгла

Генерируемое исключение:

NetConfigException

SocketServer (класс)

Конструктор:

```
SocketServer(int port, ENetwork Network=eIPv4, int QLen=15);
SocketServer(HostAddressS Addr, int QLen=15);
```

Общее описание: TCP-сервер.

Родительский класс: SocketStream

Методы:

void Accept(void (*Servlet) (const Sockets Client));	Принимает запрос на подключение и вызывает функцию Servlet(), передавая ей объект Socket
void Accept(HostAddressS Addr, void (*Servlet)(const Sockets Client));	Принимает запрос на подключение, определяя адрес вызова

Генерируемые исключения:

Exception

NetConnectException

SocketClient (класс)

Конструктор:

```
SocketClient(ENetwork Network=eIPv4);
SocketClient(HostAddress& Host, ENetwork Network=eIPv4);
```

Общее описание: TCP-клиент.
Родительский класс: SocketStream
Метод:

void Connect(HostAddress& Addr);	Подключается к узлу по указанному адресу

Генерируемое исключение:

NetConnectException

Datagram (класс)

Конструктор:

```
Datagram(HostAddress& Me, ENetwork Network=eIPv4,
         EProtocol Protocol=eDatagram);
Datagram(ENetwork Network=eIPv4, EProtocol Protocol=eDatagram);
```

Общее описание: UDP-сокет.
Родительский класс: Socket
Методы:

void MinimizeDelay(bool Setting);	Запрашивает режим минимальной задержки пакета
void MaximizeThroughput(bool Setting);	Запрашивает режим максимальной пропускной способности
void MaximizeReliability(bool Setting);	Запрашивает режим максимальной надежности
void MinimizeCost(bool Setting);	Запрашивает режим минимальной стоимости
void PermitFragNegotiation(EFrag Setting);	Разрешает процедуру фрагментации

Генерируемое исключение:

NetConfigException

Broadcast (класс)

Конструктор:

```
Broadcast(HostAddress& Me);
```

Общее описание: широковещательный сокет, работающий в рамках подсети.
Родительскийкласс:Datagram
Методы: (отсутствуют).

Генерируемое исключение:

NetConfigException .

MessageGroup (класс)

Конструктор:

```
MessageGroup(HostAddress& Me, ENetwork Network=eIPv4);
```

Общее описание: групповой сокет.
Родительский класс: Datagram
Методы;

Connect(HostAddress& Address);	Подключает сокет к адресу группового вещания
void Join(HostAddress& Address, int IFIndex=0)	Регистрирует сокет в адресной группе
void Drop(HostAddress& Address);	Отменяет регистрацию сокета в адресной группе

Генерируемые исключения:

```
NetConfigException
NetConnectException
RangeException
```

Исключения Java

Ниже описаны все исключения, которые Java-программа может сгенерировать при работе с сокетами.

Рис. Г.3. Иерархия классов исключений в Java

java.io.IOException (класс)

Конструктор:

```
IOException();
IOException(String msg);
```

Общее описание: исключение общего характера, произошедшее в процессе ввода-вывода.
Родительский класс: Exception
Дочерние исключения:

java.net.ProtocolException	Ошибка протокола в классе Socket
java.net.UnknownHostException	Имя узла не найдено в базе данных DNS-сервера
java.net.unknownServiceException	Предпринята попытка вызова неподдерживаемого сервиса

java.net.SocketException (класс)

Конструктор:

```
SocketException();
SocketException(String msg);
```

Общее описание: исключение, возникшее при попытке вызова функции bind(), connect(), listen() или accept(); генерируется классами Socket, ServerSocket, DatagramSocket и MulticastSocket.

Родительский класс: IOExceptian

Дочерние исключения:

java.net.BindException	Невозможно осуществить привязку к адресу/порту (как правило, это означает, что он уже используется другим процессом)
java.net.ConnectException	Узел недоступен, не найден, не отвечает или отсутствует процесс, прослушивающий запросы по указанному порту
java.net.NoRouteToHostException	Маршрут к указанному узлу не может быть установлен

Служебные классы Java

Ниже описан ряд служебных классов, часто используемых при работе с сокетами.

java.net.DatagramPacket (класс)

Конструктор:

```
DatagramPacket(byte[] buf, int Деп);
DatagramPacket(byte[] buf, int len, InetAddress addr, int port);
DatagramPacket(byte[] buf, int Offset, int len);
DatagramPacket(byte[] buf,int Offset,int len,InetAddress addr,int port);
```

Общее описание: класс, управляющий массивами, в которые записываются принимаемые/отправляемые дейтаграммы.

Методы:

InetAddress getAddress(); void setAddress(InetAddress addr);	Возвращает/задает адрес отправителя/получателя пакета

byte[] getData();
void SetData(byte[] buf);
void SetData(byte[] buf, int offset, int len);
	Возвращает/задает массив данных дейтаграммы

int getLength(); void setLength(int length);	Возвращает/задает длину данных дейтаграммы

`int getOffset();`	Ворвращает смещение данных в массиве, предназначенном для приёма или отправки
`int getPort();` `void setPort(int port);`	Возвращает/задает порт отпрабителя/получателя пакета

Генерируемые исключения: (отсутствуют).

java.net.InetAddress (класс)

Конструктор: (отсутствует).

Общее описание: класс, предназначенный для работы с адресами Internet; у класса нет конструктора, а его объекты создаются с помощью статических методов.

Статические методы:

InetAddressgetByName(Stringhost);	Возвращает адрес узла, заданный по умолчанию
InetAddress getAHByName(String host);	Возвращает все адреса указанного узла
InetAddress getLocalHost();	Возвращает адрес локального узла

Методы:

`String getHostAddress();`	Возвращает адрес узла в символьном виде
`byte[] getAddress();`	Возвращает адрес узла в виде массива байтов
boolean isMulticastAddress();	Проверяет, попадает ли указанный адрес в диапазон групповых адресов
String getHostName();	Возвращает имя узла

Генерируемое исключение:

UnknownHostException

Классы ввода-вывода Java

В Java имеется огромный набор классов, обрабатывающих различные аспекты ввода-вывода. К сожалению, схема организации этих классов весьма сложна и запутанна. Ниже описаны классы, Имеющие отношение к сокетам.

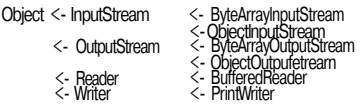

Рис. Г.4. Иерархия классов ввода-вывода в Java

java.io.InputStream (абстрактный класс)

Конструктор:

```
InputStream();
```

Общее описание: простейший входной поток.
Родительский класс: Object
Методы:

`int available();`	Возвращает число байтов, которые можно прочесть без блокирования
`void close();`	Закрывает канал
`void mark(int readlimit);`	Помечает текущую позицию потока для метода `reset()`, задавая максимальный размер буфера упреждающего чтения
`boolean markSupportedf);`	Определяет, поддерживает ли потоковый объект методы `mark()/reset()`
`int read();`	Читает одиночный байт из потока
`int read(byte [] arr);`	Читает массив байтов из потока
`int read(byte [] arr, int offset, int length);`	Читает массив байтов указанного размера, начиная с заданного смещения
`void reset();`	Возвращается к последней помеченной позиции
`long skip();`	Пропускает ближайшие n байтов потока

Генерируемое исключение:

`IOException`

java.io.ByteArraylnputStream (класс)

Конструктор:

```
ByteArrayInputStream(byte[] buf);
ByteArrayInputStream(byte[] buf, int offset, int length);
```

Общее описание: позволяет создавать виртуальный входной поток из массива байтов (например, из дейтаграммы).
Родительский класс: InputStream
Методы: (отсутствуют; много переопределенных методов класса InputStream).
Генерируемые исключения: (отсутствуют).

java.io.ObjectlnputStream (класс)

Конструктор:

```
ObjectInputstream(InputStream o);
```

Общее описание: с помощью этого класса можно читать передаваемые или сохраненные объекты; объект InputStream создается в классе Socket.

Родительский класс: InputStream
Методы:

`int available();`	Возвращает число байтов, которые можно прочесть без блокирования
`void close();`	Закрывает канал
`void defaultReadObject();`	Считывает из потока не статические и не временные поля текущего объекта

`int read();`
`int read(byte[] arr, int offset, int len);`
`int readFully(byte[] arr);`
`int readFully(byte[] arr, int offset, int len);`
	Считывает байт или массив байтов указанного размера, начиная с заданного смещения; метод `readFully()` читает все байты, необходимые для заполнения массива, блокируя программу при необходимости

`boolean readBooleanf);`
`byte readByte();`
`char readChar();`
`double readDouble();`
`float readFloat();`
`int readInt();`
`long readLong();`
`short readShort();`
`int readUnsignedByte();`
`int readUnsignedShort();`
	Читает данные соответствующего типа

`String readUTF();` `Object readObject();`	Читает экземпляр класса Object; можно определить тип объекта и выполнить соответствующую операцию приведения

Генерируемые исключения:

`IOException`
`ClassNotFoundException`
`NotActiveException`
`OptionalDataException`
`InvalidObjectException`
`SecurityException`
`StreamCorruptedException`

java.io.OutputStream (абстрактный класс)

Конструктор:

`OutputStream();`

Общее описание: простейший выходной поток.
Родительский класс: Object
Методы:

void close));	Закрывает канал
void flush();	Выталкивает записанные данные из буферов
void write(byte b);	Записывает одиночный байт в поток
int write(byte[] arr);	Записывает массив байтов в поток
int write(byte[] arr, int offset, int len);	Записывает в поток массив байтов указанного размера, начиная с заданного смещения

Генерируемое исключение:

IOException

java.io.ByteArrayOutputStream (класс)

Конструктор:

```
ByteArrayOutputStreain();
ByteArrayOutputStream(int size);
```

Общее описание: позволяет записывать потоковые данные в массив байтов.
Родительский класс: OutputStream
Методы:

void reset();	Очищает буферы и обнуляет внутренний массив
int write(byte[] arr, int offset, int len);	Записывает массив байтов указанного размера, начиная с заданного смещения
byte[] toByteArray();	Возвращает массив потоковых данных
intsize();	Возвращает текущий размер буфера
String toString(String encoder);	Возвращает внутренние данные в текстовом представлении
void write(int b);	Записывает одиночный байт
void writeTo(OutputStream o);	Передает массив байтов через объект OutputStream'

Генерируемые исключения: (отсутствуют).

java.io.ObjectOutputStream (класс)

Конструктор:

```
ObjectOutputStream(OutputStream o);
```

Общее описание: с помощью этого класса можно передавать и сохранять объекты; объект OutputStream создается в классе Socket.
Родительский класс: OutputStream
Методы:

void close();	Закрывает канал
void defaultwrite0bject();	Записывает в поток не статические и не временные поля текущего объекта; этот метод вызывается только в методе

	writeObject() в процессе сериализации
int flush();	Выталкивает записанные данные из буферов
int *reset();*	Сбрасывает информацию, записанную в поток

void write(byte b);
int write(byte[] arr);
int write(byte[] arr, int offset, int len);
	Записывает одиночный байт или массив байтов указанного размера, начиная с заданной позиции

void writeBoolean(boolean b);
void writeByte(byte b);
void writeBytes(String s);
void writeChar(int c);
void writeChars(String s);
void writeDouble(double d);
void writeFloat(float f);
void writeInt(int i);
void writeLong(long l);
void writeShort(int us);
	Записывает данные соответствующего типа

void writeUTF(String s); int writeFields();	Записывает буферизованные поля в поток
void writeObject(Object o);	Записывает экземпляр класса object

Генерируемые исключения:

IOException

SecurityException

java.io.BufferedReader(класс)

Конструктор:

```
BufferedReader(Reader i);
BufferedReader(Reader i, int size);
```

Общее описание: обеспечивает буферизацию входного потока, благодаря чему Повышается производительность; обладает средствами распознавания строк текста; параметр size задает размер буфера.

Родительский класс: Reader

Методы:

void close();	Закрывает канал
void mark(int readlimit);	Помечает текущую позицию потока, задавая максимальный размер буфера упреждающего чтения
boolean markSupported();	Проверяет, поддерживает ли потоковый объект методы mark()/reset()
int read();	Читает одиночный байт из потока
int read(byte[] arr, int offset, int length);	Читает массив байтов указанного размера, начиная с заданной позиции

`String readLine();`	Читает из потока строку текста (без символа конца строки)
`boolean ready();`	Возвращает true, если имеются данные для чтения
`void reset();`	Возвращается к последней помеченной позиции
`long skip(long n);`	Пропускает ближайшие n байтов потока

Генерируемое исключение:

IOException

java.io.PrintWriter (класс)

Конструктор:

```
PrintWriter(Writer o);
PrintWriter(Writer o, boolean autoFlush);
PrintWriter(OutputStream o);
PrintWriter(OutputStream o, boolean autoFlush);
```

Общее описание: инкапсулирует выходной символьный поток; флаг autoFlush задает автоматическое выталкивание данных из буферов при вызове метода println().

Родительский класс: Writer

Методы:

`boolean checkError();`	Выталкивает содержимое буфера и возвращает true, если произошла ошибка
`void close();`	Закрывает канал
`int flush();`	Выталкивает записанные данные из буферов

`void print(boolean b);`
`void print(char c);`
`void print(char[] s);`
`void print(double d);`
`void print(float f);`
`void print(int i);`
`void print(long l);`
`void print(Object obj);`
`void print(String s);`
	Записывает данные соответствующего типа; к объекту типа Object можно применить метод String.valueOf() для преобразования данных

`void println();`
`void println(boolean b);`
`void println(char c);`
`void println(char[] s);`
`void println(double d);`
`void println(float f);`
`void println(int i);`
`void println(long l);`
`void println(0bject obj);`
`void println(String s);`
	Записывает данные соответствующего типа, добавляя в конце символ новой строки; если установлена опция autoFlush, происходит запись содержимого буфера в поток

`void write(byte b);`	Записывает одиночный байт в поток

int write(byte[] arr);	Записывает массив символов в поток
int write(byte[] arr, int offset, int len);	Записывает в поток массив символов указанного размера, начиная с заданного смещения
int write(String s);	Записывает строку в поток
int write(String s, int offset, int len);	Записывает в поток строку указанного размера, начиная с заданного смещения

Генерируемые исключения:

IOException

SecurityException

Классы сокетов java

В Java имеется четыре класса сокетов IPv4: Socket, ServerSocket, DatagramSocket и MulticastSocket. Ниже описан каждый из них.

java.net.Socket (класс)

Конструктор:

```
Socket(String host, int port);
Socket(InetAddress addr, int port);
Socket(String host, int port, InetAddress lAddr, int lPort);
Socket(InetAddress addr, int port, InetAddress lAddr, int lPort);
```

Общее описание: класс, описывающий базовый интерфейс сетевого взаимодействия (TCP).

Родительский класс: Object

Методы:

void close();	Закрывает сокет
InetAddress getInetAddress();	Возвращает адрес узла на противоположном конце соединения
InputStream getInputstream();	Возвращает потоковый объект InputStream, предназначенный для приёма сообщений
boolean getKeepAlive();	Проверяет, активизирован ли режим поддержания активности соединения
void setKeepAlive(boolean bn);	Удерживает соединение активным
InetAddress getLocalAddress();	Возвращает локальный адрес, к которому подключен сокет
int getbocalPort();	Возвращает номер локального порта
OutputStream getOutputStream();	Возвращает потоковый объект OutputStream, предназначенный для отправки сообщений
int getPortf);	Возвращает номер порта одноранговогo компьютера

`int getReceiveBufferSize();` `void setReceiveBufferSize(int size);`	Возвращает/задает размер входного буфера
`int getSendBufferSize();` `void setSendBufferSize(int size);`	Возвращает/задает размер выходного буфера
`int getSoLinger();` `void setSoLinger(boolean on, int linger);`	Возвращает/задает длительность задержки (в секундах), в течение которой ожидается очистка буферов при закрытии сокета
`int getSoTimeout();` `void setSoTimeout(int timeout);`	Возвращает/задает период ожидания для операций ввода-вывода
`boolean getTcpNoDelay();` `void setTcpNoDelay(boolean on);`	Включает/отключает алгоритм Нейгла, который определяет процедуру отправки данных; если алгоритм отключен, сокет может посылать данные, не дожидаясь получения подтверждений
`void shutdownInput();`	Закрывает входной канал
`void shutdownOutput();`	Закрывает выходной канал

Генерируемые исключения:

IOException

SocketException

java.net.ServerSocket (класс)

Конструктор:

```
ServerSocket(int port);
ServerSocket(int port, int backlog);
ServerSocket(int port, int backlog, InetAddress bindAddr);
```

Общее описание: серверный TCP-сокет, формирующий очередь клиентских запросов.

Родительский класс: Object

Статический метод:

`setSocketPactory(SocketImplFactory fac);`	Регистрирует объект, отвечающий за создание экземпляров сокетов

Методы:

`Socket accept();`	Принимает клиентский запрос и возвращает объект класса Socket
`void close();`	Закрывает сокет
`InetAddress getInetAddress();`	Возвращает локальный адрес, к которому подключен сокет
`int getLocalPort();`	Возвращает номер локального порта
`int getSoTimeout();` `void setSoTimeout(int timeout);`	Возвращает/задает период ожидания для операций ввода-вывода

Генерируемые исключения:

IOException

SocketException

java.net.DatagramSocket(класс)

Конструктор:

```
DatagramSocket();
DatagramSocket(int port);
DatagramSocket(int port, InetAddress bindAddr);
```

Общее описание: UDP-сокет.
Родительский класс: Object
Методы:

void close();	Закрывает сокет
void connect(InetAddress addr, int port);	Подключается к одноранговому компьютеру для неявной отправки сообщений
void disconnect();	Отключается от одноранговогo компьютера
InetAddress getInetAddress();	Возвращает адрес одноранговогo компьютера
InetAddress getLocalAddress();	Возвращает локальный адрес, к которому подключен сокет
int getLocalPort();	Возвращает номер локального порта
int getPort();	Возвращает номер порта одноранговогo компьютера
int getReceiveBufferSize(); void setReceiveBufferSize(int size);	Возвращает/задает размер входного буфера
int getSendBufferSize(); void setSendBufferSize(int size);	Возвращает/задает размер выходного буфера
int getSoTimeout(); void setSoTimeout(int timeout);	Возвращает/задает период ожидания для операций ввода-вывода
void receive(DatagramPacket p);	Принимает сообщение
void send(DatagramPacket p);	Отправляет сообщение

Генерируемые исключения:

IOException

SocketException

java.net.MulticastSocket (класс)

Конструктор:

```
MulticastSocket();
MulticastSocket(int port);
```

Общее описание: групповой UDP-сокет.
Родительский класс: DatagramSocket
Методы:

InetAddress getInterf ace(); void setInterface(InetAddress addr);	Возвращает/задает локальный адрес, к которому подключейсокет
int getTimeToLive(); void setTimeToLive(int TTL);	Возвращает/задает предельное число переходов для каждого сообщения
void joinGroup(InetAddress addr);	Подключает сокет к адресной группе
void leaveGroup(InetAddress addr);	Отключает сокет от адресной группы
void send(DatagramPacket p, ist. TTL);	Посылает сообщение с указанным значением TTL

Генерируемые исключения:

`IOException`

`SocketException`

M

O

P

R

S

T

U

A

Б

В

Г

З

И

K

М

Н

О

П

С

Т

Ф

Ш

Научно-популярное издание

Шон Уолтон

Создание сетевых приложений в среде Linux

Литературный редактор *И.А. Попова*
Верстка *А.В. Говдя*
Художественный редактор *С.А. Чернокозинский*
Технический редактор *Г.Н. Горобец*
Корректоры *Л.А. Гордиенко, Л.В. Коровкина, О.В. Мишутина*

Издательский дом "Вильяме".
101509, Москва, ул. Лесная, д. 43, стр. 1.
Изд. лиц. ЛР № 090230 от 23.06.99
Госкомитета РФ по печати.

Подписано в печать 10.08.2001. Формат 70x100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 33,9. Уч.-изд. л. 27,11.
Тираж 5000 экз. Заказ № 1480.

Отпечатано с диапозитивов в ФГУП "Печатный двор"
Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.