Библиотека Socket API

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 этом методическом пособии последовательно рассказывается о том, как написать простейшее клиентское приложение, подключающееся к некоторому серверу. Так же разбирается, 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). Ниже 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.

Большинство программ приводимых ниже, можно выполнять, не имея подключения к сети, при условии, что сетевые функции ядра ОС 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 Unux содержат все необходимые средства для организации сетевого взаимодействия с использованием адреса обратной

Клиентская программа должна предпринять несколько действий для установления соединения с другим компьютером или сервером. Причем эти действия следует выполнять в определенном порядке. Конечно, возникает вопрос: "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 (Рисунок 1).

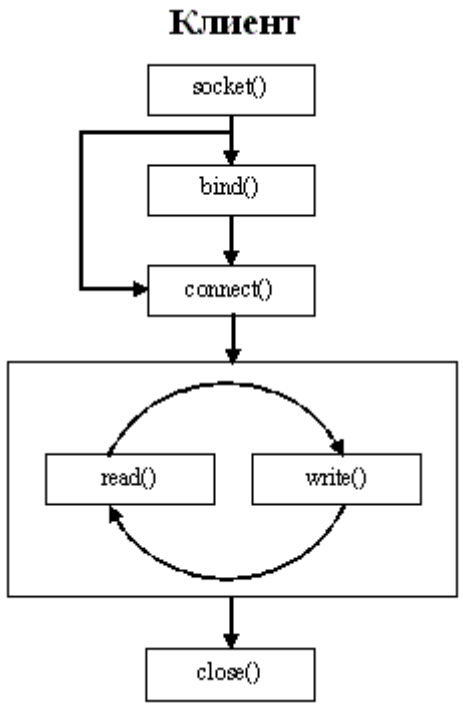

Рисунок 1. Каждый клиент взаимодействует с операционной системой, вызывая определенные функции в заданном порядке

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.

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.

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.

Ниже 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 решении вопросов приема 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 приложений большинство таких проблем вполне можно избежать.

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 по 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. 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). Номер порта чаще отделяется двоеточием, а не точкой.С каждым IP-адресом может быть связано более 65000 портов, через которые подключаются сокеты.

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 Листинг 1. В нем клиент подключается к серверу, получает приветственное сообщение и отключается.

Листинг 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 ореп(), 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 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; стек TCP/IP
type	SOCK_STREAM	Протокол последовательной передачи данных (в виде байтового потока) с подтверждением доставки (TCP)
	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. 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 с 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_STREAM, 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.

Файлы resolv.h и sys/types.h

В примерах 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 Unix и UNIX этот файл исправлен.

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).

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>

#include <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().

Абстрактная структура sockaddr

Структура sockaddr является абстрактной в 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 проверяет первое поле. Следует также отметить, что это единственное поле с серверным порядком следования байтов. 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.

Необходимо обратить внимание что размер массива __pad[] не указан. Ничего неправильного в этом нет — таково общепринятое соглашение. 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 Unix этого не требует).

Листинг 2 Использование функции connect ()

```
/*******************************************************/

/***   Фрагмент программы, демонстрирующий инициализацию   ***/

/***    параметров и вызов функции connect().          ***/

/*******************************************************/

#define PORTJTIME    13

struct sockaddr_in dest;

char *host = "127.0.0.1";

int sd;
```

```
/**** Создание сокета ****/

...

bzero(&dest, 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 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")) == NOLL ) /* 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.

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 написании 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()? 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.

•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 фрагментированных пакетов

Программа может работать гораздо быстрее, чем сеть. Иногда пакеты 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 по-| чему даже при наличии флага MSG_PEEK функция recv() при последовательных вызовах можете 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. В 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:

```
#include <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 (Таблица 2).

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

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().

Пример использования библиотеки Socket API

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.

Ниже мы остановимся на том, как предоставлять сервисы и шаг за шагом рассматрим процесс создания сервера.

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 (Рисунок 2).

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 выше, сейчас она будет описана более формально. Кроме того, будут представлены две новые функции: listen () и accept().

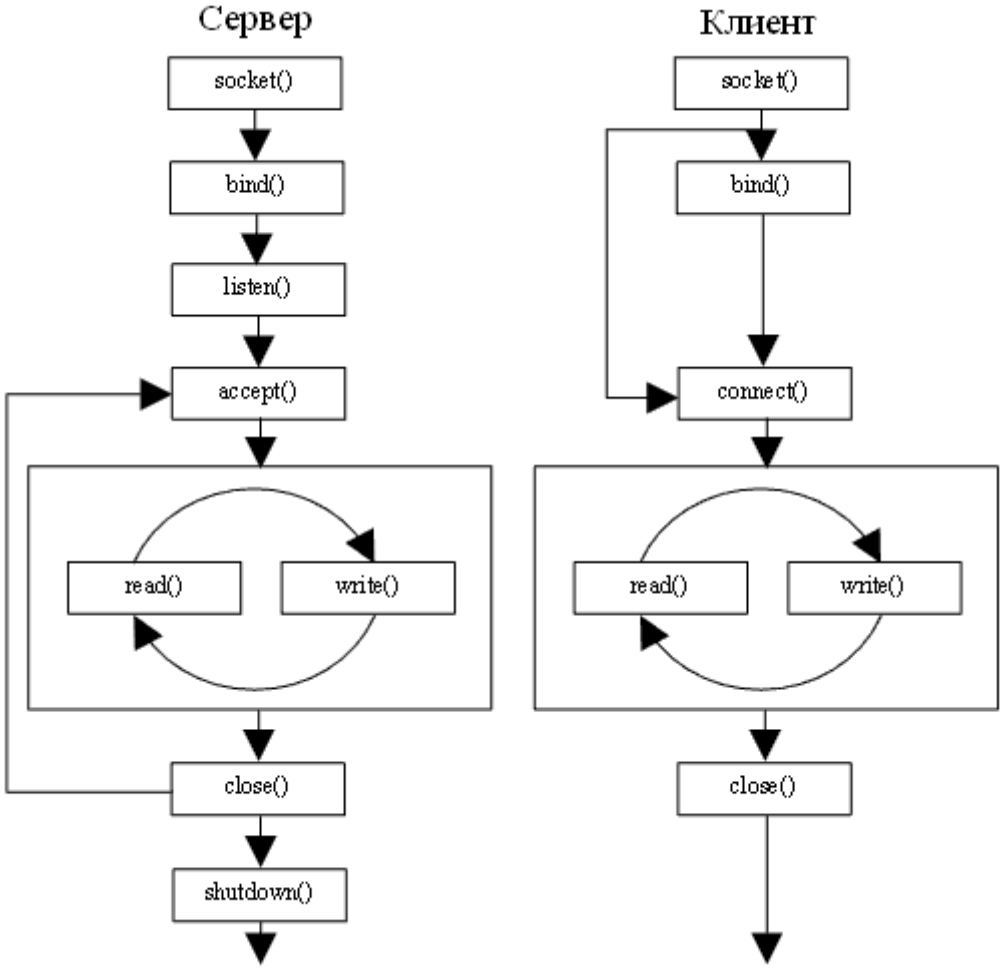

Рисунок 2. Алгоритмы построения клиента и сервера сходны, но схема подключения к сети в них разная

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.

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 выше. Как уже упоминалось, с этого начинается любая сетевая программа. 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 программа 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 программы.

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

Объявление функции bind() выглядит так:

#include <sys/socket.h>

#include <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. выше). 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 3).

Листинг 3. 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, &addr, sizeof(addr)) != 0 ) /* запрашиваем порт */

  perror("Bind AF_INET");
```

В следующем фрагменте программы (Листинг 4) осуществляется инициализация именованного сокета (семейство AF_UNIX или AF_LOCAL).

Листинг 4.Вызов функции bind() в локальном сервере

```
У************************************************/

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

/***   sockaddr_ux                                    ***/

/************************************************/

#include <linux/un.h>

struct sockaddr ux addr; /* создаем локальный именованный сокет */

bzero(&addr, 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>

#include <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 5).

Листинг 5. Пример функции listen ()

```
/*********************************************************/

/***   Пример функции listen(): перевод сокета         ***/

/***  в режим прослушивания клиентских подключений      ***/
```

```
/*****************************************************/

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.

1.EBADF. Указан неверный дескриптор сокета.

2.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. Профамма 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. 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, saddr, &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 7).

Листинг 7.Пример функции accept () с регистрацией подключений.

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

```
/*** (Внутри цикла) ***/

client = accept(sd, &addr, &size);

if ( client > 0 )

{

  if ( addr.sin_faaily == AF_INET)

    printf("Connection[%s]: %s:%d\n",            /* регистрация */

            ctime(tiine(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.

•EOPNOTSOPP. 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 8).

Листинг 8.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) != 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.

■ На главную