56

SOCKETS: INTRODUCTION

Sockets are a method of IPC that allow data to be exchanged between applications, either on the same host (computer) or on different hosts connected by a network. The first widespread implementation of the sockets API appeared with 4.2BSD in 1983, and this API has been ported to virtually every UNIX implementation, as well as most other operating systems.

> The sockets API is formally specified in POSIX.1g, which was ratified in 2000 after spending about a decade as a draft standard. This standard has been superseded by SUSv3.

This chapter and the following chapters describe the use of sockets, as follows:

- This chapter provides a general introduction to the sockets API. The following chapters assume an understanding of the general concepts presented here. We don't present any example code in this chapter. Code examples in the UNIX and Internet domains are presented in the following chapters.
- Chapter 57 describes UNIX domain sockets, which allow communication between applications on the same host system.
- Chapter 58 introduces various computer networking concepts and describes key features of the TCP/IP networking protocols. It provides background needed for the next chapters.
- Chapter 59 describes Internet domain sockets, which allow applications on different hosts to communicate via a TCP/IP network.

- Chapter 60 discusses the design of servers that use sockets.
- Chapter 61 covers a range of advanced topics, including additional features for socket I/O, a more detailed look at the TCP protocol, and the use of socket options to retrieve and modify various attributes of sockets.

These chapters merely aim to give the reader a good grounding in the use of sockets. Sockets programming, especially for network communication, is an enormous topic in its own right, and forms the subject of entire books. Sources of further information are listed in Section 59.15.

56.1 Overview

In a typical client-server scenario, applications communicate using sockets as follows:

- Each application creates a socket. A socket is the "apparatus" that allows communication, and both applications require one.
- The server binds its socket to a well-known address (name) so that clients can locate it.

A socket is created using the *socket()* system call, which returns a file descriptor used to refer to the socket in subsequent system calls:

```
fd = socket(domain, type, protocol);
```

We describe socket domains and types in the following paragraphs. For all applications described in this book, *protocol* is always specified as 0.

Communication domains

Sockets exist in a *communication domain*, which determines:

- the method of identifying a socket (i.e., the format of a socket "address"); and
- the range of communication (i.e., either between applications on the same host or between applications on different hosts connected via a network).

Modern operating systems support at least the following domains:

- The *UNIX* (`AF_UNIX`) domain allows communication between applications on the same host. (POSIX.1g used the name `AF_LOCAL` as a synonym for `AF_UNIX`, but this name is not used in SUSv3.)
- The *IPv4* (`AF_INET`) domain allows communication between applications running on hosts connected via an Internet Protocol version 4 (IPv4) network.
- The *IPv6* (`AF_INET6`) domain allows communication between applications running on hosts connected via an Internet Protocol version 6 (IPv6) network. Although IPv6 is designed as the successor to IPv4, the latter protocol is currently still the most widely used.

Table 56-1 summarizes the characteristics of these socket domains.

In some code, we may see constants with names such as PF_UNIX instead of AF_UNIX. In this context, AF stands for "address family" and PF stands for "protocol family." Initially, it was conceived that a single protocol family might support multiple address families. In practice, no protocol family supporting multiple address families has ever been defined, and all existing implementations define the PF_ constants to be synonymous with the corresponding AF_ constants. (SUSv3 specifies the AF_ constants, but not the PF_ constants.) In this book, we always use the AF_ constants. Further information about the history of these constants can be found in Section 4.2 of [Stevens et al., 2004].

Table 56-1: Socket domains

Domain	Communication performed	Communication between applications	Address format	Address structure
AF_UNIX	within kernel	on same host	pathname	*sockaddr_un*
AF_INET	via IPv4	on hosts connected via an IPv4 network	32-bit IPv4 address + 16-bit port number	*sockaddr_in*
AF_INET6	via IPv6	on hosts connected via an IPv6 network	128-bit IPv6 address + 16-bit port number	*sockaddr_in6*

Socket types

Every sockets implementation provides at least two types of sockets: stream and datagram. These socket types are supported in both the UNIX and the Internet domains. Table 56-2 summarizes the properties of these socket types.

Table 56-2: Socket types and their properties

Property	Socket type	
	Stream	Datagram
Reliable delivery?	Y	N
Message boundaries preserved?	N	Y
Connection-oriented?	Y	N

Stream sockets (SOCK_STREAM) provide a reliable, bidirectional, byte-stream communication channel. By the terms in this description, we mean the following:

- *Reliable* means that we are guaranteed that either the transmitted data will arrive intact at the receiving application, exactly as it was transmitted by the sender (assuming that neither the network link nor the receiver crashes), or that we'll receive notification of a probable failure in transmission.
- *Bidirectional* means that data may be transmitted in either direction between two sockets.
- *Byte-stream* means that, as with pipes, there is no concept of message boundaries (refer to Section 44.1).

A stream socket is similar to using a pair of pipes to allow bidirectional communication between two applications, with the difference that (Internet domain) sockets permit communication over a network.

Stream sockets operate in connected pairs. For this reason, stream sockets are described as *connection-oriented*. The term *peer socket* refers to the socket at the other end of a connection; *peer address* denotes the address of that socket; and *peer application* denotes the application utilizing the peer socket. Sometimes, the term *remote* (or *foreign*) is used synonymously with *peer*. Analogously, sometimes the term *local* is used to refer to the application, socket, or address for this end of the connection. A stream socket can be connected to only one peer.

Datagram sockets (`SOCK_DGRAM`) allow data to be exchanged in the form of messages called *datagrams*. With datagram sockets, message boundaries are preserved, but data transmission is not reliable. Messages may arrive out of order, be duplicated, or not arrive at all.

Datagram sockets are an example of the more generic concept of a *connectionless* socket. Unlike a stream socket, a datagram socket doesn't need to be connected to another socket in order to be used. (In Section 56.6.2, we'll see that datagram sockets may be connected with one another, but this has somewhat different semantics from connected stream sockets.)

In the Internet domain, datagram sockets employ the User Datagram Protocol (UDP), and stream sockets (usually) employ the Transmission Control Protocol (TCP). Instead of using the terms *Internet domain datagram socket* and *Internet domain stream socket*, we'll often just use the terms *UDP socket* and *TCP socket*, respectively.

Socket system calls

The key socket system calls are the following:

- The *socket()* system call creates a new socket.
- The *bind()* system call binds a socket to an address. Usually, a server employs this call to bind its socket to a well-known address so that clients can locate the socket.
- The *listen()* system call allows a stream socket to accept incoming connections from other sockets.
- The *accept()* system call accepts a connection from a peer application on a listening stream socket, and optionally returns the address of the peer socket.
- The *connect()* system call establishes a connection with another socket.

> On most Linux architectures (the exceptions include Alpha and IA-64), all of the sockets system calls are actually implemented as library functions multiplexed through a single system call, *socketcall()*. (This is an artifact of the original development of the Linux sockets implementation as a separate project.) Nevertheless, we refer to all of these functions as system calls in this book, since this is what they were in the original BSD implementation, as well as in many other contemporary UNIX implementations.

Socket I/O can be performed using the conventional *read()* and *write()* system calls, or using a range of socket-specific system calls (e.g., *send()*, *recv()*, *sendto()*, and *recvfrom()*). By default, these system calls block if the I/O operation can't be completed immediately. Nonblocking I/O is also possible, by using the *fcntl()* `F_SETFL` operation (Section 5.3) to enable the `O_NONBLOCK` open file status flag.

> On Linux, we can call *ioctl(fd, FIONREAD, &cnt)* to obtain the number of unread bytes available on the stream socket referred to by the file descriptor *fd*. For a datagram socket, this operation returns the number of bytes in the next unread datagram (which may be zero if the next datagram is of zero length), or zero if there are no pending datagrams. This feature is not specified in SUSv3.

56.2 Creating a Socket: *socket()*

The *socket()* system call creates a new socket.

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
                                Returns file descriptor on success, or –1 on error
```

The *domain* argument specifies the communication domain for the socket. The *type* argument specifies the socket type. This argument is usually specified as either `SOCK_STREAM`, to create a stream socket, or `SOCK_DGRAM`, to create a datagram socket.

The *protocol* argument is always specified as 0 for the socket types we describe in this book. Nonzero *protocol* values are used with some socket types that we don't describe. For example, *protocol* is specified as `IPPROTO_RAW` for raw sockets (`SOCK_RAW`).

On success, *socket()* returns a file descriptor used to refer to the newly created socket in later system calls.

> Starting with kernel 2.6.27, Linux provides a second use for the *type* argument, by allowing two nonstandard flags to be ORed with the socket type. The `SOCK_CLOEXEC` flag causes the kernel to enable the close-on-exec flag (`FD_CLOEXEC`) for the new file descriptor. This flag is useful for the same reasons as the *open()* `O_CLOEXEC` flag described in Section 4.3.1. The `SOCK_NONBLOCK` flag causes the kernel to set the `O_NONBLOCK` flag on the underlying open file description, so that future I/O operations on the socket will be nonblocking. This saves additional calls to *fcntl()* to achieve the same result.

56.3 Binding a Socket to an Address: *bind()*

The *bind()* system call binds a socket to an address.

```
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
                                Returns 0 on success, or –1 on error
```

The *sockfd* argument is a file descriptor obtained from a previous call to *socket()*. The *addr* argument is a pointer to a structure specifying the address to which this socket is to be bound. The type of structure passed in this argument depends on the socket domain. The *addrlen* argument specifies the size of the address structure. The *socklen_t* data type used for the *addrlen* argument is an integer type specified by SUSv3.

Typically, we bind a server's socket to a well-known address—that is, a fixed address that is known in advance to client applications that need to communicate with that server.

> There are other possibilities than binding a server's socket to a well-known address. For example, for an Internet domain socket, the server could omit the call to *bind()* and simply call *listen()*, which causes the kernel to choose an ephemeral port for that socket. (We describe ephemeral ports in Section 58.6.1.) Afterward, the server can use *getsockname()* (Section 61.5) to retrieve the address of its socket. In this scenario, the server must then publish that address so that clients know how to locate the server's socket. Such publication could be done by registering the server's address with a centralized directory service application that clients then contact in order to obtain the address. (For example, Sun RPC solves this problem using its *portmapper* server.) Of course, the directory service application's socket must reside at a well-known address.

56.4 Generic Socket Address Structures: *struct sockaddr*

The *addr* and *addrlen* arguments to *bind()* require some further explanation. Looking at Table 56-1, we see that each socket domain uses a different address format. For example, UNIX domain sockets use pathnames, while Internet domain sockets use the combination of an IP address plus a port number. For each socket domain, a different structure type is defined to store a socket address. However, because system calls such as *bind()* are generic to all socket domains, they must be able to accept address structures of any type. In order to permit this, the sockets API defines a generic address structure, *struct sockaddr*. The only purpose for this type is to cast the various domain-specific address structures to a single type for use as arguments in the socket system calls. The *sockaddr* structure is typically defined as follows:

```
struct sockaddr {
    sa_family_t sa_family;          /* Address family (AF_* constant) */
    char        sa_data[14];        /* Socket address (size varies
                                       according to socket domain) */
};
```

This structure serves as a template for all of the domain-specific address structures. Each of these address structures begins with a *family* field corresponding to the *sa_family* field of the *sockaddr* structure. (The *sa_family_t* data type is an integer type specified in SUSv3.) The value in the *family* field is sufficient to determine the size and format of the address stored in the remainder of the structure.

> Some UNIX implementations also define an additional field in the *sockaddr* structure, *sa_len*, that specifies the total size of the structure. SUSv3 doesn't require this field, and it is not present in the Linux implementation of the sockets API.

If we define the _GNU_SOURCE feature test macro, then *glibc* prototypes the various socket system calls in <sys/socket.h> using a *gcc* extension that eliminates the need for the *(struct sockaddr *)* cast. However, reliance on this feature is nonportable (it will result in compilation warnings on other systems).

56.5 Stream Sockets

The operation of stream sockets can be explained by analogy with the telephone system:

1. The *socket()* system call, which creates a socket, is the equivalent of installing a telephone. In order for two applications to communicate, each of them must create a socket.
2. Communication via a stream socket is analogous to a telephone call. One application must connect its socket to another application's socket before communication can take place. Two sockets are connected as follows:
   a) One application calls *bind()* in order to bind the socket to a well-known address, and then calls *listen()* to notify the kernel of its willingness to accept incoming connections. This step is analogous to having a known telephone number and ensuring that our telephone is turned on so that people can call us.
   b) The other application establishes the connection by calling *connect()*, specifying the address of the socket to which the connection is to be made. This is analogous to dialing someone's telephone number.
   c) The application that called *listen()* then accepts the connection using *accept()*. This is analogous to picking up the telephone when it rings. If the *accept()* is performed before the peer application calls *connect()*, then the *accept()* blocks ("waiting by the telephone").
3. Once a connection has been established, data can be transmitted in both directions between the applications (analogous to a two-way telephone conversation) until one of them closes the connection using *close()*. Communication is performed using the conventional *read()* and *write()* system calls or via a number of socket-specific system calls (such as *send()* and *recv()*) that provide additional functionality.

Figure 56-1 illustrates the use of the system calls used with stream sockets.

Active and passive sockets

Stream sockets are often distinguished as being either active or passive:

- By default, a socket that has been created using *socket()* is *active*. An active socket can be used in a *connect()* call to establish a connection to a passive socket. This is referred to as performing an *active open*.
- A *passive* socket (also called a *listening* socket) is one that has been marked to allow incoming connections by calling *listen()*. Accepting an incoming connection is referred to as performing a *passive open*.

In most applications that employ stream sockets, the server performs the passive open, and the client performs the active open. We presume this scenario in subsequent sections, so that instead of saying "the application that performs the active socket open," we'll often just say "the client." Similarly, we'll equate "the server" with "the application that performs the passive socket open."

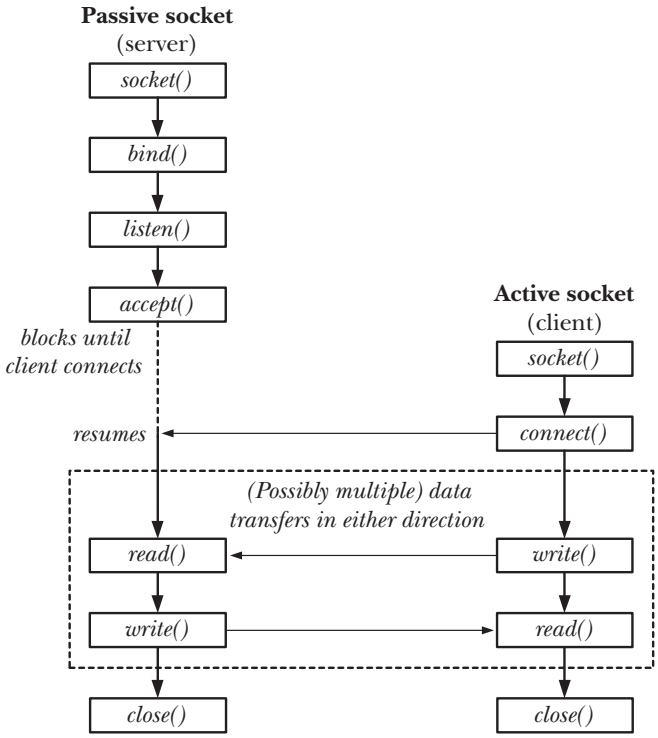

Figure 56-1: Overview of system calls used with stream sockets

56.5.1 Listening for Incoming Connections: *listen()*

The *listen()* system call marks the stream socket referred to by the file descriptor *sockfd* as *passive*. The socket will subsequently be used to accept connections from other (active) sockets.

```
#include <sys/socket.h>

int listen(int sockfd, int backlog);
                                                    Returns 0 on success, or –1 on error
```

We can't apply *listen()* to a connected socket—that is, a socket on which a *connect()* has been successfully performed or a socket returned by a call to *accept()*.

To understand the purpose of the *backlog* argument, we first observe that the client may call *connect()* before the server calls *accept()*. This could happen, for example, because the server is busy handling some other client(s). This results in a *pending connection*, as illustrated in Figure 56-2.

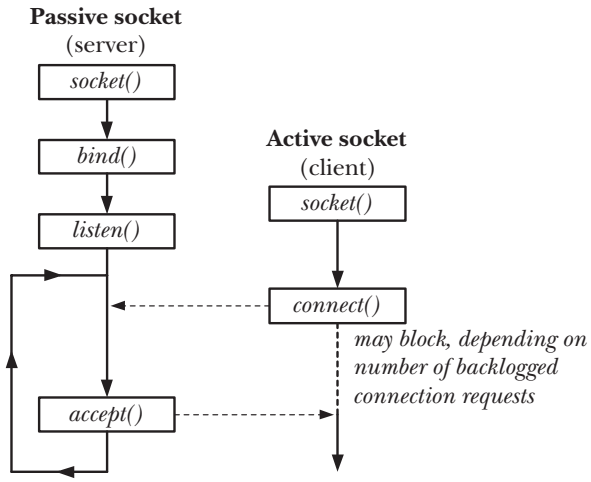

Figure 56-2: A pending socket connection

The kernel must record some information about each pending connection request so that a subsequent *accept()* can be processed. The *backlog* argument allows us to limit the number of such pending connections. Connection requests up to this limit succeed immediately. (For TCP sockets, the story is a little more complicated, as we'll see in Section 61.6.4.) Further connection requests block until a pending connection is accepted (via *accept()*), and thus removed from the queue of pending connections.

SUSv3 allows an implementation to place an upper limit on the value that can be specified for *backlog*, and permits an implementation to silently round *backlog* values down to this limit. SUSv3 specifies that the implementation should advertise this limit by defining the constant `SOMAXCONN` in `<sys/socket.h>`. On Linux, this constant is defined with the value 128. However, since kernel 2.4.25, Linux allows this limit to be adjusted at run time via the Linux-specific `/proc/sys/net/core/somaxconn` file. (In earlier kernel versions, the `SOMAXCONN` limit is immutable.)

> In the original BSD sockets implementation, the upper limit for *backlog* was 5, and we may see this number specified in older code. All modern implementations allow higher values of *backlog*, which are necessary for network servers employing TCP sockets to serve large numbers of clients.

56.5.2 Accepting a Connection: *accept()*

The *accept()* system call accepts an incoming connection on the listening stream socket referred to by the file descriptor *sockfd*. If there are no pending connections when *accept()* is called, the call blocks until a connection request arrives.

```
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
                                    Returns file descriptor on success, or –1 on error
```

The key point to understand about *accept()* is that it creates a *new* socket, and it is this new socket that is connected to the peer socket that performed the *connect()*. A file descriptor for the connected socket is returned as the function result of the *accept()* call. The listening socket (*sockfd*) remains open, and can be used to accept further connections. A typical server application creates one listening socket, binds it to a well-known address, and then handles all client requests by accepting connections via that socket.

The remaining arguments to *accept()* return the address of the peer socket. The *addr* argument points to a structure that is used to return the socket address. The type of this argument depends on the socket domain (as for *bind()*).

The *addrlen* argument is a value-result argument. It points to an integer that, prior to the call, must be initialized to the size of the buffer pointed to by *addr*, so that the kernel knows how much space is available to return the socket address. Upon return from *accept()*, this integer is set to indicate the number of bytes of data actually copied into the buffer.

If we are not interested in the address of the peer socket, then *addr* and *addrlen* should be specified as `NULL` and 0, respectively. (If desired, we can retrieve the peer's address later using the *getpeername()* system call, as described in Section 61.5.)

> Starting with kernel 2.6.28, Linux supports a new, nonstandard system call, *accept4()*. This system call performs the same task as *accept()*, but supports an additional argument, *flags*, that can be used to modify the behavior of the system call. Two flags are supported: `SOCK_CLOEXEC` and `SOCK_NONBLOCK`. The `SOCK_CLOEXEC` flag causes the kernel to enable the close-on-exec flag (`FD_CLOEXEC`) for the new file descriptor returned by the call. This flag is useful for the same reasons as the *open()* `O_CLOEXEC` flag described in Section 4.3.1. The `SOCK_NONBLOCK` flag causes the kernel to enable the `O_NONBLOCK` flag on the underlying open file description, so that future I/O operations on the socket will be nonblocking. This saves additional calls to *fcntl()* to achieve the same result.

56.5.3 Connecting to a Peer Socket: *connect()*

The *connect()* system call connects the active socket referred to by the file descriptor *sockfd* to the listening socket whose address is specified by *addr* and *addrlen*.

```
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Returns 0 on success, or −1 on error

The *addr* and *addrlen* arguments are specified in the same way as the corresponding arguments to *bind()*.

If *connect()* fails and we wish to reattempt the connection, then SUSv3 specifies that the portable method of doing so is to close the socket, create a new socket, and reattempt the connection with the new socket.

56.5.4 I/O on Stream Sockets

A pair of connected stream sockets provides a bidirectional communication channel between the two endpoints. Figure 56-3 shows what this looks like in the UNIX domain.

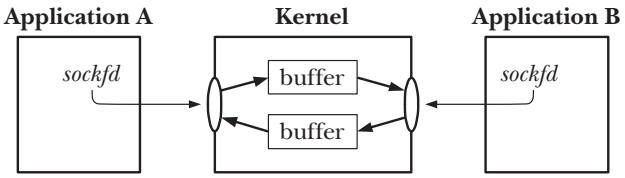

Figure 56-3: UNIX domain stream sockets provide a bidirectional communication channel

The semantics of I/O on connected stream sockets are similar to those for pipes:

- To perform I/O, we use the *read()* and *write()* system calls (or the socket-specific *send()* and *recv()*, which we describe in Section 61.3). Since sockets are bidirectional, both calls may be used on each end of the connection.
- A socket may be closed using the *close()* system call or as a consequence of the application terminating. Afterward, when the peer application attempts to read from the other end of the connection, it receives end-of-file (once all buffered data has been read). If the peer application attempts to write to its socket, it receives a `SIGPIPE` signal, and the system call fails with the error `EPIPE`. As we noted in Section 44.2, the usual way of dealing with this possibility is to ignore the `SIGPIPE` signal and find out about the closed connection via the `EPIPE` error.

56.5.5 Connection Termination: *close()*

The usual way of terminating a stream socket connection is to call *close()*. If multiple file descriptors refer to the same socket, then the connection is terminated when all of the descriptors are closed.

Suppose that, after we close a connection, the peer application crashes or otherwise fails to read or correctly process the data that we previously sent to it. In this case, we have no way of knowing that an error occurred. If we need to ensure that the data was successfully read and processed, then we must build some type of acknowledgement protocol into our application. This normally consists of an explicit acknowledgement message passed back to us from the peer.

In Section 61.2, we describe the *shutdown()* system call, which provides finer control of how a stream socket connection is closed.

56.6 Datagram Sockets

The operation of datagram sockets can be explained by analogy with the postal system:

1. The *socket()* system call is the equivalent of setting up a mailbox. (Here, we assume a system like the rural postal service in some countries, which both picks up letters from and delivers letters to the mailbox.) Each application that wants to send or receive datagrams creates a datagram socket using *socket()*.

2. In order to allow another application to send it datagrams (letters), an application uses *bind()* to bind its socket to a well-known address. Typically, a server binds its socket to a well-known address, and a client initiates communication by sending a datagram to that address. (In some domains—notably the UNIX domain—the client may also need to use *bind()* to assign an address to its socket if it wants to receive datagrams sent by the server.)
3. To send a datagram, an application calls *sendto()*, which takes as one of its arguments the address of the socket to which the datagram is to be sent. This is analogous to putting the recipient's address on a letter and posting it.
4. In order to receive a datagram, an application calls *recvfrom()*, which may block if no datagram has yet arrived. Because *recvfrom()* allows us to obtain the address of the sender, we can send a reply if desired. (This is useful if the sender's socket is bound to an address that is not well known, which is typical of a client.) Here, we stretch the analogy a little, since there is no requirement that a posted letter is marked with the sender's address.
5. When the socket is no longer needed, the application closes it using *close()*.

Just as with the postal system, when multiple datagrams (letters) are sent from one address to another, there is no guarantee that they will arrive in the order they were sent, or even arrive at all. Datagrams add one further possibility not present in the postal system: since the underlying networking protocols may sometimes retransmit a data packet, the same datagram could arrive more than once.

Figure 56-4 illustrates the use of the system calls employed with datagram sockets.

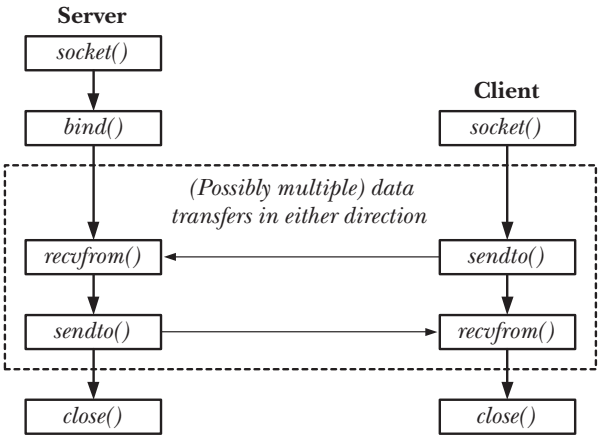

Figure 56-4: Overview of system calls used with datagram sockets

56.6.1 Exchanging Datagrams: *recvfrom()* and *sendto()*

The *recvfrom()* and *sendto()* system calls receive and send datagrams on a datagram socket.

```
#include <sys/socket.h>

ssize_t recvfrom(int sockfd, void *buffer, size_t length, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

Returns number of bytes received, 0 on EOF, or −1 on error

```
ssize_t sendto(int sockfd, const void *buffer, size_t length, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
```

Returns number of bytes sent, or −1 on error

The return value and the first three arguments to these system calls are the same as for *read()* and *write()*.

The fourth argument, *flags*, is a bit mask controlling socket-specific I/O features. We cover these features when we describe the *recv()* and *send()* system calls in Section 61.3. If we don't require any of these features, we can specify *flags* as 0.

The *src_addr* and *addrlen* arguments are used to obtain or specify the address of the peer socket with which we are communicating.

For *recvfrom()*, the *src_addr* and *addrlen* arguments return the address of the remote socket used to send the datagram. (These arguments are analogous to the *addr* and *addrlen* arguments of *accept()*, which return the address of a connecting peer socket.) The *src_addr* argument is a pointer to an address structure appropriate to the communication domain. As with *accept()*, *addrlen* is a value-result argument. Prior to the call, *addrlen* should be initialized to the size of the structure pointed to by *src_addr*; upon return, it contains the number of bytes actually written to this structure.

If we are not interested in the address of the sender, then we specify both *src_addr* and *addrlen* as `NULL`. In this case, *recvfrom()* is equivalent to using *recv()* to receive a datagram. We can also use *read()* to read a datagram, which is equivalent to using *recv()* with a *flags* argument of 0.

Regardless of the value specified for *length*, *recvfrom()* retrieves exactly one message from a datagram socket. If the size of that message exceeds *length* bytes, the message is silently truncated to *length* bytes.

> If we employ the *recvmsg()* system call (Section 61.13.2), then it is possible to find out about a truncated datagram via the `MSG_TRUNC` flag returned in the *msg_flags* field of the returned *msghdr* structure. See the *recvmsg(2)* manual page for details.

For *sendto()*, the *dest_addr* and *addrlen* arguments specify the socket to which the datagram is to be sent. These arguments are employed in the same manner as the corresponding arguments to *connect()*. The *dest_addr* argument is an address structure suitable for this communication domain. It is initialized with the address of the destination socket. The *addrlen* argument specifies the size of *addr*.

> On Linux, it is possible to use *sendto()* to send datagrams of length 0. However, not all UNIX implementations permit this.

56.6.2 Using *connect()* with Datagram Sockets

Even though datagram sockets are connectionless, the *connect()* system call serves a purpose when applied to datagram sockets. Calling *connect()* on a datagram socket causes the kernel to record a particular address as this socket's peer. The term *connected datagram socket* is applied to such a socket. The term *unconnected datagram socket* is applied to a datagram socket on which *connect()* has not been called (i.e., the default for a new datagram socket).

After a datagram socket has been connected:

- Datagrams can be sent through the socket using *write()* (or *send()*) and are automatically sent to the same peer socket. As with *sendto()*, each *write()* call results in a separate datagram.
- Only datagrams sent by the peer socket may be read on the socket.

Note that the effect of *connect()* is asymmetric for datagram sockets. The above statements apply only to the socket on which *connect()* has been called, not to the remote socket to which it is connected (unless the peer application also calls *connect()* on its socket).

We can change the peer of a connected datagram socket by issuing a further *connect()* call. It is also possible to dissolve the peer association altogether by specifying an address structure in which the address family (e.g., the *sun_family* field in the UNIX domain) is specified as `AF_UNSPEC`. Note, however, that many other UNIX implementations don't support the use of `AF_UNSPEC` for this purpose.

> SUSv3 was somewhat vague about dissolving peer associations, stating that a connection can be reset by making a *connect()* call that specifies a "null address," without defining that term. SUSv4 explicitly specifies the use of `AF_UNSPEC`.

The obvious advantage of setting the peer for a datagram socket is that we can use simpler I/O system calls when transmitting data on the socket. We no longer need to use *sendto()* with *dest_addr* and *addrlen* arguments, but can instead use *write()*. Setting the peer is useful primarily in an application that needs to send multiple datagrams to a single peer (which is typical of some datagram clients).

> On some TCP/IP implementations, connecting a datagram socket to a peer yields a performance improvement ([Stevens et al., 2004]). On Linux, connecting a datagram socket makes little difference to performance.

56.7 Summary

Sockets allow communication between applications on the same host or on different hosts connected via a network.

A socket exists within a communication domain, which determines the range of communication and the address format used to identify the socket. SUSv3 specifies the UNIX (`AF_UNIX`), IPv4 (`AF_INET`), and IPv6 (`AF_INET6`) communication domains.

Most applications use one of two socket types: stream or datagram. Stream sockets (`SOCK_STREAM`) provide a reliable, bidirectional, byte-stream communication channel between two endpoints. Datagram sockets (`SOCK_DGRAM`) provide unreliable, connectionless, message-oriented communication.

A typical stream socket server creates its socket using *socket()*, and then binds the socket to a well-known address using *bind()*. The server then calls *listen()* to allow connections to be received on the socket. Each client connection is then accepted on the listening socket using *accept()*, which returns a file descriptor for a new socket that is connected to the client's socket. A typical stream socket client creates a socket using *socket()*, and then establishes a connection by calling *connect()*, specifying the server's well-known address. After two stream sockets are connected, data can be transferred in either direction using *read()* and *write()*. Once all processes with a file descriptor referring to a stream socket endpoint have performed an implicit or explicit *close()*, the connection is terminated.

A typical datagram socket server creates a socket using *socket()*, and then binds it to a well-known address using *bind()*. Because datagram sockets are connectionless, the server's socket can be used to receive datagrams from any client. Datagrams can be received using *read()* or using the socket-specific *recvfrom()* system call, which returns the address of the sending socket. A datagram socket client creates a socket using *socket()*, and then uses *sendto()* to send a datagram to a specified (i.e., the server's) address. The *connect()* system call can be used with a datagram socket to set a peer address for the socket. After doing this, it is no longer necessary to specify the destination address for outgoing datagrams; a *write()* call can be used to send a datagram.

Further information

Refer to the sources of further information listed in Section 59.15.

57

SOCKETS: UNIX DOMAIN

This chapter looks at the use of UNIX domain sockets, which allow communication between processes on the same host system. We discuss the use of both stream and datagram sockets in the UNIX domain. We also describe the use of file permissions to control access to UNIX domain sockets, the use of *socketpair()* to create a pair of connected UNIX domain sockets, and the Linux abstract socket namespace.

57.1 UNIX Domain Socket Addresses: *struct sockaddr_un*

In the UNIX domain, a socket address takes the form of a pathname, and the domain-specific socket address structure is defined as follows:

```
struct sockaddr_un {
    sa_family_t sun_family;         /* Always AF_UNIX */
    char sun_path[108];             /* Null-terminated socket pathname */
};
```

> The prefix *sun_* in the fields of the *sockaddr_un* structure has nothing to do with Sun Microsystems; rather, it derives from *socket unix*.

SUSv3 doesn't specify the size of the *sun_path* field. Early BSD implementations used 108 and 104 bytes, and one contemporary implementation (HP-UX 11) uses 92 bytes. Portable applications should code to this lower value, and use *snprintf()* or *strncpy()* to avoid buffer overruns when writing into this field.

In order to bind a UNIX domain socket to an address, we initialize a *sockaddr_un* structure, and then pass a (cast) pointer to this structure as the *addr* argument to *bind()*, and specify *addrlen* as the size of the structure, as shown in Listing 57-1.

Listing 57-1: Binding a UNIX domain socket

```
const char *SOCKNAME = "/tmp/mysock";
int sfd;
struct sockaddr_un addr;

sfd = socket(AF_UNIX, SOCK_STREAM, 0);            /* Create socket */
if (sfd == -1)
    errExit("socket");

memset(&addr, 0, sizeof(struct sockaddr_un));     /* Clear structure */
addr.sun_family = AF_UNIX;                        /* UNIX domain address */
strncpy(addr.sun_path, SOCKNAME, sizeof(addr.sun_path) - 1);

if (bind(sfd, (struct sockaddr *) &addr, sizeof(struct sockaddr_un)) == -1)
    errExit("bind");
```

The use of the *memset()* call in Listing 57-1 ensures that all of the structure fields have the value 0. (The subsequent *strncpy()* call takes advantage of this by specifying its final argument as one less than the size of the *sun_path* field, to ensure that this field always has a terminating null byte.) Using *memset()* to zero out the entire structure, rather than initializing individual fields, ensures that any nonstandard fields that are provided by some implementations are also initialized to 0.

> The BSD-derived function *bzero()* is an alternative to *memset()* for zeroing the contents of a structure. SUSv3 specifies *bzero()* and the related *bcopy()* (which is similar to *memmove()*), but marks both functions LEGACY, noting that *memset()* and *memmove()* are preferred. SUSv4 removes the specifications of *bzero()* and *bcopy()*.

When used to bind a UNIX domain socket, *bind()* creates an entry in the file system. (Thus, a directory specified as part of the socket pathname needs to be accessible and writable.) The ownership of the file is determined according to the usual rules for file creation (Section 15.3.1). The file is marked as a socket. When *stat()* is applied to this pathname, it returns the value `S_IFSOCK` in the file-type component of the *st_mode* field of the *stat* structure (Section 15.1). When listed with *ls −l*, a UNIX domain socket is shown with the type *s* in the first column, and *ls −F* appends an equal sign (`=`) to the socket pathname.

> Although UNIX domain sockets are identified by pathnames, I/O on these sockets doesn't involve operations on the underlying device.

The following points are worth noting about binding a UNIX domain socket:

- We can't bind a socket to an existing pathname (*bind()* fails with the error `EADDRINUSE`).

- It is usual to bind a socket to an absolute pathname, so that the socket resides at a fixed address in the file system. Using a relative pathname is possible, but unusual, because it requires an application that wants to *connect()* to this socket to know the current working directory of the application that performs the *bind()*.
- A socket may be bound to only one pathname; conversely, a pathname can be bound to only one socket.
- We can't use *open()* to open a socket.
- When the socket is no longer required, its pathname entry can (and generally should) be removed using *unlink()* (or *remove()*).

In most of our example programs, we bind UNIX domain sockets to pathnames in the `/tmp` directory, because this directory is normally present and writable on every system. This makes it easy for the reader to run these programs without needing to first edit the socket pathnames. Be aware, however, that this is generally not a good design technique. As pointed out in Section 38.7, creating files in publicly writable directories such as `/tmp` can lead to various security vulnerabilities. For example, by creating a pathname in `/tmp` with the same name as that used by the application socket, we can create a simple denial-of-service attack. Real-world applications should *bind()* UNIX domain sockets to absolute pathnames in suitably secured directories.

57.2 Stream Sockets in the UNIX Domain

We now present a simple client-server application that uses stream sockets in the UNIX domain. The client program (Listing 57-4) connects to the server, and uses the connection to transfer data from its standard input to the server. The server program (Listing 57-3) accepts client connections, and transfers all data sent on the connection by the client to standard output. The server is a simple example of an *iterative* server—a server that handles one client at a time before proceeding to the next client. (We consider server design in more detail in Chapter 60.)

Listing 57-2 is the header file used by both of these programs.

Listing 57-2: Header file for `us_xfr_sv.c` and `us_xfr_cl.c`

sockets/us_xfr.h

```
#include <sys/un.h>
#include <sys/socket.h>
#include "tlpi_hdr.h"

#define SV_SOCK_PATH "/tmp/us_xfr"

#define BUF_SIZE 100
```

sockets/us_xfr.h

In the following pages, we first present the source code of the server and client, and then discuss the details of these programs and show an example of their use.

Listing 57-3: A simple UNIX domain stream socket server

sockets/us_xfr_sv.c

```
#include "us_xfr.h"

#define BACKLOG 5

int
main(int argc, char *argv[])
{
    struct sockaddr_un addr;
    int sfd, cfd;
    ssize_t numRead;
    char buf[BUF_SIZE];

    sfd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sfd == -1)
        errExit("socket");

    /* Construct server socket address, bind socket to it,
       and make this a listening socket */

    if (remove(SV_SOCK_PATH) == -1 && errno != ENOENT)
        errExit("remove-%s", SV_SOCK_PATH);

    memset(&addr, 0, sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, SV_SOCK_PATH, sizeof(addr.sun_path) - 1);

    if (bind(sfd, (struct sockaddr *) &addr, sizeof(struct sockaddr_un)) == -1)
        errExit("bind");

    if (listen(sfd, BACKLOG) == -1)
        errExit("listen");

    for (;;) {          /* Handle client connections iteratively */

        /* Accept a connection. The connection is returned on a new
           socket, 'cfd'; the listening socket ('sfd') remains open
           and can be used to accept further connections. */

        cfd = accept(sfd, NULL, NULL);
        if (cfd == -1)
            errExit("accept");

        /* Transfer data from connected socket to stdout until EOF */

        while ((numRead = read(cfd, buf, BUF_SIZE)) > 0)
            if (write(STDOUT_FILENO, buf, numRead) != numRead)
                fatal("partial/failed write");

        if (numRead == -1)
            errExit("read");
```

```
        if (close(cfd) == -1)
            errMsg("close");
    }
}
```
sockets/us_xfr_sv.c

Listing 57-4: A simple UNIX domain stream socket client

sockets/us_xfr_cl.c
```
#include "us_xfr.h"

int
main(int argc, char *argv[])
{
    struct sockaddr_un addr;
    int sfd;
    ssize_t numRead;
    char buf[BUF_SIZE];

    sfd = socket(AF_UNIX, SOCK_STREAM, 0);       /* Create client socket */
    if (sfd == -1)
        errExit("socket");

    /* Construct server address, and make the connection */

    memset(&addr, 0, sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, SV_SOCK_PATH, sizeof(addr.sun_path) - 1);

    if (connect(sfd, (struct sockaddr *) &addr,
                sizeof(struct sockaddr_un)) == -1)
        errExit("connect");

    /* Copy stdin to socket */

    while ((numRead = read(STDIN_FILENO, buf, BUF_SIZE)) > 0)
        if (write(sfd, buf, numRead) != numRead)
            fatal("partial/failed write");

    if (numRead == -1)
        errExit("read");

    exit(EXIT_SUCCESS);         /* Closes our socket; server sees EOF */
}
```
sockets/us_xfr_cl.c

The server program is shown in Listing 57-3. The server performs the following steps:

- Create a socket.
- Remove any existing file with the same pathname as that to which we want to bind the socket.

- Construct an address structure for the server's socket, bind the socket to that address, and mark the socket as a listening socket.
- Execute an infinite loop to handle incoming client requests. Each loop iteration performs the following steps:
  - Accept a connection, obtaining a new socket, *cfd*, for the connection.
  - Read all of the data from the connected socket and write it to standard output.
  - Close the connected socket *cfd*.

The server must be terminated manually (e.g., by sending it a signal).

The client program (Listing 57-4) performs the following steps:

- Create a socket.
- Construct the address structure for the server's socket and connect to the socket at that address.
- Execute a loop that copies its standard input to the socket connection. Upon encountering end-of-file in its standard input, the client terminates, with the result that its socket is closed and the server sees end-of-file when reading from the socket on the other end of the connection.

The following shell session log demonstrates the use of these programs. We begin by running the server in the background:

```
$ ./us_xfr_sv > b &
[1] 9866
$ ls -lF /tmp/us_xfr                          Examine socket file with ls
srwxr-xr-x    1 mtk      users         0 Jul 18 10:48 /tmp/us_xfr=
```

We then create a test file to be used as input for the client, and run the client:

```
$ cat *.c > a
$ ./us_xfr_cl < a                             Client takes input from test file
```

At this point, the child has completed. Now we terminate the server as well, and check that the server's output matches the client's input:

```
$ kill %1                                     Terminate server
 [1]+  Terminated   ./us_xfr_sv >b           Shell sees server's termination
$ diff a b
$
```

The *diff* command produces no output, indicating that the input and output files are identical.

Note that after the server terminates, the socket pathname continues to exist. This is why the server uses *remove()* to remove any existing instance of the socket pathname before calling *bind()*. (Assuming we have appropriate permissions, this *remove()* call would remove any type of file with this pathname, even if it wasn't a socket.) If we did not do this, then the *bind()* call would fail if a previous invocation of the server had already created this socket pathname.

57.3 Datagram Sockets in the UNIX Domain

In the generic description of datagram sockets that we provided in Section 56.6, we stated that communication using datagram sockets is unreliable. This is the case for datagrams transferred over a network. However, for UNIX domain sockets, datagram transmission is carried out within the kernel, and is reliable. All messages are delivered in order and unduplicated.

Maximum datagram size for UNIX domain datagram sockets

SUSv3 doesn't specify a maximum size for datagrams sent via a UNIX domain socket. On Linux, we can send quite large datagrams. The limits are controlled via the `SO_SNDBUF` socket option and various `/proc` files, as described in the *socket(7)* manual page. However, some other UNIX implementations impose lower limits, such as 2048 bytes. Portable applications employing UNIX domain datagram sockets should consider imposing a low upper limit on the size of datagrams used.

Example program

Listing 57-6 and Listing 57-7 show a simple client-server application using UNIX domain datagram sockets. Both of these programs make use of the header file shown in Listing 57-5.

Listing 57-5: Header file used by `ud_ucase_sv.c` and `ud_ucase_cl.c`

`sockets/ud_ucase.h`

```
#include <sys/un.h>
#include <sys/socket.h>
#include <ctype.h>
#include "tlpi_hdr.h"

#define BUF_SIZE 10             /* Maximum size of messages exchanged
                                   between client to server */

#define SV_SOCK_PATH "/tmp/ud_ucase"
```

`sockets/ud_ucase.h`

The server program (Listing 57-6) first creates a socket and binds it to a well-known address. (Beforehand, the server unlinks the pathname matching that address, in case the pathname already exists.) The server then enters an infinite loop, using *recvfrom()* to receive datagrams from clients, converting the received text to uppercase, and returning the converted text to the client using the address obtained via *recvfrom()*.

The client program (Listing 57-7) creates a socket and binds the socket to an address, so that the server can send its reply. The client address is made unique by including the client's process ID in the pathname. The client then loops, sending each of its command-line arguments as a separate message to the server. After sending each message, the client reads the server response and displays it on standard output.

Listing 57-6: A simple UNIX domain datagram server

sockets/ud_ucase_sv.c

```
#include "ud_ucase.h"

int
main(int argc, char *argv[])
{
    struct sockaddr_un svaddr, claddr;
    int sfd, j;
    ssize_t numBytes;
    socklen_t len;
    char buf[BUF_SIZE];

    sfd = socket(AF_UNIX, SOCK_DGRAM, 0);       /* Create server socket */
    if (sfd == -1)
        errExit("socket");

    /* Construct well-known address and bind server socket to it */

    if (remove(SV_SOCK_PATH) == -1 && errno != ENOENT)
        errExit("remove-%s", SV_SOCK_PATH);

    memset(&svaddr, 0, sizeof(struct sockaddr_un));
    svaddr.sun_family = AF_UNIX;
    strncpy(svaddr.sun_path, SV_SOCK_PATH, sizeof(svaddr.sun_path) - 1);

    if (bind(sfd, (struct sockaddr *) &svaddr, sizeof(struct sockaddr_un)) == -1)
        errExit("bind");

    /* Receive messages, convert to uppercase, and return to client */

    for (;;) {
        len = sizeof(struct sockaddr_un);
        numBytes = recvfrom(sfd, buf, BUF_SIZE, 0,
                            (struct sockaddr *) &claddr, &len);
        if (numBytes == -1)
            errExit("recvfrom");

        printf("Server received %ld bytes from %s\n", (long) numBytes,
                claddr.sun_path);

        for (j = 0; j < numBytes; j++)
            buf[j] = toupper((unsigned char) buf[j]);

        if (sendto(sfd, buf, numBytes, 0, (struct sockaddr *) &claddr, len) !=
                numBytes)
            fatal("sendto");
    }
}
```

sockets/ud_ucase_sv.c

Listing 57-7: A simple UNIX domain datagram client

sockets/ud_ucase_cl.c

```
#include "ud_ucase.h"

int
main(int argc, char *argv[])
{
    struct sockaddr_un svaddr, claddr;
    int sfd, j;
    size_t msgLen;
    ssize_t numBytes;
    char resp[BUF_SIZE];

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s msg...\n", argv[0]);

    /* Create client socket; bind to unique pathname (based on PID) */

    sfd = socket(AF_UNIX, SOCK_DGRAM, 0);
    if (sfd == -1)
        errExit("socket");

    memset(&claddr, 0, sizeof(struct sockaddr_un));
    claddr.sun_family = AF_UNIX;
    snprintf(claddr.sun_path, sizeof(claddr.sun_path),
            "/tmp/ud_ucase_cl.%ld", (long) getpid());

    if (bind(sfd, (struct sockaddr *) &claddr, sizeof(struct sockaddr_un)) == -1)
        errExit("bind");

    /* Construct address of server */

    memset(&svaddr, 0, sizeof(struct sockaddr_un));
    svaddr.sun_family = AF_UNIX;
    strncpy(svaddr.sun_path, SV_SOCK_PATH, sizeof(svaddr.sun_path) - 1);

    /* Send messages to server; echo responses on stdout */

    for (j = 1; j < argc; j++) {
        msgLen = strlen(argv[j]);       /* May be longer than BUF_SIZE */
        if (sendto(sfd, argv[j], msgLen, 0, (struct sockaddr *) &svaddr,
                sizeof(struct sockaddr_un)) != msgLen)
            fatal("sendto");

        numBytes = recvfrom(sfd, resp, BUF_SIZE, 0, NULL, NULL);
        if (numBytes == -1)
            errExit("recvfrom");
        printf("Response %d: %.*s\n", j, (int) numBytes, resp);
    }

    remove(claddr.sun_path);            /* Remove client socket pathname */
    exit(EXIT_SUCCESS);
}
```

sockets/ud_ucase_cl.c

The following shell session log demonstrates the use of the server and client programs:

```
$ ./ud_ucase_sv &
[1] 20113
$ ./ud_ucase_cl hello world                 Send 2 messages to server
Server received 5 bytes from /tmp/ud_ucase_cl.20150
Response 1: HELLO
Server received 5 bytes from /tmp/ud_ucase_cl.20150
Response 2: WORLD
$ ./ud_ucase_cl 'long message'              Send 1 longer message to server
Server received 10 bytes from /tmp/ud_ucase_cl.20151
Response 1: LONG MESSA
$ kill %1                                   Terminate server
```

The second invocation of the client program was designed to show that when a *recvfrom()* call specifies a *length* (`BUF_SIZE`, defined in Listing 57-5 with the value 10) that is shorter than the message size, the message is silently truncated. We can see that this truncation occurred, because the server prints a message saying it received just 10 bytes, while the message sent by the client consisted of 12 bytes.

57.4 UNIX Domain Socket Permissions

The ownership and permissions of the socket file determine which processes are able to communicate with that socket:

- To connect to a UNIX domain stream socket, write permission is required on the socket file.
- To send a datagram to a UNIX domain datagram socket, write permission is required on the socket file.

In addition, execute (search) permission is required on each of the directories in the socket pathname.

By default, a socket is created (by *bind()*) with all permissions granted to owner (user), group, and other. To change this, we can precede the call to *bind()* with a call to *umask()* to disable the permissions that we do not wish to grant.

Some systems ignore the permissions on the socket file (SUSv3 allows this). Thus, we can't portably use socket file permissions to control access to the socket, although we can portably use permissions on the hosting directory for this purpose.

57.5 Creating a Connected Socket Pair: *socketpair()*

Sometimes, it is useful for a single process to create a pair of sockets and connect them together. This could be done using two calls to *socket()*, a call to *bind()*, and then either calls to *listen()*, *connect()*, and *accept()* (for stream sockets), or a call to *connect()* (for datagram sockets). The *socketpair()* system call provides a shorthand for this operation.

```
#include <sys/socket.h>

int socketpair(int domain, int type, int protocol, int sockfd[2]);
                                                    Returns 0 on success, or –1 on error
```

This *socketpair()* system call can be used only in the UNIX domain; that is, *domain* must be specified as AF_UNIX. (This restriction applies on most implementations, and is logical, since the socket pair is created on a single host system.) The socket *type* may be specified as either SOCK_DGRAM or SOCK_STREAM. The *protocol* argument must be specified as 0. The *sockfd* array returns the file descriptors referring to the two connected sockets.

Specifying *type* as SOCK_STREAM creates the equivalent of a bidirectional pipe (also known as a *stream pipe*). Each socket can be used for both reading and writing, and separate data channels flow in each direction between the two sockets. (On BSD-derived implementations, *pipe()* is implemented as a call to *socketpair()*.)

Typically, a socket pair is used in a similar fashion to a pipe. After the *socketpair()* call, the process then creates a child via *fork()*. The child inherits copies of the parent's file descriptors, including the descriptors referring to the socket pair. Thus, the parent and child can use the socket pair for IPC.

One way in which the use of *socketpair()* differs from creating a pair of connected sockets manually is that the sockets are not bound to any address. This can help us avoid a whole class of security vulnerabilities, since the sockets are not visible to any other process.

> Starting with kernel 2.6.27, Linux provides a second use for the *type* argument, by allowing two nonstandard flags to be ORed with the socket type. The SOCK_CLOEXEC flag causes the kernel to enable the close-on-exec flag (FD_CLOEXEC) for the two new file descriptors. This flag is useful for the same reasons as the *open()* O_CLOEXEC flag described in Section 4.3.1. The SOCK_NONBLOCK flag causes the kernel to set the O_NONBLOCK flag on both underlying open file descriptions, so that future I/O operations on the socket will be nonblocking. This saves additional calls to *fcntl()* to achieve the same result.

57.6 The Linux Abstract Socket Namespace

The so-called *abstract namespace* is a Linux-specific feature that allows us to bind a UNIX domain socket to a name without that name being created in the file system. This provides a few potential advantages:

- We don't need to worry about possible collisions with existing names in the file system.
- It is not necessary to unlink the socket pathname when we have finished using the socket. The abstract name is automatically removed when the socket is closed.
- We don't need to create a file-system pathname for the socket. This may be useful in a *chroot* environment, or if we don't have write access to a file system.

To create an abstract binding, we specify the first byte of the *sun_path* field as a null byte (\0). This distinguishes abstract socket names from conventional UNIX domain socket pathnames, which consist of a string of one or more nonnull bytes terminated by a null byte. The name of the abstract socket is then defined by the remaining bytes (including any null bytes) in *sun_path* up to the length defined for the size of the address structure (i.e., *addrlen − sizeof(sa_family_t)*).

Listing 57-8 demonstrates the creation of an abstract socket binding.

Listing 57-8: Creating an abstract socket binding

from **sockets/us_abstract_bind.c**

```
struct sockaddr_un addr;

memset(&addr, 0, sizeof(struct sockaddr_un));  /* Clear address structure */
addr.sun_family = AF_UNIX;                     /* UNIX domain address */

/* addr.sun_path[0] has already been set to 0 by memset() */

str = "xyz";         /* Abstract name is "\0xyz" */
strncpy(&addr.sun_path[1], str, strlen (str));

sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
if (sockfd == -1)
    errExit("socket");

if (bind(sockfd, (struct sockaddr *) &addr,
        sizeof(sa_family_t) + strlen(str) + 1) == -1)
    errExit("bind");
```

from **sockets/us_abstract_bind.c**

The fact that an initial null byte is used to distinguish an abstract socket name from a conventional socket name can have an unusual consequence. Suppose that the variable *name* happens to point to a zero-length string and that we attempt to bind a UNIX domain socket to a *sun_path* initialized as follows:

```
strncpy(addr.sun_path, name, sizeof(addr.sun_path) - 1);
```

On Linux, we'll inadvertently create an abstract socket binding. However, such a code sequence is probably unintentional (i.e., a bug). On other UNIX implementations, the subsequent *bind()* would fail.

57.7 Summary

UNIX domain sockets allow communication between applications on the same host. The UNIX domain supports both stream and datagram sockets.

A UNIX domain socket is identified by a pathname in the file system. File permissions can be used to control access to a UNIX domain socket.

The *socketpair()* system call creates a pair of connected UNIX domain sockets. This avoids the need for multiple system calls to create, bind, and connect the sockets. A socket pair is normally used in a similar fashion to a pipe: one process creates

the socket pair and then forks to create a child that inherits descriptors referring to the sockets. The two processes can then communicate via the socket pair.

The Linux-specific abstract socket namespace allows us to bind a UNIX domain socket to a name that doesn't appear in the file system.

Further information

Refer to the sources of further information listed in Section 59.15.

57.8 Exercises

57-1. In Section 57.3, we noted that UNIX domain datagram sockets are reliable. Write programs to show that if a sender transmits datagrams to a UNIX domain datagram socket faster than the receiver reads them, then the sender is eventually blocked, and remains blocked until the receiver reads some of the pending datagrams.

57-2. Rewrite the programs in Listing 57-3 (`us_xfr_sv.c`) and Listing 57-4 (`us_xfr_cl.c`) to use the Linux-specific abstract socket namespace (Section 57.6).

57-3. Reimplement the sequence-number server and client of Section 44.8 using UNIX domain stream sockets.

57-4. Suppose that we create two UNIX domain datagram sockets bound to the paths `/somepath/a` and `/somepath/b`, and that we connect the socket `/somepath/a` to `/somepath/b`. What happens if we create a third datagram socket and try to send (*sendto()*) a datagram via that socket to `/somepath/a`? Write a program to determine the answer. If you have access to other UNIX systems, test the program on those systems to see if the answer differs.

58

SOCKETS: FUNDAMENTALS OF TCP/IP NETWORKS

This chapter provides an introduction to computer networking concepts and the TCP/IP networking protocols. An understanding of these topics is necessary to make effective use of Internet domain sockets, which are described in the next chapter.

Starting in this chapter, we begin mentioning various *Request for Comments* (RFC) documents. Each of the networking protocols discussed in this book is formally described in an RFC. We provide further information about RFCs, as well as a list of RFCs of particular relevance to the material covered in this book, in Section 58.7.

58.1 Internets

An *internetwork* or, more commonly, *internet* (with a lowercase *i*), connects different computer networks, allowing hosts on all of the networks to communicate with one another. In other words, an internet is a network of computer networks. The term *subnetwork*, or *subnet*, is used to refer to one of the networks composing an internet. An internet aims to hide the details of different physical networks in order to present a unified network architecture to all hosts on the connected networks. This means, for example, that a single address format is used to identify all hosts in the internet.

Although various internetworking protocols have been devised, TCP/IP has become the dominant protocol suite, supplanting even the proprietary networking

protocols that were formerly common on local and wide area networks. The term *Internet* (with an uppercase *I*) is used to refer to the TCP/IP internet that connects millions of computers globally.

The first widespread implementation of TCP/IP appeared with 4.2BSD in 1983. Several implementations of TCP/IP are derived directly from the BSD code; other implementations, including the Linux implementation, are written from scratch, taking the operation of the BSD code as a reference standard defining the operation of TCP/IP.

> TCP/IP grew out of a project sponsored by the US Department of Defense Advanced Research Projects Agency (ARPA, later DARPA, with the *D* for Defense) to devise a computer networking architecture to be used in the ARPANET, an early wide area network. During the 1970s, a new family of protocols was designed for the ARPANET. Accurately, these protocols are known as the DARPA Internet protocol suite, but more usually they are known as the TCP/IP protocol suite, or simply TCP/IP.
>
> The web page *http://www.isoc.org/internet/history/brief.shtml* provides a brief history of the Internet and TCP/IP.

Figure 58-1 shows a simple internet. In this diagram, the machine `tekapo` is an example of a *router*, a computer whose function is to connect one subnetwork to another, transferring data between them. As well as understanding the internet protocol being used, a router must also understand the (possibly) different data-link-layer protocols used on each of the subnets that it connects.

A router has multiple network interfaces, one for each of the subnets to which it is connected. The more general term *multihomed host* is used for any host—not necessarily a router—with multiple network interfaces. (Another way of describing a router is to say that it is a multihomed host that forwards packets from one subnet to another.) A multihomed host has a different network address for each of its interfaces (i.e., a different address on each of the subnets to which it is connected).

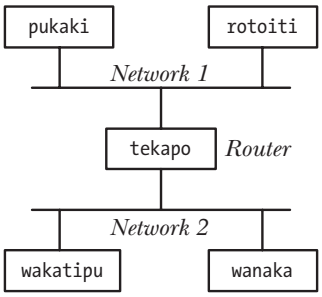

Figure 58-1: An internet using a router to connect two networks

58.2 Networking Protocols and Layers

A *networking protocol* is a set of rules defining how information is to be transmitted across a network. Networking protocols are generally organized as a series of *layers*, with each layer building on the layer below it to add features that are made available to higher layers.

The *TCP/IP protocol suite* is a layered networking protocol (Figure 58-2). It includes the *Internet Protocol* (IP) and various protocols layered above it. (The code that implements these various layers is commonly referred to as a *protocol stack*.) The name TCP/IP derives from the fact that the *Transmission Control Protocol* (TCP) is the most heavily used transport-layer protocol.

> We have omitted a range of other TCP/IP protocols from Figure 58-2 because they are not relevant to this chapter. The *Address Resolution Protocol* (ARP) is concerned with mapping Internet addresses to hardware (e.g., Ethernet) addresses. The *Internet Control Message Protocol* (ICMP) is used to convey error and control information across the network. (ICMP is used by the *ping* program, which is frequently employed to check whether a particular host is alive and visible on a TCP/IP network, and by *traceroute*, which traces the path of an IP packet through the network.) The *Internet Group Management Protocol* (IGMP) is used by hosts and routers that support multicasting of IP datagrams.

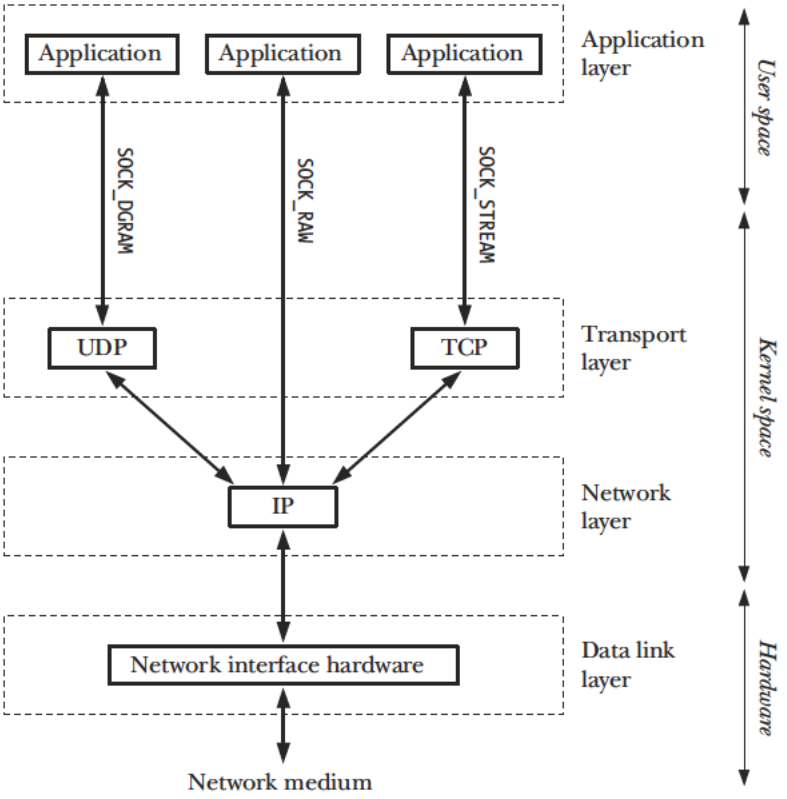

Figure 58-2: Protocols in the TCP/IP suite

One of the notions that lends great power and flexibility to protocol layering is *transparency*—each protocol layer shields higher layers from the operation and complexity of lower layers. Thus, for example, an application making use of TCP only needs to use the standard sockets API and to know that it is employing a reliable, byte-stream transport service. It doesn't need to understand the details of the operation of TCP. (When we look at socket options in Section 61.9, we'll see that this doesn't

always strictly hold true; occasionally, an application does need to know some of the details of the operation of the underlying transport protocol.) Nor does the application need to know the details of the operation of IP or of the data-link layer. From the point of view of the applications, it is as though they are communicating directly with each other via the sockets API, as shown in Figure 58-3, where the dashed horizontal lines represent the virtual communication paths between corresponding application, TCP, and IP entities on the two hosts.

Encapsulation

Encapsulation is an important principle of a layered networking protocol. Figure 58-4 shows an example of encapsulation in the TCP/IP protocol layers. The key idea of encapsulation is that the information (e.g., application data, a TCP segment, or an IP datagram) passed from a higher layer to a lower layer is treated as opaque data by the lower layer. In other words, the lower layer makes no attempt to interpret information sent from the upper layer, but merely places that information inside whatever type of packet is used in the lower layer and adds its own layer-specific header before passing the packet down to the next lower layer. When data is passed up from a lower layer to a higher layer, a converse unpacking process takes place.

> We don't show it in Figure 58-4, but the concept of encapsulation also extends down into the data-link layer, where IP datagrams are encapsulated inside network frames. Encapsulation may also extend up into the application layer, where the application may perform its own packaging of data.

58.3 The Data-Link Layer

The lowest layer in Figure 58-2 is the *data-link layer*, which consists of the device driver and the hardware interface (network card) to the underlying physical medium (e.g., a telephone line, a coaxial cable, or a fiber-optic cable). The data-link layer is concerned with transferring data across a physical link in a network.

To transfer data, the data-link layer encapsulates datagrams from the network layer into units called *frames*. In addition to the data to be transmitted, each frame includes a header containing, for example, the destination address and frame size. The data-link layer transmits the frames across the physical link and handles acknowledgements from the receiver. (Not all data-link layers use acknowledgements.) This layer may perform error detection, retransmission, and flow control. Some data-link layers also split large network packets into multiple frames and reassemble them at the receiver.

From an application-programming point of view, we can generally ignore the data-link layer, since all communication details are handled in the driver and hardware.

One characteristic of the data-link layer that is important for our discussion of IP is the *maximum transmission unit* (MTU). A data-link layer's MTU is the upper limit that the layer places on the size of a frame. Different data-link layers have different MTUs.

> The command *netstat −i* displays a list of the system's network interfaces, along with their MTUs.

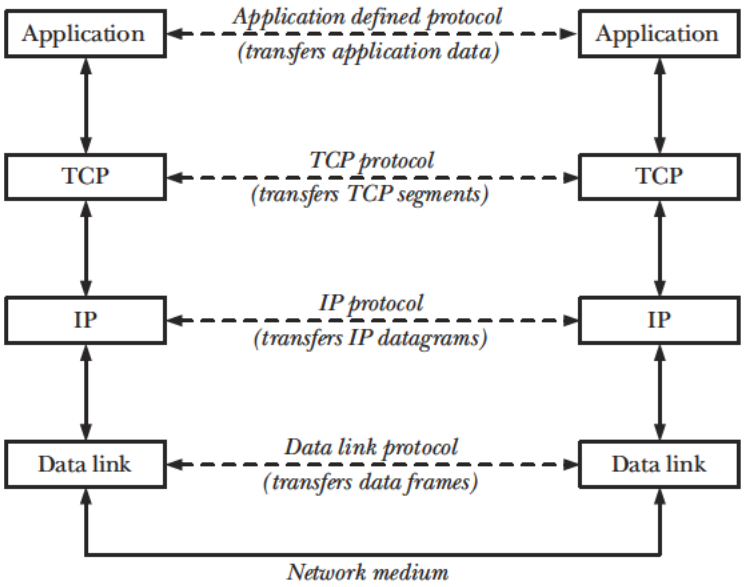

Figure 58-3: Layered communication via the TCP/IP protocols

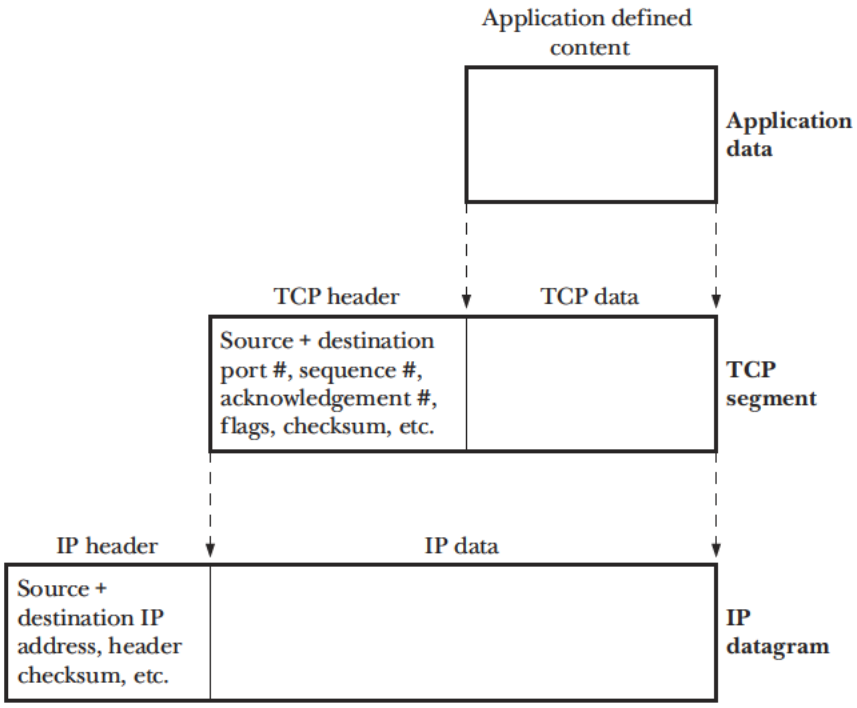

Figure 58-4: Encapsulation within the TCP/IP protocol layers

58.4 The Network Layer: IP

Above the data-link layer is the *network layer*, which is concerned with delivering packets (data) from the source host to the destination host. This layer performs a variety of tasks, including:

- breaking data into fragments small enough for transmission via the data-link layer (if necessary);
- routing data across the internet; and
- providing services to the transport layer.

In the TCP/IP protocol suite, the principal protocol in the network layer is IP. The version of IP that appeared in the 4.2BSD implementation was IP version 4 (IPv4). In the early 1990s, a revised version of IP was devised: IP version 6 (IPv6). The most notable difference between the two versions is that IPv4 identifies subnets and hosts using 32-bit addresses, while IPv6 uses 128-bit addresses, thus providing a much larger range of addresses to be assigned to hosts. Although IPv4 is still the predominant version of IP in use on the Internet, in coming years, it should be supplanted by IPv6. Both IPv4 and IPv6 support the higher UDP and TCP transport-layer protocols (as well as many other protocols).

> Although a 32-bit address space theoretically permits billions of IPv4 network addresses to be assigned, the manner in which addresses were structured and allocated meant that the practical number of available addresses was far lower. The possible exhaustion of the IPv4 address space was one of the primary motivations for the creation of IPv6.
>
> A short history of IPv6 can be found at *http://www.laynetworks.com/IPv6.htm*.
>
> The existence of IPv4 and IPv6 begs the question, "What about IPv5?" There never was an IPv5 as such. Each IP datagram header includes a 4-bit version number field (thus, IPv4 datagrams always have the number 4 in this field), and the version number 5 was assigned to an experimental protocol, *Internet Stream Protocol*. (Version 2 of this protocol, abbreviated as ST-II, is described in RFC 1819.) Initially conceived in the 1970s, this connection-oriented protocol was designed to support voice and video transmission, and distributed simulation. Since the IP datagram version number 5 was already assigned, the successor to IPv4 was assigned the version number 6.

Figure 58-2 shows a *raw* socket type (`SOCK_RAW`), which allows an application to communicate directly with the IP layer. We don't describe the use of raw sockets, since most applications employ sockets over one of the transport-layer protocols (TCP or UDP). Raw sockets are described in Chapter 28 of [Stevens et al., 2004]. One instructive example of the use of raw sockets is the *sendip* program (*http://www.earth.li/projectpurple/progs/sendip.html*), which is a command-line-driven tool that allows the construction and transmission of IP datagrams with arbitrary contents (including options to construct UDP datagrams and TCP segments).

IP transmits datagrams

IP transmits data in the form of datagrams (packets). Each datagram sent between two hosts travels independently across the network, possibly taking a different

route. An IP datagram includes a header, which ranges in size from 20 to 60 bytes. The header contains the address of the target host, so that the datagram can be routed through the network to its destination, and also includes the originating address of the packet, so that the receiving host knows the origin of the datagram.

> It is possible for a sending host to spoof the originating address of a packet, and this forms the basis of a TCP denial-of-service attack known as SYN-flooding. [Lemon, 2002] describes the details of this attack and the measures used by modern TCP implementations to deal with it.

An IP implementation may place an upper limit on the size of datagrams that it supports. All IP implementations must permit datagrams at least as large as the limit specified by IP's *minimum reassembly buffer size*. In IPv4, this limit is 576 bytes; in IPv6, it is 1500 bytes.

IP is connectionless and unreliable

IP is described as a *connectionless* protocol, since it doesn't provide the notion of a virtual circuit connecting two hosts. IP is also an *unreliable* protocol: it makes a "best effort" to transmit datagrams from the sender to the receiver, but doesn't guarantee that packets will arrive in the order they were transmitted, that they won't be duplicated, or even that they will arrive at all. Nor does IP provide error recovery (packets with header errors are silently discarded). Reliability must be provided either by using a reliable transport-layer protocol (e.g., TCP) or within the application itself.

> IPv4 provides a checksum for the IP header, which allows the detection of errors in the header, but doesn't provide any error detection for the data transmitted within the packet. IPv6 doesn't provide a checksum in the IP header, relying on higher-layer protocols to provide error checking and reliability as required. (UDP checksums are optional with IPv4, but generally enabled; UDP checksums are mandatory with IPv6. TCP checksums are mandatory with both IPv4 and IPv6.)
>
> Duplication of IP datagrams may occur because of techniques employed by some data-link layers to ensure reliability or when IP datagrams are tunneled through some non-TCP/IP network that employs retransmission.

IP may fragment datagrams

IPv4 datagrams can be up to 65,535 bytes. By default, IPv6 allows datagrams of up to 65,575 bytes (40 bytes for the header, 65,535 bytes for data), and provides an option for larger datagrams (so-called *jumbograms*).

We noted earlier that most data-link layers impose an upper limit (the MTU) on the size of data frames. For example, this upper limit is 1500 bytes on the commonly used Ethernet network architecture (i.e., much smaller than the maximum size of an IP datagram). IP also defines the notion of the *path MTU*. This is the minimum MTU on all of the data-link layers traversed on the route from the source to the destination. (In practice, the Ethernet MTU is often the minimum MTU in a path.)

When an IP datagram is larger than the MTU, IP fragments (breaks up) the datagram into suitably sized units for transmission across the network. These fragments are then reassembled at the final destination to re-create the original datagram.

(Each IP fragment is itself an IP datagram that contains an offset field giving the location of that fragment within the original datagram.)

IP fragmentation occurs transparently to higher protocol layers, but nevertheless is generally considered undesirable ([Kent & Mogul, 1987]). The problem is that, because IP doesn't perform retransmission, and a datagram can be reassembled at the destination only if all fragments arrive, the entire datagram is unusable if any fragment is lost or contains transmission errors. In some cases, this can lead to significant rates of data loss (for higher protocol layers that don't perform retransmission, such as UDP) or degraded transfer rates (for higher protocol layers that do perform retransmission, such as TCP). Modern TCP implementations employ algorithms (*path MTU discovery*) to determine the MTU of a path between hosts, and accordingly break up the data they pass to IP, so that IP is not asked to transmit datagrams that exceed this size. UDP provides no such mechanism, and we consider how UDP-based applications can deal with the possibility of IP fragmentation in Section 58.6.2.

58.5 IP Addresses

An IP address consists of two parts: a network ID, which specifies the network on which a host resides, and a host ID, which identifies the host within that network.

IPv4 addresses

An IPv4 address consists of 32 bits (Figure 58-5). When expressed in human-readable form, these addresses are normally written in *dotted-decimal notation*, with the 4 bytes of the address being written as decimal numbers separated by dots, as in `204.152.189.116`.

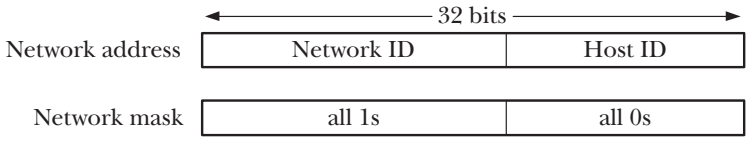

Figure 58-5: An IPv4 network address and corresponding network mask

When an organization applies for a range of IPv4 addresses for its hosts, it receives a 32-bit network address and a corresponding 32-bit *network mask*. In binary form, this mask consists of a sequence of 1s in the leftmost bits, followed by a sequence of 0s to fill out the remainder of the mask. The 1s indicate which part of the address contains the assigned network ID, while the 0s indicate which part of the address is available to the organization to assign as unique host IDs on its network. The size of the network ID part of the mask is determined when the address is assigned. Since the network ID component always occupies the leftmost part of the mask, the following notation is sufficient to specify the range of assigned addresses:

```
204.152.189.0/24
```

The `/24` indicates that the network ID part of the assigned address consists of the leftmost 24 bits, with the remaining 8 bits specifying the host ID. Alternatively, we could say that the network mask in this case is `255.255.255.0` in dotted-decimal notation.

An organization holding this address can assign 254 unique Internet addresses to its computers—204.152.189.1 through 204.152.189.254. Two addresses can't be assigned. One of these is the address whose host ID is all 0 bits, which is used to identify the network itself. The other is the address whose host ID is all 1 bits—204.152.189.255 in this example—which is the *subnet broadcast address*.

Certain IPv4 addresses have special meanings. The special address 127.0.0.1 is normally defined as the *loopback address*, and is conventionally assigned the hostname localhost. (Any address on the network 127.0.0.0/8 can be designated as the IPv4 loopback address, but 127.0.0.1 is the usual choice.) A datagram sent to this address never actually reaches the network, but instead automatically loops back to become input to the sending host. Using this address is convenient for testing client and server programs on the same host. For use in a C program, the integer constant INADDR_LOOPBACK is defined for this address.

The constant INADDR_ANY is the so-called IPv4 *wildcard address*. The wildcard IP address is useful for applications that bind Internet domain sockets on multihomed hosts. If an application on a multihomed host binds a socket to just one of its host's IP addresses, then that socket can receive only UDP datagrams or TCP connection requests sent to that IP address. However, we normally want an application on a multihomed host to be able to receive datagrams or connection requests that specify any of the host's IP addresses, and binding the socket to the wildcard IP address makes this possible. SUSv3 doesn't specify any particular value for INADDR_ANY, but most implementations define it as 0.0.0.0 (all zeros).

Typically, IPv4 addresses are *subnetted*. Subnetting divides the host ID part of an IPv4 address into two parts: a subnet ID and a host ID (Figure 58-6). (The choice of how the bits of the host ID are divided is made by the local network administrator.) The rationale for subnetting is that an organization often doesn't attach all of its hosts to a single network. Instead, the organization may operate a set of subnetworks (an "internal internetwork"), with each subnetwork being identified by the combination of the network ID plus the subnet ID. This combination is usually referred to as the *extended network ID*. Within a subnet, the subnet mask serves the same role as described earlier for the network mask, and we can use a similar notation to indicate the range of addresses assigned to a particular subnet.

For example, suppose that our assigned network ID is 204.152.189.0/24, and we choose to subnet this address range by splitting the 8 bits of the host ID into a 4-bit subnet ID and a 4-bit host ID. Under this scheme, the subnet mask would consist of 28 leading ones, followed by 4 zeros, and the subnet with the ID of 1 would be designated as 204.152.189.16/28.

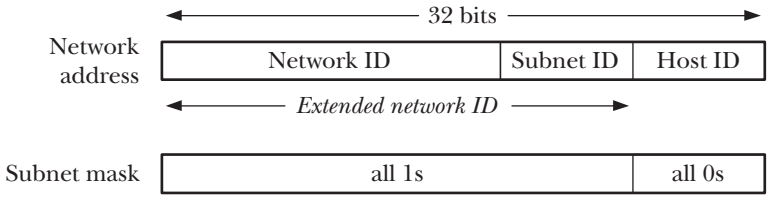

Figure 58-6: IPv4 subnetting

IPv6 addresses

The principles of IPv6 addresses are similar to IPv4 addresses. The key difference is that IPv6 addresses consist of 128 bits, and the first few bits of the address are a *format prefix*, indicating the address type. (We won't go into the details of these address types; see Appendix A of [Stevens et al., 2004] and RFC 3513 for details.)

IPv6 addresses are typically written as a series of 16-bit hexadecimal numbers separated by colons, as in the following:

```
FOOO:0:0:0:0:0:A:1
```

IPv6 addresses often include a sequence of zeros and, as a notational convenience, two colons (`::`) can be employed to indicate such a sequence. Thus, the above address can be rewritten as:

```
FOOO::A:1
```

Only one instance of the double-colon notation can appear in an IPv6 address; more than one instance would be ambiguous.

IPv6 also provides equivalents of the IPv4's loopback address (127 zeros, followed by a one, thus `::1`) and wildcard address (all zeros, written as either `0::0` or `::`).

In order to allow IPv6 applications to communicate with hosts supporting only IPv4, IPv6 provides so-called *IPv4-mapped IPv6 addresses*. The format of these addresses is shown in Figure 58-7.

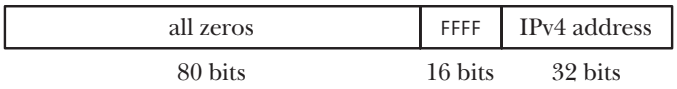

Figure 58-7: Format of an IPv4-mapped IPv6 address

When writing an IPv4-mapped IPv6 address, the IPv4 part of the address (i.e., the last 4 bytes) is written in IPv4 dotted-decimal notation. Thus, the IPv4-mapped IPv6 address equivalent to `204.152.189.116` is `::FFFF:204.152.189.116`.

58.6 The Transport Layer

There are two widely used transport-layer protocols in the TCP/IP suite:

- *User Datagram Protocol* (UDP) is the protocol used for datagram sockets.
- *Transmission Control Protocol* (TCP) is the protocol used for stream sockets.

Before considering these protocols, we first need to describe port numbers, a concept used by both protocols.

58.6.1 Port Numbers

The task of the transport protocol is to provide an end-to-end communication service to applications residing on different hosts (or sometimes on the same host). In order to do this, the transport layer requires a method of differentiating the applications on a host. In TCP and UDP, this differentiation is provided by a 16-bit *port number*.

Well-known, registered, and privileged ports

Some *well-known port numbers* are permanently assigned to specific applications (also known as *services*). For example, the *ssh* (secure shell) daemon uses the well-known port 22, and HTTP (the protocol used for communication between web servers and browsers) uses the well-known port 80. Well-known ports are assigned numbers in the range 0 to 1023 by a central authority, the Internet Assigned Numbers Authority (IANA, *http://www.iana.org/*). Assignment of a well-known port number is contingent on an approved network specification (typically in the form of an RFC).

IANA also records *registered ports*, which are allocated to application developers on a less stringent basis (which also means that an implementation doesn't need to guarantee the availability of these ports for their registered purpose). The range of IANA registered ports is 1024 to 41951. (Not all port numbers in this range are registered.)

The up-to-date list of IANA well-known and registered port assignments can be obtained online at *http://www.iana.org/assignments/port-numbers*.

In most TCP/IP implementations (including Linux), the port numbers in the range 0 to 1023 are also *privileged*, meaning that only privileged (`CAP_NET_BIND_SERVICE`) processes may bind to these ports. This prevents a normal user from implementing a malicious application that, for example, spoofs as *ssh* in order to obtain passwords. (Sometimes, privileged ports are referred to as *reserved* ports.)

Although TCP and UDP ports with the same number are distinct entities, the same well-known port number is usually assigned to a service under both TCP and UDP, even if, as is often the case, that service is available under only one of these protocols. This convention avoids confusion of port numbers across the two protocols.

Ephemeral ports

If an application doesn't select a particular port (i.e., in sockets terminology, it doesn't *bind()* its socket to a particular port), then TCP and UDP assign a unique *ephemeral port* (i.e., short-lived) number to the socket. In this case, the application—typically a client—doesn't care which port number it uses, but assigning a port is necessary so that the transport-layer protocols can identify the communication endpoints. It also has the result that the peer application at the other end of the communication channel knows how to communicate with this application. TCP and UDP also assign an ephemeral port number if we bind a socket to port 0.

IANA specifies the ports in the range 49152 to 65535 as *dynamic* or *private*, with the intention that these ports can be used by local applications and assigned as ephemeral ports. However, various implementations allocate ephemeral ports from different ranges. On Linux, the range is defined by (and can be modified via) two numbers contained in the file `/proc/sys/net/ipv4/ip_local_port_range`.

58.6.2 User Datagram Protocol (UDP)

UDP adds just two features to IP: port numbers and a data checksum to allow the detection of errors in the transmitted data.

Like IP, UDP is connectionless. Since it adds no reliability to IP, UDP is likewise unreliable. If an application layered on top of UDP requires reliability, then this must be implemented within the application. Despite this unreliability, we may sometimes prefer to use UDP instead of TCP, for the reasons detailed in Section 61.12.

The checksums used by both UDP and TCP are just 16 bits long, and are simple "add-up" checksums that can fail to detect certain classes of errors. Consequently, they do not provide extremely strong error detection. Busy Internet servers typically see an average of one undetected transmission error every few days ([Stone & Partridge, 2000]). Applications that need stronger assurances of data integrity can use the Secure Sockets Layer (SSL) protocol, which provides not only secure communication, but also much more rigorous detection of errors. Alternatively, an application could implement its own error-control scheme.

Selecting a UDP datagram size to avoid IP fragmentation

In Section 58.4, we described the IP fragmentation mechanism, and noted that it is usually best to avoid IP fragmentation. While TCP contains mechanisms for avoiding IP fragmentation, UDP does not. With UDP, we can easily cause IP fragmentation by transmitting a datagram that exceeds the MTU of the local data link.

A UDP-based application generally doesn't know the MTU of the path between the source and destination hosts. UDP-based applications that aim to avoid IP fragmentation typically adopt a conservative approach, which is to ensure that the transmitted IP datagram is less than the IPv4 minimum reassembly buffer size of 576 bytes. (This value is likely to be lower than the path MTU.) From these 576 bytes, 8 bytes are required by UDP's own header, and an additional minimum of 20 bytes are required for the IP header, leaving 548 bytes for the UDP datagram itself. In practice, many UDP-based applications opt for a still lower limit of 512 bytes for their datagrams ([Stevens, 1994]).

58.6.3 Transmission Control Protocol (TCP)

TCP provides a reliable, connection-oriented, bidirectional, byte-stream communication channel between two endpoints (i.e., applications), as shown in Figure 58-8. In order to provide these features, TCP must perform the tasks described in this section. (A detailed description of all of these features can be found in [Stevens, 1994].)

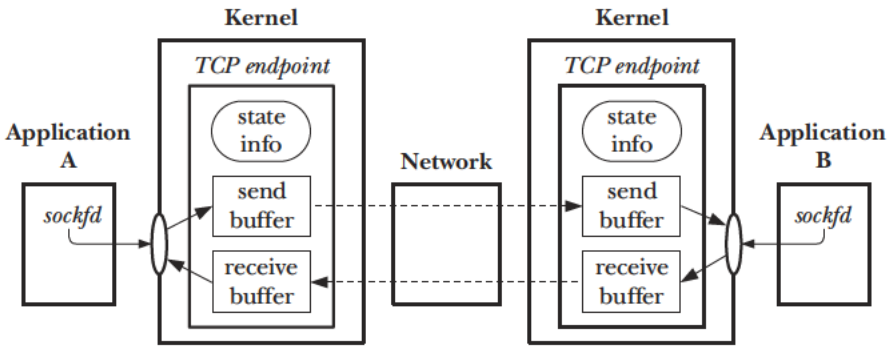

Figure 58-8: Connected TCP sockets

We use the term *TCP endpoint* to denote the information maintained by the kernel for one end of a TCP connection. (Often, we abbreviate this term further, for example, writing just "a TCP," to mean "a TCP endpoint," or "the client TCP" to mean "the TCP endpoint maintained for the client application.") This information includes the send and receive buffers for this end of the connection, as well as state

information that is maintained in order to synchronize the operation of the two connected endpoints. (We describe this state information in further detail when we consider the TCP state transition diagram in Section 61.6.3.) In the remainder of this book, we use the terms *receiving TCP* and *sending TCP* to denote the TCP endpoints maintained for the receiving and sending applications on either end of a stream socket connection that is being used to transmit data in a particular direction.

Connection establishment

Before communication can commence, TCP establishes a communication channel between the two endpoints. During connection establishment, the sender and receiver can exchange options to advertise parameters for the connection.

Packaging of data in segments

Data is broken into segments, each of which contains a checksum to allow the detection of end-to-end transmission errors. Each segment is transmitted in a single IP datagram.

Acknowledgements, retransmissions, and timeouts

When a TCP segment arrives at its destination without errors, the receiving TCP sends a positive acknowledgement to the sender, informing it of the successfully delivered data. If a segment arrives with errors, then it is discarded, and no acknowledgement is sent. To handle the possibility of segments that never arrive or are discarded, the sender starts a timer when each segment is transmitted. If an acknowledgement is not received before the timer expires, the segment is retransmitted.

> Since the time taken to transmit a segment and receive its acknowledgement varies according to the range of the network and the current traffic loading, TCP employs an algorithm to dynamically adjust the size of the retransmission timeout (RTO).
>
> The receiving TCP may not send acknowledgements immediately, but instead wait for a fraction of a second to see if the acknowledgement can be piggybacked inside any response that the receiver may send straight back to the sender. (Every TCP segment includes an acknowledgement field, allowing for such piggybacking.) The aim of this technique, called *delayed ACK*, is to save sending a TCP segment, thus decreasing the number of packets in the network and decreasing the load on the sending and receiving hosts.

Sequencing

Each byte that is transmitted over a TCP connection is assigned a logical sequence number. This number indicates the position of that byte in the data stream for the connection. (Each of the two streams in the connection has its own sequence numbering.) When a TCP segment is transmitted, it includes a field containing the sequence number of the first byte in the segment.

Attaching sequence numbers to each segment serves a variety of purposes:

- The sequence number allows TCP segments to be assembled in the correct order at the destination, and then passed as a byte stream to the application layer. (At any moment, multiple TCP segments may be in transit between sender and receiver, and these segments may arrive out of order.)

- The acknowledgement message passed from the receiver back to the sender can use the sequence number to identify which TCP segment was received.
- The receiver can use the sequence number to eliminate duplicate segments. Such duplicates may occur either because of the duplication of IP datagrams or because of TCP's own retransmission algorithm, which could retransmit a successfully delivered segment if the acknowledgement for that segment was lost or was not received in a timely fashion.

The initial sequence number (ISN) for a stream doesn't start at 0. Instead, it is generated via an algorithm that increases the ISN assigned to successive TCP connections (to prevent the possibility of old segments from a previous incarnation of the connection being confused with segments for this connection). This algorithm is also designed to make guessing the ISN difficult. The sequence number is a 32-bit value that is wrapped around to 0 when the maximum value is reached.

Flow control

Flow control prevents a fast sender from overwhelming a slow receiver. To implement flow control, the receiving TCP maintains a buffer for incoming data. (Each TCP advertises the size of this buffer during connection establishment.) Data accumulates in this buffer as it is received from the sending TCP, and is removed as the application reads data. With each acknowledgement, the receiver advises the sender of how much space is available in its incoming data buffer (i.e., how many bytes the sender can transmit). The TCP flow-control algorithm employs a so-called *sliding window* algorithm, which allows unacknowledged segments containing a total of up *N* (the offered window size) bytes to be in transit between the sender and receiver. If a receiving TCP's incoming data buffer fills completely, then the window is said to be closed, and the sending TCP stops transmitting.

> The receiver can override the default size for the incoming data buffer using the `SO_RCVBUF` socket option (see the *socket(7)* manual page).

Congestion control: slow-start and congestion-avoidance algorithms

TCP's congestion-control algorithms are designed to prevent a fast sender from overwhelming a network. If a sending TCP transmits packets faster than they can be relayed by an intervening router, that router will start dropping packets. This could lead to high rates of packet loss and, consequently, serious performance degradation, if the sending TCP kept retransmitting these dropped segments at the same rate. TCP's congestion-control algorithms are important in two circumstances:

- *After connection establishment*: At this time (or when transmission resumes on a connection that has been idle for some time), the sender could start by immediately injecting as many segments into the network as would be permitted by the window size advertised by the receiver. (In fact, this is what was done in early TCP implementations.) The problem here is that if the network can't handle this flood of segments, the sender risks overwhelming the network immediately.

- *When congestion is detected*: If the sending TCP detects that congestion is occurring, then it must reduce its transmission rate. TCP detects that congestion is occurring based on the assumption that segment loss because of transmission errors is very low; thus, if a packet is lost, the cause is assumed to be congestion.

TCP's congestion-control strategy employs two algorithms in combination: slow start and congestion avoidance.

The *slow-start* algorithm causes the sending TCP to initially transmit segments at a slow rate, but allows it to exponentially increase the rate as these segments are acknowledged by the receiving TCP. Slow start attempts to prevent a fast TCP sender from overwhelming a network. However, if unrestrained, slow start's exponential increase in the transmission rate could mean that the sender would soon overwhelm the network. TCP's *congestion-avoidance* algorithm prevents this, by placing a governor on the rate increase.

With congestion avoidance, at the beginning of a connection, the sending TCP starts with a small *congestion window*, which limits the amount of unacknowledged data that it can transmit. As the sender receives acknowledgements from the peer TCP, the congestion window initially grows exponentially. However, once the congestion window reaches a certain threshold believed to be close to the transmission capacity of the network, its growth becomes linear, rather than exponential. (An estimate of the capacity of the network is derived from a calculation based on the transmission rate that was in operation when congestion was detected, or is set at a fixed value after initial establishment of the connection.) At all times, the quantity of data that the sending TCP will transmit remains additionally constrained by the receiving TCP's advertised window and the local TCP's send buffer.

In combination, the slow-start and congestion-avoidance algorithms allow the sender to rapidly raise its transmission speed up to the available capacity of the network, without overshooting that capacity. The effect of these algorithms is to allow data transmission to quickly reach a state of equilibrium, where the sender transmits packets at the same rate as it receives acknowledgements from the receiver.

58.7 Requests for Comments (RFCs)

Each of the Internet protocols that we discuss in this book is defined in an RFC document—a formal protocol specification. RFCs are published by the *RFC Editor* (*http://www.rfc-editor.org/*), which is funded by the *Internet Society* (*http://www.isoc.org/*). RFCs that describe Internet standards are developed under the auspices of the *Internet Engineering Task Force* (IETF, *http://www.ietf.org/*), a community of network designers, operators, vendors, and researchers concerned with the evolution and smooth operation of the Internet. Membership of the IETF is open to any interested individual.

The following RFCs are of particular relevance to the material covered in this book:

- RFC 791, *Internet Protocol*. J. Postel (ed.), 1981.
- RFC 950, *Internet Standard Subnetting Procedure*. J. Mogul and J. Postel, 1985.

- RFC 793, *Transmission Control Protocol*. J. Postel (ed.), 1981.
- RFC 768, *User Datagram Protocol*. J. Postel (ed.), 1980.
- RFC 1122, *Requirements for Internet Hosts–Communication Layers*. R. Braden (ed.), 1989.

  > RFC 1122 extends (and corrects) various earlier RFCs describing the TCP/IP protocols. It is one of a pair of RFCs that are often simply known as the *Host Requirements RFCs*. The other member of the pair is RFC 1123, which covers application-layer protocols such as *telnet*, FTP, and SMTP.

Among the RFCs that describe IPv6 are the following:

- RFC 2460, *Internet Protocol, Version 6*. S. Deering and R. Hinden, 1998.
- RFC 4291, *IP Version 6 Addressing Architecture*. R. Hinden and S. Deering, 2006.
- RFC 3493, *Basic Socket Interface Extensions for IPv6*. R. Gilligan, S. Thomson, J. Bound, J. McCann, and W. Stevens, 2003.
- RFC 3542, *Advanced Sockets API for IPv6*. W. Stevens, M. Thomas, E. Nordmark, and T. Jinmei, 2003.

A number of RFCs and papers provide improvements and extensions to the original TCP specification, including the following:

- *Congestion Avoidance and Control*. V. Jacobsen, 1988. This was the initial paper describing the congestion-control and slow-start algorithms for TCP. Originally published in *Proceedings of SIGCOMM '88*, a slightly revised version is available at *ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z*. This paper is largely superseded by some of the following RFCs.
- RFC 1323, *TCP Extensions for High Performance*. V. Jacobson, R. Braden, and D. Borman, 1992.
- RFC 2018, *TCP Selective Acknowledgment Options*. M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, 1996.
- RFC 2581, *TCP Congestion Control*. M. Allman, V. Paxson, and W. Stevens, 1999.
- RFC 2861, *TCP Congestion Window Validation*. M. Handley, J. Padhye, and S. Floyd, 2000.
- RFC 2883, *An Extension to the Selective Acknowledgement (SACK) Option for TCP*. S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky, 2000.
- RFC 2988, *Computing TCP's Retransmission Timer*. V. Paxson and M. Allman, 2000.
- RFC 3168, *The Addition of Explicit Congestion Notification (ECN) to IP*. K. Ramakrishnan, S. Floyd, and D. Black, 2001.
- RFC 3390, *Increasing TCP's Initial Window*. M. Allman, S. Floyd, and C. Partridge, 2002.

58.8 Summary

TCP/IP is a layered networking protocol suite. At the bottom layer of the TCP/IP protocol stack is the IP network-layer protocol. IP transmits data in the form of datagrams. IP is connectionless, meaning that datagrams transmitted between source and destination hosts may take different routes across the network. IP is unreliable, in that it provides no guarantee that datagrams will arrive in order or unduplicated, or even arrive at all. If reliability is required, then it must be provided via the use of a reliable higher-layer protocol (e.g., TCP), or within an application.

The original version of IP is IPv4. In the early 1990s, a new version of IP, IPv6, was devised. The most notable difference between IPv4 and IPv6 is that IPv4 uses 32 bits to represent a host address, while IPv6 uses 128 bits, thus allowing for a much larger number of hosts on the world-wide Internet. Currently, IPv4 remains the most widely used version of IP, although in coming years, it is likely to be supplanted by IPv6.

Various transport-layer protocols are layered on top of IP, of which the most widely used are UDP and TCP. UDP is an unreliable datagram protocol. TCP is a reliable, connection-oriented, byte-stream protocol. TCP handles all of the details of connection establishment and termination. TCP also packages data into segments for transmission by IP, and provides sequence numbering for these segments so that they can be acknowledged and assembled in the correct order by the receiver. In addition, TCP provides flow control, to prevent a fast sender from overwhelming a slow receiver, and congestion control, to prevent a fast sender from overwhelming the network.

Further information

Refer to the sources of further information listed in Section 59.15.

59

SOCKETS: INTERNET DOMAINS

Having looked at generic sockets concepts and the TCP/IP protocol suite in previous chapters, we are now ready in this chapter to look at programming with sockets in the IPv4 (`AF_INET`) and IPv6 (`AF_INET6`) domains.

As noted in Chapter 58, Internet domain socket addresses consist of an IP address and a port number. Although computers use binary representations of IP addresses and port numbers, humans are much better at dealing with names than with numbers. Therefore, we describe the techniques used to identify host computers and ports using names. We also examine the use of library functions to obtain the IP address(es) for a particular hostname and the port number that corresponds to a particular service name. Our discussion of hostnames includes a description of the Domain Name System (DNS), which implements a distributed database that maps hostnames to IP addresses and vice versa.

59.1 Internet Domain Sockets

Internet domain stream sockets are implemented on top of TCP. They provide a reliable, bidirectional, byte-stream communication channel.

Internet domain datagram sockets are implemented on top of UDP. UDP sockets are similar to their UNIX domain counterparts, but note the following differences:

- UNIX domain datagram sockets are reliable, but UDP sockets are not—datagrams may be lost, duplicated, or arrive in a different order from that in which they were sent.
- Sending on a UNIX domain datagram socket will block if the queue of data for the receiving socket is full. By contrast, with UDP, if the incoming datagram would overflow the receiver's queue, then the datagram is silently dropped.

59.2 Network Byte Order

IP addresses and port numbers are integer values. One problem we encounter when passing these values across a network is that different hardware architectures store the bytes of a multibyte integer in different orders. As shown in Figure 59-1, architectures that store integers with the most significant byte first (i.e., at the lowest memory address) are termed *big endian*; those that store the least significant byte first are termed *little endian*. (The terms derive from Jonathan Swift's 1726 satirical novel *Gulliver's Travels*, in which the terms refer to opposing political factions who open their boiled eggs at opposite ends.) The most notable example of a little-endian architecture is x86. (Digital's VAX architecture was another historically important example, since BSD was widely used on that machine.) Most other architectures are big endian. A few hardware architectures are switchable between the two formats. The byte ordering used on a particular machine is called the *host byte order*.

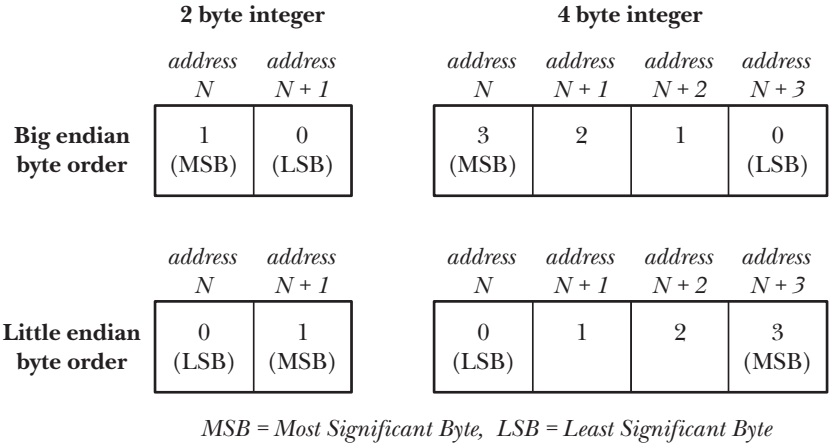

Figure 59-1: Big-endian and little-endian byte order for 2-byte and 4-byte integers

Since port numbers and IP addresses must be transmitted between, and understood by, all hosts on a network, a standard ordering must be used. This ordering is called *network byte order*, and happens to be big endian.

Later in this chapter, we look at various functions that convert hostnames (e.g., `www.kernel.org`) and service names (e.g., *http*) into the corresponding numeric forms. These functions generally return integers in network byte order, and these integers can be copied directly into the relevant fields of a socket address structure.

However, we sometimes make direct use of integer constants for IP addresses and port numbers. For example, we may choose to hard-code a port number into our program, specify a port number as a command-line argument to a program, or use constants such as INADDR_ANY and INADDR_LOOPBACK when specifying an IPv4 address. These values are represented in C according to the conventions of the host machine, so they are in host byte order. We must convert these values to network byte order before storing them in socket address structures.

The *htons()*, *htonl()*, *ntohs()*, and *ntohl()* functions are defined (typically as macros) for converting integers in either direction between host and network byte order.

```
#include <arpa/inet.h>

uint16_t htons(uint16_t host_uint16);
                                Returns host_uint16 converted to network byte order
uint32_t htonl(uint32_t host_uint32);
                                Returns host_uint32 converted to network byte order
uint16_t ntohs(uint16_t net_uint16);
                                Returns net_uint16 converted to host byte order
uint32_t ntohl(uint32_t net_uint32);
                                Returns net_uint32 converted to host byte order
```

In earlier times, these functions had prototypes such as the following:

```
unsigned long htonl(unsigned long hostlong);
```

This reveals the origin of the function names—in this case, *host to network long*. On most early systems on which sockets were implemented, short integers were 16 bits, and long integers were 32 bits. This no longer holds true on modern systems (at least for long integers), so the prototypes given above provide a more exact definition of the types dealt with by these functions, although the names remain unchanged. The *uint16_t* and *uint32_t* data types are 16-bit and 32-bit unsigned integers.

Strictly speaking, the use of these four functions is necessary only on systems where the host byte order differs from network byte order. However, these functions should always be used, so that programs are portable to different hardware architectures. On systems where the host byte order is the same as network byte order, these functions simply return their arguments unchanged.

59.3 Data Representation

When writing network programs, we need to be aware of the fact that different computer architectures use different conventions for representing various data types. We have already noted that integer types can be stored in big-endian or little-endian form. There are also other possible differences. For example, the C *long* data type may be 32 bits on some systems and 64 bits on others. When we consider structures, the issue is further complicated by the fact that different implementations

employ different rules for aligning the fields of a structure to address boundaries on the host system, leaving different numbers of padding bytes between the fields.

Because of these differences in data representation, applications that exchange data between heterogeneous systems over a network must adopt some common convention for encoding that data. The sender must encode data according to this convention, while the receiver decodes following the same convention. The process of putting data into a standard format for transmission across a network is referred to as *marshalling*. Various marshalling standards exist, such as XDR (External Data Representation, described in RFC 1014), ASN.1-BER (Abstract Syntax Notation 1, *http://www.asn1.org/*), CORBA, and XML. Typically, these standards define a fixed format for each data type (defining, for example, byte order and number of bits used). As well as being encoded in the required format, each data item is tagged with extra field(s) identifying its type (and, possibly, length).

However, a simpler approach than marshalling is often employed: encode all transmitted data in text form, with separate data items delimited by a designated character, typically a newline character. One advantage of this approach is that we can use *telnet* to debug an application. To do this, we use the following command:

```
$ telnet host port
```

We can then type lines of text to be transmitted to the application, and view the responses sent by the application. We demonstrate this technique in Section 59.11.

> The problems associated with differences in representation across heterogeneous systems apply not only to data transfer across a network, but also to any mechanism of data exchange between such systems. For example, we face the same problems when transferring files on disk or tape between heterogeneous systems. Network programming is simply the most common programming context in which we are nowadays likely to encounter this issue.

If we encode data transmitted on a stream socket as newline-delimited text, then it is convenient to define a function such as *readLine()*, shown in Listing 59-1.

```
#include "read_line.h"

ssize_t readLine(int fd, void *buffer, size_t n);
```

Returns number of bytes copied into *buffer* (excluding terminating null byte), or 0 on end-of-file, or −1 on error

The *readLine()* function reads bytes from the file referred to by the file descriptor argument *fd* until a newline is encountered. The input byte sequence is returned in the location pointed to by *buffer*, which must point to a region of at least *n* bytes of memory. The returned string is always null-terminated; thus, at most *(n − 1)* bytes of actual data will be returned. On success, *readLine()* returns the number of bytes of data placed in *buffer*; the terminating null byte is not included in this count.

Listing 59-1: Reading data a line at a time

sockets/read_line.c

```
#include <unistd.h>
#include <errno.h>
#include "read_line.h"                  /* Declaration of readLine() */

ssize_t
readLine(int fd, void *buffer, size_t n)
{
    ssize_t numRead;                    /* # of bytes fetched by last read() */
    size_t totRead;                     /* Total bytes read so far */
    char *buf;
    char ch;

    if (n <= 0 || buffer == NULL) {
        errno = EINVAL;
        return -1;
    }

    buf = buffer;                       /* No pointer arithmetic on "void *" */

    totRead = 0;
    for (;;) {
        numRead = read(fd, &ch, 1);

        if (numRead == -1) {
            if (errno == EINTR)        /* Interrupted --> restart read() */
                continue;
            else
                return -1;              /* Some other error */

        } else if (numRead == 0) {      /* EOF */
            if (totRead == 0)           /* No bytes read; return 0 */
                return 0;
            else                        /* Some bytes read; add '\0' */
                break;

        } else {                        /* 'numRead' must be 1 if we get here */
            if (totRead < n - 1) {      /* Discard > (n - 1) bytes */
                totRead++;
                *buf++ = ch;
            }

            if (ch == '\n')
                break;
        }
    }

    *buf = '\0';
    return totRead;
}
```

sockets/read_line.c

If the number of bytes read before a newline is encountered is greater than or equal to *(n − 1)*, then the *readLine()* function discards the excess bytes (including the newline). If a newline was read within the first *(n − 1)* bytes, then it is included in the returned string. (Thus, we can determine if bytes were discarded by checking if a newline precedes the terminating null byte in the returned *buffer*.) We take this approach so that application protocols that rely on handling input in units of lines don't end up processing a long line as though it were multiple lines. This would likely break the protocol, as the applications on either end would become desynchronized. An alternative approach would be to have *readLine()* read only sufficient bytes to fill the supplied buffer, leaving any remaining bytes up to the next newline for the next call to *readLine()*. In this case, the caller of *readLine()* would need to handle the possibility of a partial line being read.

We employ the *readLine()* function in the example programs presented in Section 59.11.

59.4 Internet Socket Addresses

There are two types of Internet domain socket addresses: IPv4 and IPv6.

IPv4 socket addresses: *struct sockaddr_in*

An IPv4 socket address is stored in a *sockaddr_in* structure, defined in `<netinet/in.h>` as follows:

```
struct in_addr {                    /* IPv4 4-byte address */
    in_addr_t s_addr;               /* Unsigned 32-bit integer */
};

struct sockaddr_in {                /* IPv4 socket address */
    sa_family_t    sin_family;      /* Address family (AF_INET) */
    in_port_t      sin_port;        /* Port number */
    struct in_addr sin_addr;        /* IPv4 address */
    unsigned char  __pad[X];        /* Pad to size of 'sockaddr'
                                       structure (16 bytes) */
};
```

In Section 56.4, we saw that the generic *sockaddr* structure commences with a field identifying the socket domain. This corresponds to the *sin_family* field in the *sockaddr_in* structure, which is always set to `AF_INET`. The *sin_port* and *sin_addr* fields are the port number and the IP address, both in network byte order. The *in_port_t* and *in_addr_t* data types are unsigned integer types, 16 and 32 bits in length, respectively.

IPv6 socket addresses: *struct sockaddr_in6*

Like an IPv4 address, an IPv6 socket address includes an IP address plus a port number. The difference is that an IPv6 address is 128 bits instead of 32 bits. An IPv6 socket address is stored in a *sockaddr_in6* structure, defined in `<netinet/in.h>` as follows:

```
struct in6_addr {                   /* IPv6 address structure */
    uint8_t s6_addr[16];            /* 16 bytes == 128 bits */
};
```

```
struct sockaddr_in6 {               /* IPv6 socket address */
    sa_family_t sin6_family;        /* Address family (AF_INET6) */
    in_port_t   sin6_port;          /* Port number */
    uint32_t    sin6_flowinfo;      /* IPv6 flow information */
    struct in6_addr sin6_addr;      /* IPv6 address */
    uint32_t    sin6_scope_id;      /* Scope ID (new in kernel 2.4) */
};
```

The *sin_family* field is set to AF_INET6. The *sin6_port* and *sin6_addr* fields are the port number and the IP address. (The *uint8_t* data type, used to type the bytes of the *in6_addr* structure, is an 8-bit unsigned integer.) The remaining fields, *sin6_flowinfo* and *sin6_scope_id*, are beyond the scope of this book; for our purposes, they are always set to 0. All of the fields in the *sockaddr_in6* structure are in network byte order.

> IPv6 addresses are described in RFC 4291. Information about IPv6 flow control (*sin6_flowinfo*) can be found in Appendix A of [Stevens et al., 2004] and in RFCs 2460 and 3697. RFCs 3493 and 4007 provide information about *sin6_scope_id*.

IPv6 has equivalents of the IPv4 wildcard and loopback addresses. However, their use is complicated by the fact that an IPv6 address is stored in an array (rather than using a scalar type). We use the IPv6 wildcard address (0::0) to illustrate this point. The constant IN6ADDR_ANY_INIT is defined for this address as follows:

```
#define IN6ADDR_ANY_INIT { { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 } }
```

> On Linux, some details in the header files differ from our description in this section. In particular, the *in6_addr* structure contains a union definition that divides the 128-bit IPv6 address into 16 bytes, eight 2-byte integers, or four 32-byte integers. Because of the presence of this definition, the *glibc* definition of the IN6ADDR_ANY_INIT constant actually includes one more set of nested braces than is shown in the main text.

We can use the IN6ADDR_ANY_INIT constant in the initializer that accompanies a variable declaration, but can't use it on the right-hand side of an assignment statement, since C syntax doesn't permit structured constants to be used in assignments. Instead, we must use a predefined variable, *in6addr_any*, which is initialized as follows by the C library:

```
const struct in6_addr in6addr_any = IN6ADDR_ANY_INIT;
```

Thus, we can initialize an IPv6 socket address structure using the wildcard address as follows:

```
struct sockaddr_in6 addr;

memset(&addr, 0, sizeof(struct sockaddr_in6));
addr.sin6_family = AF_INET6;
addr.sin6_addr = in6addr_any;
addr.sin6_port = htons(SOME_PORT_NUM);
```

The corresponding constant and variable for the IPv6 loopback address (::1) are IN6ADDR_LOOPBACK_INIT and *in6addr_loopback*.

Unlike their IPv4 counterparts, the IPv6 constant and variable initializers are in network byte order. But, as shown in the above code, we still must ensure that the port number is in network byte order.

If IPv4 and IPv6 coexist on a host, they share the same port-number space. This means that if, for example, an application binds an IPv6 socket to TCP port 2000 (using the IPv6 wildcard address), then an IPv4 TCP socket can't be bound to the same port. (The TCP/IP implementation ensures that sockets on other hosts are able to communicate with this socket, regardless of whether those hosts are running IPv4 or IPv6.)

The *sockaddr_storage* structure

With the IPv6 sockets API, the new generic *sockaddr_storage* structure was introduced. This structure is defined to be large enough to hold any type of socket address (i.e., any type of socket address structure can be cast and stored in it). In particular, this structure allows us to transparently store either an IPv4 or an IPv6 socket address, thus removing IP version dependencies from our code. The *sockaddr_storage* structure is defined on Linux as follows:

```
#define __ss_aligntype uint32_t         /* On 32-bit architectures */
struct sockaddr_storage {
    sa_family_t ss_family;
    __ss_aligntype __ss_align;          /* Force alignment */
    char __ss_padding[SS_PADSIZE];      /* Pad to 128 bytes */
};
```

59.5 Overview of Host and Service Conversion Functions

Computers represent IP addresses and port numbers in binary. However, humans find names easier to remember than numbers. Employing symbolic names also provides a useful level of indirection; users and programs can continue to use the same name even if the underlying numeric value changes.

A *hostname* is the symbolic identifier for a system that is connected to a network (possibly with multiple IP addresses). A *service name* is the symbolic representation of a port number.

The following methods are available for representing host addresses and ports:

- A host address can be represented as a binary value, as a symbolic hostname, or in presentation format (dotted-decimal for IPv4 or hex-string for IPv6).
- A port can be represented as a binary value or as a symbolic service name.

Various library functions are provided for converting between these formats. This section briefly summarizes these functions. The following sections describe the modern APIs (*inet_ntop()*, *inet_pton()*, *getaddrinfo()*, *getnameinfo()*, and so on) in detail. In Section 59.13, we briefly discuss the obsolete APIs (*inet_aton()*, *inet_ntoa()*, *gethostbyname()*, *getservbyname()*, and so on).

Converting IPv4 addresses between binary and human-readable forms

The *inet_aton()* and *inet_ntoa()* functions convert an IPv4 address in dotted-decimal notation to binary and vice versa. We describe these functions primarily because

they appear in historical code. Nowadays, they are obsolete. Modern programs that need to do such conversions should use the functions that we describe next.

Converting IPv4 and IPv6 addresses between binary and human-readable forms

The *inet_pton()* and *inet_ntop()* functions are like *inet_aton()* and *inet_ntoa()*, but differ in that they also handle IPv6 addresses. They convert binary IPv4 and IPv6 addresses to and from *presentation* format—that is, either dotted-decimal or hex-string notation.

Since humans deal better with names than with numbers, we normally use these functions only occasionally in programs. One use of *inet_ntop()* is to produce a printable representation of an IP address for logging purposes. Sometimes, it is preferable to use this function instead of converting ("resolving") an IP address to a hostname, for the following reasons:

- Resolving an IP address to a hostname involves a possibly time-consuming request to a DNS server.
- In some circumstances, there may not be a DNS (PTR) record that maps the IP address to a corresponding hostname.

We describe these functions (in Section 59.6) before *getaddrinfo()* and *getnameinfo()*, which perform conversions between binary representations and the corresponding symbolic names, principally because they present a much simpler API. This allows us to quickly show some working examples of the use of Internet domain sockets.

Converting host and service names to and from binary form (obsolete)

The *gethostbyname()* function returns the binary IP address(es) corresponding to a hostname and the *getservbyname()* function returns the port number corresponding to a service name. The reverse conversions are performed by *gethostbyaddr()* and *getservbyport()*. We describe these functions because they are widely used in existing code. However, they are now obsolete. (SUSv3 marks these functions obsolete, and SUSv4 removes their specifications.) New code should use the *getaddrinfo()* and *getnameinfo()* functions (described next) for such conversions.

Converting host and service names to and from binary form (modern)

The *getaddrinfo()* function is the modern successor to both *gethostbyname()* and *getservbyname()*. Given a hostname and a service name, *getaddrinfo()* returns a set of structures containing the corresponding binary IP address(es) and port number. Unlike *gethostbyname()*, *getaddrinfo()* transparently handles both IPv4 and IPv6 addresses. Thus, we can use it to write programs that don't contain dependencies on the IP version being employed. All new code should use *getaddrinfo()* for converting hostnames and service names to binary representation.

The *getnameinfo()* function performs the reverse translation, converting an IP address and port number into the corresponding hostname and service name.

We can also use *getaddrinfo()* and *getnameinfo()* to convert binary IP addresses to and from presentation format.

The discussion of *getaddrinfo()* and *getnameinfo()*, in Section 59.10, requires an accompanying description of DNS (Section 59.8) and the `/etc/services` file (Section 59.9). DNS allows cooperating servers to maintain a distributed database

that maps binary IP addresses to hostnames and vice versa. The existence of a system such as DNS is essential to the operation of the Internet, since centralized management of the enormous set of Internet hostnames would be impossible. The `/etc/services` file maps port numbers to symbolic service names.

59.6 The *inet_pton()* and *inet_ntop()* Functions

The *inet_pton()* and *inet_ntop()* functions allow conversion of both IPv4 and IPv6 addresses between binary form and dotted-decimal or hex-string notation.

```
#include <arpa/inet.h>

int inet_pton(int domain, const char *src_str, void *addrptr);
                                Returns 1 on successful conversion, 0 if src_str is not in
                                                    presentation format, or −1 on error
const char *inet_ntop(int domain, const void *addrptr, char *dst_str, size_t len);
                                Returns pointer to dst_str on success, or NULL on error
```

The *p* in the names of these functions stands for "presentation," and the *n* stands for "network." The presentation form is a human-readable string, such as the following:

- `204.152.189.116` (IPv4 dotted-decimal address);
- `::1` (an IPv6 colon-separated hexadecimal address); or
- `::FFFF:204.152.189.116` (an IPv4-mapped IPv6 address).

The *inet_pton()* function converts the presentation string contained in *src_str* into a binary IP address in network byte order. The *domain* argument should be specified as either `AF_INET` or `AF_INET6`. The converted address is placed in the structure pointed to by *addrptr*, which should point to either an *in_addr* or an *in6_addr* structure, according to the value specified in *domain*.

The *inet_ntop()* function performs the reverse conversion. Again, *domain* should be specified as either `AF_INET` or `AF_INET6`, and *addrptr* should point to an *in_addr* or *in6_addr* structure that we wish to convert. The resulting null-terminated string is placed in the buffer pointed to by *dst_str*. The *len* argument must specify the size of this buffer. On success, *inet_ntop()* returns *dst_str*. If *len* is too small, then *inet_ntop()* returns `NULL`, with *errno* set to `ENOSPC`.

To correctly size the buffer pointed to by *dst_str*, we can employ two constants defined in `<netinet/in.h>`. These constants indicate the maximum lengths (including the terminating null byte) of the presentation strings for IPv4 and IPv6 addresses:

```
#define INET_ADDRSTRLEN  16     /* Maximum IPv4 dotted-decimal string */
#define INET6_ADDRSTRLEN 46     /* Maximum IPv6 hexadecimal string */
```

We provide examples of the use of *inet_pton()* and *inet_ntop()* in the next section.

59.7 Client-Server Example (Datagram Sockets)

In this section, we take the case-conversion server and client programs shown in Section 57.3 and modify them to use datagram sockets in the AF_INET6 domain. We present these programs with a minimum of commentary, since their structure is similar to the earlier programs. The main differences in the new programs lie in the declaration and initialization of the IPv6 socket address structure, which we described in Section 59.4.

The client and server both employ the header file shown in Listing 59-2. This header file defines the server's port number and the maximum size of messages that the client and server can exchange.

Listing 59-2: Header file used by i6d_ucase_sv.c and i6d_ucase_cl.c

sockets/i6d_ucase.h

```
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <ctype.h>
#include "tlpi_hdr.h"

#define BUF_SIZE 10                     /* Maximum size of messages exchanged
                                           between client and server */

#define PORT_NUM 50002                  /* Server port number */
```

sockets/i6d_ucase.h

Listing 59-3 shows the server program. The server uses the *inet_ntop()* function to convert the host address of the client (obtained via the *recvfrom()* call) to printable form.

The client program shown in Listing 59-4 contains two notable modifications from the earlier UNIX domain version (Listing 57-7, on page 1173). The first difference is that the client interprets its initial command-line argument as the IPv6 address of the server. (The remaining command-line arguments are passed as separate datagrams to the server.) The client converts the server address to binary form using *inet_pton()*. The other difference is that the client doesn't bind its socket to an address. As noted in Section 58.6.1, if an Internet domain socket is not bound to an address, the kernel binds the socket to an ephemeral port on the host system. We can observe this in the following shell session log, where we run the server and the client on the same host:

```
$ ./i6d_ucase_sv &
[1] 31047
$ ./i6d_ucase_cl ::1 ciao                    Send to server on local host
Server received 4 bytes from (::1, 32770)
Response 1: CIAO
```

From the above output, we see that the server's *recvfrom()* call was able to obtain the address of the client's socket, including the ephemeral port number, despite the fact that the client did not do a *bind()*.

Listing 59-3: IPv6 case-conversion server using datagram sockets

sockets/i6d_ucase_sv.c

```
#include "i6d_ucase.h"

int
main(int argc, char *argv[])
{
    struct sockaddr_in6 svaddr, claddr;
    int sfd, j;
    ssize_t numBytes;
    socklen_t len;
    char buf[BUF_SIZE];
    char claddrStr[INET6_ADDRSTRLEN];

    sfd = socket(AF_INET6, SOCK_DGRAM, 0);
    if (sfd == -1)
        errExit("socket");

    memset(&svaddr, 0, sizeof(struct sockaddr_in6));
    svaddr.sin6_family = AF_INET6;
    svaddr.sin6_addr = in6addr_any;                    /* Wildcard address */
    svaddr.sin6_port = htons(PORT_NUM);

    if (bind(sfd, (struct sockaddr *) &svaddr,
                sizeof(struct sockaddr_in6)) == -1)
        errExit("bind");

    /* Receive messages, convert to uppercase, and return to client */

    for (;;) {
        len = sizeof(struct sockaddr_in6);
        numBytes = recvfrom(sfd, buf, BUF_SIZE, 0,
                            (struct sockaddr *) &claddr, &len);
        if (numBytes == -1)
            errExit("recvfrom");

        if (inet_ntop(AF_INET6, &claddr.sin6_addr, claddrStr,
                    INET6_ADDRSTRLEN) == NULL)
            printf("Couldn't convert client address to string\n");
        else
            printf("Server received %ld bytes from (%s, %u)\n",
                    (long) numBytes, claddrStr, ntohs(claddr.sin6_port));

        for (j = 0; j < numBytes; j++)
            buf[j] = toupper((unsigned char) buf[j]);

        if (sendto(sfd, buf, numBytes, 0, (struct sockaddr *) &claddr, len) !=
                numBytes)
            fatal("sendto");
    }
}
```

sockets/i6d_ucase_sv.c

Listing 59-4: IPv6 case-conversion client using datagram sockets

sockets/i6d_ucase_cl.c

```
#include "i6d_ucase.h"

int
main(int argc, char *argv[])
{
    struct sockaddr_in6 svaddr;
    int sfd, j;
    size_t msgLen;
    ssize_t numBytes;
    char resp[BUF_SIZE];

    if (argc < 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s host-address msg...\n", argv[0]);

    sfd = socket(AF_INET6, SOCK_DGRAM, 0);      /* Create client socket */
    if (sfd == -1)
        errExit("socket");

    memset(&svaddr, 0, sizeof(struct sockaddr_in6));
    svaddr.sin6_family = AF_INET6;
    svaddr.sin6_port = htons(PORT_NUM);
    if (inet_pton(AF_INET6, argv[1], &svaddr.sin6_addr) <= 0)
        fatal("inet_pton failed for address '%s'", argv[1]);

    /* Send messages to server; echo responses on stdout */

    for (j = 2; j < argc; j++) {
        msgLen = strlen(argv[j]);
        if (sendto(sfd, argv[j], msgLen, 0, (struct sockaddr *) &svaddr,
                    sizeof(struct sockaddr_in6)) != msgLen)
            fatal("sendto");

        numBytes = recvfrom(sfd, resp, BUF_SIZE, 0, NULL, NULL);
        if (numBytes == -1)
            errExit("recvfrom");

        printf("Response %d: %.*s\n", j - 1, (int) numBytes, resp);
    }

    exit(EXIT_SUCCESS);
}
```

sockets/i6d_ucase_cl.c

59.8 Domain Name System (DNS)

In Section 59.10, we describe *getaddrinfo()*, which obtains the IP address(es) corresponding to a hostname, and *getnameinfo()*, which performs the converse task. However, before looking at these functions, we explain how DNS is used to maintain the mappings between hostnames and IP addresses.

Before the advent of DNS, mappings between hostnames and IP addresses were defined in a manually maintained local file, `/etc/hosts`, containing records of the following form:

```
# IP-address    canonical hostname      [aliases]
127.0.0.1       localhost
```

The *gethostbyname()* function (the predecessor to *getaddrinfo()*) obtained an IP address by searching this file, looking for a match on either the canonical hostname (i.e., the official or primary name of the host) or one of the (optional, space-delimited) aliases.

However, the `/etc/hosts` scheme scales poorly, and then becomes impossible, as the number of hosts in the network increases (e.g., the Internet, with millions of hosts).

DNS was devised to address this problem. The key ideas of DNS are the following:

- Hostnames are organized into a hierarchical namespace (Figure 59-2). Each *node* in the DNS hierarchy has a *label* (name), which may be up to 63 characters. At the root of the hierarchy is an unnamed node, the "anonymous root."
- A node's *domain name* consists of all of the names from that node up to the root concatenated together, with each name separated by a period (`.`). For example, `google.com` is the domain name for the node `google`.
- A *fully qualified domain name* (FQDN), such as `www.kernel.org.`, identifies a host within the hierarchy. A fully qualified domain name is distinguished by being terminated by a period, although in many contexts the period may be omitted.
- No single organization or system manages the entire hierarchy. Instead, there is a hierarchy of DNS servers, each of which manages a branch (a *zone*) of the tree. Normally, each zone has a *primary master name server*, and one or more *slave name servers* (sometimes also known as *secondary master name servers*), which provide backup in the event that the primary master name server crashes. Zones may themselves be divided into separately managed smaller zones. When a host is added within a zone, or the mapping of a hostname to an IP address is changed, the administrator responsible for the corresponding local name server updates the name database on that server. (No manual changes are required on any other name-server databases in the hierarchy.)

  > The DNS server implementation employed on Linux is the widely used Berkeley Internet Name Domain (BIND) implementation, *named(8)*, maintained by the *Internet Systems Consortium* (*http://www.isc.org/*). The operation of this daemon is controlled by the file `/etc/named.conf` (see the *named.conf(5)* manual page). The key reference on DNS and BIND is [Albitz & Liu, 2006]. Information about DNS can also be found in Chapter 14 of [Stevens, 1994], Chapter 11 of [Stevens et al., 2004], and Chapter 24 of [Comer, 2000].

- When a program calls *getaddrinfo()* to *resolve* (i.e., obtain the IP address for) a domain name, *getaddrinfo()* employs a suite of library functions (the *resolver library*) that communicate with the local DNS server. If this server can't supply the required information, then it communicates with other DNS servers within the hierarchy in order to obtain the information. Occasionally, this resolution process may take a noticeable amount of time, and DNS servers employ caching techniques to avoid unnecessary communication for frequently queried domain names.

Using the above approach allows DNS to cope with large namespaces, and does not require centralized management of names.

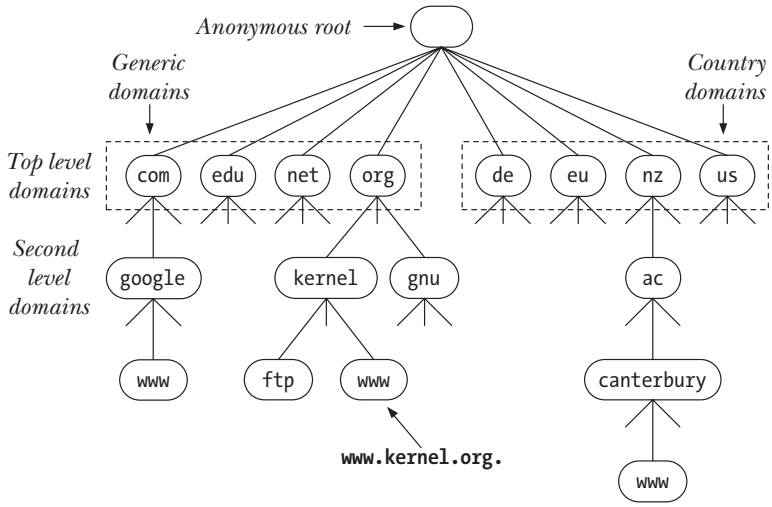

Figure 59-2: A subset of the DNS hierarchy

Recursive and iterative resolution requests

DNS resolution requests fall into two categories: *recursive* and *iterative*. In a recursive request, the requester asks the server to handle the entire task of resolution, including the task of communicating with any other DNS servers, if necessary. When an application on the local host calls *getaddrinfo()*, that function makes a recursive request to the local DNS server. If the local DNS server does not itself have the information to perform the resolution, it resolves the domain name iteratively.

We explain iterative resolution via an example. Suppose that the local DNS server is asked to resolve the name `www.otago.ac.nz`. To do this, it first communicates with one of a small set of *root name servers* that every DNS server is required to know about. (We can obtain a list of these servers using the command *dig . NS* or from the web page at *http://www.root-servers.org/*.) Given the name `www.otago.ac.nz`, the root name server refers the local DNS server to one of the `nz` DNS servers. The local DNS server then queries the `nz` server with the name `www.otago.ac.nz`, and receives a response referring it to the `ac.nz` server. The local DNS server then queries the `ac.nz` server with the name `www.otago.ac.nz`, and is referred to the `otago.ac.nz` server. Finally, the local DNS server queries the `otago.ac.nz` server with the name `www.otago.ac.nz`, and obtains the required IP address.

If we supply an incomplete domain name to *gethostbyname()*, the resolver will attempt to complete it before resolving it. The rules on how a domain name is completed are defined in `/etc/resolv.conf` (see the *resolv.conf(5)* manual page). By default, the resolver will at least try completion using the domain name of the local host. For example, if we are logged in on the machine `oghma.otago.ac.nz` and we type the command *ssh octavo*, the resulting DNS query will be for the name `octavo.otago.ac.nz`.

Top-level domains

The nodes immediately below the anonymous root form the so-called *top-level domains* (TLDs). (Below these are the *second-level domains*, and so on.) TLDs fall into two categories: *generic* and *country*.

Historically, there were seven *generic* TLDs, most of which can be considered international. We have shown four of the original generic TLDs in Figure 59-2. The other three are `int`, `mil`, and `gov`; the latter two are reserved for the United States. In more recent times, a number of new generic TLDs have been added (e.g., `info`, `name`, and `museum`).

Each nation has a corresponding *country* (or *geographical*) TLD (standardized as ISO 3166-1), with a 2-character name. In Figure 59-2, we have shown a few of these: `de` (Germany, *Deutschland*), `eu` (a supra-national geographical TLD for the European Union), `nz` (New Zealand), and `us` (United States of America). Several countries divide their TLD into a set of second-level domains in a manner similar to the generic domains. For example, New Zealand has `ac.nz` (academic institutions), `co.nz` (commercial), and `govt.nz` (government).

59.9 The `/etc/services` File

As noted in Section 58.6.1, well-known port numbers are centrally registered by IANA. Each of these ports has a corresponding *service name*. Because service numbers are centrally managed and are less volatile than IP addresses, an equivalent of the DNS server is usually not necessary. Instead, the port numbers and service names are recorded in the file `/etc/services`. The *getaddrinfo()* and *getnameinfo()* functions use the information in this file to convert service names to port numbers and vice versa.

The `/etc/services` file consists of lines containing three columns, as shown in the following examples:

```
# Service name  port/protocol  [aliases]
echo            7/tcp          Echo     # echo service
echo            7/udp          Echo
ssh             22/tcp                  # Secure Shell
ssh             22/udp
telnet          23/tcp                  # Telnet
telnet          23/udp
smtp            25/tcp                  # Simple Mail Transfer Protocol
smtp            25/udp
domain          53/tcp                  # Domain Name Server
domain          53/udp
http            80/tcp                  # Hypertext Transfer Protocol
http            80/udp
ntp             123/tcp                 # Network Time Protocol
ntp             123/udp
login           513/tcp                 # rlogin(1)
who             513/udp                 # rwho(1)
shell           514/tcp                 # rsh(1)
syslog          514/udp                 # syslog
```

The *protocol* is typically either `tcp` or `udp`. The optional (space-delimited) *aliases* specify alternative names for the service. In addition to the above, lines may include comments starting with the # character.

As noted previously, a given port number refers to distinct entities for UDP and TCP, but IANA policy assigns both port numbers to a service, even if that service uses only one protocol. For example, *telnet*, *ssh*, HTTP, and SMTP all use TCP, but the corresponding UDP port is also assigned to these services. Conversely, NTP uses only UDP, but the TCP port 123 is also assigned to this service. In some cases, a service uses both UDP and TCP; DNS and *echo* are examples of such services. Finally, there are a very few cases where the UDP and TCP ports with the same number are assigned to different services; for example, *rsh* uses TCP port 514, while the *syslog* daemon (Section 37.5) uses UDP port 514. This is because these port numbers were assigned before the adoption of the present IANA policy.

> The `/etc/services` file is merely a record of name-to-number mappings. It is not a reservation mechanism: the appearance of a port number in `/etc/services` doesn't guarantee that it will actually be available for binding by a particular service.

59.10 Protocol-Independent Host and Service Conversion

The *getaddrinfo()* function converts host and service names to IP addresses and port numbers. It was defined in POSIX.1g as the (reentrant) successor to the obsolete *gethostbyname()* and *getservbyname()* functions. (Replacing the use of *gethostbyname()* with *getaddrinfo()* allows us to eliminate IPv4-versus-IPv6 dependencies from our programs.)

The *getnameinfo()* function is the converse of *getaddrinfo()*. It translates a socket address structure (either IPv4 or IPv6) to strings containing the corresponding host and service name. This function is the (reentrant) equivalent of the obsolete *gethostbyaddr()* and *getservbyport()* functions.

> Chapter 11 of [Stevens et al., 2004] describes *getaddrinfo()* and *getnameinfo()* in detail, and provides implementations of these functions. These functions are also described in RFC 3493.

59.10.1 The *getaddrinfo()* Function

Given a host name and a service name, *getaddrinfo()* returns a list of socket address structures, each of which contains an IP address and port number.

```
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *host, const char *service,
                const struct addrinfo *hints, struct addrinfo **result);
```

Returns 0 on success, or nonzero on error

As input, *getaddrinfo()* takes the arguments *host*, *service*, and *hints*. The *host* argument contains either a hostname or a numeric address string, expressed in IPv4 dotted-decimal notation or IPv6 hex-string notation. (To be precise, *getaddrinfo()* accepts IPv4 numeric strings in the more general numbers-and-dots notation described in Section 59.13.1.) The *service* argument contains either a service name or a decimal port number. The *hints* argument points to an *addrinfo* structure that specifies further criteria for selecting the socket address structures returned via *result*. We describe the *hints* argument in more detail below.

As output, *getaddrinfo()* dynamically allocates a linked list of *addrinfo* structures and sets *result* pointing to the beginning of this list. Each of these *addrinfo* structures includes a pointer to a socket address structure corresponding to *host* and *service* (Figure 59-3). The *addrinfo* structure has the following form:

```
struct addrinfo {
    int    ai_flags;            /* Input flags (AI_* constants) */
    int    ai_family;           /* Address family */
    int    ai_socktype;         /* Type: SOCK_STREAM, SOCK_DGRAM */
    int    ai_protocol;         /* Socket protocol */
    size_t ai_addrlen;          /* Size of structure pointed to by ai_addr */
    char  *ai_canonname;        /* Canonical name of host */
    struct sockaddr *ai_addr;   /* Pointer to socket address structure */
    struct addrinfo *ai_next;   /* Next structure in linked list */
};
```

The *result* argument returns a list of structures, rather than a single structure, because there may be multiple combinations of host and service corresponding to the criteria specified in *host*, *service*, and *hints*. For example, multiple address structures could be returned for a host with more than one network interface. Furthermore, if *hints.ai_socktype* was specified as 0, then two structures could be returned—one for a `SOCK_DGRAM` socket, the other for a `SOCK_STREAM` socket—if the given *service* was available for both UDP and TCP.

The fields of each *addrinfo* structure returned via *result* describe properties of the associated socket address structure. The *ai_family* field is set to either `AF_INET` or `AF_INET6`, informing us of the type of the socket address structure. The *ai_socktype* field is set to either `SOCK_STREAM` or `SOCK_DGRAM`, indicating whether this address structure is for a TCP or a UDP service. The *ai_protocol* field returns a protocol value appropriate for the address family and socket type. (The three fields *ai_family*, *ai_socktype*, and *ai_protocol* supply the values required for the arguments used when calling *socket()* to create a socket for this address.) The *ai_addrlen* field gives the size (in bytes) of the socket address structure pointed to by *ai_addr*. The *in_addr* field points to the socket address structure (an *in_addr* structure for IPv4 or an *in6_addr* structure for IPv6). The *ai_flags* field is unused (it is used for the *hints* argument). The *ai_canonname* field is used only in the first *addrinfo* structure, and only if the `AI_CANONNAME` flag is employed in *hints.ai_flags*, as described below.

As with *gethostbyname()*, *getaddrinfo()* may need to send a request to a DNS server, and this request may take some time to complete. The same applies for *getnameinfo()*, which we describe in Section 59.10.4.

We demonstrate the use of *getaddrinfo()* in Section 59.11.

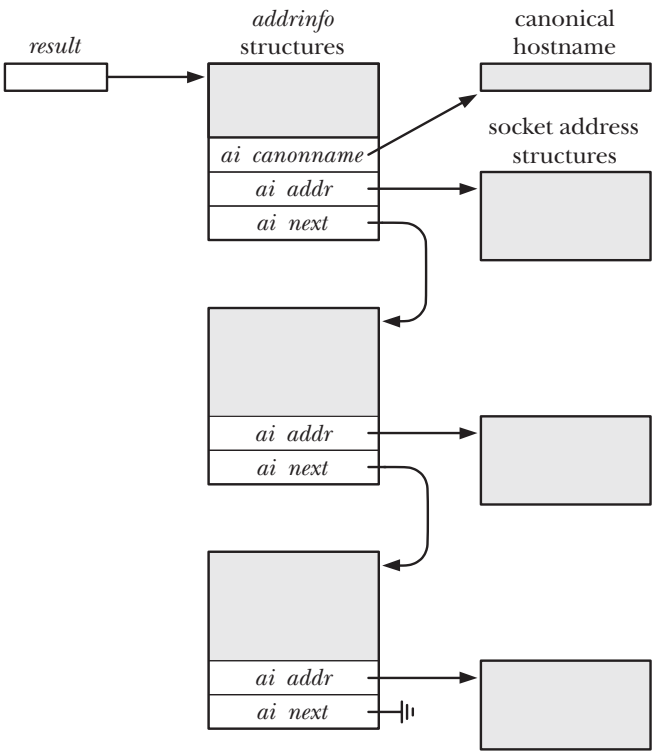

Figure 59-3: Structures allocated and returned by *getaddrinfo()*

The *hints* argument

The *hints* argument specifies further criteria for selecting the socket address structures returned by *getaddrinfo()*. When used as the *hints* argument, only the *ai_flags*, *ai_family*, *ai_socktype*, and *ai_protocol* fields of the *addrinfo* structure can be set. The other fields are unused, and should be initialized to 0 or `NULL`, as appropriate.

The *hints.ai_family* field selects the domain for the returned socket address structures. It may be specified as `AF_INET` or `AF_INET6` (or some other `AF_*` constant, if the implementation supports it). If we are interested in getting back all types of socket address structures, we can specify the value `AF_UNSPEC` for this field.

The *hints.ai_socktype* field specifies the type of socket for which the returned address structure is to be used. If we specify this field as `SOCK_DGRAM`, then a lookup is performed for the UDP service, and a corresponding socket address structure is returned via *result*. If we specify `SOCK_STREAM`, a lookup for the TCP service is performed. If *hints.ai_socktype* is specified as 0, any socket type is acceptable.

The *hints.ai_protocol* field selects the socket protocol for the returned address structures. For our purposes, this field is always specified as 0, meaning that the caller will accept any protocol.

The *hints.ai_flags* field is a bit mask that modifies the behavior of *getaddrinfo()*. This field is formed by ORing together zero or more of the following values:

`AI_ADDRCONFIG`
: Return IPv4 addresses only if there is at least one IPv4 address configured for the local system (other than the IPv4 loopback address), and return IPv6 addresses only if there is at least one IPv6 address configured for the local system (other than the IPv6 loopback address).

`AI_ALL`
: See the description of `AI_V4MAPPED` below.

`AI_CANONNAME`
: If *host* is not `NULL`, return a pointer to a null-terminated string containing the canonical name of the host. This pointer is returned in a buffer pointed to by the *ai_canonname* field of the first of the *addrinfo* structures returned via *result*.

`AI_NUMERICHOST`
: Force interpretation of *host* as a numeric address string. This is used to prevent name resolution in cases where it is unnecessary, since name resolution can be time-consuming.

`AI_NUMERICSERV`
: Interpret *service* as a numeric port number. This flag prevents the invocation of any name-resolution service, which is not required if *service* is a numeric string.

`AI_PASSIVE`
: Return socket address structures suitable for a passive open (i.e., a listening socket). In this case, *host* should be `NULL`, and the IP address component of the socket address structure(s) returned by *result* will contain a wildcard IP address (i.e., `INADDR_ANY` or `IN6ADDR_ANY_INIT`). If this flag is not set, then the address structure(s) returned via *result* will be suitable for use with *connect()* and *sendto()*; if *host* is `NULL`, then the IP address in the returned socket address structures will be set to the loopback IP address (either `INADDR_LOOPBACK` or `IN6ADDR_LOOPBACK_INIT`, according to the domain).

`AI_V4MAPPED`
: If `AF_INET6` was specified in the *ai_family* field of *hints*, then IPv4-mapped IPv6 address structures should be returned in *result* if no matching IPv6 address could be found. If `AI_ALL` is specified in conjunction with `AI_V4MAPPED`, then both IPv6 and IPv4 address structures are returned in *result*, with IPv4 addresses being returned as IPv4-mapped IPv6 address structures.

As noted above for `AI_PASSIVE`, *host* can be specified as `NULL`. It is also possible to specify *service* as `NULL`, in which case the port number in the returned address structures is set to 0 (i.e., we are just interested in resolving hostnames to addresses). It is not permitted, however, to specify both *host* and *service* as `NULL`.

If we don't need to specify any of the above selection criteria in hints, then *hints* may be specified as `NULL`, in which case *ai_socktype* and *ai_protocol* are assumed

as 0, *ai_flags* is assumed as (AI_V4MAPPED | AI_ADDRCONFIG), and *ai_family* is assumed as AF_UNSPEC. (The *glibc* implementation deliberately deviates from SUSv3, which states that if *hints* is NULL, *ai_flags* is assumed as 0.)

59.10.2 Freeing *addrinfo* Lists: *freeaddrinfo()*

The *getaddrinfo()* function dynamically allocates memory for all of the structures referred to by *result* (Figure 59-3). Consequently, the caller must deallocate these structures when they are no longer needed. The *freeaddrinfo()* function is provided to conveniently perform this deallocation in a single step.

```
#include <sys/socket.h>
#include <netdb.h>

void freeaddrinfo(struct addrinfo *result);
```

If we want to preserve a copy of one of the *addrinfo* structures or its associated socket address structure, then we must duplicate the structure(s) before calling *freeaddrinfo()*.

59.10.3 Diagnosing Errors: *gai_strerror()*

On error, *getaddrinfo()* returns one of the nonzero error codes shown in Table 59-1.

Table 59-1: Error returns for *getaddrinfo()* and *getnameinfo()*

Error constant	Description
EAI_ADDRFAMILY	No addresses for *host* exist in *hints.ai_family* (not in SUSv3, but defined on most implementations; *getaddrinfo()* only)
EAI_AGAIN	Temporary failure in name resolution (try again later)
EAI_BADFLAGS	An invalid flag was specified in *hints.ai_flags*
EAI_FAIL	Unrecoverable failure while accessing name server
EAI_FAMILY	Address family specified in *hints.ai_family* is not supported
EAI_MEMORY	Memory allocation failure
EAI_NODATA	No address associated with *host* (not in SUSv3, but defined on most implementations; *getaddrinfo()* only)
EAI_NONAME	Unknown *host* or *service*, or both *host* and *service* were NULL, or AI_NUMERICSERV specified and *service* didn't point to numeric string
EAI_OVERFLOW	Argument buffer overflow
EAI_SERVICE	Specified *service* not supported for *hints.ai_socktype* (*getaddrinfo()* only)
EAI_SOCKTYPE	Specified *hints.ai_socktype* is not supported (*getaddrinfo()* only)
EAI_SYSTEM	System error returned in *errno*

Given one of the error codes in Table 59-1, the *gai_strerror()* function returns a string describing the error. (This string is typically briefer than the description shown in Table 59-1.)

```
#include <netdb.h>

const char *gai_strerror(int errcode);
                                    Returns pointer to string containing error message
```

We can use the string returned by *gai_strerror()* as part of an error message displayed by an application.

59.10.4 The *getnameinfo()* Function

The *getnameinfo()* function is the converse of *getaddrinfo()*. Given a socket address structure (either IPv4 or IPv6), it returns strings containing the corresponding host and service name, or numeric equivalents if the names can't be resolved.

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *addr, socklen_t addrlen, char *host,
                size_t hostlen, char *service, size_t servlen, int flags);
                                    Returns 0 on success, or nonzero on error
```

The *addr* argument is a pointer to the socket address structure that is to be converted. The length of that structure is given in *addrlen*. Typically, the values for *addr* and *addrlen* are obtained from a call to *accept()*, *recvfrom()*, *getsockname()*, or *getpeername()*.

The resulting host and service names are returned as null-terminated strings in the buffers pointed to by *host* and *service*. These buffers must be allocated by the caller, and their sizes must be passed in *hostlen* and *servlen*. The <netdb.h> header file defines two constants to assist in sizing these buffers. NI_MAXHOST indicates the maximum size, in bytes, for a returned hostname string. It is defined as 1025. NI_MAXSERV indicates the maximum size, in bytes, for a returned service name string. It is defined as 32. These two constants are not specified in SUSv3, but they are defined on all UNIX implementations that provide *getnameinfo()*. (Since *glibc* 2.8, we must define one of the feature text macros _BSD_SOURCE, _SVID_SOURCE, or _GNU_SOURCE to obtain the definitions of NI_MAXHOST and NI_MAXSERV.)

If we are not interested in obtaining the hostname, we can specify *host* as NULL and *hostlen* as 0. Similarly, if we don't need the service name, we can specify *service* as NULL and *servlen* as 0. However, at least one of *host* and *service* must be non-NULL (and the corresponding length argument must be nonzero).

The final argument, *flags*, is a bit mask that controls the behavior of *getnameinfo()*. The following constants may be ORed together to form this bit mask:

NI_DGRAM

> By default, *getnameinfo()* returns the name corresponding to a *stream* socket (i.e., TCP) service. Normally, this doesn't matter, because, as noted in Section 59.9, the service names are usually the same for corresponding

TCP and UDP ports. However, in the few instances where the names differ, the NI_DGRAM flag forces the name of the datagram socket (i.e., UDP) service to be returned.

NI_NAMEREQD
: By default, if the hostname can't be resolved, a numeric address string is returned in *host*. If the NI_NAMEREQD flag is specified, an error (EAI_NONAME) is returned instead.

NI_NOFQDN
: By default, the fully qualified domain name for the host is returned. Specifying the NI_NOFQDN flag causes just the first (i.e., the hostname) part of the name to be returned, if this is a host on the local network.

NI_NUMERICHOST
: Force a numeric address string to be returned in *host*. This is useful if we want to avoid a possibly time-consuming call to the DNS server.

NI_NUMERICSERV
: Force a decimal port number string to be returned in *service*. This is useful in cases where we know that the port number doesn't correspond to a service name—for example, if it is an ephemeral port number assigned to the socket by the kernel—and we want to avoid the inefficiency of unnecessarily searching /etc/services.

On success, *getnameinfo()* returns 0. On error, it returns one of the nonzero error codes shown in Table 59-1.

59.11 Client-Server Example (Stream Sockets)

We now have enough information to look at a simple client-server application using TCP sockets. The task performed by this application is the same as that performed by the FIFO client-server application presented in Section 44.8: allocating unique sequence numbers (or ranges of sequence numbers) to clients.

In order to handle the possibility that integers may be represented in different formats on the server and client hosts, we encode all transmitted integers as strings terminated by a newline, and use our *readLine()* function (Listing 59-1) to read these strings.

Common header file

Both the server and the client include the header file shown in Listing 59-5. This file includes various other header files, and defines the TCP port number to be used by the application.

Server program

The server program shown in Listing 59-6 performs the following steps:

- Initialize the server's sequence number either to 1 or to the value supplied in the optional command-line argument ①.

- Ignore the SIGPIPE signal ②. This prevents the server from receiving the SIGPIPE signal if it tries to write to a socket whose peer has been closed; instead, the *write()* fails with the error EPIPE.
- Call *getaddrinfo()* ④ to obtain a set of socket address structures for a TCP socket that uses the port number PORT_NUM. (Instead of using a hard-coded port number, we would more typically use a service name.) We specify the AI_PASSIVE flag ③ so that the resulting socket will be bound to the wildcard address (Section 58.5). As a result, if the server is run on a multihomed host, it can accept connection requests sent to any of the host's network addresses.
- Enter a loop that iterates through the socket address structures returned by the previous step ⑤. The loop terminates when the program finds an address structure that can be used to successfully create and bind a socket ⑦.
- Set the SO_REUSEADDR option for the socket created in the previous step ⑥. We defer discussion of this option until Section 61.10, where we note that a TCP server should usually set this option on its listening socket.
- Mark the socket as a listening socket ⑧.
- Commence an infinite for loop ⑨ that services clients iteratively (Chapter 60). Each client's request is serviced before the next client's request is accepted. For each client, the server performs the following steps:
  - Accept a new connection ⑩. The server passes non-NULL pointers for the second and third arguments to *accept()*, in order to obtain the address of the client. The server displays the client's address (IP address plus port number) on standard output ⑪.
  - Read the client's message ⑫, which consists of a newline-terminated string specifying how many sequence numbers the client wants. The server converts this string to an integer and stores it in the variable *reqLen* ⑬.
  - Send the current value of the sequence number (*seqNum*) back to the client, encoding it as a newline-terminated string ⑭. The client can assume that it has been allocated all of the sequence numbers in the range *seqNum* to *(seqNum + reqLen − 1)*.
  - Update the value of the server's sequence number by adding *reqLen* to *seqNum* ⑮.

Listing 59-5: Header file used by is_seqnum_sv.c and is_seqnum_cl.c

sockets/is_seqnum.h

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <signal.h>
#include "read_line.h"          /* Declaration of readLine() */
#include "tlpi_hdr.h"

#define PORT_NUM "50000"        /* Port number for server */

#define INT_LEN 30              /* Size of string able to hold largest
                                   integer (including terminating '\n') */
```

sockets/is_seqnum.h

Listing 59-6: An iterative server that uses a stream socket to communicate with clients

sockets/is_seqnum_sv.c

```
#define _BSD_SOURCE             /* To get definitions of NI_MAXHOST and
                                   NI_MAXSERV from <netdb.h> */
#include <netdb.h>
#include "is_seqnum.h"

#define BACKLOG 50

int
main(int argc, char *argv[])
{
    uint32_t seqNum;
    char reqLenStr[INT_LEN];            /* Length of requested sequence */
    char seqNumStr[INT_LEN];            /* Start of granted sequence */
    struct sockaddr_storage claddr;
    int lfd, cfd, optval, reqLen;
    socklen_t addrlen;
    struct addrinfo hints;
    struct addrinfo *result, *rp;
#define ADDRSTRLEN (NI_MAXHOST + NI_MAXSERV + 10)
    char addrStr[ADDRSTRLEN];
    char host[NI_MAXHOST];
    char service[NI_MAXSERV];

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [init-seq-num]\n", argv[0]);

①   seqNum = (argc > 1) ? getInt(argv[1], 0, "init-seq-num") : 0;

②   if (signal(SIGPIPE, SIG_IGN) == SIG_ERR)
        errExit("signal");

    /* Call getaddrinfo() to obtain a list of addresses that
       we can try binding to */

    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_canonname = NULL;
    hints.ai_addr = NULL;
    hints.ai_next = NULL;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_family = AF_UNSPEC;        /* Allows IPv4 or IPv6 */
③   hints.ai_flags = AI_PASSIVE | AI_NUMERICSERV;
                        /* Wildcard IP address; service name is numeric */
④   if (getaddrinfo(NULL, PORT_NUM, &hints, &result) != 0)
        errExit("getaddrinfo");

    /* Walk through returned list until we find an address structure
       that can be used to successfully create and bind a socket */

    optval = 1;
⑤   for (rp = result; rp != NULL; rp = rp->ai_next) {
        lfd = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
        if (lfd == -1)
            continue;                   /* On error, try next address */
```

```
⑥          if (setsockopt(lfd, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval))
                   == -1)
               errExit("setsockopt");

⑦          if (bind(lfd, rp->ai_addr, rp->ai_addrlen) == 0)
               break;                       /* Success */

           /* bind() failed: close this socket and try next address */

           close(lfd);
       }

       if (rp == NULL)
           fatal("Could not bind socket to any address");

⑧      if (listen(lfd, BACKLOG) == -1)
           errExit("listen");

       freeaddrinfo(result);

⑨      for (;;) {                  /* Handle clients iteratively */

           /* Accept a client connection, obtaining client's address */

           addrlen = sizeof(struct sockaddr_storage);
⑩          cfd = accept(lfd, (struct sockaddr *) &claddr, &addrlen);
           if (cfd == -1) {
               errMsg("accept");
               continue;
           }

⑪          if (getnameinfo((struct sockaddr *) &claddr, addrlen,
                       host, NI_MAXHOST, service, NI_MAXSERV, 0) == 0)
               snprintf(addrStr, ADDRSTRLEN, "(%s, %s)", host, service);
           else
               snprintf(addrStr, ADDRSTRLEN, "(?UNKNOWN?)");
           printf("Connection from %s\n", addrStr);

           /* Read client request, send sequence number back */

⑫          if (readLine(cfd, reqLenStr, INT_LEN) <= 0) {
               close(cfd);
               continue;                    /* Failed read; skip request */
           }

⑬          reqLen = atoi(reqLenStr);
           if (reqLen <= 0) {              /* Watch for misbehaving clients */
               close(cfd);
               continue;                    /* Bad request; skip it */
           }

⑭          snprintf(seqNumStr, INT_LEN, "%d\n", seqNum);
           if (write(cfd, &seqNumStr, strlen(seqNumStr)) != strlen(seqNumStr))
               fprintf(stderr, "Error on write");
```

```
⑮          seqNum += reqLen;               /* Update sequence number */

           if (close(cfd) == -1)           /* Close connection */
               errMsg("close");
       }
   }
```

sockets/is_seqnum_sv.c

Client program

The client program is shown in Listing 59-7. This program accepts two arguments. The first argument, which is the name of the host on which the server is running, is mandatory. The optional second argument is the length of the sequence desired by the client. The default length is 1. The client performs the following steps:

- Call *getaddrinfo()* to obtain a set of socket address structures suitable for connecting to a TCP server bound to the specified host ①. For the port number, the client specifies PORT_NUM.
- Enter a loop ② that iterates through the socket address structures returned by the previous step, until the client finds one that can be used to successfully create ③ and connect ④ a socket to the server. Since the client has not bound its socket, the *connect()* call causes the kernel to assign an ephemeral port to the socket.
- Send an integer specifying the length of the client's desired sequence ⑤. This integer is sent as a newline-terminated string.
- Read the sequence number sent back by the server (which is likewise a newline-terminated string) ⑥ and print it on standard output ⑦.

When we run the server and the client on the same host, we see the following:

```
$ ./is_seqnum_sv &
[1] 4075
$ ./is_seqnum_cl localhost              Client 1: requests 1 sequence number
Connection from (localhost, 33273)      Server displays client address + port
Sequence number: 0                      Client displays returned sequence number
$ ./is_seqnum_cl localhost 10           Client 2: requests 10 sequence numbers
Connection from (localhost, 33274)
Sequence number: 1
$ ./is_seqnum_cl localhost              Client 3: requests 1 sequence number
Connection from (localhost, 33275)
Sequence number: 11
```

Next, we demonstrate the use of *telnet* for debugging this application:

```
$ telnet localhost 50000                Our server uses this port number
                                        Empty line printed by telnet
Trying 127.0..0.1...
Connection from (localhost, 33276)
Connected to localhost.
Escape character is '^]'.
1                                       Enter length of requested sequence
12                                      telnet displays sequence number and
Connection closed by foreign host.      detects that server closed connection
```

In the shell session log, we see that the kernel cycles sequentially through the ephemeral port numbers. (Other implementations exhibit similar behavior.) On Linux, this behavior is the result of an optimization to minimize hash lookups in the kernel's table of local socket bindings. When the upper limit for these numbers is reached, the kernel recommences allocating an available number starting at the low end of the range (defined by the Linux-specific `/proc/sys/net/ipv4/ip_local_port_range` file).

Listing 59-7: A client that uses stream sockets

sockets/is_seqnum_cl.c

```
#include <netdb.h>
#include "is_seqnum.h"

int
main(int argc, char *argv[])
{
    char *reqLenStr;                    /* Requested length of sequence */
    char seqNumStr[INT_LEN];            /* Start of granted sequence */
    int cfd;
    ssize_t numRead;
    struct addrinfo hints;
    struct addrinfo *result, *rp;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s server-host [sequence-len]\n", argv[0]);

    /* Call getaddrinfo() to obtain a list of addresses that
       we can try connecting to */

    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_canonname = NULL;
    hints.ai_addr = NULL;
    hints.ai_next = NULL;
    hints.ai_family = AF_UNSPEC;                /* Allows IPv4 or IPv6 */
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_NUMERICSERV;

①   if (getaddrinfo(argv[1], PORT_NUM, &hints, &result) != 0)
        errExit("getaddrinfo");

    /* Walk through returned list until we find an address structure
       that can be used to successfully connect a socket */

②   for (rp = result; rp != NULL; rp = rp->ai_next) {
③       cfd = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
        if (cfd == -1)
            continue;                           /* On error, try next address */

④       if (connect(cfd, rp->ai_addr, rp->ai_addrlen) != -1)
            break;                              /* Success */
```

```
            /* Connect failed: close this socket and try next address */

            close(cfd);
        }

        if (rp == NULL)
            fatal("Could not connect socket to any address");

        freeaddrinfo(result);

        /* Send requested sequence length, with terminating newline */

⑤       reqLenStr = (argc > 2) ? argv[2] : "1";
        if (write(cfd, reqLenStr, strlen(reqLenStr)) !=  strlen(reqLenStr))
            fatal("Partial/failed write (reqLenStr)");
        if (write(cfd, "\n", 1) != 1)
            fatal("Partial/failed write (newline)");

        /* Read and display sequence number returned by server */

⑥       numRead = readLine(cfd, seqNumStr, INT_LEN);
        if (numRead == -1)
            errExit("readLine");
        if (numRead == 0)
            fatal("Unexpected EOF from server");

⑦       printf("Sequence number: %s", seqNumStr);          /* Includes '\n' */

        exit(EXIT_SUCCESS);                                /* Closes 'cfd' */
    }
```

sockets/is_seqnum_cl.c

59.12 An Internet Domain Sockets Library

In this section, we use the functions presented in Section 59.10 to implement a library of functions to perform tasks commonly required for Internet domain sockets. (This library abstracts many of the steps shown in the example programs presented in Section 59.11.) Since these functions employ the protocol-independent *getaddrinfo()* and *getnameinfo()* functions, they can be used with both IPv4 and IPv6. Listing 59-8 shows the header file that declares these functions.

Many of the functions in this library have similar arguments:

- The *host* argument is a string containing either a hostname or a numeric address (in IPv4 dotted-decimal, or IPv6 hex-string notation). Alternatively, *host* can be specified as a `NULL` pointer to indicate that the loopback IP address is to be used.
- The *service* argument is either a service name or a port number specified as a decimal string.
- The *type* argument is a socket type, specified as either `SOCK_STREAM` or `SOCK_DGRAM`.

Listing 59-8: Header file for `inet_sockets.c`

```
sockets/inet_sockets.h
#ifndef INET_SOCKETS_H
#define INET_SOCKETS_H          /* Prevent accidental double inclusion */

#include <sys/socket.h>
#include <netdb.h>

int inetConnect(const char *host, const char *service, int type);

int inetListen(const char *service, int backlog, socklen_t *addrlen);

int inetBind(const char *service, int type, socklen_t *addrlen);

char *inetAddressStr(const struct sockaddr *addr, socklen_t addrlen,
                char *addrStr, int addrStrLen);

#define IS_ADDR_STR_LEN 4096
                        /* Suggested length for string buffer that caller
                           should pass to inetAddressStr(). Must be greater
                           than (NI_MAXHOST + NI_MAXSERV + 4) */
#endif
sockets/inet_sockets.h
```

The *inetConnect()* function creates a socket with the given socket *type*, and connects it to the address specified by *host* and *service*. This function is designed for TCP or UDP clients that need to connect their socket to a server socket.

```
#include "inet_sockets.h"

int inetConnect(const char *host, const char *service, int type);
```

Returns a file descriptor on success, or –1 on error

The file descriptor for the new socket is returned as the function result.

The *inetListen()* function creates a listening stream (`SOCK_STREAM`) socket bound to the wildcard IP address on the TCP port specified by *service*. This function is designed for use by TCP servers.

```
#include "inet_sockets.h"

int inetListen(const char *service, int backlog, socklen_t *addrlen);
```

Returns a file descriptor on success, or –1 on error

The file descriptor for the new socket is returned as the function result.

The *backlog* argument specifies the permitted backlog of pending connections (as for *listen()*).

If *addrlen* is specified as a non-NULL pointer, then the location it points to is used to return the size of the socket address structure corresponding to the returned file descriptor. This value allows us to allocate a socket address buffer of the appropriate size to be passed to a later *accept()* call if we want to obtain the address of a connecting client.

The *inetBind()* function creates a socket of the given *type*, bound to the wildcard IP address on the port specified by *service* and *type*. (The socket *type* indicates whether this is a TCP or UDP service.) This function is designed (primarily) for UDP servers and clients to create a socket bound to a specific address.

```
#include "inet_sockets.h"

int inetBind(const char *service, int type, socklen_t *addrlen);
```

Returns a file descriptor on success, or −1 on error

The file descriptor for the new socket is returned as the function result.

As with *inetListen()*, *inetBind()* returns the length of the associated socket address structure for this socket in the location pointed to by *addrlen*. This is useful if we want to allocate a buffer to pass to *recvfrom()* in order to obtain the address of the socket sending a datagram. (Many of the steps required for *inetListen()* and *inetBind()* are the same, and these steps are implemented within the library by a single function, *inetPassiveSocket()*.)

The *inetAddressStr()* function converts an Internet socket address to printable form.

```
#include "inet_sockets.h"

char *inetAddressStr(const struct sockaddr *addr, socklen_t addrlen,
                     char *addrStr, int addrStrLen);
```

Returns pointer to *addrStr*, a string containing host and service name

Given a socket address structure in *addr*, whose length is specified in *addrlen*, *inetAddressStr()* returns a null-terminated string containing the corresponding hostname and port number in the following form:

```
(hostname, port-number)
```

The string is returned in the buffer pointed to by *addrStr*. The caller must specify the size of this buffer in *addrStrLen*. If the returned string would exceed *(addrStrLen − 1)* bytes, it is truncated. The constant IS_ADDR_STR_LEN defines a suggested size for the *addrStr* buffer that should be large enough to handle all possible return strings. As its function result, *inetAddressStr()* returns *addrStr*.

The implementation of the functions described in this section is shown in Listing 59-9.

Listing 59-9: An Internet domain sockets library

sockets/inet_sockets.c

```
#define _BSD_SOURCE             /* To get NI_MAXHOST and NI_MAXSERV
                                   definitions from <netdb.h> */
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include "inet_sockets.h"       /* Declares functions defined here */
#include "tlpi_hdr.h"

int
inetConnect(const char *host, const char *service, int type)
{
    struct addrinfo hints;
    struct addrinfo *result, *rp;
    int sfd, s;

    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_canonname = NULL;
    hints.ai_addr = NULL;
    hints.ai_next = NULL;
    hints.ai_family = AF_UNSPEC;        /* Allows IPv4 or IPv6 */
    hints.ai_socktype = type;

    s = getaddrinfo(host, service, &hints, &result);
    if (s != 0) {
        errno = ENOSYS;
        return -1;
    }

    /* Walk through returned list until we find an address structure
       that can be used to successfully connect a socket */

    for (rp = result; rp != NULL; rp = rp->ai_next) {
        sfd = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
        if (sfd == -1)
            continue;                   /* On error, try next address */

        if (connect(sfd, rp->ai_addr, rp->ai_addrlen) != -1)
            break;                      /* Success */

        /* Connect failed: close this socket and try next address */

        close(sfd);
    }

    freeaddrinfo(result);

    return (rp == NULL) ? -1 : sfd;
}
```

```
static int              /* Public interfaces: inetBind() and inetListen() */
inetPassiveSocket(const char *service, int type, socklen_t *addrlen,
                  Boolean doListen, int backlog)
{
    struct addrinfo hints;
    struct addrinfo *result, *rp;
    int sfd, optval, s;

    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_canonname = NULL;
    hints.ai_addr = NULL;
    hints.ai_next = NULL;
    hints.ai_socktype = type;
    hints.ai_family = AF_UNSPEC;        /* Allows IPv4 or IPv6 */
    hints.ai_flags = AI_PASSIVE;        /* Use wildcard IP address */

    s = getaddrinfo(NULL, service, &hints, &result);
    if (s != 0)
        return -1;

    /* Walk through returned list until we find an address structure
       that can be used to successfully create and bind a socket */

    optval = 1;
    for (rp = result; rp != NULL; rp = rp->ai_next) {
        sfd = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
        if (sfd == -1)
            continue;                   /* On error, try next address */

        if (doListen) {
            if (setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR, &optval,
                    sizeof(optval)) == -1) {
                close(sfd);
                freeaddrinfo(result);
                return -1;
            }
        }

        if (bind(sfd, rp->ai_addr, rp->ai_addrlen) == 0)
            break;                      /* Success */

        /* bind() failed: close this socket and try next address */

        close(sfd);
    }

    if (rp != NULL && doListen) {
        if (listen(sfd, backlog) == -1) {
            freeaddrinfo(result);
            return -1;
        }
    }

    if (rp != NULL && addrlen != NULL)
        *addrlen =  rp->ai_addrlen;     /* Return address structure size */
```

```
    freeaddrinfo(result);

    return (rp == NULL) ? -1 : sfd;
}

int
inetListen(const char *service, int backlog, socklen_t *addrlen)
{
    return inetPassiveSocket(service, SOCK_STREAM, addrlen, TRUE, backlog);
}

int
inetBind(const char *service, int type, socklen_t *addrlen)
{
    return inetPassiveSocket(service, type, addrlen, FALSE, 0);
}

char *
inetAddressStr(const struct sockaddr *addr, socklen_t addrlen,
               char *addrStr, int addrStrLen)
{
    char host[NI_MAXHOST], service[NI_MAXSERV];

    if (getnameinfo(addr, addrlen, host, NI_MAXHOST,
                    service, NI_MAXSERV, NI_NUMERICSERV) == 0)
        snprintf(addrStr, addrStrLen, "(%s, %s)", host, service);
    else
        snprintf(addrStr, addrStrLen, "(?UNKNOWN?)");

    addrStr[addrStrLen - 1] = '\0';     /* Ensure result is null-terminated */
    return addrStr;
}
```

sockets/inet_sockets.c

59.13 Obsolete APIs for Host and Service Conversions

In the following sections, we describe the older, now obsolete functions for converting host names and service names to and from binary and presentation formats. Although new programs should perform these conversions using the modern functions described earlier in this chapter, a knowledge of the obsolete functions is useful because we may encounter them in older code.

59.13.1 The *inet_aton()* and *inet_ntoa()* Functions

The *inet_aton()* and *inet_ntoa()* functions convert IPv4 addresses between dotted-decimal notation and binary form (in network byte order). These functions are nowadays made obsolete by *inet_pton()* and *inet_ntop()*.

The *inet_aton()* ("ASCII to network") function converts the dotted-decimal string pointed to by *str* into an IPv4 address in network byte order, which is returned in the *in_addr* structure pointed to by *addr*.

```
#include <arpa/inet.h>

int inet_aton(const char *str, struct in_addr *addr);
```

Returns 1 (true) if *str* is a valid dotted-decimal address, or 0 (false) on error

The *inet_aton()* function returns 1 if the conversion was successful, or 0 if *str* was invalid.

The numeric components of the string given to *inet_aton()* need not be decimal. They can be octal (specified by a leading 0) or hexadecimal (specified by a leading 0x or 0X). Furthermore, *inet_aton()* supports shorthand forms that allow an address to be specified using fewer than four numeric components. (See the *inet(3)* manual page for details.) The term *numbers-and-dots notation* is used for the more general address strings that employ these features.

SUSv3 doesn't specify *inet_aton()*. Nevertheless, this function is available on most implementations. On Linux, we must define one of the feature test macros `_BSD_SOURCE`, `_SVID_SOURCE`, or `_GNU_SOURCE` in order to obtain the declaration of *inet_aton()* from `<arpa/inet.h>`.

The *inet_ntoa()* ("network to ASCII") function performs the converse of *inet_aton()*.

```
#include <arpa/inet.h>

char *inet_ntoa(struct in_addr addr);
```

Returns pointer to (statically allocated) dotted-decimal string version of *addr*

Given an *in_addr* structure (a 32-bit IPv4 address in network byte order), *inet_ntoa()* returns a pointer to a (statically allocated) string containing the address in dotted-decimal notation.

Because the string returned by *inet_ntoa()* is statically allocated, it is overwritten by successive calls.

59.13.2 The *gethostbyname()* and *gethostbyaddr()* Functions

The *gethostbyname()* and *gethostbyaddr()* functions allow conversion between hostnames and IP addresses. These functions are nowadays made obsolete by *getaddrinfo()* and *getnameinfo()*.

```
#include <netdb.h>

extern int h_errno;

struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const char *addr, socklen_t len, int type);
```

Both return pointer to (statically allocated) *hostent* structure on success, or `NULL` on error

The *gethostbyname()* function resolves the hostname given in *name*, returning a pointer to a statically allocated *hostent* structure containing information about that hostname. This structure has the following form:

```
struct hostent {
    char  *h_name;              /* Official (canonical) name of host */
    char **h_aliases;           /* NULL-terminated array of pointers
                                   to alias strings */
    int    h_addrtype;          /* Address type (AF_INET or AF_INET6) */
    int    h_length;            /* Length (in bytes) of addresses pointed
                                   to by h_addr_list (4 bytes for AF_INET,
                                   16 bytes for AF_INET6) */
    char **h_addr_list;         /* NULL-terminated array of pointers to
                                   host IP addresses (in_addr or in6_addr
                                   structures) in network byte order */
};

#define h_addr  h_addr_list[0]
```

The *h_name* field returns the official name of the host, as a null-terminated string. The *h_aliases* fields points to an array of pointers to null-terminated strings containing aliases (alternative names) for this hostname.

The *h_addr_list* field is an array of pointers to IP address structures for this host. (A multihomed host has more than one address.) This list consists of either *in_addr* or *in6_addr* structures. We can determine the type of these structures from the *h_addrtype* field, which contains either `AF_INET` or `AF_INET6`, and their length from the *h_length* field. The *h_addr* definition is provided for backward compatibility with earlier implementations (e.g., 4.2BSD) that returned just one address in the *hostent* structure. Some existing code relies on this name (and thus is not multihomed-host aware).

With modern versions of *gethostbyname()*, *name* can also be specified as a numeric IP address string; that is, numbers-and-dots notation for IPv4 or hex-string notation for IPv6. In this case, no lookup is performed; instead, *name* is copied into the *h_name* field of the *hostent* structure, and *h_addr_list* is set to the binary equivalent of *name*.

The *gethostbyaddr()* function performs the converse of *gethostbyname()*. Given a binary IP address, it returns a *hostent* structure containing information about the host with that address.

On error (e.g., a name could not be resolved), both *gethostbyname()* and *gethostbyaddr()* return a `NULL` pointer and set the global variable *h_errno*. As the name suggests, this variable is analogous to *errno* (possible values placed in this variable are described in the *gethostbyname(3)* manual page), and the *herror()* and *hstrerror()* functions are analogous to *perror()* and *strerror()*.

The *herror()* function displays (on standard error) the string given in *str*, followed by a colon (`:`), and then a message for the current error in *h_errno*. Alternatively, we can use *hstrerror()* to obtain a pointer to a string corresponding to the error value specified in *err*.

```
#define _BSD_SOURCE           /* Or _SVID_SOURCE or _GNU_SOURCE */
#include <netdb.h>

void herror(const char *str);

const char *hstrerror(int err);
                        Returns pointer to h_errno error string corresponding to err
```

Listing 59-10 demonstrates the use of *gethostbyname()*. This program displays *hostent* information for each of the hosts named on its command line. The following shell session demonstrates the use of this program:

```
$ ./t_gethostbyname www.jambit.com
Canonical name: jamjam1.jambit.com
        alias(es):      www.jambit.com
        address type:   AF_INET
        address(es):    62.245.207.90
```

Listing 59-10: Using *gethostbyname()* to retrieve host information

sockets/t_gethostbyname.c

```
#define _BSD_SOURCE     /* To get hstrerror() declaration from <netdb.h> */
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    struct hostent *h;
    char **pp;
    char str[INET6_ADDRSTRLEN];

    for (argv++; *argv != NULL; argv++) {
        h = gethostbyname(*argv);
        if (h == NULL) {
            fprintf(stderr, "gethostbyname() failed for '%s': %s\n",
                    *argv, hstrerror(h_errno));
            continue;
        }

        printf("Canonical name: %s\n", h->h_name);

        printf("        alias(es):     ");
        for (pp = h->h_aliases; *pp != NULL; pp++)
            printf(" %s", *pp);
        printf("\n");
```

```
        printf("        address type:   %s\n",
                (h->h_addrtype == AF_INET) ? "AF_INET" :
                (h->h_addrtype == AF_INET6) ? "AF_INET6" : "???");

        if (h->h_addrtype == AF_INET || h->h_addrtype == AF_INET6) {
            printf("        address(es):   ");
            for (pp = h->h_addr_list; *pp != NULL; pp++)
                printf(" %s", inet_ntop(h->h_addrtype, *pp,
                                        str, INET6_ADDRSTRLEN));
            printf("\n");
        }
    }

    exit(EXIT_SUCCESS);
}
```

sockets/t_gethostbyname.c

59.13.3 The *getservbyname()* and *getservbyport()* Functions

The *getservbyname()* and *getservbyport()* functions retrieve records from the /etc/services file (Section 59.9). These functions are nowadays made obsolete by *getaddrinfo()* and *getnameinfo()*.

```
#include <netdb.h>

struct servent *getservbyname(const char *name, const char *proto);
struct servent *getservbyport(int port, const char *proto);
```

Both return pointer to a (statically allocated) *servent* structure on success, or NULL on not found or error

The *getservbyname()* function looks up the record whose service name (or one of its aliases) matches *name* and whose protocol matches *proto*. The *proto* argument is a string such as *tcp* or *udp*, or it can be NULL. If *proto* is specified as NULL, any record whose service name matches *name* is returned. (This is usually sufficient since, where both UDP and TCP records with the same name exist in the /etc/services file, they normally have the same port number.) If a matching record is found, then *getservbyname()* returns a pointer to a statically allocated structure of the following type:

```
struct servent {
    char  *s_name;          /* Official service name */
    char **s_aliases;       /* Pointers to aliases (NULL-terminated) */
    int    s_port;          /* Port number (in network byte order) */
    char  *s_proto;         /* Protocol */
};
```

Typically, we call *getservbyname()* only in order to obtain the port number, which is returned in the *s_port* field.

The *getservbyport()* function performs the converse of *getservbyname()*. It returns a *servent* record containing information from the /etc/services record whose port number matches *port* and whose protocol matches *proto*. Again, we can specify *proto* as NULL, in which case the call will return any record whose port number matches

the one specified in *port*. (This may not return the desired result in the few cases mentioned above where the same port number maps to different service names in UDP and TCP.)

> An example of the use of the *getservbyname()* function is provided in the file `files/t_getservbyname.c` in the source code distribution for this book.

59.14 UNIX Versus Internet Domain Sockets

When writing applications that communicate over a network, we must necessarily use Internet domain sockets. However, when using sockets to communicate between applications on the same system, we have the choice of using either Internet or UNIX domain sockets. In the case, which domain should we use and why?

Writing an application using just Internet domain sockets is often the simplest approach, since it will work on both a single host and across a network. However, there are some reasons why we may choose to use UNIX domain sockets:

- On some implementations, UNIX domain sockets are faster than Internet domain sockets.
- We can use directory (and, on Linux, file) permissions to control access to UNIX domain sockets, so that only applications with a specified user or group ID can connect to a listening stream socket or send a datagram to a datagram socket. This provides a simple method of authenticating clients. With Internet domain sockets, we need to do rather more work if we wish to authenticate clients.
- Using UNIX domain sockets, we can pass open file descriptors and sender credentials, as summarized in Section 61.13.3.

59.15 Further Information

There is a wealth of printed and online resources on TCP/IP and the sockets API:

- The key book on network programming with the sockets API is [Stevens at al., 2004]. [Snader, 2000] adds some useful guidelines on sockets programming.
- [Stevens, 1994] and [Wright & Stevens, 1995] describe TCP/IP in detail. [Comer, 2000], [Comer & Stevens, 1999], [Comer & Stevens, 2000], [Kozierok, 2005], and [Goralksi, 2009] also provide good coverage of the same material.
- [Tanenbaum, 2002] provides general background on computer networks.
- [Herbert, 2004] describes the details of the Linux 2.6 TCP/IP stack.
- The GNU C library manual (online at *http://www.gnu.org/*) has an extensive discussion of the sockets API.
- The IBM Redbook, *TCP/IP Tutorial and Technical Overview*, provides lengthy coverage of networking concepts, TCP/IP internals, the sockets API, and a host of related topics. It is freely downloadable from *http://www.redbooks.ibm.com/*.
- [Gont, 2008] and [Gont, 2009b] provide security assessments of IPv4 and TCP.
- The Usenet newsgroup *comp.protocols.tcp-ip* is dedicated to questions related to the TCP/IP networking protocols.

- [Sarolahti & Kuznetsov, 2002] describes congestion control and other details of the Linux TCP implementation.
- Linux-specific information can be found in the following manual pages: *socket(7)*, *ip(7)*, *raw(7)*, *tcp(7)*, *udp(7)*, and *packet(7)*.
- See also the RFC list in Section 58.7.

59.16 Summary

Internet domain sockets allow applications on different hosts to communicate via a TCP/IP network. An Internet domain socket address consists of an IP address and a port number. In IPv4, an IP address is a 32-bit number; in IPv6, it is a 128-bit number. Internet domain datagram sockets operate over UDP, providing connectionless, unreliable, message-oriented communication. Internet domain stream sockets operate over TCP, and provide a reliable, bidirectional, byte-stream communication channel between two connected applications.

Different computer architectures use different conventions for representing data types. For example, integers may be stored in little-endian or big-endian form, and different computers may use different numbers of bytes to represent numeric types such as *int* or *long*. These differences mean that we need to employ some architecture-independent representation when transferring data between heterogeneous machines connected via a network. We noted that various marshalling standards exist to deal this problem, and also described a simple solution used by many applications: encoding all transmitted data in text form, with fields delimited by a designated character (usually a newline).

We looked at a range of functions that can be used to convert between (numeric) string representations of IP addresses (dotted-decimal for IPv4 and hex-string for IPv6) and their binary equivalents. However, it is generally preferable to use host and service names rather than numbers, since names are easier to remember and continue to be usable, even if the corresponding number is changed. We looked at various functions that convert host and service names to their numeric equivalents and vice versa. The modern function for translating host and service names into socket addresses is *getaddrinfo()*, but it is common to see the historical functions *gethostbyname()* and *getservbyname()* in existing code.

Consideration of hostname conversions led us into a discussion of DNS, which implements a distributed database for a hierarchical directory service. The advantage of DNS is that the management of the database is not centralized. Instead, local zone administrators update changes for the hierarchical component of the database for which they are responsible, and DNS servers communicate with one another in order to resolve a hostname.

59.17 Exercises

59-1. When reading large quantities of data, the *readLine()* function shown in Listing 59-1 is inefficient, since a system call is required to read each character. A more efficient interface would read a block of characters into a buffer and extract a line at a time from this buffer. Such an interface might consist of two functions. The first of these functions, which might be called *readLineBufInit(fd, &rlbuf)*, initializes the bookkeeping

data structure pointed to by *rlbuf*. This structure includes space for a data buffer, the size of that buffer, and a pointer to the next "unread" character in that buffer. It also includes a copy of the file descriptor given in the argument *fd*. The second function, *readLineBuf(&rlbuf)*, returns the next line from the buffer associated with *rlbuf*. If required, this function reads a further block of data from the file descriptor saved in *rlbuf*. Implement these two functions. Modify the programs in Listing 59-6 (`is_seqnum_sv.c`) and Listing 59-7 (`is_seqnum_cl.c`) to use these functions.

59-2. Modify the programs in Listing 59-6 (`is_seqnum_sv.c`) and Listing 59-7 (`is_seqnum_cl.c`) to use the *inetListen()* and *inetConnect()* functions provided in Listing 59-9 (`inet_sockets.c`).

59-3. Write a UNIX domain sockets library with an API similar to the Internet domain sockets library shown in Section 59.12. Rewrite the programs in Listing 57-3 (`us_xfr_sv.c`, on page 1168) and Listing 57-4 (`us_xfr_cl.c`, on page 1169) to use this library.

59-4. Write a network server that stores name-value pairs. The server should allow names to be added, deleted, modified, and retrieved by clients. Write one or more client programs to test the server. Optionally, implement some kind of security mechanism that allows only the client that created the name to delete it or to modify the value associated with it.

59-5. Suppose that we create two Internet domain datagram sockets, bound to specific addresses, and connect the first socket to the second. What happens if we create a third datagram socket and try to send (*sendto()*) a datagram via that socket to the first socket? Write a program to determine the answer.

60

SOCKETS: SERVER DESIGN

This chapter discusses the fundamentals of designing iterative and concurrent servers and describes *inetd*, a special daemon designed to facilitate the creation of Internet servers.

60.1 Iterative and Concurrent Servers

Two common designs for network servers using sockets are the following:

- *Iterative*: The server handles one client at a time, processing that client's request(s) completely, before proceeding to the next client.
- *Concurrent*: The server is designed to handle multiple clients simultaneously.

We have already seen an example of an iterative server using FIFOs in Section 44.8 and an example of a concurrent server using System V message queues in Section 46.8.

Iterative servers are usually suitable only when client requests can be handled quickly, since each client must wait until all of the preceding clients have been serviced. A typical scenario for employing an iterative server is where the client and server exchange a single request and response.

Concurrent servers are suitable when a significant amount of processing time is required to handle each request, or where the client and server engage in an extended conversation, passing messages back and forth. In this chapter, we mainly focus on the traditional (and simplest) method of designing a concurrent server: creating a new child process for each new client. Each server child performs all tasks necessary to service a single client and then terminates. Since each of these processes can operate independently, multiple clients can be handled simultaneously. The principal task of the main server process (the parent) is to create a new child process for each new client. (A variation on this approach is to create a new thread for each client.)

In the following sections, we look at examples of an iterative and a concurrent server using Internet domain sockets. These two servers implement the *echo* service (RFC 862), a rudimentary service that returns a copy of whatever the client sends it.

60.2 An Iterative UDP *echo* Server

In this and the next section, we present servers for the *echo* service. The *echo* service operates on both UDP and TCP port 7. (Since this is a reserved port, the *echo* server must be run with superuser privileges.)

The UDP *echo* server continuously reads datagrams, returning a copy of each datagram to the sender. Since the server needs to handle only a single message at a time, an iterative server design suffices. The header file for the server is shown in Listing 60-1.

Listing 60-1: Header file for `id_echo_sv.c` and `id_echo_cl.c`

sockets/id_echo.h

```
#include "inet_sockets.h"       /* Declares our socket functions */
#include "tlpi_hdr.h"

#define SERVICE "echo"          /* Name of UDP service */

#define BUF_SIZE 500            /* Maximum size of datagrams that can
                                   be read by client and server */
```

sockets/id_echo.h

Listing 60-2 shows the implementation of the server. Note the following points regarding the server implementation:

- We use the *becomeDaemon()* function of Section 37.2 to turn the server into a daemon.
- To shorten this program, we employ the Internet domain sockets library developed in Section 59.12.
- If the server can't send a reply to the client, it logs a message using *syslog()*.

> In a real-world application, we would probably apply some rate limit to the messages written with *syslog()*, both to prevent the possibility of an attacker filling the system log and because each call to *syslog()* is expensive, since (by default) *syslog()* in turn calls *fsync()*.

Listing 60-2: An iterative server that implements the UDP *echo* service

sockets/id_echo_sv.c

```
#include <syslog.h>
#include "id_echo.h"
#include "become_daemon.h"

int
main(int argc, char *argv[])
{
    int sfd;
    ssize_t numRead;
    socklen_t addrlen, len;
    struct sockaddr_storage claddr;
    char buf[BUF_SIZE];
    char addrStr[IS_ADDR_STR_LEN];

    if (becomeDaemon(0) == -1)
        errExit("becomeDaemon");

    sfd = inetBind(SERVICE, SOCK_DGRAM, &addrlen);
    if (sfd == -1) {
        syslog(LOG_ERR, "Could not create server socket (%s)", strerror(errno));
        exit(EXIT_FAILURE);
    }

    /* Receive datagrams and return copies to senders */

    for (;;) {
        len = sizeof(struct sockaddr_storage);
        numRead = recvfrom(sfd, buf, BUF_SIZE, 0,
                           (struct sockaddr *) &claddr, &len);
        if (numRead == -1)
            errExit("recvfrom");

        if (sendto(sfd, buf, numRead, 0, (struct sockaddr *) &claddr, len)
                    != numRead)
            syslog(LOG_WARNING, "Error echoing response to %s (%s)",
                    inetAddressStr((struct sockaddr *) &claddr, len,
                                   addrStr, IS_ADDR_STR_LEN),
                    strerror(errno));
    }
}
```

sockets/id_echo_sv.c

To test the server, we use the client program shown in Listing 60-3. This program also employs the Internet domain sockets library developed in Section 59.12. As its first command-line argument, the client program expects the name of the host on which the server resides. The client executes a loop in which it sends each of its remaining command-line arguments to the server as separate datagrams, and reads and prints each response datagram sent back by the server.

Listing 60-3: A client for the UDP *echo* service

sockets/id_echo_cl.c

```
#include "id_echo.h"

int
main(int argc, char *argv[])
{
    int sfd, j;
    size_t len;
    ssize_t numRead;
    char buf[BUF_SIZE];

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s: host msg...\n", argv[0]);

    /* Construct server address from first command-line argument */

    sfd = inetConnect(argv[1], SERVICE, SOCK_DGRAM);
    if (sfd == -1)
        fatal("Could not connect to server socket");

    /* Send remaining command-line arguments to server as separate datagrams */

    for (j = 2; j < argc; j++) {
        len = strlen(argv[j]);
        if (write(sfd, argv[j], len) != len)
            fatal("partial/failed write");

        numRead = read(sfd, buf, BUF_SIZE);
        if (numRead == -1)
            errExit("read");

        printf("[%ld bytes] %.*s\n", (long) numRead, (int) numRead, buf);
    }

    exit(EXIT_SUCCESS);
}
```

sockets/id_echo_cl.c

Here is an example of what we see when we run the server and two instances of the client:

```
$ su                                    Need privilege to bind reserved port
Password:
# ./id_echo_sv                          Server places itself in background
# exit                                  Cease to be superuser
$ ./id_echo_cl localhost hello world    This client sends two datagrams
[5 bytes] hello                         Client prints responses from server
[5 bytes] world
$ ./id_echo_cl localhost goodbye        This client sends one datagram
[7 bytes] goodbye
```

60.3 A Concurrent TCP *echo* Server

The TCP *echo* service also operates on port 7. The TCP *echo* server accepts a connection and then loops continuously, reading all transmitted data and sending it back to the client on the same socket. The server continues reading until it detects end-of-file, at which point it closes its socket (so that the client sees end-of-file if it is still reading from its socket).

Since the client may send an indefinite amount of data to the server (and thus servicing the client may take an indefinite amount of time), a concurrent server design is appropriate, so that multiple clients can be simultaneously served. The server implementation is shown in Listing 60-4. (We show an implementation of a client for this service in Section 61.2.) Note the following points about the implementation:

- The server becomes a daemon by calling the *becomeDaemon()* function shown in Section 37.2.
- To shorten this program, we employ the Internet domain sockets library shown in Listing 59-9 (page 1228).
- Since the server creates a child process for each client connection, we must ensure that zombies are reaped. We do this within a `SIGCHLD` handler.
- The main body of the server consists of a `for` loop that accepts a client connection and then uses *fork()* to create a child process that invokes the *handleRequest()* function to handle that client. In the meantime, the parent continues around the `for` loop to accept the next client connection.

  > In a real-world application, we would probably include some code in our server to place an upper limit on the number of child processes that the server could create, in order to prevent an attacker from attempting a remote fork bomb by using the service to create so many processes on the system that it becomes unusable. We could impose this limit by adding extra code in the server to count the number of children currently executing (this count would be incremented after a successful *fork()* and decremented as each child was reaped in the `SIGCHLD` handler). If the limit on the number of children were reached, we could then temporarily stop accepting connections (or alternatively, accept connections and then immediately close them).

- After each *fork()*, the file descriptors for the listening and connected sockets are duplicated in the child (Section 24.2.1). This means that both the parent and the child could communicate with the client using the connected socket. However, only the child needs to perform such communication, and so the parent closes the file descriptor for the connected socket immediately after the *fork()*. (If the parent did not do this, then the socket would never actually be closed; furthermore, the parent would eventually run out of file descriptors.) Since the child doesn't accept new connections, it closes its duplicate of the file descriptor for the listening socket.
- Each child process terminates after handling a single client.

Listing 60-4: A concurrent server that implements the TCP *echo* service

sockets/is_echo_sv.c

```
#include <signal.h>
#include <syslog.h>
#include <sys/wait.h>
#include "become_daemon.h"
#include "inet_sockets.h"       /* Declarations of inet*() socket functions */
#include "tlpi_hdr.h"

#define SERVICE "echo"          /* Name of TCP service */
#define BUF_SIZE 4096

static void             /* SIGCHLD handler to reap dead child processes */
grimReaper(int sig)
{
    int savedErrno;             /* Save 'errno' in case changed here */

    savedErrno = errno;
    while (waitpid(-1, NULL, WNOHANG) > 0)
        continue;
    errno = savedErrno;
}

/* Handle a client request: copy socket input back to socket */

static void
handleRequest(int cfd)
{
    char buf[BUF_SIZE];
    ssize_t numRead;

    while ((numRead = read(cfd, buf, BUF_SIZE)) > 0) {
        if (write(cfd, buf, numRead) != numRead) {
            syslog(LOG_ERR, "write() failed: %s", strerror(errno));
            exit(EXIT_FAILURE);
        }
    }

    if (numRead == -1) {
        syslog(LOG_ERR, "Error from read(): %s", strerror(errno));
        exit(EXIT_FAILURE);
    }
}

int
main(int argc, char *argv[])
{
    int lfd, cfd;               /* Listening and connected sockets */
    struct sigaction sa;

    if (becomeDaemon(0) == -1)
        errExit("becomeDaemon");
```

```
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    sa.sa_handler = grimReaper;
    if (sigaction(SIGCHLD, &sa, NULL) == -1) {
        syslog(LOG_ERR, "Error from sigaction(): %s", strerror(errno));
        exit(EXIT_FAILURE);
    }

    lfd = inetListen(SERVICE, 10, NULL);
    if (lfd == -1) {
        syslog(LOG_ERR, "Could not create server socket (%s)", strerror(errno));
        exit(EXIT_FAILURE);
    }

    for (;;) {
        cfd = accept(lfd, NULL, NULL);  /* Wait for connection */
        if (cfd == -1) {
            syslog(LOG_ERR, "Failure in accept(): %s", strerror(errno));
            exit(EXIT_FAILURE);
        }

        /* Handle each client request in a new child process */

        switch (fork()) {
        case -1:
            syslog(LOG_ERR, "Can't create child (%s)", strerror(errno));
            close(cfd);                 /* Give up on this client */
            break;                      /* May be temporary; try next client */

        case 0:                         /* Child */
            close(lfd);                 /* Unneeded copy of listening socket */
            handleRequest(cfd);
            _exit(EXIT_SUCCESS);

        default:                        /* Parent */
            close(cfd);                 /* Unneeded copy of connected socket */
            break;                      /* Loop to accept next connection */
        }
    }
}
```

sockets/is_echo_sv.c

60.4 Other Concurrent Server Designs

The traditional concurrent server model described in the previous section is adequate for many applications that need to simultaneously handle multiple clients via TCP connections. However, for very high-load servers (for example, web servers handling thousands of requests per minute), the cost of creating a new child (or even thread) for each client imposes a significant burden on the server (refer to Section 28.3), and alternative designs need to be employed. We briefly consider some of these alternatives.

Preforked and prethreaded servers

Preforked and prethreaded servers are described in some detail in Chapter 30 of [Stevens et al., 2004]. The key ideas are the following:

- Instead of creating a new child process (or thread) for each client, the server precreates a fixed number of child processes (or threads) immediately on startup (i.e., before any client requests are even received). These children constitute a so-called *server pool*.
- Each child in the server pool handles one client at a time, but instead of terminating after handling the client, the child fetches the next client to be serviced and services it, and so on.

Employing the above technique requires some careful management within the server application. The server pool should be large enough to ensure adequate response to client requests. This means that the server parent must monitor the number of unoccupied children, and, in times of peak load, increase the size of the pool so that there are always enough child processes available to immediately serve new clients. If the load decreases, then the size of the server pool should be reduced, since having excess processes on the system can degrade overall system performance.

In addition, the children in the server pool must follow some protocol to allow them to exclusively select individual client connections. On most UNIX implementations (including Linux), it is sufficient to have each child in the pool block in an *accept()* call on the listening descriptor. In other words, the server parent creates the listening socket before creating any children, and each of the children inherits a file descriptor for the socket during the *fork()*. When a new client connection arrives, only one of the children will complete the *accept()* call. However, because *accept()* is not an atomic system call on some older implementations, the call may need to be bracketed by some mutual-exclusion technique (e.g., a file lock) to ensure that only one child at a time performs the call ([Stevens et al., 2004]).

> There are alternatives to having all of the children in the server pool perform *accept()* calls. If the server pool consists of separate processes, the server parent can perform the *accept()* call, and then pass the file descriptor containing the new connection to one of the free processes in the pool, using a technique that we briefly describe in Section 61.13.3. If the server pool consists of threads, the main thread can perform the *accept()* call, and then inform one of the free server threads that a new client is available on the connected descriptor.

Handling multiple clients from a single process

In some cases, we can design a single server process to handle multiple clients. To do this, we must employ one of the I/O models (I/O multiplexing, signal-driven I/O, or *epoll*) that allow a single process to simultaneously monitor multiple file descriptors for I/O events. These models are described in Chapter 63.

In a single-server design, the server process must take on some of the scheduling tasks that are normally handled by the kernel. In a solution that involves one server process per client, we can rely on the kernel to ensure that each server process (and thus client) gets a fair share of access to the resources of the server host. But when we use a single server process to handle multiple clients, the server must do some work

to ensure that one or a few clients don't monopolize access to the server while other clients are starved. We say a little more about this point in Section 63.4.6.

Using server farms

Other approaches to handling high client loads involve the use of multiple server systems—a *server farm*.

One of the simplest approaches to building a server farm (employed by some web servers) is *DNS round-robin load sharing* (or *load distribution*), where the authoritative name server for a zone maps the same domain name to several IP addresses (i.e., several servers share the same domain name). Successive requests to the DNS server to resolve the domain name return these IP addresses in round-robin order. Further information about DNS round-robin load sharing can be found in [Albitz & Liu, 2006].

Round-robin DNS has the advantage of being inexpensive and easy to set up. However, it does have some shortcomings. A DNS server performing iterative resolution may cache its results (see Section 59.8), with the result that future queries on the domain name return the same IP address, instead of the round-robin sequence generated by the authoritative DNS server. Also, round-robin DNS doesn't have any built-in mechanisms for ensuring good load balancing (different clients may place different loads on a server) or ensuring high availability (what if one of the servers dies or the server application that it is running crashes?). Another issue that we may need to consider—one that is faced by many designs that employ multiple server machines—is ensuring *server affinity*; that is, ensuring that a sequence of requests from the same client are all directed to the same server, so that any state information maintained by the server about the client remains accurate.

A more flexible, but also more complex, solution is *server load balancing*. In this scenario, a single load-balancing server routes incoming client requests to one of the members of the server farm. (To ensure high availability, there may be a backup server that takes over if the primary load-balancing server crashes.) This eliminates the problems associated with remote DNS caching, since the server farm presents a single IP address (that of the load-balancing server) to the outside world. The load-balancing server incorporates algorithms to measure or estimate server load (perhaps based on metrics supplied by the members of the server farm) and intelligently distribute the load across the members of the server farm. The load-balancing server also automatically detects failures in members of the server farm (and the addition of new servers, if demand requires it). Finally, a load-balancing server may also provide support for server affinity. Further information about server load balancing can be found in [Kopparapu, 2002].

60.5 The *inetd* (Internet Superserver) Daemon

If we look through the contents of `/etc/services`, we see literally hundreds of different services listed. This implies that a system could theoretically be running a large number of server processes. However, most of these servers would usually be doing nothing but waiting for infrequent connection requests or datagrams. All of these server processes would nevertheless occupy slots in the kernel process table, and consume some memory and swap space, thus placing a load on the system.

The *inetd* daemon is designed to eliminate the need to run large numbers of infrequently used servers. Using *inetd* provides two main benefits:

- Instead of running a separate daemon for each service, a single process—the *inetd* daemon—monitors a specified set of socket ports and starts other servers as required. Thus, the number of processes running on the system is reduced.
- The programming of the servers started by *inetd* is simplified, because *inetd* performs several of the steps that are commonly required by all network servers on startup.

Since it oversees a range of services, invoking other servers as required, *inetd* is sometimes known as the *Internet superserver*.

> An extended version of *inetd*, *xinetd*, is provided in some Linux distributions. Among other things, *xinetd* adds a number of security enhancements. Information about *xinetd* can be found at *http://www.xinetd.org/*.

Operation of the *inetd* daemon

The *inetd* daemon is normally started during system boot. After becoming a daemon process (Section 37.2), *inetd* performs the following steps:

1. For each of the services specified in its configuration file, `/etc/inetd.conf`, *inetd* creates a socket of the appropriate type (i.e., stream or datagram) and binds it to the specified port. Each TCP socket is additionally marked to permit incoming connections via a call to *listen()*.
2. Using the *select()* system call (Section 63.2.1), *inetd* monitors all of the sockets created in the preceding step for datagrams or incoming connection requests.
3. The *select()* call blocks until either a UDP socket has a datagram available to read or a connection request is received on a TCP socket. In the case of a TCP connection, *inetd* performs an *accept()* for the connection before proceeding to the next step.
4. To start the server specified for this socket, *inetd()* calls *fork()* to create a new process that then does an *exec()* to start the server program. Before performing the *exec()*, the child process performs the following steps:
   a) Close all of the file descriptors inherited from its parent, except the one for the socket on which the UDP datagram is available or the TCP connection has been accepted.
   b) Use the techniques described in Section 5.5 to duplicate the socket file descriptor on file descriptors 0, 1, and 2, and close the socket file descriptor itself (since it is no longer required). After this step, the execed server is able to communicate on the socket by using the three standard file descriptors.
   c) Optionally, set the user and group IDs for the execed server to values specified in `/etc/inetd.conf`.
5. If a connection was accepted on a TCP socket in step 3, *inetd* closes the connected socket (since it is needed only in the execed server).
6. The *inetd* server returns to step 2.

The /etc/inetd.conf file

The operation of the *inetd* daemon is controlled by a configuration file, normally /etc/inetd.conf. Each line in this file describes one of the services to be handled by *inetd*. Listing 60-5 shows some examples of entries in the /etc/inetd.conf file that comes with one Linux distribution.

Listing 60-5: Example lines from /etc/inetd.conf

```
# echo  stream  tcp  nowait  root    internal
# echo  dgram   udp  wait    root    internal
ftp     stream  tcp  nowait  root    /usr/sbin/tcpd   in.ftpd
telnet  stream  tcp  nowait  root    /usr/sbin/tcpd   in.telnetd
login   stream  tcp  nowait  root    /usr/sbin/tcpd   in.rlogind
```

The first two lines of Listing 60-5 are commented out by the initial # character; we show them now since we'll refer to the *echo* service shortly.

Each line of /etc/inetd.conf consists of the following fields, delimited by white space:

- *Service name*: This specifies the name of a service from the /etc/services file. In conjunction with the *protocol* field, this is used to look up /etc/services to determine which port number *inetd* should monitor for this service.
- *Socket type*: This specifies the type of socket used by this service—for example, stream or dgram.
- *Protocol*: This specifies the protocol to be used by this socket. This field can contain any of the Internet protocols listed in the file /etc/protocols (documented in the *protocols(5)* manual page), but almost every service specifies either tcp (for TCP) or udp (for UDP).
- *Flags*: This field contains either wait or nowait. This field specifies whether or not the server execed by *inetd* (temporarily) takes over management of the socket for this service. If the execed server manages the socket, then this field is specified as wait. This causes *inetd* to remove this socket from the file descriptor set that it monitors using *select()* until the execed server exits (*inetd* detects this via a handler for SIGCHLD). We say some more about this field below.
- *Login name*: This field consists of a username from /etc/passwd, optionally followed by a period (.) and a group name from /etc/group. These determine the user and group IDs under which the execed server is run. (Since *inetd* runs with an effective user ID of *root*, its children are also privileged and can thus use calls to *setuid()* and *setgid()* to change process credentials if desired.)
- *Server program*: This specifies the pathname of the server program to be execed.
- *Server program arguments*: This field specifies one or more arguments, separated by white space, to be used as the argument list when execing the server program. The first of these corresponds to *argv[0]* in the execed program and is thus usually the same as the basename part of the *server program* name. The next argument corresponds to *argv[1]*, and so on.

> In the example lines shown in Listing 60-5 for the *ftp*, *telnet*, and *login* services, we see the server program and arguments are set up differently than just described. All three of these services cause *inetd* to invoke the same program, *tcpd(8)* (the TCP daemon wrapper), which performs some logging and access-control checks before in turn execing the appropriate program, based on the value specified as the first server program argument (which is available to *tcpd* via *argv[0]*). Further information about *tcpd* can be found in the *tcpd(8)* manual page and in [Mann & Mitchell, 2003].

Stream socket (TCP) servers invoked by *inetd* are normally designed to handle just a single client connection and then terminate, leaving *inetd* with the job of listening for further connections. For such servers, *flags* should be specified as `nowait`. (If, instead, the execed server is to accept connections, then `wait` should be specified, in which case *inetd* does not accept the connection, but instead passes the file descriptor for the *listening* socket to the execed server as descriptor 0.)

For most UDP servers, the *flags* field should be specified as `wait`. A UDP server invoked by *inetd* is normally designed to read and process all outstanding datagrams on the socket and then terminate. (This usually requires some sort of timeout when reading the socket, so that the server terminates when no new datagrams arrive within a specified interval.) By specifying `wait`, we prevent the *inetd* daemon from simultaneously trying to *select()* on the socket, which would have the unintended consequence that *inetd* would race the UDP server to check for datagrams and, if it won the race, start another instance of the UDP server.

> Because the operation of *inetd* and the format of its configuration file are not specified by SUSv3, there are some (generally small) variations in the values that can be specified in the fields of `/etc/inetd.conf`. Most versions of *inetd* provide at least the syntax that we describe in the main text. For further details, see the *inetd.conf(8)* manual page.

As an efficiency measure, *inetd* implements a few simple services itself, instead of execing separate servers to perform the task. The UDP and TCP *echo* services are examples of services that *inetd* implements. For such services, the *server program* field of the corresponding `/etc/inetd.conf` record is specified as `internal`, and the *server program arguments* are omitted. (In the example lines in Listing 60-5, we saw that the *echo* service entries were commented out. To enable the *echo* service, we need to remove the # character at the start of these lines.)

Whenever we change the `/etc/inetd.conf` file, we need to send a `SIGHUP` signal to *inetd* to request it to reread the file:

```
# killall -HUP inetd
```

Example: invoking a TCP *echo* service via *inetd*

We noted earlier that *inetd* simplifies the programming of servers, especially concurrent (usually TCP) servers. It does this by carrying out the following steps on behalf of the servers it invokes:

1. Perform all socket-related initialization, calling *socket()*, *bind()*, and (for TCP servers) *listen()*.
2. For a TCP service, perform an *accept()* for the new connection.

3. Create a new process to handle the incoming UDP datagram or TCP connection. The process is automatically set up as a daemon. The *inetd* program handles all details of process creation via *fork()* and the reaping of dead children via a handler for SIGCHLD.
4. Duplicate the file descriptor of the UDP socket or the connected TCP socket on file descriptors 0, 1, and 2, and close all other file descriptors (since they are unused in the execed server).
5. Exec the server program.

(In the description of the above steps, we assume the usual cases that the *flags* field of the service entry in /etc/inetd.conf is specified as nowait for TCP services and wait for UDP services.)

As an example of how *inetd* simplifies the programming of a TCP service, in Listing 60-6, we show the *inetd*-invoked equivalent of the TCP *echo* server from Listing 60-4. Since *inetd* performs all of the above steps, all that remains of the server is the code executed by the child process to handle the client request, which can be read from file descriptor 0 (STDIN_FILENO).

If the server resides in the directory /bin (for example), then we would need to create the following entry in /etc/inetd.conf in order to have *inetd* invoke the server:

```
echo stream tcp nowait root /bin/is_echo_inetd_sv is_echo_inetd_sv
```

Listing 60-6: TCP *echo* server designed to be invoked via *inetd*

sockets/is_echo_inetd_sv.c

```
#include <syslog.h>
#include "tlpi_hdr.h"

#define BUF_SIZE 4096

int
main(int argc, char *argv[])
{
    char buf[BUF_SIZE];
    ssize_t numRead;

    while ((numRead = read(STDIN_FILENO, buf, BUF_SIZE)) > 0) {
        if (write(STDOUT_FILENO, buf, numRead) != numRead) {
            syslog(LOG_ERR, "write() failed: %s", strerror(errno));
            exit(EXIT_FAILURE);
        }
    }

    if (numRead == -1) {
        syslog(LOG_ERR, "Error from read(): %s", strerror(errno));
        exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
}
```

sockets/is_echo_inetd_sv.c

60.6 Summary

An iterative server handles one client at a time, processing that client's request(s) completely, before proceeding to the next client. A concurrent server handles multiple clients simultaneously. In high-load scenarios, a traditional concurrent server design that creates a new child process (or thread) for each client may not perform well enough, and we outlined a range of other approaches for concurrently handling large numbers of clients.

The Internet superserver daemon, *inetd*, monitors multiple sockets and starts the appropriate servers in response to incoming UDP datagrams or TCP connections. Using *inetd* allows us to decrease system load by minimizing the number of network server processes on the system, and also simplifies the programming of server processes, since it performs most of the initialization steps required by a server.

Further information

Refer to the sources of further information listed in Section 59.15.

60.7 Exercises

60-1. Add code to the program in Listing 60-4 (`is_echo_sv.c`) to place a limit on the number of simultaneously executing children.

60-2. Sometimes, it may be necessary to write a socket server so that it can be invoked either directly from the command line or indirectly via *inetd*. In this case, a command-line option is used to distinguish the two cases. Modify the program in Listing 60-4 so that, if it is given a *−i* command-line option, it assumes that it is being invoked by *inetd* and handles a single client on the connected socket, which *inetd* supplies via `STDIN_FILENO`. If the *−i* option is not supplied, then the program can assume it is being invoked from the command line, and operate in the usual fashion. (This change requires only the addition of a few lines of code.) Modify `/etc/inetd.conf` to invoke this program for the *echo* service.

61

SOCKETS: ADVANCED TOPICS

This chapter considers a range of more advanced topics relating to sockets programming, including the following:

- the circumstances in which partial reads and writes can occur on stream sockets;
- the use of *shutdown()* to close one half of the bidirectional channel between two connected sockets;
- the *recv()* and *send()* I/O system calls, which provide socket-specific functionality not available with *read()* and *write()*;
- the *sendfile()* system call, which is used in certain circumstances to efficiently output data on a socket;
- details of the operation of the TCP protocol, with the aim of eliminating some common misunderstandings that lead to mistakes when writing programs that use TCP sockets;
- the use of the *netstat* and *tcpdump* commands for monitoring and debugging applications that use sockets; and
- the use of the *getsockopt()* and *setsockopt()* system calls to retrieve and modify options affecting the operation of a socket.

We also consider a number of other more minor topics, and conclude the chapter with a summary of some advanced sockets features.

61.1 Partial Reads and Writes on Stream Sockets

When we first introduced the *read()* and *write()* system calls in Chapter 4, we noted that, in some circumstances, they may transfer fewer bytes than requested. Such partial transfers can occur when performing I/O on stream sockets. We now consider why they can occur and show a pair of functions that transparently handle partial transfers.

A partial read may occur if there are fewer bytes available in the socket than were requested in the *read()* call. In this case, *read()* simply returns the number of bytes available. (This is the same behavior that we saw with pipes and FIFOs in Section 44.10.)

A partial write may occur if there is insufficient buffer space to transfer all of the requested bytes and one of the following is true:

- A signal handler interrupted the *write()* call (Section 21.5) after it transferred some of the requested bytes.
- The socket was operating in nonblocking mode (`O_NONBLOCK`), and it was possible to transfer only some of the requested bytes.
- An asynchronous error occurred after only some of the requested bytes had been transferred. By an *asynchronous error*, we mean an error that occurs asynchronously with respect to the application's use of calls in the sockets API. An asynchronous error can arise, for example, because of a problem with a TCP connection, perhaps resulting from a crash by the peer application.

In all of the above cases, assuming that there was space to transfer at least 1 byte, the *write()* is successful, and returns the number of bytes that were transferred to the output buffer.

If a partial I/O occurs—for example, if a *read()* returns fewer bytes than requested or a blocked *write()* is interrupted by a signal handler after transferring only part of the requested data—then it is sometimes useful to restart the system call to complete the transfer. In Listing 61-1, we provide two functions that do this: *readn()* and *writen()*. (The ideas for these functions are drawn from functions of the same name presented in [Stevens et al., 2004].)

```
#include "rdwrn.h"

ssize_t readn(int fd, void *buffer, size_t count);
                                Returns number of bytes read, 0 on EOF, or –1 on error
ssize_t writen(int fd, void *buffer, size_t count);
                                Returns number of bytes written, or –1 on error
```

The *readn()* and *writen()* functions take the same arguments as *read()* and *write()*. However, they use a loop to restart these system calls, thus ensuring that the requested number of bytes is always transferred (unless an error occurs or end-of-file is detected on a *read()*).

Listing 61-1: Implementation of *readn()* and *writen()*

sockets/rdwrn.c

```
#include <unistd.h>
#include <errno.h>
#include "rdwrn.h"                      /* Declares readn() and writen() */

ssize_t
readn(int fd, void *buffer, size_t n)
{
    ssize_t numRead;                    /* # of bytes fetched by last read() */
    size_t totRead;                     /* Total # of bytes read so far */
    char *buf;

    buf = buffer;                       /* No pointer arithmetic on "void *" */
    for (totRead = 0; totRead < n; ) {
        numRead = read(fd, buf, n - totRead);

        if (numRead == 0)               /* EOF */
            return totRead;             /* May be 0 if this is first read() */
        if (numRead == -1) {
            if (errno == EINTR)
                continue;               /* Interrupted --> restart read() */
            else
                return -1;              /* Some other error */
        }
        totRead += numRead;
        buf += numRead;
    }
    return totRead;                     /* Must be 'n' bytes if we get here */
}

ssize_t
writen(int fd, const void *buffer, size_t n)
{
    ssize_t numWritten;                 /* # of bytes written by last write() */
    size_t totWritten;                  /* Total # of bytes written so far */
    const char *buf;

    buf = buffer;                       /* No pointer arithmetic on "void *" */
    for (totWritten = 0; totWritten < n; ) {
        numWritten = write(fd, buf, n - totWritten);

        if (numWritten <= 0) {
            if (numWritten == -1 && errno == EINTR)
                continue;               /* Interrupted --> restart write() */
            else
                return -1;              /* Some other error */
        }
        totWritten += numWritten;
        buf += numWritten;
    }
    return totWritten;                  /* Must be 'n' bytes if we get here */
}
```

sockets/rdwrn.c

61.2 The *shutdown()* System Call

Calling *close()* on a socket closes both halves of the bidirectional communication channel. Sometimes, it is useful to close one half of the connection, so that data can be transmitted in just one direction through the socket. The *shutdown()* system call provides this functionality.

```
#include <sys/socket.h>

int shutdown(int sockfd, int how);
```

Returns 0 on success, or −1 on error

The *shutdown()* system call closes one or both channels of the socket *sockfd*, depending on the value of *how*, which is specified as one of the following:

SHUT_RD
: Close the reading half of the connection. Subsequent reads will return end-of-file (0). Data can still be written to the socket. After a SHUT_RD on a UNIX domain stream socket, the peer application receives a SIGPIPE signal and the EPIPE error if it makes further attempts to write to the peer socket. As discussed in Section 61.6.6, SHUT_RD can't be used meaningfully for TCP sockets.

SHUT_WR
: Close the writing half of the connection. Once the peer application has read all outstanding data, it will see end-of-file. Subsequent writes to the local socket yield the SIGPIPE signal and an EPIPE error. Data written by the peer can still be read from the socket. In other words, this operation allows us to signal end-of-file to the peer while still being able to read data that the peer sends back to us. The SHUT_WR operation is employed by programs such as *ssh* and *rsh* (refer to Section 18.5 of [Stevens, 1994]). The SHUT_WR operation is the most common use of *shutdown()*, and is sometimes referred to as a *socket half-close*.

SHUT_RDWR
: Close both the read and the write halves of the connection. This is the same as performing a SHUT_RD followed by a SHUT_WR.

Aside from the semantics of the *how* argument, *shutdown()* differs from *close()* in another important respect: it closes the socket channel(s) regardless of whether there are other file descriptors referring to the socket. (In other words, *shutdown()* is performing an operation on the open file description, rather than the file descriptor. See Figure 5-1, on page 91.) Suppose, for example, that *sockfd* refers to a connected stream socket. If we make the following calls, then the connection remains open, and we can still perform I/O on the connection via the file descriptor *fd2*:

```
fd2 = dup(sockfd);
close(sockfd);
```

However, if we make the following sequence of calls, then both channels of the connection are closed, and I/O can no longer be performed via *fd2*:

```
fd2 = dup(sockfd);
shutdown(sockfd, SHUT_RDWR);
```

A similar scenario holds if a file descriptor for a socket is duplicated during a *fork()*. If, after the *fork()*, one process does a SHUT_RDWR on its copy of the descriptor, then the other process also can no longer perform I/O on its descriptor.

Note that *shutdown()* doesn't close the file descriptor, even if *how* is specified as SHUT_RDWR. To close the file descriptor, we must additionally call *close()*.

Example program

Listing 61-2 demonstrates the use of the *shutdown()* SHUT_WR operation. This program is a TCP client for the *echo* service. (We presented a TCP server for the *echo* service in Section 60.3.) To shorten the implementation, we make use of functions in the Internet domain sockets library shown in Section 59.12.

> In some Linux distributions, the *echo* service is not enabled by default, and therefore we must enable it before running the program in Listing 61-2. Typically, this service is implemented internally by the *inetd(8)* daemon (Section 60.5), and, to enable the *echo* service, we must edit the file /etc/inetd.conf to uncomment the two lines corresponding to the UDP and TCP *echo* services (see Listing 60-5, on page 1249), and then send a SIGHUP signal to the *inetd* daemon.
>
> Many distributions supply the more modern *xinetd(8)* instead of *inetd(8)*. Consult the *xinetd* documentation for information about how to make the equivalent changes under *xinetd*.

As its single command-line argument, the program takes the name of the host on which the *echo* server is running. The client performs a *fork()*, yielding parent and child processes.

The client parent writes the contents of standard input to the socket, so that it can be read by the *echo* server. When the parent detects end-of-file on standard input, it uses *shutdown()* to close the writing half of its socket. This causes the *echo* server to see end-of-file, at which point it closes its socket (which causes the client child in turn to see end-of-file). The parent then terminates.

The client child reads the *echo* server's response from the socket and echoes the response on standard output. The child terminates when it sees end-of-file on the socket.

The following shows an example of what we see when running this program:

```
$ cat > tell-tale-heart.txt                          Create a file for testing
It is impossible to say how the idea entered my brain;
but once conceived, it haunted me day and night.
Type Control-D
$ ./is_echo_cl tekapo < tell-tale-heart.txt
It is impossible to say how the idea entered my brain;
but once conceived, it haunted me day and night.
```

Listing 61-2: A client for the *echo* service

sockets/is_echo_cl.c

```
#include "inet_sockets.h"
#include "tlpi_hdr.h"

#define BUF_SIZE 100

int
main(int argc, char *argv[])
{
    int sfd;
    ssize_t numRead;
    char buf[BUF_SIZE];

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s host\n", argv[0]);

    sfd = inetConnect(argv[1], "echo", SOCK_STREAM);
    if (sfd == -1)
        errExit("inetConnect");

    switch (fork()) {
    case -1:
        errExit("fork");

    case 0:             /* Child: read server's response, echo on stdout */
        for (;;) {
            numRead = read(sfd, buf, BUF_SIZE);
            if (numRead <= 0)           /* Exit on EOF or error */
                break;
            printf("%.*s", (int) numRead, buf);
        }
        exit(EXIT_SUCCESS);

    default:            /* Parent: write contents of stdin to socket */
        for (;;) {
            numRead = read(STDIN_FILENO, buf, BUF_SIZE);
            if (numRead <= 0)           /* Exit loop on EOF or error */
                break;
            if (write(sfd, buf, numRead) != numRead)
                fatal("write() failed");
        }

        /* Close writing channel, so server sees EOF */

        if (shutdown(sfd, SHUT_WR) == -1)
            errExit("shutdown");
        exit(EXIT_SUCCESS);
    }
}
```

sockets/is_echo_cl.c

61.3 Socket-Specific I/O System Calls: *recv()* and *send()*

The *recv()* and *send()* system calls perform I/O on connected sockets. They provide socket-specific functionality that is not available with the traditional *read()* and *write()* system calls.

```
#include <sys/socket.h>

ssize_t recv(int sockfd, void *buffer, size_t length, int flags);
                                    Returns number of bytes received, 0 on EOF, or –1 on error
ssize_t send(int sockfd, const void *buffer, size_t length, int flags);
                                    Returns number of bytes sent, or –1 on error
```

The return value and the first three arguments to *recv()* and *send()* are the same as for *read()* and *write()*. The last argument, *flags*, is a bit mask that modifies the behavior of the I/O operation. For *recv()*, the bits that may be ORed in *flags* include the following:

`MSG_DONTWAIT`
: Perform a nonblocking *recv()*. If no data is available, then instead of blocking, return immediately with the error `EAGAIN`. We can obtain the same behavior by using *fcntl()* to set nonblocking mode (`O_NONBLOCK`) on the socket, with the difference that `MSG_DONTWAIT` allows us to control nonblocking behavior on a per-call basis.

`MSG_OOB`
: Receive out-of-band data on the socket. We briefly describe this feature in Section 61.13.1.

`MSG_PEEK`
: Retrieve a copy of the requested bytes from the socket buffer, but don't actually remove them from the buffer. The data can later be reread by another *recv()* or *read()* call.

`MSG_WAITALL`
: Normally, a *recv()* call returns the lesser of the number of bytes requested (*length*) and the number of bytes actually available in the socket. Specifying the `MSG_WAITALL` flag causes the system call to block until *length* bytes have been received. However, even when this flag is specified, the call may return fewer bytes than requested if: (a) a signal is caught; (b) the peer on a stream socket terminated the connection; (c) an out-of-band data byte (Section 61.13.1) was encountered; (d) the received message from a datagram socket is less than *length* bytes; or (e) an error occurs on the socket. (The `MSG_WAITALL` flag can replace the *readn()* function that we show in Listing 61-1, with the difference that our *readn()* function does restart itself if interrupted by a signal handler.)

All of the above flags are specified in SUSv3, except for MSG_DONTWAIT, which is nevertheless available on some other UNIX implementations. The MSG_WAITALL flag was a later addition to the sockets API, and is not present in some older implementations.

For *send()*, the bits that may be ORed in *flags* include the following:

MSG_DONTWAIT
: Perform a nonblocking *send()*. If the data can't be immediately transferred (because the socket send buffer is full), then, instead of blocking, fail with the error EAGAIN. As with *recv()*, the same effect can be achieved by setting the O_NONBLOCK flag for the socket.

MSG_MORE (since Linux 2.4.4)
: This flag is used with TCP sockets to achieve the same effect as the TCP_CORK socket option (Section 61.4), with the difference that it provides corking of data on a per-call basis. Since Linux 2.6, this flag can also be used with datagram sockets, where it has a different meaning. Data transmitted in successive *send()* or *sendto()* calls specifying MSG_MORE is packaged into a single datagram that is transmitted only when a further call is made that does not specify this flag. (Linux also provides an analogous UDP_CORK socket option that causes data from successive *send()* or *sendto()* calls to be accumulated into a single datagram that is transmitted when UDP_CORK is disabled.) The MSG_MORE flag has no effect for UNIX domain sockets.

MSG_NOSIGNAL
: When sending data on a connected stream socket, don't generate a SIGPIPE signal if the other end of the connection has been closed. Instead, the *send()* call fails with the error EPIPE. This is the same behavior as can be obtained by ignoring the SIGPIPE signal, with the difference that the MSG_NOSIGNAL flag controls the behavior on a per-call basis.

MSG_OOB
: Send out-of-band data on a stream socket. Refer to Section 61.13.1.

Of the above flags, only MSG_OOB is specified by SUSv3. SUSv4 adds a specification for MSG_NOSIGNAL. MSG_DONTWAIT is not standardized, but appears on a few other UNIX implementations. MSG_MORE is Linux-specific. The *send(2)* and *recv(2)* manual pages describe further flags that we don't cover here.

61.4 The *sendfile()* System Call

Applications such as web servers and file servers frequently need to transfer the unaltered contents of a disk file through a (connected) socket. One way to do this would be a loop of the following form:

```
while ((n = read(diskfilefd, buf, BUZ_SIZE)) > 0)
    write(sockfd, buf, n);
```

For many applications, such a loop is perfectly acceptable. However, if we frequently transfer large files via a socket, this technique is inefficient. In order to transmit the file, we must use two system calls (possibly multiple times within a loop): one to copy the file contents from the kernel buffer cache into user space,

and the other to copy the user-space buffer back to kernel space in order to be transmitted via the socket. This scenario is shown on the left side of Figure 61-1. Such a two-step process is wasteful if the application doesn't perform any processing of the file contents before transmitting them. The *sendfile()* system call is designed to eliminate this inefficiency. When an application calls *sendfile()*, the file contents are transferred directly to the socket, without passing through user space, as shown on the right side of Figure 61-1. This is referred to as a *zero-copy transfer*.

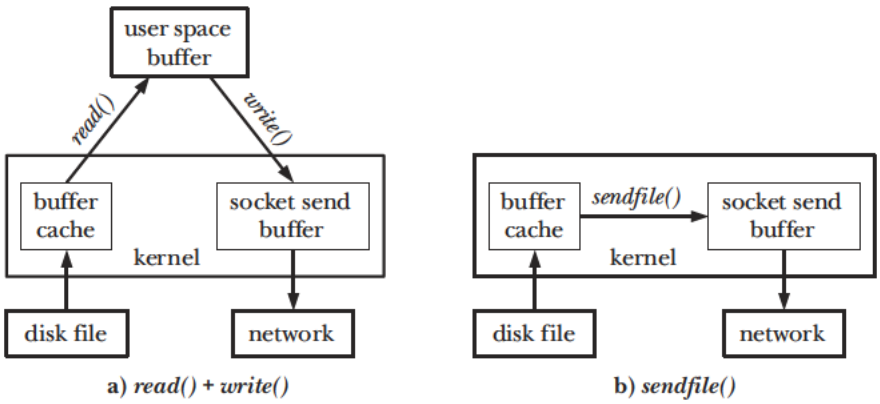

Figure 61-1: Transferring the contents of a file to a socket

```
#include <sys/sendfile.h>

ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);
                                    Returns number of bytes transferred, or –1 on error
```

The *sendfile()* system call transfers bytes from the file referred to by the descriptor *in_fd* to the file referred to by the descriptor *out_fd*. The *out_fd* descriptor must refer to a socket. The *in_fd* argument must refer to a file to which *mmap()* can be applied; in practice, this usually means a regular file. This somewhat restricts the use of *sendfile()*. We can use it to pass data from a file to a socket, but not vice versa. And we can't use *sendfile()* to pass data directly from one socket to another.

> Performance benefits could also be obtained if *sendfile()* could be used to transfer bytes between two regular files. On Linux 2.4 and earlier, *out_fd* could refer to a regular file. Some reworking of the underlying implementation meant that this possibility disappeared in the 2.6 kernel. However, this feature may be reinstated in a future kernel version.

If *offset* is not `NULL`, then it should point to an *off_t* value that specifies the starting file offset from which bytes should be transferred from *in_fd*. This is a value-result argument. On return, it contains the offset of the next byte following the last byte that was transferred from *in_fd*. In this case, *sendfile()* doesn't change the file offset for *in_fd*.

If *offset* is `NULL`, then bytes are transferred from *in_fd* starting at the current file offset, and the file offset is updated to reflect the number of bytes transferred.

The *count* argument specifies the number of bytes to be transferred. If end-of-file is encountered before *count* bytes are transferred, only the available bytes are transferred. On success, *sendfile()* returns the number of bytes actually transferred.

SUSv3 doesn't specify *sendfile()*. Versions of *sendfile()* are available on some other UNIX implementations, but the argument list is typically different from the version on Linux.

> Starting with kernel 2.6.17, Linux provides three new (nonstandard) system calls—*splice()*, *vmsplice()*, and *tee()*—that provide a superset of the functionality of *sendfile()*. See the manual pages for details.

The TCP_CORK socket option

To further improve the efficiency of TCP applications using *sendfile()*, it is sometimes useful to employ the Linux-specific TCP_CORK socket option. As an example, consider a web server delivering a page in response to a request by a web browser. The web server's response consists of two parts: HTTP headers, perhaps output using *write()*, followed by the page data, perhaps output using *sendfile()*. In this scenario, normally *two* TCP segments are transmitted: the headers are sent in the first (rather small) segment, and then the page data is sent in a second segment. This is an inefficient use of network bandwidth. It probably also creates unnecessary work for both the sending and the receiving TCP, since in many cases the HTTP headers and the page data would be small enough to fit inside a single TCP segment. The TCP_CORK option is designed to address this inefficiency.

When the TCP_CORK option is enabled on a TCP socket, all subsequent output is buffered into a single TCP segment until either the upper limit on the size of a segment is reached, the TCP_CORK option is disabled, the socket is closed, or a maximum of 200 milliseconds passes from the time that the first corked byte is written. (The timeout ensures that the corked data is transmitted if the application forgets to disable the TCP_CORK option.)

We enable and disable the TCP_CORK option using the *setsockopt()* system call (Section 61.9). The following code (which omits error checking) demonstrates the use of TCP_CORK for our hypothetical HTTP server example:

```
int optval;

/* Enable TCP_CORK option on 'sockfd' - subsequent TCP output is corked
   until this option is disabled. */

optval = 1;
setsockopt(sockfd, IPPROTO_TCP, TCP_CORK, &optval, sizeof(optval));

write(sockfd, ...);                     /* Write HTTP headers */
sendfile(sockfd, ...);                  /* Send page data */

/* Disable TCP_CORK option on 'sockfd' - corked output is now transmitted
   in a single TCP segment. */

optval = 0
setsockopt(sockfd, IPPROTO_TCP, TCP_CORK, &optval, sizeof(optval));
```

We could avoid the possibility of two segments being transmitted by building a single data buffer within our application, and then transmitting that buffer with a single *write()*. (Alternatively, we could use *writev()* to combine two distinct buffers in a single output operation.) However, if we want to combine the zero-copy efficiency of *sendfile()* with the ability to include a header as part of the first segment of transmitted file data, then we need to use TCP_CORK.

> In Section 61.3, we noted that the MSG_MORE flag provides similar functionality to TCP_CORK, but on a per-system-call basis. This is not necessarily an advantage. It is possible to set the TCP_CORK option on the socket, and then exec a program that performs output on the inherited file descriptor without being aware of the TCP_CORK option. By contrast, the use of MSG_MORE requires explicit changes to the source code of a program.
>
> FreeBSD provides an option similar to TCP_CORK in the form of TCP_NOPUSH.

61.5 Retrieving Socket Addresses

The *getsockname()* and *getpeername()* system calls return, respectively, the local address to which a socket is bound and the address of the peer socket to which the local socket is connected.

```
#include <sys/socket.h>

int getsockname(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
int getpeername(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Both return 0 on success, or −1 on error

For both calls, *sockfd* is a file descriptor referring to a socket, and *addr* is a pointer to a suitably sized buffer that is used to return a structure containing the socket address. The size and type of this structure depend on the socket domain. The *addrlen* argument is a value-result argument. Before the call, it should be initialized to the length of the buffer pointed to by *addr*; on return, it contains the number of bytes actually written to this buffer.

The *getsockname()* function returns a socket's address family and the address to which a socket is bound. This is useful if the socket was bound by another program (e.g., *inetd(8)*) and the socket file descriptor was then preserved across an *exec()*.

Calling *getsockname()* is also useful if we want to determine the ephemeral port number that the kernel assigned to a socket when performing an implicit bind of an Internet domain socket. The kernel performs an implicit bind in the following circumstances:

- after a *connect()* or a *listen()* call on a TCP socket that has not previously been bound to an address by *bind()*;
- on the first *sendto()* on a UDP socket that had not previously been bound to an address; or
- after a *bind()* call where the port number (*sin_port*) was specified as 0. In this case, the *bind()* specifies the IP address for the socket, but the kernel selects an ephemeral port number.

The *getpeername()* system call returns the address of the peer socket on a stream socket connection. This is useful primarily with TCP sockets, if the server wants to find out the address of the client that has made a connection. This information could also be obtained when the *accept()* call is performed; however, if the server was execed by the program that did the *accept()* (e.g., *inetd*), then it inherits the socket file descriptor, but the address information returned by *accept()* is no longer available.

Listing 61-3 demonstrates the use of *getsockname()* and *getpeername()*. This program employs the functions that we defined in Listing 59-9 (on page 1228), and performs the following steps:

1. Use our *inetListen()* function to create a listening socket, *listenFd*, bound to the wildcard IP address and the port specified in the program's sole command-line argument. (The port can be specified numerically or as a service name.) The *len* argument returns the length of the address structure for this socket's domain. This value is passed in a later call to *malloc()* to allocate a buffer that is used to return a socket address from calls to *getsockname()* and *getpeername()*.
2. Use our *inetConnect()* function to create a second socket, *connFd*, which is used to send a connection request to the socket created in step 1.
3. Call *accept()* on the listening socket in order to create a third socket, *acceptFd*, that is connected to the socket created in the previous step.
4. Use calls to *getsockname()* and *getpeername()* to obtain the local and peer addresses for the two connected sockets, *connFd* and *acceptFd*. After each of these calls, the program uses our *inetAddressStr()* function to convert the socket address into printable form.
5. Sleep for a few seconds so that we can run *netstat* in order to confirm the socket address information. (We describe *netstat* in Section 61.7.)

The following shell session log shows an example run of this program:

```
$ ./socknames 55555 &
getsockname(connFd):   (localhost, 32835)
getsockname(acceptFd): (localhost, 55555)
getpeername(connFd):   (localhost, 55555)
getpeername(acceptFd): (localhost, 32835)
[1] 8171
$ netstat -a | egrep '(Address|55555)'
Proto Recv-Q Send-Q Local Address    Foreign Address  State
tcp        0      0 *:55555          *:*              LISTEN
tcp        0      0 localhost:32835  localhost:55555  ESTABLISHED
tcp        0      0 localhost:55555  localhost:32835  ESTABLISHED
```

From the above output, we can see that the connected socket (*connFd*) was bound to the ephemeral port 32835. The *netstat* command shows us information about all three sockets created by the program, and allows us to confirm the port information for the two connected sockets, which are in the ESTABLISHED state (described in Section 61.6.3).

Listing 61-3: Using *getsockname()* and *getpeername()*

sockets/socknames.c

```
#include "inet_sockets.h"              /* Declares our socket functions */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int listenFd, acceptFd, connFd;
    socklen_t len;                     /* Size of socket address buffer */
    void *addr;                        /* Buffer for socket address */
    char addrStr[IS_ADDR_STR_LEN];

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s service\n", argv[0]);

    listenFd = inetListen(argv[1], 5, &len);
    if (listenFd == -1)
        errExit("inetListen");

    connFd = inetConnect(NULL, argv[1], SOCK_STREAM);
    if (connFd == -1)
        errExit("inetConnect");

    acceptFd = accept(listenFd, NULL, NULL);
    if (acceptFd == -1)
        errExit("accept");

    addr = malloc(len);
    if (addr == NULL)
        errExit("malloc");

    if (getsockname(connFd, addr, &len) == -1)
        errExit("getsockname");
    printf("getsockname(connFd):   %s\n",
            inetAddressStr(addr, len, addrStr, IS_ADDR_STR_LEN));
    if (getsockname(acceptFd, addr, &len) == -1)
        errExit("getsockname");
    printf("getsockname(acceptFd): %s\n",
            inetAddressStr(addr, len, addrStr, IS_ADDR_STR_LEN));

    if (getpeername(connFd, addr, &len) == -1)
        errExit("getpeername");
    printf("getpeername(connFd):   %s\n",
            inetAddressStr(addr, len, addrStr, IS_ADDR_STR_LEN));
    if (getpeername(acceptFd, addr, &len) == -1)
        errExit("getpeername");
    printf("getpeername(acceptFd): %s\n",
            inetAddressStr(addr, len, addrStr, IS_ADDR_STR_LEN));

    sleep(30);                         /* Give us time to run netstat(8) */
    exit(EXIT_SUCCESS);
}
```

sockets/socknames.c

61.6 A Closer Look at TCP

Knowing some of the details of the operation of TCP helps us to debug applications that use TCP sockets, and, in some cases, to make such applications more efficient. In the following sections, we look at:

- the format of TCP segments;
- the TCP acknowledgement scheme;
- the TCP state machine;
- TCP connection establishment and termination; and
- the TCP TIME_WAIT state.

61.6.1 Format of a TCP Segment

Figure 61-2 shows the format of the TCP segments that are exchanged between the endpoints of a TCP connection. The meanings of these fields are as follows:

- *Source port number*: This is the port number of the sending TCP.
- *Destination port number*: This is the port number of the destination TCP.
- *Sequence number*: This is the sequence number for this segment. This is the offset of the first byte of data in this segment within the stream of data being transmitted in this direction over the connection, as described in Section 58.6.3.

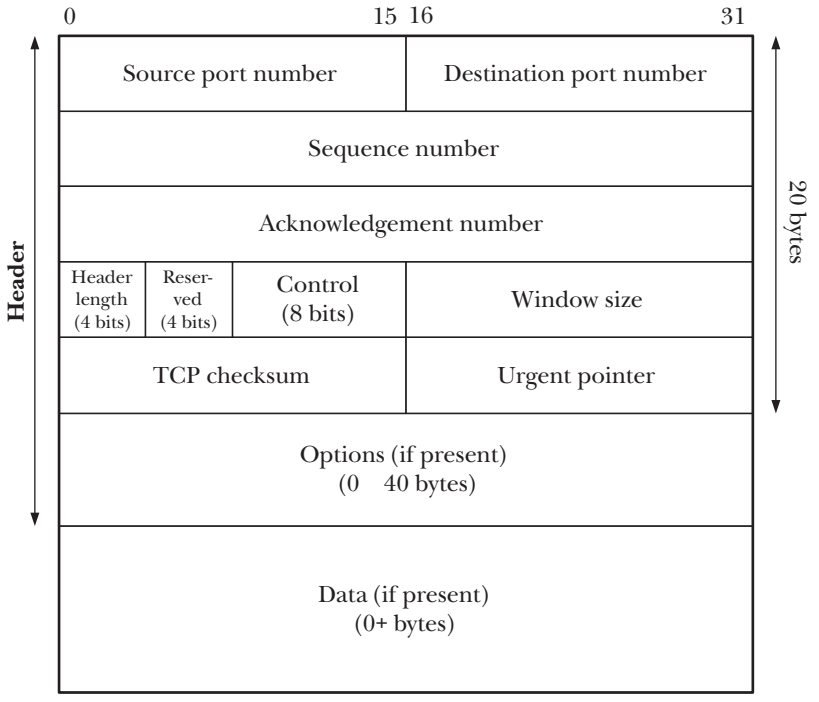

Figure 61-2: Format of a TCP segment

- *Acknowledgement number*: If the ACK bit (see below) is set, then this field contains the sequence number of the next byte of data that the receiver expects to receive from the sender.
- *Header length*: This is the length of the header, in units of 32-bit words. Since this is a 4-bit field, the total header length can be up to 60 bytes (15 words). This field enables the receiving TCP to determine the length of the variable-length *options* field and the starting point of the *data*.
- *Reserved*: This consists of 4 unused bits (must be set to 0).
- *Control bits*: This field consists of 8 bits that further specify the meaning of the segment:
  - *CWR*: the *congestion window reduced* flag.
  - *ECE*: the *explicit congestion notification echo* flag. The CWR and ECE flags are used as part of TCP/IP's Explicit Congestion Notification (ECN) algorithm. ECN is a relatively recent addition to TCP/IP and is described in RFC 3168 and in [Floyd, 1994]. ECN is implemented in Linux from kernel 2.4 onward, and enabled by placing a nonzero value in the Linux-specific `/proc/sys/net/ipv4/tcp_ecn` file.
  - *URG*: if set, then the *urgent pointer* field contains valid information.
  - *ACK*: if set, then the *acknowledgement number* field contains valid information (i.e., this segment acknowledges data previously sent by the peer).
  - *PSH*: push all received data to the receiving process. This flag is described in RFC 993 and in [Stevens, 1994].
  - *RST*: reset the connection. This is used to handle various error situations.
  - *SYN*: synchronize sequence numbers. Segments with this flag set are exchanged during connection establishment to allow the two TCPs to specify the initial sequence numbers to be used for transferring data in each direction.
  - *FIN*: used by a sender to indicate that it has finished sending data.

  Multiple control bits (or none at all) may be set in a segment, which allows a single segment to serve multiple purposes. For example, we'll see later that a segment with both the SYN and the ACK bits set is exchanged during TCP connection establishment.
- *Window size*: This field is used when a receiver sends an ACK to indicate the number of bytes of data that the receiver has space to accept. (This relates to the sliding window scheme briefly described in Section 58.6.3.)
- *Checksum*: This is a 16-bit checksum covering both the TCP header and the TCP data.

  > The TCP checksum covers not just the TCP header and data, but also 12 bytes usually referred to as the TCP *pseudoheader*. The pseudoheader consists of the following: the source and destination IP address (4 bytes each); 2 bytes specifying the size of the TCP segment (this value is computed, but doesn't form part of either the IP or the TCP header); 1 byte containing the value 6, which is TCP's unique protocol number within the TCP/IP suite of protocols; and 1 padding byte containing 0 (so that the length of the pseudoheader is a multiple of 16 bits). The purpose of including the pseudoheader in the checksum calculation is to

allow the receiving TCP to double-check that an incoming segment has arrived at the correct destination (i.e., that IP has not wrongly accepted a datagram that was addressed to another host or passed TCP a packet that should have gone to another upper layer). UDP calculates the checksum in its packet headers in a similar manner and for similar reasons. See [Stevens, 1994] for further details on the pseudoheader.

- *Urgent pointer*: If the URG control bit is set, then this field indicates the location of so-called urgent data within the stream of data being transmitted from the sender to the receiver. We briefly discuss urgent data in Section 61.13.1.
- *Options*: This is a variable-length field containing options controlling the operation of the TCP connection.
- *Data*: This field contains the user data transmitted in this segment. This field may be of length 0 if this segment doesn't contain any data (e.g., if it is simply an ACK segment).

61.6.2 TCP Sequence Numbers and Acknowledgements

Each byte that is transmitted over a TCP connection is assigned a logical sequence number by TCP. (Each of the two streams in a connection has its own sequence numbering.) When a segment is transmitted, its *sequence number* field is set to the logical offset of the first byte of data in the segment within the stream of data being transmitted in this direction over the connection. This allows the receiving TCP to assemble the received segments in the correct order, and to indicate which data was received when sending an acknowledgement to the sender.

To implement reliable communication, TCP uses positive acknowledgements; that is, when a segment is successfully received, an acknowledgement message (i.e., a segment with the ACK bit set) is sent from the receiving TCP to the sending TCP, as shown in Figure 61-3. The *acknowledgement number* field of this message is set to indicate the logical sequence number of the next byte of data that the receiver expects to receive. (In other words, the value in the acknowledgement number field is the sequence number of the last byte in the segment that it acknowledges, plus 1.)

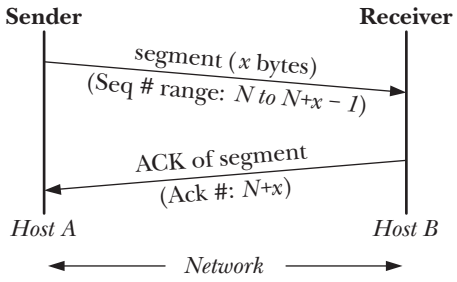

Figure 61-3: Acknowledgements in TCP

When the sending TCP transmits a segment, it sets a timer. If an acknowledgement is not received before the timer expires, the segment is retransmitted.

Figure 61-3 and later similar diagrams are intended to illustrate the exchange of TCP segments between two endpoints. An implicit time dimension is assumed when reading these diagrams from top to bottom.

61.6.3 TCP State Machine and State Transition Diagram

Maintaining a TCP connection requires the coordination of the TCPs at both ends of the connection. To reduce the complexity of this task, a TCP endpoint is modeled as a *state machine*. This means that the TCP can be in one of a fixed set of *states*, and it moves from one state to another in response to *events*, such as system calls by the application above the TCP or the arrival of TCP segments from the peer TCP. The TCP states are the following:

- LISTEN: The TCP is waiting for a connection request from a peer TCP.
- SYN_SENT: The TCP has sent a SYN on behalf of an application performing an active open and is waiting for a reply from the peer in order to complete the connection.
- SYN_RECV: The TCP, formerly in the LISTEN state, has received a SYN and has responded with a SYN/ACK (i.e., a TCP segment with both the SYN and ACK bits set), and is now waiting for an ACK from the peer TCP in order to complete the connection.
- ESTABLISHED: Establishment of the connection to the peer TCP has been completed. Data segments can now be exchanged in either direction between the two TCPs.
- FIN_WAIT1: The application has closed the connection. The TCP has sent a FIN to the peer TCP in order to terminate its side of the connection and is waiting for an ACK from the peer. This and the next three states are associated with an application performing an active close—that is, the first application to close its side of the connection.
- FIN_WAIT2: The TCP, formerly in the FIN_WAIT1 state, has now received an ACK from the peer TCP.
- CLOSING: The TCP, formerly awaiting an ACK in the FIN_WAIT1 state, instead received a FIN from its peer indicating that the peer simultaneously tried to perform an active close. (In other words, the two TCPs sent FIN segments at almost the same time. This is a rare scenario.)
- TIME_WAIT: Having done an active close, the TCP has received a FIN, indicating that the peer TCP has performed a passive close. This TCP now spends a fixed period of time in the TIME_WAIT state, in order to ensure reliable termination of the TCP connection and to ensure that any old duplicate segments expire in the network before a new incarnation of the same connection is created. (We explain the TIME_WAIT state in more detail in Section 61.6.7.) When this fixed time period expires, the connection is closed, and the associated kernel resources are freed.
- CLOSE_WAIT: The TCP has received a FIN from the peer TCP. This and the following state are associated with an application performing a passive close—that is, the second application to close the connection.

- LAST_ACK: The application performed a passive close, and the TCP, formerly in the CLOSE_WAIT state, sent a FIN to the peer TCP and is waiting for it to be acknowledged. When this ACK is received, the connection is closed, and the associated kernel resources are freed.

To the above states, RFC 793 adds one further, fictional state, CLOSED, representing the state when there is no connection (i.e., no kernel resources are allocated to describe a TCP connection).

> In the above list we use the spellings for the TCP states as defined in the Linux source code. These differ slightly from the spellings in RFC 793.

Figure 61-4 shows the *state transition diagram* for TCP. (This figure is based on diagrams in RFC 793 and [Stevens et al., 2004].) This diagram shows how a TCP endpoint moves from one state to another in response to various events. Each arrow indicates a possible transition and is labeled with the event that triggers the transition. This label is either an action by the application (in boldface) or the string *recv*, indicating the receipt of a segment from the peer TCP. As a TCP moves from one state to another, it may transmit a segment to the peer, and this is indicated by the *send* label on the transition. For example, the arrow for the transition from the ESTABLISHED to the FIN_WAIT1 state shows that the triggering event is a *close()* by the local application, and that, during the transition, the TCP sends a FIN segment to its peer.

In Figure 61-4, the usual transition path for a client TCP is shown with heavy solid arrows, and the usual transition path for a server TCP is shown with heavy dashed arrows. (Other arrows indicate paths less traveled.) Looking at the parenthetical numbering on the arrows in these paths, we can see that the segments sent and received by the two TCPs are mirror images of one another. (After the ESTABLISHED state, the paths traveled by the server TCP and the client TCP may be the opposite of those indicated, if it is the server that performs the active close.)

> Figure 61-4 doesn't show all possible transitions for the TCP state machine; it illustrates just those of principal interest. A more detailed TCP state transition diagram can be found at *http://www.cl.cam.ac.uk/~pes20/Netsem/poster.pdf*.

61.6.4 TCP Connection Establishment

At the sockets API level, two stream sockets are connected via the following steps (see Figure 56-1, on page 1156):

1. The server calls *listen()* to perform a passive open of a socket, and then calls *accept()*, which blocks until a connection is established.
2. The client calls *connect()* to perform an active open of a socket in order to establish a connection to the server's passive socket.

The steps performed by TCP to establish a connection are shown in Figure 61-5. These steps are often referred to as the *three-way handshake*, since three segments pass between the two TCPs. The steps are as follows:

1. The *connect()* causes the client TCP to send a SYN segment to the server TCP. This segment informs the server TCP of the client TCP's initial sequence number

(labeled *M* in the diagram). This information is necessary because sequence numbers don't begin at 0, as noted in Section 58.6.3.

2. The server TCP must both acknowledge the client TCP's SYN segment and inform the client TCP of its own initial sequence number (labeled *N* in the diagram). (Two sequence numbers are required because a stream socket is bidirectional.) The server TCP can perform both operations by returning a single segment with both the SYN and the ACK control bits set. (We say that the ACK is *piggybacked* on the SYN.)
3. The client TCP sends an ACK segment to acknowledge the server TCP's SYN segment.

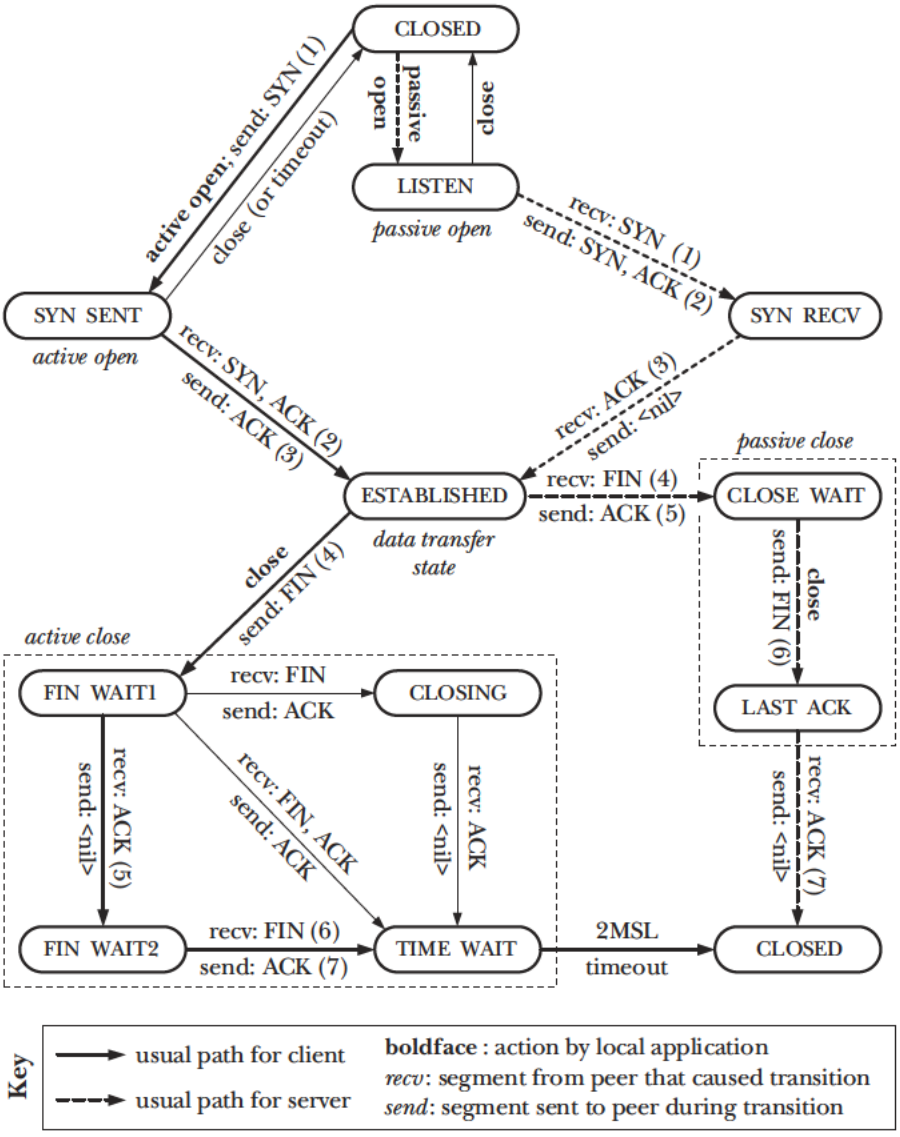

Figure 61-4: TCP state transition diagram

The SYN segments exchanged in the first two steps of the three-way handshake may contain information in the *options* field of the TCP header that is used to determine various parameters for the connection. See [Stevens et al., 2004], [Stevens, 1994], and [Wright & Stevens, 1995] for details.

The labels inside angle brackets (e.g., <LISTEN>) in Figure 61-5 indicate the states of the TCPs on either side of the connection.

The SYN flag consumes a byte of the sequence-number space for the connection. This is necessary so that this flag can be acknowledged unambiguously, since segments with this flag set may also contain data bytes. This is why we show the acknowledgement of the *SYN M* segment as *ACK M+1* in Figure 61-5.

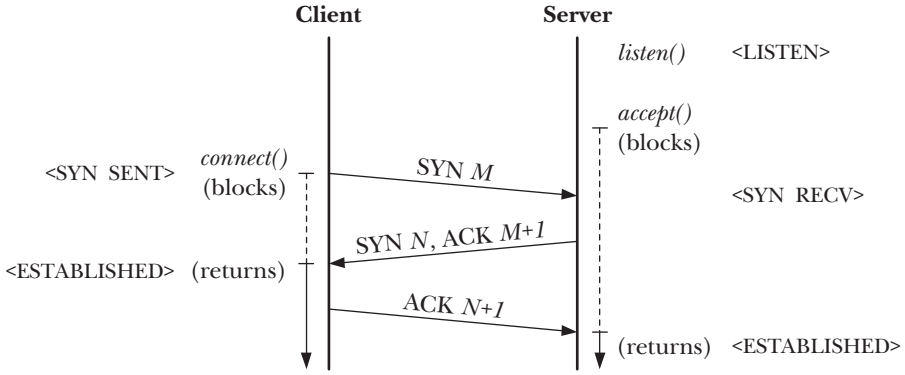

Figure 61-5: Three-way handshake for TCP connection establishment

61.6.5 TCP Connection Termination

Closing a TCP connection normally occurs in the following manner:

1. An application on one end of the connection performs a *close()*. (This is often, but not necessarily, the client.) We say that this application is performing an *active close*.
2. Later, the application on the other end of the connection (the server) also performs a *close()*. This is termed a *passive close*.

Figure 61-6 shows the corresponding steps performed by the underlying TCPs (here, we assume that it is the client that does the active close). These steps are as follows:

1. The client performs an active close, which causes the client TCP to send a FIN to the server TCP.
2. After receipt of the FIN, the server TCP responds with an ACK. Any subsequent attempt by the server to *read()* from the socket yields end-of-file (i.e., a 0 return).
3. When the server later closes its end of the connection, the server TCP sends a FIN to the client TCP.
4. The client TCP responds with an ACK to acknowledge the server's FIN.

As with the SYN flag, and for the same reasons, the FIN flag consumes a byte of the sequence-number space for the connection. This is why we show the acknowledgement of the *FIN M* segment as *ACK M+1* in Figure 61-6.

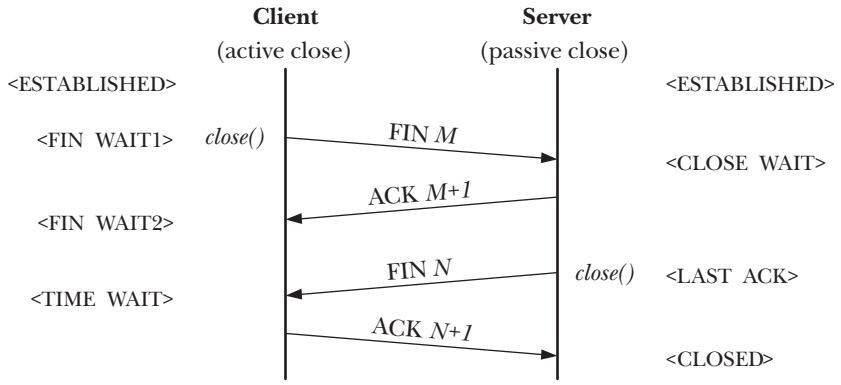

Figure 61-6: TCP connection termination

61.6.6 Calling *shutdown()* on a TCP Socket

The discussion in the preceding section assumed a full-duplex close; that is, an application closes both the sending and receiving channels of the TCP socket using *close()*. As noted in Section 61.2, we can use *shutdown()* to close just one channel of the connection (a half-duplex close). This section notes some specific details for *shutdown()* on a TCP socket.

Specifying *how* as `SHUT_WR` or `SHUT_RDWR` initiates the TCP connection termination sequence (i.e., the active close) described in Section 61.6.5, regardless of whether there are other file descriptors referring to the socket. Once this sequence has been initiated, the local TCP moves into the FIN_WAIT1 state, and then into the FIN_WAIT2 state, while the peer TCP moves into the CLOSE_WAIT state (Figure 61-6). If *how* is specified as `SHUT_WR`, then, since the socket file descriptor remains valid and the reading half of the connection remains open, the peer can continue to send data back to us.

The `SHUT_RD` operation can't be meaningfully used with TCP sockets. This is because most TCP implementations don't provide the expected behavior for `SHUT_RD`, and the effect of `SHUT_RD` varies across implementations. On Linux and a few other implementations, following a `SHUT_RD` (and after any outstanding data has been read), a *read()* returns end-of-file, as we expect from the description of `SHUT_RD` in Section 61.2. However, if the peer application subsequently writes data on its socket, then it is still possible to read that data on the local socket.

On some other implementations (e.g., the BSDs), `SHUT_RD` does indeed cause subsequent calls to *read()* to always return 0. However, on those implementations, if the peer continues to *write()* to the socket, then the data channel will eventually fill until the point where a further (blocking) call to *write()* by the peer will block. (With UNIX domain stream sockets, a peer would receive a `SIGPIPE` signal and the `EPIPE` error if it continued writing to its socket after a `SHUT_RD` had been performed on the local socket.)

In summary, the use of `SHUT_RD` should be avoided for portable TCP applications.

61.6.7 The TIME_WAIT State

The TCP TIME_WAIT state is a frequent source of confusion in network programming. Looking at Figure 61-4, we can see that a TCP performing an active close goes through this state. The TIME_WAIT state exists to serve two purposes:

- to implement reliable connection termination; and
- to allow expiration of old duplicate segments in the network so that they are not accepted by a new incarnation of the connection.

The TIME_WAIT state differs from the other states in that the event that causes a transition out of this state (to CLOSED) is a timeout. This timeout has a duration of twice the MSL (2MSL), where MSL (*maximum segment lifetime*) is the assumed maximum lifetime of a TCP segment in the network.

> An 8-bit time-to-live (TTL) field in the IP header ensures that all IP packets are eventually discarded if they don't reach their destination within a fixed number of hops (routers traversed) while traveling from the source to the destination host. The MSL is an estimate of the maximum time that an IP packet could take to exceed the TTL limit. Since it is represented using 8 bits, the TTL permits a maximum of 255 hops. Normally, an IP packet requires considerably fewer hops than this to complete its journey. A packet could encounter this limit because of certain types of router anomalies (e.g., a router configuration problem) that cause the packet to get caught in a network loop until it exceeds the TTL limit.

The BSD sockets implementation assumes a value of 30 seconds for the MSL, and Linux follows the BSD norm. Thus, the TIME_WAIT state has a lifetime of 60 seconds on Linux. However, RFC 1122 recommends a value of 2 minutes for the MSL, and, on implementations following this recommendation, the TIME_WAIT state can thus last 4 minutes.

We can understand the first purpose of the TIME_WAIT state—ensuring reliable connection termination—by looking at Figure 61-6. In this diagram, we can see that four segments are usually exchanged during the termination of a TCP connection. The last of these is an ACK sent from the TCP performing the active close to the TCP performing the passive close. Suppose that this ACK gets lost in the network. If this occurs, then the TCP performing the passive close will eventually retransmit its FIN. Having the TCP that performs the active close remain in the TIME_WAIT state for a fixed period ensures that it is available to resend the final ACK in this case. If the TCP that performs the active close did not still exist, then—since it wouldn't have any state information for the connection—the TCP protocol would respond to the resent FIN by sending an RST (reset) segment to the TCP performing the passive close, and this RST would be interpreted as an error. (This explains why the duration of the TIME_WAIT state is *twice* the MSL: one MSL for the final ACK to reach the peer TCP, plus a further MSL in case a further FIN must be sent.)

> An equivalent of the TIME_WAIT state is not required for the TCP performing the passive close, because it is the initiator of the final exchange in the connection termination. After sending the FIN, this TCP will wait for the ACK from its peer, and retransmit the FIN if its timer expires before the ACK is received.

To understand the second purpose of the TIME_WAIT state—ensuring the expiration of old duplicate segments in the network—we must remember that the retransmission algorithm used by TCP means that duplicate segments may be generated, and that, depending on routing decisions, these duplicates could arrive after the connection has been closed. For example, suppose that we have a TCP connection between two socket addresses, say, `204.152.189.116` port 21 (the FTP port) and `200.0.0.1` port 50,000. Suppose also that this connection is closed, and that later a new connection is established using exactly the same IP addresses and ports. This is referred to as a new incarnation of the connection. In this case, TCP must ensure that no old duplicate segments from the previous incarnation are accepted as valid data in the new incarnation. This is done by preventing a new incarnation from being established while there is an existing TCP in the TIME_WAIT state on one of the endpoints.

A frequent question posted to online forums is how to disable the TIME_WAIT state, since it can lead to the error `EADDRINUSE` ("Address already in use") when a restarted server tries to bind a socket to an address that has a TCP in the TIME_WAIT state. Although there are ways of doing this (see [Stevens et al., 2004]), and also ways of assassinating a TCP in this state (i.e., causing the TIME_WAIT state to terminate prematurely, see [Snader, 2000]), this should be avoided, since it would thwart the reliability guarantees that the TIME_WAIT state provides. In Section 61.10, we look at the use of the `SO_REUSEADDR` socket option, which can be used to avoid the usual causes of the `EADDRINUSE` error, while still allowing the TIME_WAIT to provide its reliability guarantees.

61.7 Monitoring Sockets: *netstat*

The *netstat* program displays the state of Internet and UNIX domain sockets on a system. It is a useful debugging tool when writing socket applications. Most UNIX implementations provide a version of *netstat*, although there is some variation in the syntax of its command-line arguments across implementations.

By default, when executed with no command-line options, *netstat* displays information for connected sockets in both the UNIX and Internet domains. We can use a number of command-line options to change the information displayed. Some of these options are listed in Table 61-1.

Table 61-1: Options for the *netstat* command

Option	Description
`-a`	Display information about all sockets, including listening sockets
`-e`	Display extended information (includes user ID of socket owner)
`-c`	Redisplay socket information continuously (each second)
`-l`	Display information only about listening sockets
`-n`	Display IP addresses, port numbers, and usernames in numerical form
`-p`	Show the process ID and name of program to which socket belongs
`--inet`	Display information for Internet domain sockets
`--tcp`	Display information for Internet domain TCP (stream) sockets
`--udp`	Display information for Internet domain UDP (datagram) sockets
`--unix`	Display information for UNIX domain sockets

Here is an abridged example of the output that we see when using *netstat* to list all Internet domain sockets on the system:

```
$ netstat -a --inet
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address    Foreign Address  State
tcp        0      0 *:50000          *:*              LISTEN
tcp        0      0 *:55000          *:*              LISTEN
tcp        0      0 localhost:smtp   *:*              LISTEN
tcp        0      0 localhost:32776  localhost:58000  TIME_WAIT
tcp    34767      0 localhost:55000  localhost:32773  ESTABLISHED
tcp        0 115680 localhost:32773  localhost:55000  ESTABLISHED
udp        0      0 localhost:61000  localhost:60000  ESTABLISHED
udp      684      0 *:60000          *:*
```

For each Internet domain socket, we see the following information:

- `Proto`: This is the socket protocol—for example, `tcp` or `udp`.
- `Recv-Q`: This is the number of bytes in the socket receive buffer that are as yet unread by the local application. For UDP sockets, this field counts not just data, but also bytes in UDP headers and other metadata.
- `Send-Q`: This is the number of bytes queued for transmission in the socket send buffer. As with the `Recv-Q` field, for UDP sockets, this field includes bytes in UDP headers and other metadata.
- `Local Address`: This is the address to which the socket is bound, expressed in the form *host-IP-address:port*. By default, both components of the address are displayed as names, unless the numeric values can't be resolved to corresponding host and service names. An asterisk (*) in the host part of the address means the wildcard IP address.
- `Foreign Address`: This is the address of the peer socket to which this socket is bound. The string `*:*` indicates no peer address.
- `State`: This is the current state of the socket. For a TCP socket, this state is one of those described in Section 61.6.3.

For further details, see the *netstat(8)* manual page.

Various Linux-specific files in the directory `/proc/net` allow a program to read much of the same information that is displayed by *netstat*. These files are named `tcp`, `udp`, `tcp6`, `udp6`, and `unix`, with the obvious purposes. For further details, see the *proc(5)* manual page.

61.8 Using *tcpdump* to Monitor TCP Traffic

The *tcpdump* program is a useful debugging tool that allows the superuser to monitor the Internet traffic on a live network, generating a real-time textual equivalent of diagrams such as Figure 61-3. Despite its name, *tcpdump* can be used to display traffic for all kinds of network packets (e.g., TCP segments, UDP datagrams, and ICMP packets). For each network packet, *tcpdump* displays information such as timestamps, the source and destination IP addresses, and further protocol-specific details. It is possible to select the packets to be monitored by protocol type, source

and destination IP address and port number, and a range of other criteria. Full details are provided in the *tcpdump* manual page.

> The *wireshark* (formerly *ethereal*; *http://www.wireshark.org/*) program performs a similar task to *tcpdump*, but displays traffic information via a graphical interface.

For each TCP segment, *tcpdump* displays a line of the following form:

---

src > *dst*: *flags data-seqno ack window urg* <*options*>

These fields have the following meanings:

- *src*: This is the source IP address and port.
- *dst*: This is the destination IP address and port.
- *flags*: This field contains zero or more of the following letters, each of which corresponds to one of the TCP control bits described in Section 61.6.1: S (SYN), F (FIN), P (PSH), R (RST), E (ECE), and C (CWR).
- *data-seqno*: This is the range of the sequence-number space covered by the bytes in this packet.

  > By default, the sequence-number range is displayed relative to the first byte monitored for this direction of the data stream. The *tcpdump* *−S* option causes sequence numbers to be displayed in absolute format.

- *ack*: This is a string of the form "`ack` *num*" indicating the sequence number of the next byte expected from the other direction on this connection.
- *window*: This is a string of the form "`win` *num*" indicating the number of bytes of receive buffer space available for transmission in the opposite direction on this connection.
- *urg*: This is a string of the form "`urg` *num*" indicating that this segment contains urgent data at the specified offset within the segment.
- *options*: This string describes any TCP options contained in the segment.

The *src*, *dst*, and *flags* fields always appear. The remaining fields are displayed only if appropriate.

The shell session below shows how *tcpdump* can be used to monitor the traffic between a client (running on the host `pukaki`) and a server (running on `tekapo`). In this shell session, we use two *tcpdump* options that make the output less verbose. The *−t* option suppresses the display of timestamp information. The *−N* option causes hostnames to be displayed without a qualifying domain name. Furthermore, for brevity, and because we don't describe the details of TCP options, we have removed the *options* fields from the lines of *tcpdump* output.

The server operates on port 55555, so our *tcpdump* command selects traffic for that port. The output shows the three segments exchanged during connection establishment:

```
$ tcpdump -t -N 'port 55555'
IP pukaki.60391 > tekapo.55555: S 3412991013:3412991013(0) win 5840
IP tekapo.55555 > pukaki.60391: S 1149562427:1149562427(0) ack 3412991014 win 5792
IP pukaki.60391 > tekapo.55555: . ack 1 win 5840
```

These three segments are the SYN, SYN/ACK, and ACK segments exchanged for the three-way handshake (see Figure 61-5).

In the following output, the client sends the server two messages, containing 16 and 32 bytes, respectively, and the server responds in each case with a 4-byte message:

```
IP pukaki.60391 > tekapo.55555: P 1:17(16) ack 1 win 5840
IP tekapo.55555 > pukaki.60391: . ack 17 win 1448
IP tekapo.55555 > pukaki.60391: P 1:5(4) ack 17 win 1448
IP pukaki.60391 > tekapo.55555: . ack 5 win 5840
IP pukaki.60391 > tekapo.55555: P 17:49(32) ack 5 win 5840
IP tekapo.55555 > pukaki.60391: . ack 49 win 1448
IP tekapo.55555 > pukaki.60391: P 5:9(4) ack 49 win 1448
IP pukaki.60391 > tekapo.55555: . ack 9 win 5840
```

For each of the data segments, we see an ACK sent in the opposite direction.

Lastly, we show the segments exchanged during connection termination (first, the client closes its end of the connection, and then the server closes the other end):

```
IP pukaki.60391 > tekapo.55555: F 49:49(0) ack 9 win 5840
IP tekapo.55555 > pukaki.60391: . ack 50 win 1448
IP tekapo.55555 > pukaki.60391: F 9:9(0) ack 50 win 1448
IP pukaki.60391 > tekapo.55555: . ack 10 win 5840
```

The above output shows the four segments exchanged during connection termination (see Figure 61-6).

61.9 Socket Options

Socket options affect various features of the operation of a socket. In this book, we describe just a couple of the many socket options that are available. An extensive discussion covering most standard socket options is provided in [Stevens et al., 2004]. See the *tcp(7)*, *udp(7)*, *ip(7)*, *socket(7)*, and *unix(7)* manual pages for additional Linux-specific details.

The *setsockopt()* and *getsockopt()* system calls set and retrieve socket options.

```
#include <sys/socket.h>

int getsockopt(int sockfd, int level, int optname, void *optval,
               socklen_t *optlen);
int setsockopt(int sockfd, int level, int optname, const void *optval,
               socklen_t optlen);
```

Both return 0 on success, or −1 on error

For both *setsockopt()* and *getsockopt()*, *sockfd* is a file descriptor referring to a socket.

The *level* argument specifies the protocol to which the socket option applies—for example, IP or TCP. For most of the socket options that we describe in this book, *level* is set to `SOL_SOCKET`, which indicates an option that applies at the sockets API level.

The *optname* argument identifies the option whose value we wish to set or retrieve. The *optval* argument is a pointer to a buffer used to specify or return the option value; this argument is a pointer to an integer or a structure, depending on the option.

The *optlen* argument specifies the size (in bytes) of the buffer pointed to by *optval*. For *setsockopt()*, this argument is passed by value. For *getsockopt()*, *optlen* is a value-result argument. Before the call, we initialize it to the size of the buffer pointed to by *optval*; upon return, it is set to the number of bytes actually written to that buffer.

As detailed in Section 61.11, the socket file descriptor returned by a call to *accept()* inherits the values of settable socket options from the listening socket.

Socket options are associated with an open file description (refer to Figure 5-2, on page 95). This means that file descriptors duplicated as a consequence of *dup()* (or similar) or *fork()* share the same set of socket options.

A simple example of a socket option is SO_TYPE, which can be used to find out the type of a socket, as follows:

```
int optval;
socklen_t optlen;

optlen = sizeof(optval);
if (getsockopt(sfd, SOL_SOCKET, SO_TYPE, &optval, &optlen) == -1)
    errExit("getsockopt");
```

After this call, *optval* contains the socket type—for example, SOCK_STREAM or SOCK_DGRAM. Using this call can be useful in a program that inherited a socket file descriptor across an *exec()*—for example, a program execed by *inetd*—since that program may not know which type of socket it inherited.

SO_TYPE is an example of a read-only socket option. It is not possible to use *setsockopt()* to change a socket's type.

61.10 The SO_REUSEADDR Socket Option

The SO_REUSEADDR socket option serves a number of purposes (see Chapter 7 of [Stevens et al., 2004] for details). We'll concern ourselves with only one common use: to avoid the EADDRINUSE ("Address already in use") error when a TCP server is restarted and tries to bind a socket to a port that currently has an associated TCP. There are two scenarios in which this usually occurs:

- A previous invocation of the server that was connected to a client performed an active close, either by calling *close()*, or by crashing (e.g., it was killed by a signal). This leaves a TCP endpoint that remains in the TIME_WAIT state until the 2MSL timeout expires.
- A previous invocation of the server created a child process to handle a connection to a client. Later, the server terminated, while the child continues to serve the client, and thus maintain a TCP endpoint using the server's well-known port.

In both of these scenarios, the outstanding TCP endpoint is unable to accept new connections. Nevertheless, in both cases, by default, most TCP implementations prevent a new listening socket from being bound to the server's well-known port.

> The EADDRINUSE error doesn't usually occur with clients, since they typically use an ephemeral port that won't be one of those ports currently in the TIME_WAIT state. However, if a client binds to a specific port number, then it also can encounter this error.

To understand the operation of the SO_REUSEADDR socket option, it can help to return to our earlier telephone analogy for stream sockets (Section 56.5). Like a telephone call (we ignore the notion of conference calls), a TCP socket connection is identifiable by the *combination* of a pair of connected endpoints. The operation of *accept()* is analogous to the task performed by a telephone operator on an internal company switchboard ("a server"). When an external telephone call arrives, the operator transfers it to some internal telephone ("a new socket") within the organization. From an outside perspective, there is no way of identifying that internal telephone. When multiple external calls are being handled by the switchboard, the only way of distinguishing them is via the combination of the external caller's number and the switchboard number. (The latter is necessary when we consider that there will be multiple company switchboards within the telephone network as a whole.) Analogously, each time we accept a socket connection on a listening socket, a new socket is created. All of these sockets are associated with the same local address as the listening socket. The only way of distinguishing them is via their connections to different peer sockets.

In other words, a connected TCP socket is identified by a 4-tuple (i.e., a combination of four values) of the following form:

```
{ local-IP-address, local-port, foreign-IP-address, foreign-port }
```

The TCP specification requires that each such tuple be unique; that is, only one corresponding connection incarnation ("telephone call") can exist. The problem is that most implementations (including Linux) enforce a stricter constraint: a local port can't be reused (i.e., specified in a call to *bind()*) if any TCP connection incarnation with a matching local port exists on the host. This rule is enforced even when the TCP could not accept new connections, as in the scenarios described at the start of this section.

Enabling the SO_REUSEADDR socket option relaxes this constraint, bringing it closer to the TCP requirement. By default, this option has the value 0, meaning that it is disabled. We enable the option by giving it a nonzero value before binding a socket, as shown in Listing 61-4.

Setting the SO_REUSEADDR option means that we can bind a socket to a local port even if another TCP is bound to the same port in either of the scenarios described at the start of this section. Most TCP servers should enable this option. We have already seen some examples of the use of this option in Listing 59-6 (page 1221) and Listing 59-9 (page 1228).

Listing 61-4: Setting the SO_REUSEADDR socket option

```
int sockfd, optval;

sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd == -1)
    errExit("socket");

optval = 1;
if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &optval,
        sizeof(optval)) == -1)
    errExit("socket");

if (bind(sockfd, &addr, addrlen) == -1)
    errExit("bind");
if (listen(sockfd, backlog) == -1)
    errExit("listen");
```

61.11 Inheritance of Flags and Options Across *accept()*

Various flags and settings can be associated with open file descriptions and file descriptors (Section 5.4). Furthermore, as described in Section 61.9, various options can be set for a socket. If these flags and options are set on a listening socket, are they inherited by the new socket returned by *accept()*? We describe the details here.

On Linux, the following attributes are not inherited by the new file descriptor returned by *accept()*:

- The status flags associated with an open file description—the flags that can be altered using the *fcntl()* F_SETFL operation (Section 5.3). These include flags such as O_NONBLOCK and O_ASYNC.
- The file descriptor flags—the flags that can be altered using the *fcntl()* F_SETFD operation. The only such flag is the close-on-exec flag (FD_CLOEXEC, described in Section 27.4).
- The *fcntl()* F_SETOWN (owner process ID) and F_SETSIG (generated signal) file descriptor attributes associated with signal-driven I/O (Section 63.3).

On the other hand, the new descriptor returned by *accept()* inherits a copy of most of the socket options that can be set using *setsockopt()* (Section 61.9).

SUSv3 is silent on the details described here, and the inheritance rules for the new connected socket returned by *accept()* vary across UNIX implementations. Most notably, on some UNIX implementations, if open file status flags such as O_NONBLOCK and O_ASYNC are set on a listening socket, then they are inherited by the new socket returned by *accept()*. For portability, it may be necessary to explicitly reset these attributes on a socket returned by *accept()*.

61.12 TCP Versus UDP

Given that TCP provides reliable delivery of data, while UDP does not, an obvious question is, "Why use UDP at all?" The answer to this question is covered at some length in Chapter 22 of [Stevens et al., 2004]. Here, we summarize some of the points that may lead us to choose UDP over TCP:

- A UDP server can receive (and reply to) datagrams from multiple clients, without needing to create and terminate a connection for each client (i.e., transmission of single messages using UDP has a lower overhead than is required when using TCP).
- For simple request-response communications, UDP can be faster than TCP, since it doesn't require connection establishment and termination. Appendix A of [Stevens, 1996] notes that in the best-case scenario, the time using TCP is

  ```
  2 * RTT + SPT
  ```

  In this formula, RTT is the round-trip time (the time required to send a request and receive a response), and SPT is the time spent by the server processing the request. (On a wide area network, the SPT value may be small compared to the RTT.) For UDP, the best-case scenario for a single request-response communication is

  ```
  RTT + SPT
  ```

  This is one RTT less than the time required for TCP. Since the RTT between hosts separated by large (i.e., intercontinental) distances or many intervening routers is typically several tenths of a second, this difference can make UDP attractive for some types of request-response communication. DNS is a good example of an application that uses UDP for this reason—using UDP allows name lookup to be performed by transmitting a single packet in each direction between servers.
- UDP sockets permit broadcasting and multicasting. *Broadcasting* allows a sender to transmit a datagram to the same destination port on all of the hosts connected to a network. *Multicasting* is similar, but allows a datagram to be sent to a specified set of hosts. For further details see Chapters 21 and 22 of [Stevens et al., 2004].
- Certain types of applications (e.g., streaming video and audio transmission) can function acceptably without the reliability provided by TCP. On the other hand, the delay that may occur after TCP tries to recover from a lost segment may result in transmission delays that are unacceptably long. (A delay in streaming media transmission may be worse than a brief loss of the transmission stream.) Therefore, such applications may prefer UDP, and adopt application-specific recovery strategies to deal with occasional packet loss.

An application that uses UDP, but nevertheless requires reliability, must implement reliability features itself. Usually, this requires at least sequence numbers, acknowledgements, retransmission of lost packets, and duplicate detection. An example of how to do this is shown in [Stevens et al., 2004]. However, if more

advanced features such as flow control and congestion control are also required, then it is probably best to use TCP instead. Trying to implement all of these features on top of UDP is complex, and, even when well implemented, the result is unlikely to perform better than TCP.

61.13 Advanced Features

UNIX and Internet domain sockets have many other features that we have not detailed in this book. We summarize a few of these features in this section. For full details, see [Stevens et al., 2004].

61.13.1 Out-of-Band Data

Out-of-band data is a feature of stream sockets that allows a sender to mark transmitted data as high priority; that is, the receiver can obtain notification of the availability of out-of-band data without needing to read all of the intervening data in the stream. This feature is used in programs such as *telnet*, *rlogin*, and *ftp* to make it possible to abort previously transmitted commands. Out-of-band data is sent and received using the `MSG_OOB` flag in calls to *send()* and *recv()*. When a socket receives notification of the availability of out-of-band data, the kernel generates the `SIGURG` signal for the socket owner (normally the process using the socket), as set by the *fcntl()* `F_SETOWN` operation.

When employed with TCP sockets, at most 1 byte of data may be marked as being out-of-band at any one time. If the sender transmits an additional byte of out-of-band data before the receiver has processed the previous byte, then the indication for the earlier out-of-band byte is lost.

> TCP's limitation of out-of-band data to a single byte is an artifact of the mismatch between the generic out-of-band model of the sockets API and its specific implementation using TCP's *urgent mode*. We touched on TCP's urgent mode when looking at the format of TCP segments in Section 61.6.1. TCP indicates the presence of urgent (out-of-band) data by setting the URG bit in the TCP header and setting the urgent pointer field to point to the urgent data. However, TCP has no way of indicating the length of an urgent data sequence, so the urgent data is considered to consist of a single byte.
>
> Further information about TCP urgent data can be found in RFC 793.

Under some UNIX implementations, out-of-band data is supported for UNIX domain stream sockets. Linux doesn't support this.

The use of out-of-band data is nowadays discouraged, and it may be unreliable in some circumstances (see [Gont & Yourtchenko, 2009]). An alternative is to maintain a pair of stream sockets for communication. One of these is used for normal communication, while the other is used for high-priority communication. An application can monitor both channels using one of the techniques described in Chapter 63. This approach allows multiple bytes of priority data to be transmitted. Furthermore, it can be employed with stream sockets in any communication domain (e.g., UNIX domain sockets).

61.13.2 The *sendmsg()* and *recvmsg()* System Calls

The *sendmsg()* and *recvmsg()* system calls are the most general purpose of the socket I/O system calls. The *sendmsg()* system call can do everything that is done by *write()*, *send()*, and *sendto()*; the *recvmsg()* system call can do everything that is done by *read()*, *recv()*, and *recvfrom()*. In addition, these calls allow the following:

- We can perform scatter-gather I/O, as with *readv()* and *writev()* (Section 5.7). When we use *sendmsg()* to perform gather output on a datagram socket (or *writev()* on a connected datagram socket), a single datagram is generated. Conversely, *recvmsg()* (and *readv()*) can be used to perform scatter input on a datagram socket, dispersing the bytes of a single datagram into multiple user-space buffers.
- We can transmit messages containing domain-specific *ancillary data* (also known as control information). Ancillary data can be passed via both stream and datagram sockets. We describe some examples of ancillary data below.

> Linux 2.6.33 adds a new system call, *recvmmsg()*. This system call is similar to *recvmsg()*, but allows multiple datagrams to be received in a single system call. This reduces the system-call overhead in applications that deal with high levels of network traffic. An analogous *sendmmsg()* system call is likely to be added in a future kernel version.

61.13.3 Passing File Descriptors

Using *sendmsg()* and *recvmsg()*, we can pass ancillary data containing a file descriptor from one process to another process on the same host via a UNIX domain socket. Any type of file descriptor can be passed in this manner—for example, one obtained from a call to *open()* or *pipe()*. An example that is more relevant to sockets is that a master server could accept a client connection on a TCP listening socket and pass that descriptor to one of the members of a pool of server child processes (Section 60.4), which would then respond to the client request.

Although this technique is commonly referred to as passing a file descriptor, what is really being passed between the two processes is a reference to the same open file description (Figure 5-2, on page 95). The file descriptor number employed in the receiving process would typically be different from the number employed in the sender.

> An example of passing file descriptors is provided in the files `scm_rights_send.c` and `scm_rights_recv.c` in the `sockets` subdirectory in the source code distribution for this book.

61.13.4 Receiving Sender Credentials

Another example of the use of ancillary data is for receiving sender credentials via a UNIX domain socket. These credentials consist of the user ID, the group ID, and the process ID of the sending process. The sender may specify its user and group IDs as the corresponding real, effective, or saved set IDs. This allows the receiving process to authenticate a sender on the same host. For further details, see the *socket(7)* and *unix(7)* manual pages.

Unlike passing file credentials, passing sender credentials is not specified in SUSv3. Aside from Linux, this feature is implemented in some of the modern BSDs

(where the credentials structure contains somewhat more information than on Linux), but is available on few other UNIX implementations. The details of credential passing on FreeBSD are described in [Stevens et al., 2004].

On Linux, a privileged process can fake the user ID, group ID, and process ID that are passed as credentials if it has, respectively, the CAP_SETUID, CAP_SETGID, and CAP_SYS_ADMIN capabilities.

> An example of passing credentials is provided in the files scm_cred_send.c and scm_cred_recv.c in the sockets subdirectory in the source code distribution for this book.

61.13.5 Sequenced-Packet Sockets

Sequenced-packet sockets combine features of both stream and datagram sockets:

- Like stream sockets, sequenced-packet sockets are connection-oriented. Connections are established in the same way as for stream sockets, using *bind()*, *listen()*, *accept()*, and *connect()*.
- Like datagram sockets, message boundaries are preserved. A *read()* from a sequenced-packet socket returns exactly one message (as written by the peer). If the message is longer than the buffer supplied by the caller, the excess bytes are discarded.
- Like stream sockets, and unlike datagram sockets, communication via sequenced-packet sockets is reliable. Messages are delivered to the peer application error-free, in order, and unduplicated, and they are guaranteed to arrive (assuming that there is not a system or application crash, or a network outage).

A sequenced-packet socket is created by calling *socket()* with the *type* argument specified as SOCK_SEQPACKET.

Historically, Linux, like most UNIX implementations, did not support sequenced-packet sockets in either the UNIX or the Internet domains. However, starting with kernel 2.6.4, Linux supports SOCK_SEQPACKET for UNIX domain sockets.

In the Internet domain, the UDP and TCP protocols do not support SOCK_SEQPACKET, but the SCTP protocol (described in the next section) does.

We don't show an example of the use of sequenced-packet sockets in this book, but, other than the preservation of message boundaries, their use is very similar to stream sockets.

61.13.6 SCTP and DCCP Transport-Layer Protocols

SCTP and DCCP are two newer transport-layer protocols that are likely to become increasingly common in the future.

The *Stream Control Transmission Protocol* (SCTP, *http://www.sctp.org/*) was designed to support telephony signaling in particular, but is also general purpose. Like TCP, SCTP provides reliable, bidirectional, connection-oriented transport. Unlike TCP, SCTP preserves message boundaries. One of the distinctive features of SCTP is multistream support, which allows multiple logical data streams to be employed over a single connection.

SCTP is described in [Stewart & Xie, 2001], [Stevens et al., 2004], and in RFCs 4960, 3257, and 3286.

SCTP is available on Linux since kernel 2.6. Further information about this implementation can be found at *http://lksctp.sourceforge.net/*.

Throughout the preceding chapters that describe the sockets API, we equated Internet domain stream sockets with TCP. However, SCTP provides an alternative protocol for implementing stream sockets, created using the following call:

```
socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP);
```

Starting in kernel 2.6.14, Linux supports a new datagram protocol, the *Datagram Congestion Control Protocol* (DCCP). Like TCP, DCCP provides congestion control (removing the need to implement congestion control at the application level) to prevent a fast transmitter from overwhelming the network. (We explained congestion control when describing TCP in Section 58.6.3.) However, unlike TCP (but like UDP), DCCP doesn't provide guarantees about reliable or in-order delivery, and thus allows applications that don't need these features to avoid the delays that they can incur. Information about DCCP can be found at *http://www.read.cs.ucla.edu/dccp/* and RFCs 4336 and 4340.

61.14 Summary

In various circumstances, partial reads and writes can occur when performing I/O on stream sockets. We showed the implementation of two functions, *readn()* and *writen()*, that can be used to ensure a complete buffer of data is read or written.

The *shutdown()* system call provides more precise control over connection termination. Using *shutdown()*, we can forcibly shut down either or both halves of a bidirectional communication stream, regardless of whether there are other open file descriptors referring to the socket.

Like *read()* and *write()*, *recv()* and *send()* can be used to perform I/O on a socket, but calls provide an extra argument, *flags*, that controls socket-specific I/O functionality.

The *sendfile()* system call allows us to efficiently copy the contents of a file to a socket. This efficiency is gained because we don't need to copy the file data to and from user memory, as would be required with calls to *read()* and *write()*.

The *getsockname()* and *getpeername()* system calls retrieve, respectively, the local address to which a socket is bound and the address of the peer to which that socket is connected.

We considered some details of the operation of TCP, including TCP states and the TCP state transition diagram, and TCP connection establishment and termination. As part of this discussion, we saw why the TIME_WAIT state is an important part of TCP's reliability guarantee. Although this state can lead to the "Address already in use" error when restarting a server, we later saw that the `SO_REUSEADDR` socket option can be used to avoid this error, while nevertheless allowing the TIME_WAIT state to serve its intended purpose.

The *netstat* and *tcpdump* commands are useful tools for monitoring and debugging applications that use sockets.

The *getsockopt()* and *setsockopt()* system calls retrieve and modify options affecting the operation of a socket.

On Linux, when a new socket is created by *accept()*, it does not inherit the listening sockets open file status flags, file descriptor flags, or file descriptor attributes related to signal-driven I/O. However, it does inherit the settings of socket options. We noted that SUSv3 is silent on these details, which vary across implementations.

Although UDP doesn't provide the reliability guarantees of TCP, we saw that there are nevertheless reasons why UDP can be preferable for some applications.

Finally, we outlined a few advanced features of sockets programming that we don't describe in detail in this book.

Further information

Refer to the sources of further information listed in Section 59.15.

61.15 Exercises

61-1. Suppose that the program in Listing 61-2 (`is_echo_cl.c`) was modified so that, instead of using *fork()* to create two processes that operate concurrently, it instead used just one process that first copies its standard input to the socket and then reads the server's response. What problem might occur when running this client? (Look at Figure 58-8, on page 1190.)

61-2. Implement *pipe()* in terms of *socketpair()*. Use *shutdown()* to ensure that the resulting pipe is unidirectional.

61-3. Implement a replacement for *sendfile()* using *read()*, *write()*, and *lseek()*.

61-4. Write a program that uses *getsockname()* to show that, if we call *listen()* on a TCP socket without first calling *bind()*, the socket is assigned an ephemeral port number.

61-5. Write a client and a server that permit the client to execute arbitrary shell commands on the server host. (If you don't implement any security mechanism in this application, you should ensure that the server is operating under a user account where it can do no damage if invoked by malicious users.) The client should be executed with two command-line arguments:

```
$ ./is_shell_cl server-host 'some-shell-command'
```

After connecting to the server, the client sends the given command to the server, and then closes its writing half of the socket using *shutdown()*, so that the server sees end-of-file. The server should handle each incoming connection in a separate child process (i.e., a concurrent design). For each incoming connection, the server should read the command from the socket (until end-of-file), and then exec a shell to perform the command. Here are a couple hints:

- See the implementation of *system()* in Section 27.7 for an example of how to execute a shell command.
- By using *dup2()* to duplicate the socket on standard output and standard error, the execed command will automatically write to the socket.

61-6. Section 61.13.1 noted that an alternative to out-of-band data would be to create two socket connections between the client and server: one for normal data and one for priority data. Write client and server programs that implement this framework. Here are a few hints:

- The server needs some way of knowing which two sockets belong to the same client. One way to do this is to have the client first create a listening socket using an ephemeral port (i.e., binding to port 0). After obtaining the ephemeral port number of its listening socket (using *getsockname()*), the client connects its "normal" socket to the server's listening socket and sends a message containing the port number of the client's listening socket. The client then waits for the server to use the client's listening socket to make a connection in the opposite direction for the "priority" socket. (The server can obtain the client's IP address during the *accept()* of the normal connection.)
- Implement some type of security mechanism to prevent a rogue process from trying to connect to the client's listening socket. To do this, the client could send a cookie (i.e., some type of unique message) to the server using the normal socket. The server would then return this cookie via the priority socket so that the client could verify it.
- In order to experiment with transmitting normal and priority data from the client to the server, you will need to code the server to multiplex the input from the two sockets using *select()* or *poll()* (described in Section 63.2).