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 This standard has been superseded by SUSv3.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."

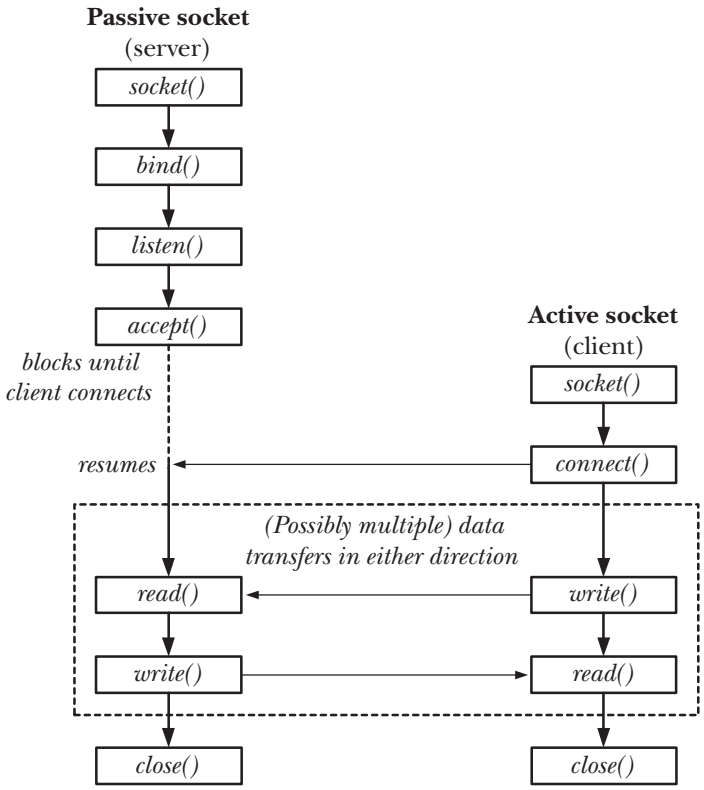

Figure 56-1: Overview of system calls used with stream sockets

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.

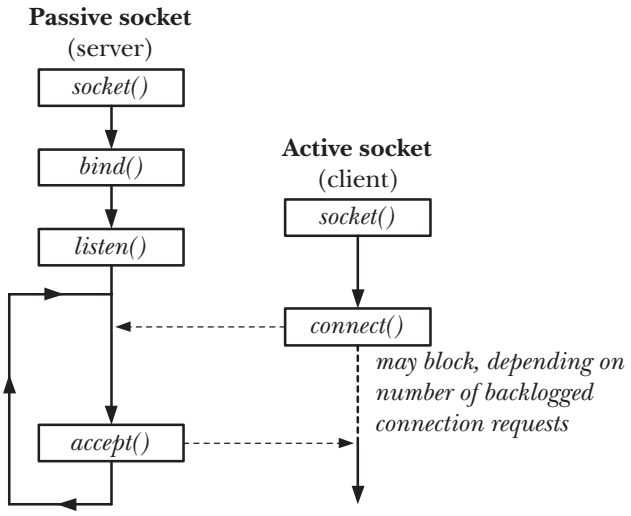

Figure 56-2: A pending socket connection

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.

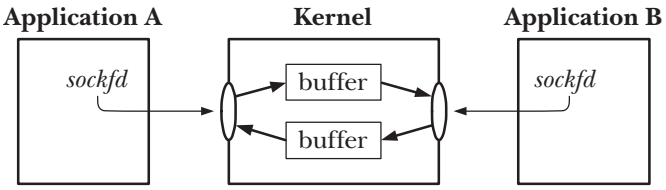

Figure 56-3: UNIX domain stream sockets provide a bidirectional communication channel

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.

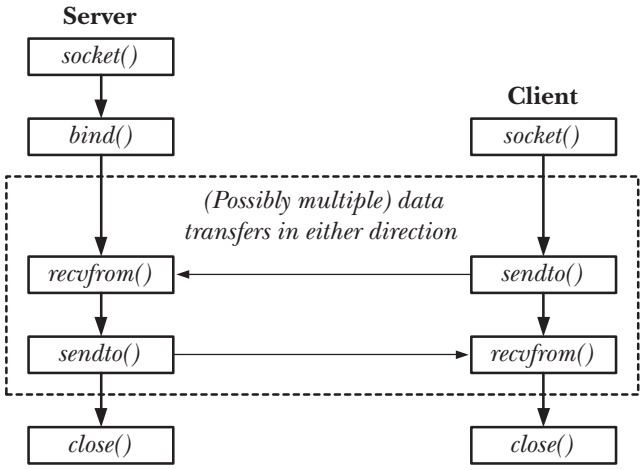

Figure 56-4: Overview of system calls used with datagram sockets

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.