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. `MSG_DONTWAIT` appears on a few other UNIX implementations, and `MSG_NOSIGNAL` and `MSG_MORE` are 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*.

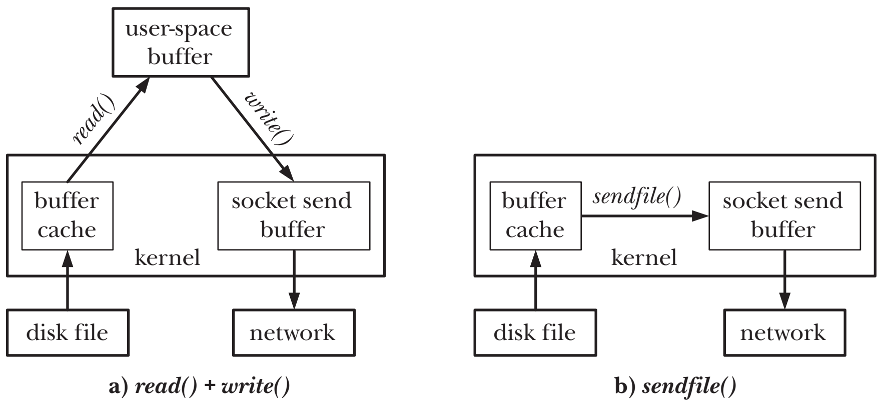

Figure 61-1: Transferring the contents of a file to a socket

```
#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, 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, 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.

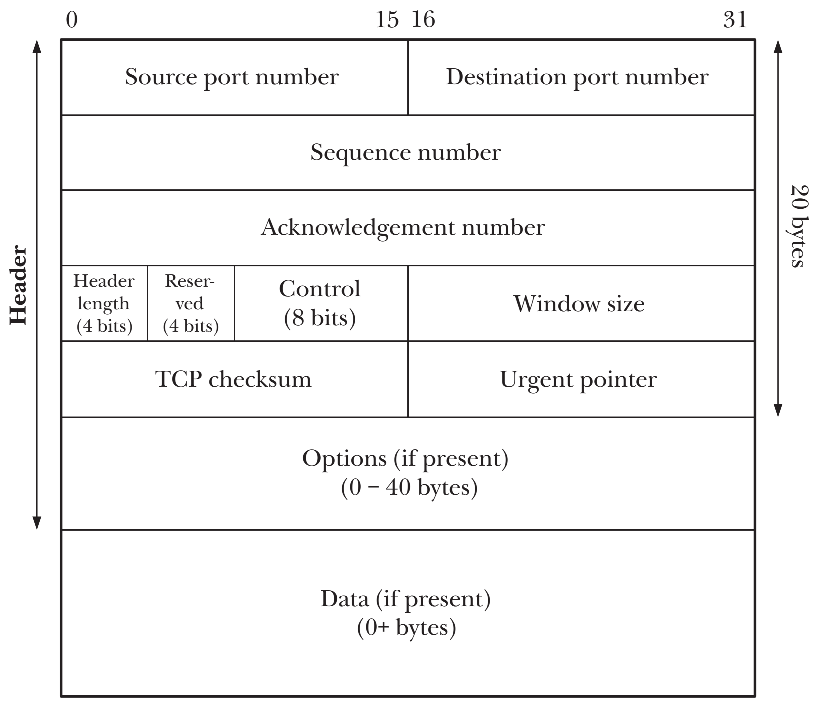

Figure 61-2: Format of a TCP segment

- *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.)

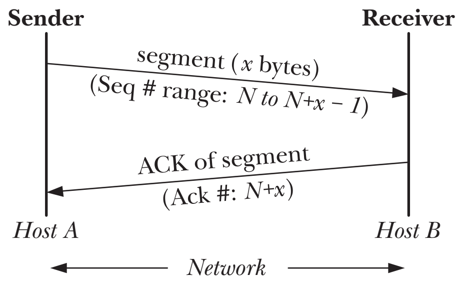

Figure 61-3: Acknowledgements in TCP

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.

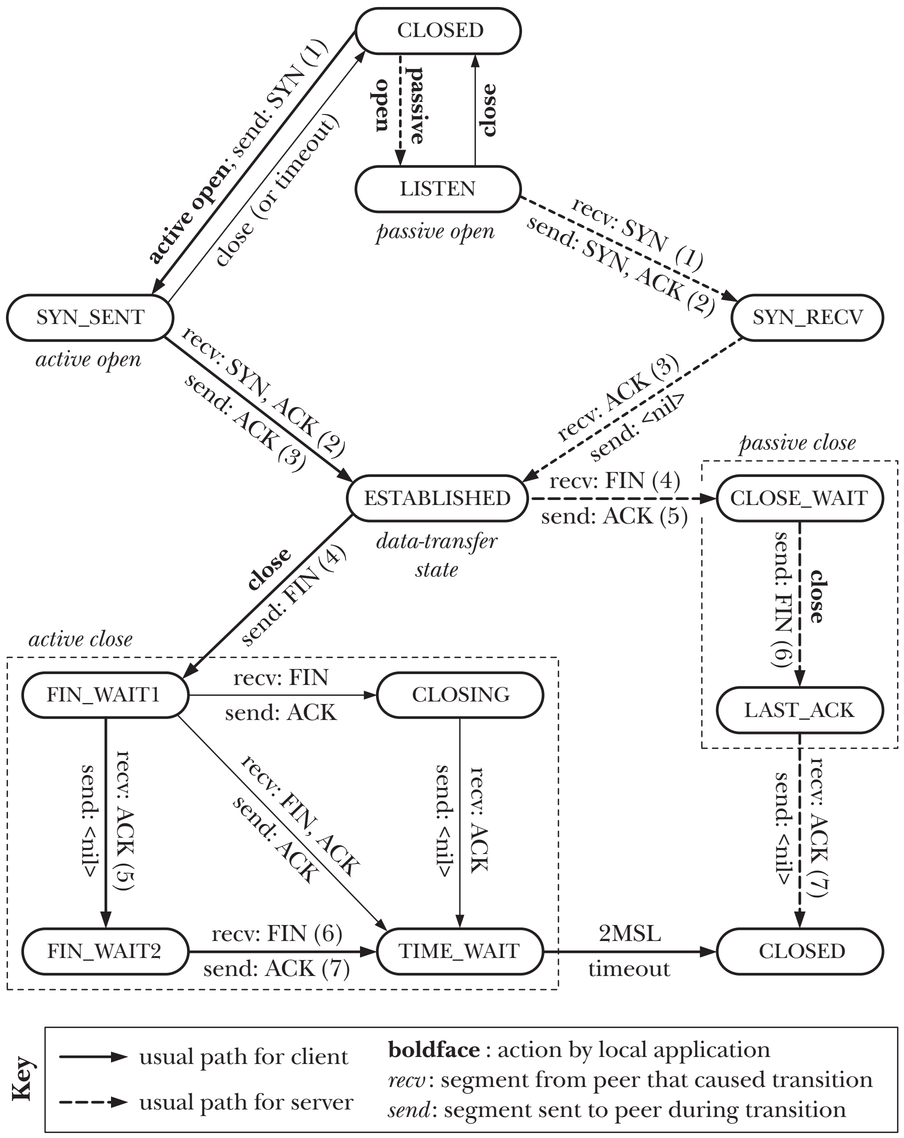

Figure 61-4: TCP state transition diagram

> 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.

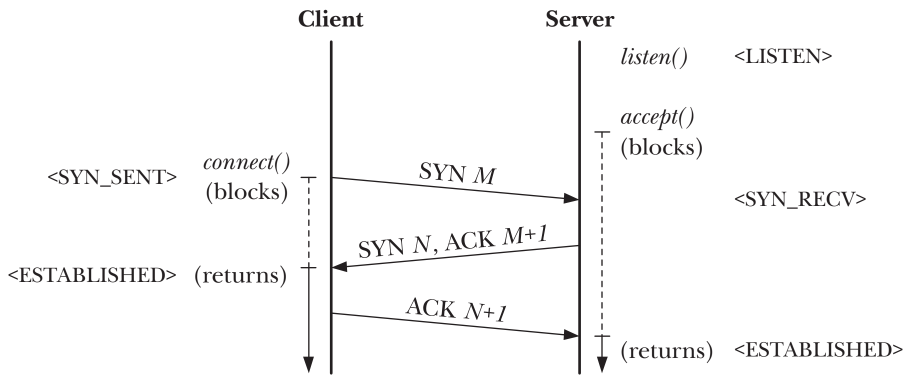

Figure 61-5: Three-way handshake for TCP connection establishment

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.

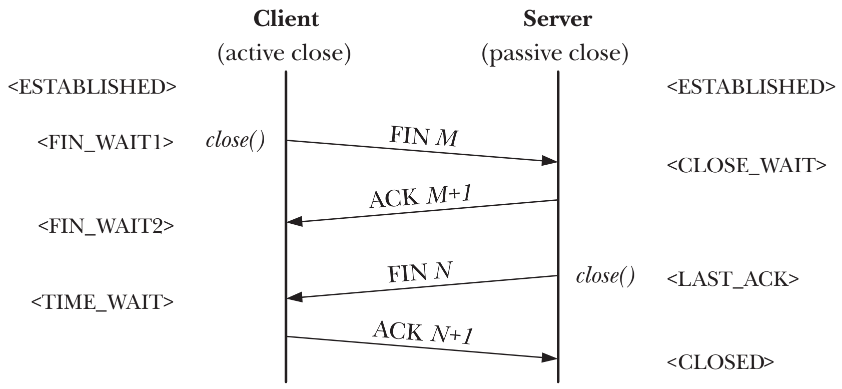

Figure 61-6: TCP connection termination

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 TCP/IP 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).