

## chapter 7

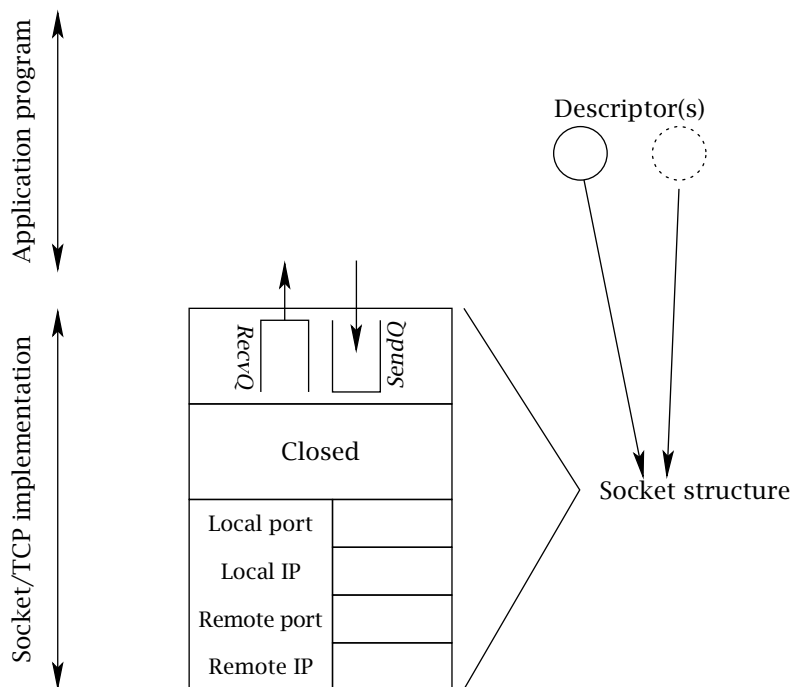
# Under the Hood

**S**ome of the subtleties of network programming are difficult to grasp without some understanding of the data structures associated with each socket in the implementation and certain details of how the underlying protocols work. This is especially true of stream (TCP) sockets. This chapter describes some of what goes on “under the hood” when you create and use a socket. The initial discussion and Section 7.5 apply to both datagram (UDP) and stream (TCP) sockets; the rest applies only to TCP sockets. Please note that this description covers only the normal sequence of events and glosses over many details. Nevertheless, we believe that even this basic level of understanding is helpful. Readers who want the full story are referred to the TCP specification [13] or to one of the more comprehensive treatises on the subject [2, 17].

Figure 7.1 is a simplified view of some of the information associated with a socket—that is, the object created by a call to `socket()`. The integer returned by `socket()` is best thought of as a “handle” that identifies the collection of data structures for one communication endpoint that we refer to in this chapter as the “socket structure.” As the figure indicates, more than one descriptor can refer to the same socket structure. In fact, descriptors in *different processes* can refer to the same underlying socket structure.

By “socket structure” here we mean all data structures in the socket layer and TCP implementation that contain state information relevant to this socket abstraction. Thus, the socket structure contains send and receive queues and other information, including the following:

- The local and remote Internet addresses and port numbers associated with the socket. The local Internet address (labeled “Local IP” in the figure) is one of those assigned to the



**Figure 7.1:** Data structures associated with a socket.

local host; the local port is set at `bind()` time. The remote address and port identify the remote socket, if any, to which the local socket is connected. We will say more about how and when these values are determined shortly. (Section 7.5 contains a concise summary.)

- A FIFO queue (“*RecvQ*”) of received data waiting to be delivered and a FIFO queue (“*SendQ*”) for data waiting to be transmitted.
- For a TCP socket, additional protocol state information relevant to the opening and closing TCP handshakes. In Figure 7.1, the state is “Closed”; all sockets start out in the Closed state.

Some general-purpose operating systems provide tools that enable users to obtain a “snapshot” of these underlying data structures. One such tool is `netstat`, which is typically available on both UNIX (Linux) and Windows platforms. Given appropriate options, `netstat` displays exactly the information indicated in Figure 7.1: number of bytes in *SendQ* and *RecvQ*, local and remote IP addresses and port numbers, and the connection state. Command-line options may vary, but the output should look something like this:

```

Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp      0      0 0.0.0.0:36045           0.0.0.0:*               LISTEN
tcp      0      0 0.0.0.0:111             0.0.0.0:*               LISTEN
tcp      0      0 0.0.0.0:53363           0.0.0.0:*               LISTEN
tcp      0      0 127.0.0.1:25            0.0.0.0:*               LISTEN
tcp      0      0 128.133.190.219:34077   4.71.104.187:80        TIME_WAIT
tcp      0      0 128.133.190.219:43346   79.62.132.8:22         ESTABLISHED
tcp      0      0 128.133.190.219:875     128.133.190.43:2049    ESTABLISHED
tcp6     0      0 :::22                   :::*                     LISTEN

```

The first four lines and the last line depict server sockets listening for connections. (The last line is a listening socket bound to an IPv6 address.) The fifth line corresponds to a connection to a Web server (port 80) that is partially shut down (see Section 7.4.2). The next-to-last two lines are existing TCP connections. You may want to play with `netstat`, if it is available on your system, to examine the status of connections in the scenarios depicted in Figures 7.8–7.11. Be aware, however, that because the transitions between states depicted in the figures happen so quickly, it may be difficult to catch them in the “snapshot” provided by `netstat`.

Knowing that these data structures exist and how they are affected by the underlying protocols is useful because they control various aspects of the behavior of the socket. For example, because TCP provides a *reliable* byte-stream service, a copy of any data sent over a TCP socket must be kept *by the TCP implementation* until it has been successfully received at the other end of the connection. Completion of a call to `send()` on a TCP socket does *not*, in general, imply that the data has actually been transmitted—only that it has been copied into the local buffer. Under normal conditions, it will be transmitted soon, but the exact moment is under the control of TCP, not the application. Moreover, the nature of the byte-stream service means that message boundaries are *not* necessarily preserved in the input stream. As we saw in Section 5.2.1, this means that most application protocols need a *framing* mechanism, so the receiver can tell when it has received an entire message.

On the other hand, with a datagram (UDP) socket, packets are *not* buffered for retransmission, and by the time a call to `send/sendto()` returns, the data has been given to the network subsystem for transmission. If the network subsystem cannot handle the message for some reason, the packet is silently dropped (but this is rare).

The next three sections deal with some of the subtleties of sending and receiving with TCP’s byte-stream service. Then, Section 7.4 considers the connection establishment and termination of the TCP protocol. Finally, Section 7.5 discusses the process of matching incoming packets to sockets and the rules about binding to port numbers.

## 7.1 Buffering and TCP

As a programmer, the most important thing to remember when using a TCP socket is this:

**You cannot assume any correspondence between the sizes of writes to one end of the connection and sizes of reads from the other end.**



In particular, data passed in a single invocation of `send()` at the sender can be spread across multiple invocations of `recv()` at the other end; a single call to `recv()` may return data passed in multiple calls to `send()`.

To see this, consider a program that does the following:

```
rv = connect(s,...);
...
rv = send(s, buffer0, 1000, 0);
...
rv = send(s, buffer1, 2000, 0);
...
rv = send(s, buffer2, 5000, 0);
...
close(s);
```

where the ellipses represent code that sets up the data in the buffers but contains no other calls to `send()`. This TCP connection transfers 8000 bytes to the receiver. The way these 8000 bytes are grouped for delivery at the receiving end of the connection depends on the timing between the calls to `send()` and `recv()` at the two ends of the connection—as well as the size of the buffers provided to the `recv()` calls.

We can think of the sequence of all bytes sent (in one direction) on a TCP connection up to a particular instant in time as being divided into three FIFO queues:

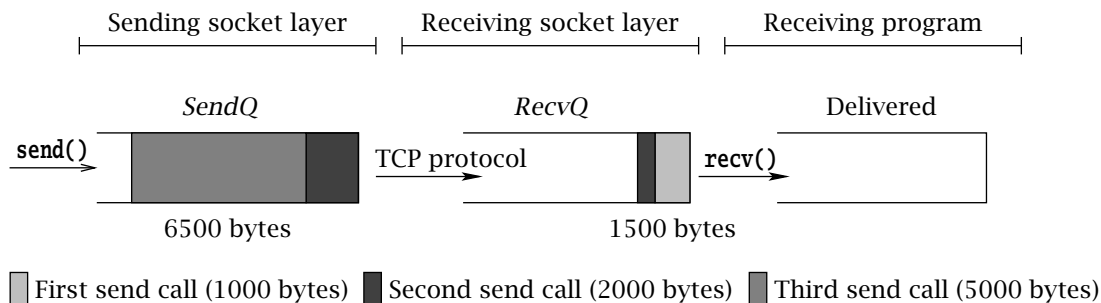
1. *SendQ*: Bytes buffered in the underlying implementation at the sender that have been written to the output stream but not yet successfully transmitted to the receiving host.
2. *RecvQ*: Bytes buffered in the underlying implementation at the receiver waiting to be delivered to the receiving program—that is, read from the input stream.
3. *Delivered*: Bytes already read from the input stream by the receiver.

A call to `send()` at the sender appends bytes to *SendQ*. The TCP protocol is responsible for moving bytes—in order—from *SendQ* to *RecvQ*. It is important to realize that this transfer cannot be controlled or directly observed by the user program, and that it occurs in chunks whose sizes are more or less independent of the size of the buffers passed in sends. Bytes are moved from *RecvQ* to *Delivered* by calls to `recv()`; the size of the transferred chunks depends on the amount of data in *RecvQ* and the size of the buffer given to `recv()`.

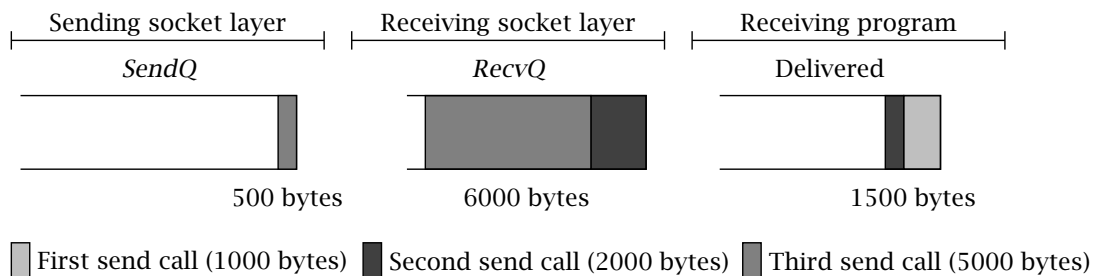
Figure 7.2 shows one possible state of the three queues *after* the three sends in the example above, but *before* any calls to `recv()` at the other end. The different shading patterns denote bytes passed in the three different invocations of `send()` shown above.

The output of `netstat` on the sending host at the instant depicted in this figure would contain a line like:

Active Internet connections					
Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	6500	10.21.44.33:43346	192.0.2.8:22	ESTABLISHED



**Figure 7.2:** State of the three queues after three `send()` calls.



**Figure 7.3:** After first `recv()`.

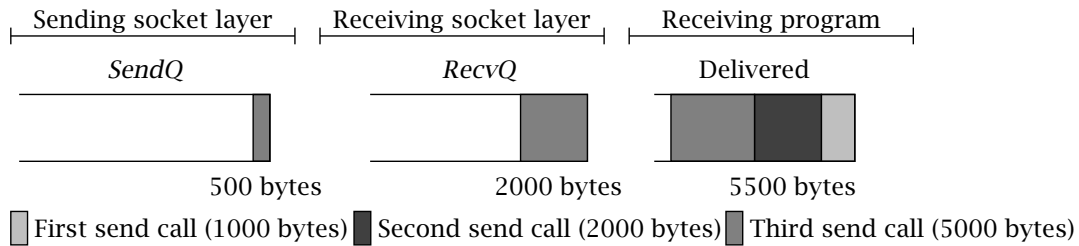
On the receiving host, `netstat` shows:

```
Active Internet connections
Proto Recv-Q Send-Q Local Address      Foreign Address    State
tcp      1500      0 192.0.2.8:22      10.21.44.33:43346 ESTABLISHED
```

Now suppose the receiver calls `recv()` with a byte array of size 2000. The `recv()` call will move all of the 1500 bytes present in the waiting-for-delivery (*RecvQ*) queue into the byte array and return the value 1500. Note that this data includes bytes passed in both the first and second calls to `send()`. At some time later, after TCP has completed transfer of more data, the three partitions might be in the state shown in Figure 7.3.

If the receiver now calls `recv()` with a buffer of size 4000, that many bytes will be moved from the waiting-for-delivery (*RecvQ*) queue to the already-delivered (*Delivered*) queue; this includes the remaining 1500 bytes from the second `send()`, plus the first 2500 bytes from the third `send()`. The resulting state of the queues is shown in Figure 7.4.

The number of bytes returned by the next call to `recv()` depends on the size of the buffer and the timing of the transfer of data over the network from the send-side socket/TCP implementation to the receive-side implementation. The movement of data from the *SendQ* to



**Figure 7.4:** After another `recv()`.

the *RecvQ* buffer has important implications for the design of application protocols. We have already encountered the challenge of parsing messages as they are received via a socket when in-band delimiters are used for framing (see Section 5.2). In the following sections, we consider two more subtle ramifications.

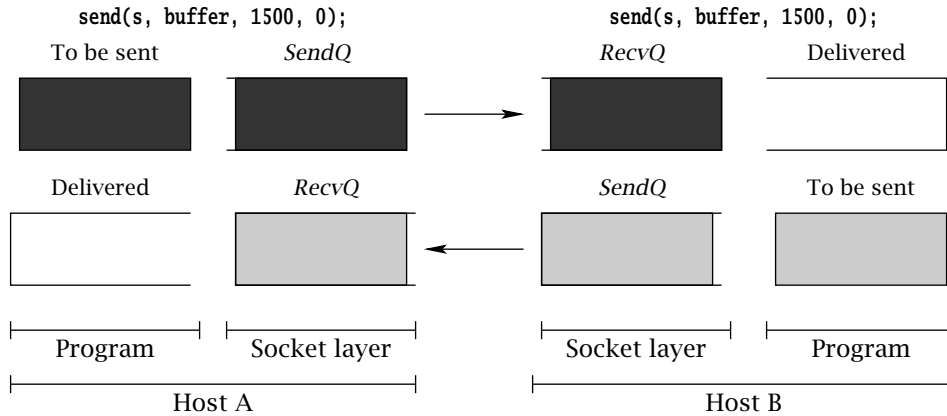
## 7.2 Deadlock Danger

Application protocols have to be designed with some care to avoid *deadlock*—that is, a state in which each peer is blocked waiting for the other to do something. For example, it is pretty obvious that if both client and server try to receive immediately after a connection is established, deadlock will result. Deadlock can also occur in less immediate ways.

The buffers *SendQ* and *RecvQ* in the implementation have limits on their capacity. Although the actual amount of memory they use may grow and shrink dynamically, a hard limit is necessary to prevent all of the system's memory from being gobbled up by a single TCP connection under control of a misbehaving program. Because these buffers are finite, they can fill up. It is this fact, coupled with TCP's *flow control* mechanism, that leads to the possibility of another form of deadlock.

Once *RecvQ* is full, the TCP flow control mechanism kicks in and prevents the transfer of any bytes from the sending host's *SendQ* until space becomes available in *RecvQ* as a result of the receiver calling `recv()`. (The purpose of the flow control mechanism is to ensure that the sender does not transmit data faster than the receiving system can handle.) A sending program can continue to call `send()` until *SendQ* is full. However, once *SendQ* is full, a `send()` will block until space becomes available, that is, until some bytes are transferred to the receiving socket's *RecvQ*. If *RecvQ* is also full, everything stops until the receiving program calls `recv` and some bytes are transferred to *Delivered*.

Let's assume the sizes of *SendQ* and *RecvQ* are *SQS* and *RQS*, respectively. A call to `send()` passing in a buffer of size *n* such that  $n > SQS$  will not return until at least  $n - SQS$  bytes have been transferred to *RecvQ* at the receiving host. If *n* exceeds  $(SQS + RQS)$ , `send()` cannot return until after the receiving program has read at least  $n - (SQS + RQS)$  bytes from the input stream. If the receiving program does not call `recv()`, a large `send()` may not complete successfully.



**Figure 7.5:** Deadlock due to simultaneous sends to output streams at opposite ends of the connection.

In particular, if both ends of the connection call `send()` simultaneously, each passing a buffer bigger than  $SQS + RQS$  bytes, deadlock *will* result: neither write will ever complete, and both programs will remain blocked forever.

As a concrete example, consider a connection between a program on Host A and a program on Host B. Assume  $SQS$  and  $RQS$  are 500 at both A and B. Figure 7.5 shows what happens when both programs try to send 1500 bytes at the same time. The first 500 bytes of data in the program at Host A have been transferred to the other end; another 500 bytes have been copied into *SendQ* at Host A. The remaining 500 bytes cannot be sent—and therefore `send()` will not return—until space frees up in *RecvQ* at Host B. Unfortunately, the same situation holds in the program at Host B. Therefore, neither program's call to `send()` call will ever return!

The moral of the story: Design the protocol carefully to avoid sending large quantities of data simultaneously in both directions.



## 7.3 Performance Implications

The TCP implementation's need to copy user data into *SendQ* before sending it also has implications for performance. In particular, the sizes of the *SendQ* and *RecvQ* buffers affect the throughput achievable over a TCP connection. “Throughput” is the *rate* at which bytes of user data from the sender are made available to the receiving program; in programs that transfer a large amount of data, we want to maximize the number of bytes delivered per second. In the absence of network capacity or other limitations, *bigger buffers generally result in higher throughput*.

The reason for this has to do with the cost of transferring data into and out of the buffers in the underlying implementation. If you want to transfer  $n$  bytes of data (where  $n$  is large), it

is generally much more efficient to call `send()` once with a buffer of size  $n$  than it is to call it  $n$  times with a single byte.<sup>1</sup> However, if you call `send()` with a size parameter that is much larger than  $SQS$  (the size of *SendQ*), the system has to transfer the data from the user address space in  $SQS$ -sized chunks. That is, the socket implementation fills up the *SendQ* buffer, waits for data to be transferred out of it by the TCP protocol, refills *SendQ*, waits some more, and so on. Each time the socket implementation has to wait for data to be removed from *SendQ*, some time is wasted in the form of overhead (a context switch occurs). This overhead is comparable to that incurred by a completely new call to `send()`. Thus the *effective* size of a call to `send()` is limited by the actual  $SQS$ . The same thing applies at the receiving end: however large the buffer we pass to `recv()`, data will be copied out in chunks no larger than  $RQS$ , with overhead incurred between chunks.

If you are writing a program for which throughput is an important performance metric, you will want to change the send and receive buffer sizes using the `SO_RCVBUF` and `SO_SNDBUF` socket options. Although there is always a system-imposed maximum size for each buffer, it is typically significantly larger than the default on modern systems. Remember that these considerations apply only if your program needs to send an amount of data significantly larger than the buffer size, all at once. Note also that wrapping a TCP socket in a **FILE**-stream adds another stage of buffering and additional overhead, and thus may negatively affect throughput.

## 7.4 TCP Socket Life Cycle

When a new TCP **socket** is created, it cannot be used immediately for sending and receiving data. First it needs to be connected to a remote endpoint. Let us therefore consider in more detail how the underlying structure gets to and from the connected, or “Established”, state. As we’ll see later, these details affect the definition of reliability and the ability to bind a socket to a particular port that was in use earlier.

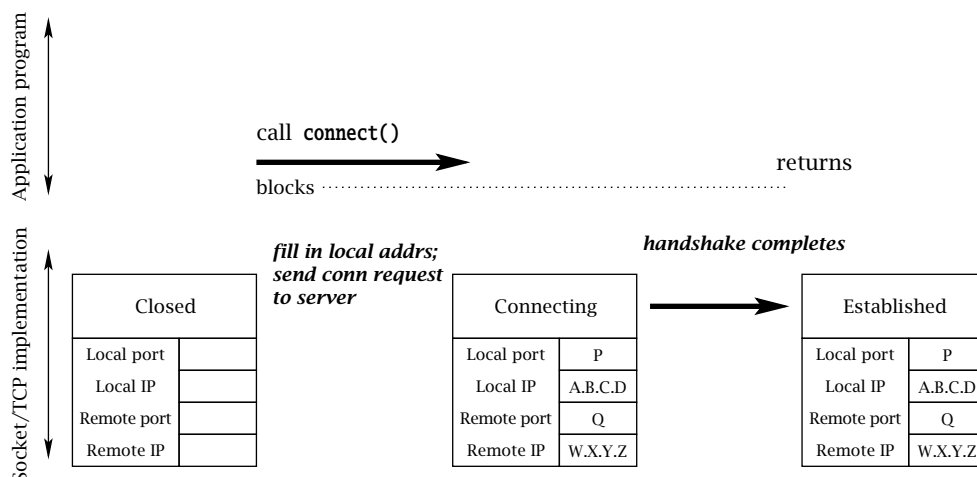
### 7.4.1 Connecting

The relationship between the `connect()` call and the protocol events associated with connection establishment at the client are illustrated in Figure 7.6. In this and the remaining figures of this section, the large arrows depict external events that cause the underlying socket structures to change state. Events that occur in the application program—that is, method calls and returns—are shown in the upper part of the figure; events such as message arrivals are shown in the lower part of the figure. Time proceeds left to right in these figures. The client’s Internet address is depicted as A.B.C.D, while the server’s is W.X.Y.Z; the server’s port number is Q. (We have depicted IPv4 addresses, but everything here applies to both IPv4 and IPv6.)

---

<sup>1</sup>The same thing generally applies to receiving, although calling `recv()` with a larger buffer does not guarantee that more data will be returned—in general, only the data present at the time of a call will be returned.



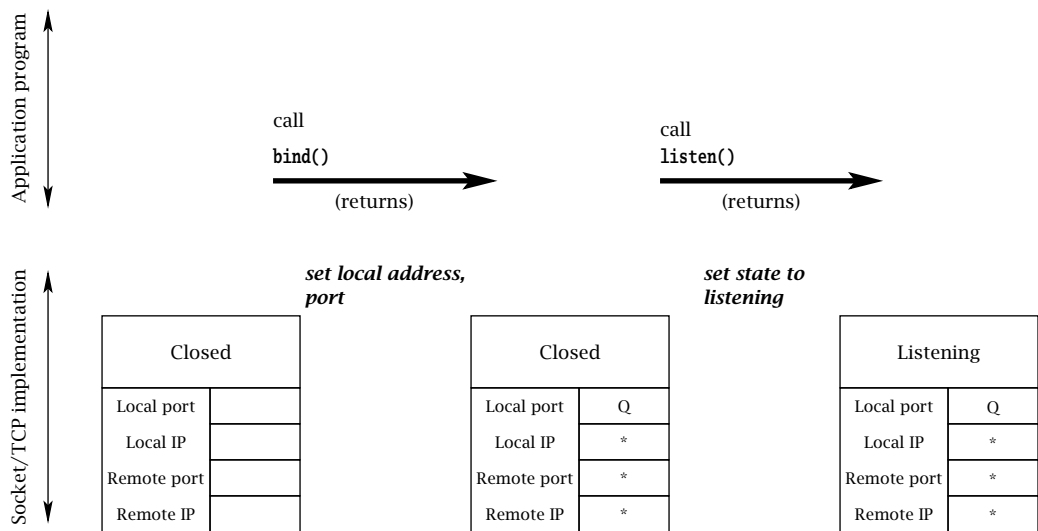


**Figure 7.6:** Client-side connection establishment. (Events proceed left to right in these figures.)

When the client calls `connect()` with the server's Internet address, `W.X.Y.Z`, and port, `Q`, the underlying implementation creates a socket instance; it is initially in the Closed state. If the client did not specify the local address/port with `bind()`, a local port number (`P`), not already in use by another TCP socket, is chosen by the implementation. The local Internet address is also assigned; if not explicitly specified, the address of the network interface through which packets will be sent to the server is used. The implementation copies the local and remote addresses and ports into the underlying socket structure, and initiates the TCP connection establishment handshake.

The TCP opening handshake is known as a *three-way handshake* because it typically involves three messages: a connection request from client to server, an acknowledgment from server to client, and another acknowledgment from client back to server. The client TCP considers the connection to be established as soon as it receives the acknowledgment from the server. In the normal case, this happens quickly. However, the Internet is a best-effort network, and either the client's initial message or the server's response can get lost. For this reason, the TCP implementation retransmits handshake messages multiple times, at increasing intervals. If the client TCP does not receive a response from the server after some time, it *times out* and gives up. In this case `connect()` returns `-1` and sets `errno` to `ETIMEDOUT`. The implementation tries very hard to complete the connection before giving up, and thus it can take on the order of minutes for a `connect()` call to fail. After the initial handshake message is sent and before the reply from the server is received (i.e., the middle part of Figure 7.6), the output from `netstat` on the client host would look something like:

```
Active Internet connections
Proto Recv-Q Send-Q Local Address   Foreign Address  State
tcp        0      0 A.B.C.D:P       W.X.Y.Z:Q        SYN_SENT
```



**Figure 7.7:** Server-side socket setup.

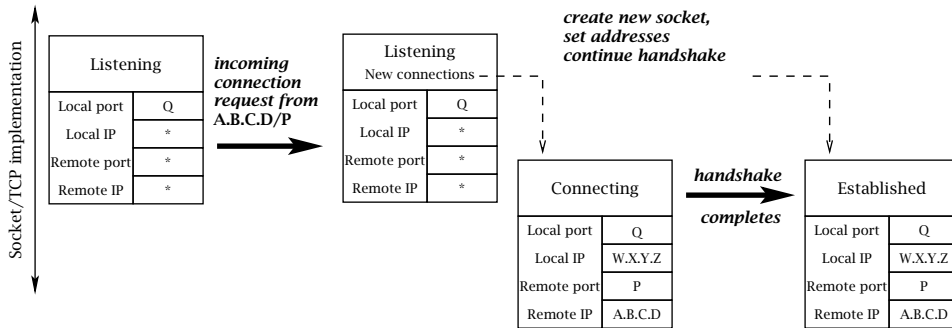
where SYN\_SENT is the technical name of the client’s state between the first and second messages of the handshake.

If the server is not accepting connections—say, if there is no program associated with the given port at the destination—the server-side TCP will respond (immediately) with a rejection message instead of an acknowledgment, and connect() returns -1 with *errno* set to ECONNREFUSED. Otherwise, after the client receives a positive reply from the server, the netstat output would look like:

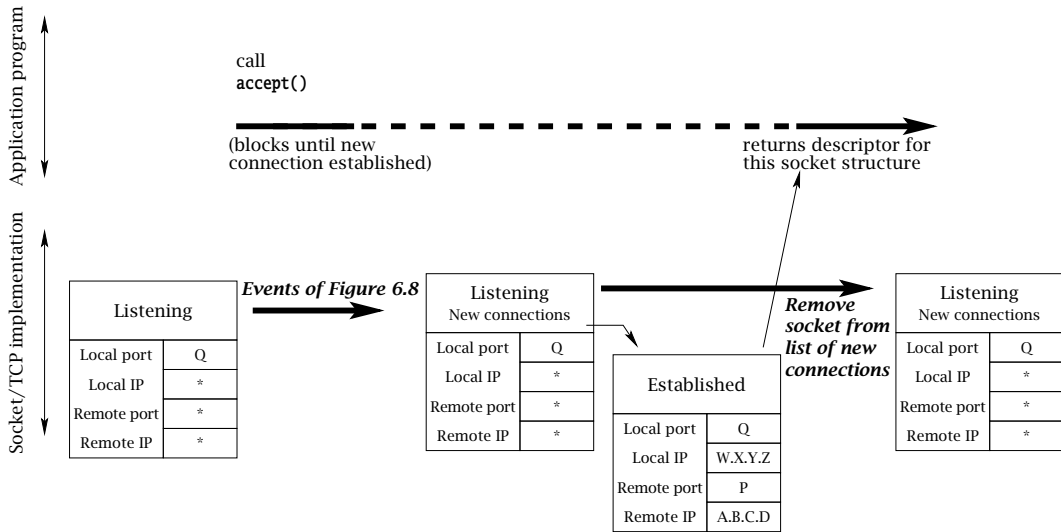
```
Active Internet connections
Proto Recv-Q Send-Q Local Address   Foreign Address  State
tcp      0      0 A.B.C.D:P       W.X.Y.Z:Q        ESTABLISHED
```

The sequence of events at the server side is rather different; we describe it in Figures 7.7, 7.8, and 7.9. The server needs to bind to the particular TCP port known to the client. Typically, the server specifies only the port number (here, Q) in the bind() call and gives the special wildcard address INADDR\_ANY for the local IP address. In case the server host has more than one IP address, this technique allows the socket to receive connections addressed to any of its IP addresses. When the server calls listen(), the state of the socket is changed to “Listening”, indicating that it is ready to accept new connections. These events are depicted in Figure 7.7. The output from netstat on the server after this sequence would include a line like:

```
Active Internet connections
Proto Recv-Q Send-Q Local Address   Foreign Address  State
tcp      0      0 0.0.0.0:Q       0.0.0.0:0        LISTENING
```



**Figure 7.8:** Incoming connection request processing.



**Figure 7.9:** `accept()`.

Note that any client connection request that arrives at the server before the call to `listen()` will be rejected, even if it arrives after the call to `bind()`.

The next thing the server does is call `accept()`, which blocks until a connection with a client is established. Therefore in Figure 7.8 we focus on the events that occur in the TCP implementation when a client connection request arrives. Note that everything depicted in this figure happens “under the covers” in the TCP implementation.

When the request for a connection arrives from the client, a new socket structure is created for the connection. The new socket’s addresses are filled in based on the arriving packet.

The packet's destination Internet address and port (W.X.Y.Z and Q, respectively) become the socket's local address and port; the packet's source address and port (A.B.C.D and P) become the socket's remote Internet address and port. Note that the local port number of the new socket is always the same as that of the listening socket. The new socket's state is set to Connecting (actually called SYN\_RCVD in the implementation), and it is added to a list of not-quite-connected sockets associated with the original server socket. Note well that the original server socket does not change state. At this point the output of netstat should show *both* the original, listening socket and the newly created one:

```
Active Internet connections
Proto Recv-Q Send-Q Local Address   Foreign Address  State
tcp      0      0 0.0.0.0:Q       0.0.0.0:0        LISTENING
tcp      0      0 W.X.Y.Z:Q       A.B.C.D:P        SYN_RCVD
```

In addition to creating a new underlying socket structure, the server-side TCP implementation sends an acknowledging TCP handshake message back to the client. However, the server TCP does not consider the handshake complete until the third message of the three-way handshake is received from the client. When that message eventually arrives, the new structure's state is set to "Established", and it is then (and only then) moved to a list of socket structures associated with the original socket, which represent established connections ready to be accepted. (If the third handshake message fails to arrive, eventually the "Connecting" structure is deleted.) Output from netstat would then include:

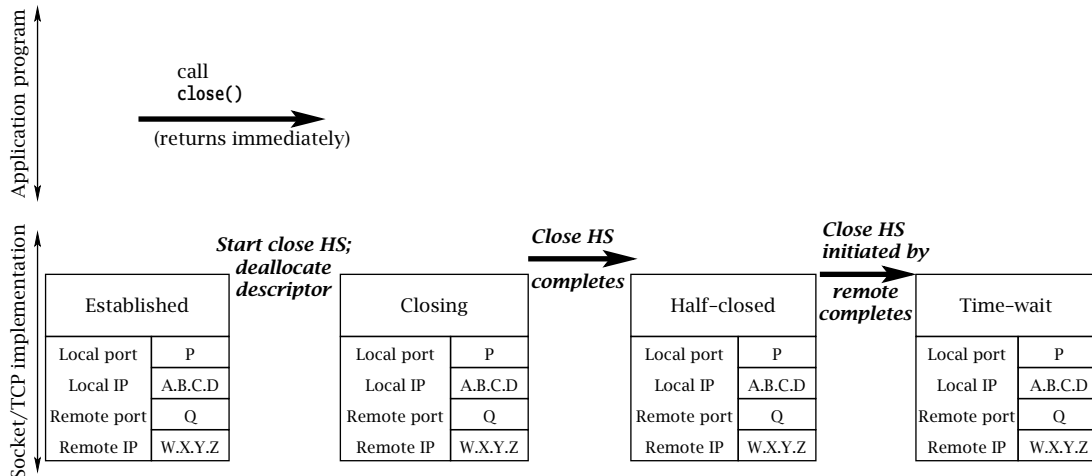
```
Active Internet connections
Proto Recv-Q Send-Q Local Address   Foreign Address  State
tcp      0      0 0.0.0.0:Q       0.0.0.0:0        LISTENING
tcp      0      0 W.X.Y.Z:Q       A.B.C.D:P        ESTABLISHED
```

Now we can consider (in Figure 7.9) what happens when the server program calls `accept()`. The call unblocks as soon as there is something in the listening socket's list of new connections. (Note that this list may already be nonempty when `accept()` is called.) At that time, the new socket structure is removed from the list, and a socket descriptor is allocated and returned as the result of `accept()`.

It is important to note that each structure in the server socket's associated list represents a fully established TCP connection with a client at the other end. Indeed, the client can send data as soon as it receives the second message of the opening handshake—which may be long before the server accepts the client connection!

## 7.4.2 Closing a TCP Connection

TCP has a *graceful close* mechanism that allows applications to terminate a connection without having to worry about loss of data that might still be in transit. The mechanism is also designed to allow data transfers in each direction to be terminated independently. It works like this: the application indicates that it is finished sending data on a connected socket by calling `close()` or `shutdown()`. At that point, the underlying TCP implementation first transmits



**Figure 7.10:** Closing a TCP connection first.

any data remaining in *SendQ* (subject to available space in *RecvQ* at the other end), and then sends a closing TCP handshake message to the other end. This closing handshake message can be thought of as an end-of-stream marker: it tells the receiving TCP that no more bytes will be placed in *RecvQ*. (Note that the closing handshake message itself is *not* passed to the receiving application, but that its position in the byte stream is indicated by *recv()* returning 0.) The closing TCP waits for an acknowledgment of its closing handshake message, which indicates that all data sent on the connection made it safely to *RecvQ*. Once that acknowledgment is received, the connection is “Half closed”. The connection is not *completely* closed until a symmetric handshake happens in the other direction—that is, until *both* ends have indicated that they have no more data to send.

The closing event sequence in TCP can happen in two ways: either one application calls *close()* (or *shutdown()*) and completes its closing handshake before the other does, or both close simultaneously, so that their closing handshake messages cross in the network. Figure 7.10 shows the sequence of events in the implementation when the application invokes *close()* *before* the other end closes. The closing handshake (HS) message is sent, the state of the socket structure is set to “Closing”, and the call returns. After this point, further attempts to perform any operation on the socket result in error returns. When the acknowledgment for the close handshake is received, the state changes to “Half closed”, where it remains until the other end’s close handshake message is received. At this point, the output of *netstat* on the client would show the status of the connection as:

```
Active Internet connections
Proto Recv-Q Send-Q Local Address   Foreign Address  State
tcp      0      0 A.B.C.D:P       W.X.Y.Z:Q        FIN_WAIT_2
```

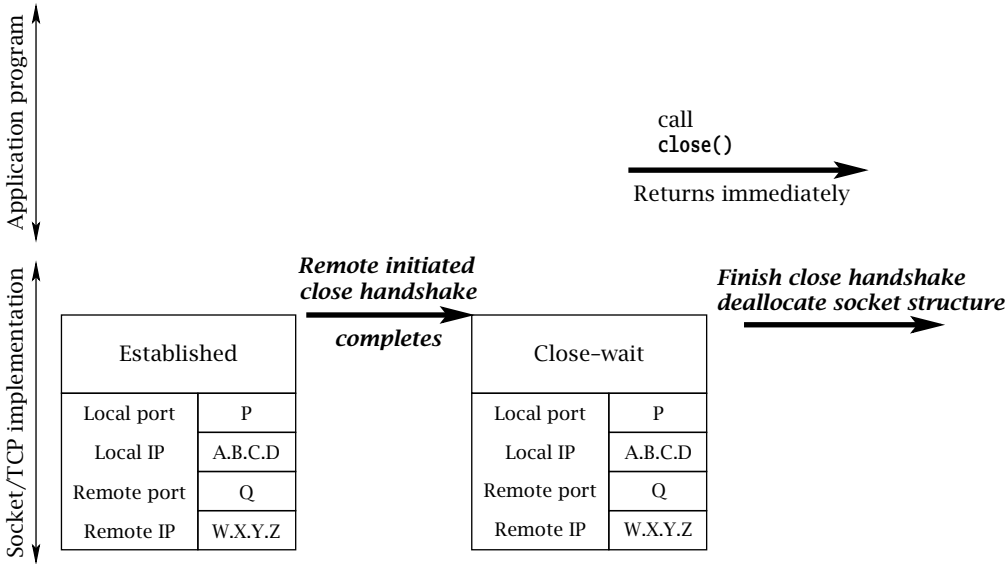
(FIN\_WAIT\_2 is the technical name for the “half-closed” state at the host that initiates close first. The state denoted by “closing” in the figure is technically called FIN\_WAIT\_1, but it is transient and is difficult to catch with netstat.) Note that if the remote endpoint goes away while the connection is in this state, the local underlying structure will stay around indefinitely. Otherwise, when the other end’s close handshake message arrives, an acknowledgment is sent and the state is changed to “Time-Wait”. Although the descriptor in the application program may have long since been reclaimed (and even reused), the associated underlying socket structure continues to exist in the implementation for a minute or more; the reasons for this are discussed at the end of this section.

The output of netstat at the right end of Figure 7.10 includes:

```
Active Internet connections
Proto Recv-Q Send-Q Local Address   Foreign Address  State
tcp      0      0 A.B.C.D:P      W.X.Y.Z:Q      TIME_WAIT
```

Figure 7.11 shows the simpler sequence of events at the endpoint that does not close first. When the closing handshake message arrives, an acknowledgment is sent immediately, and the connection state becomes “Close-Wait.” The output of netstat on this host shows:

```
Active Internet connections
Proto Recv-Q Send-Q Local Address   Foreign Address  State
tcp      0      0 W.X.Y.Z:Q      A.B.C.D:P      CLOSE_WAIT
```




**Figure 7.11:** Closing after the other end closes.

At this point, it's all over: the implementation is just waiting for the application to call `close()`. When it does, the socket descriptor is deallocated and the final close handshake is initiated. When it completes, the underlying socket structure is deallocated.

Although most applications use `close()`, `shutdown()` actually provides more flexibility. A call to `close()` terminates *both* directions of transfer and causes the file descriptor associated with the socket to be deallocated. Any undelivered data remaining in *RecvQ* is discarded, and the flow control mechanism prevents any further transfer of data from the other end's *SendQ*. All trace of the socket disappears from the calling program. Underneath, however, the associated socket structure continues to exist until the other end initiates its closing handshake. In contrast, `shutdown()` allows the sending and receiving streams to be terminated independently. It takes an additional argument, one of `SHUT_RD`, `SHUT_WR`, or `SHUT_RDWR`, indicating which stream(s) are to be shut down. A program calling `shutdown()` with second argument `SHUT_WR` can continue to receive data on the socket; only sending is prohibited. The fact that the other end of the connection has closed is indicated by `recv()` returning 0 (once *RecvQ* is empty, of course) to indicate that there will be no more data available on the connection.

In view of the fact that both `close()` and `shutdown()` return without waiting for the closing handshake to complete, you may wonder how the sender can be assured that sent data has actually made it to the receiving program (i.e., to *Delivered*). In fact, it is possible for an application to call `close()` or `shutdown()` and have it complete successfully (i.e., not return `-1`) *while there is still data in SendQ*. If either end of the connection then crashes before the data makes it to *RecvQ*, data may be lost without the sending application knowing about it!

**The best solution is to design the application protocol so that whichever side closes first, does so *only after* receiving application-level assurance that its data was received.** For example, when our `TCPEchoClient.c` program receives the same number of bytes as it sent, there should be nothing more in transit in either direction, so it is safe for it to close the connection. Note that there is no *guarantee* that the bytes received were those sent; the client is assuming that the server implements the echo protocol. In a real application the client should certainly *not* trust the server to “do the right thing.” 

The other solution is to modify the semantics of `close()` by setting the `SO_LINGER` socket option before calling it. The `SO_LINGER` option specifies an amount of time for the TCP implementation to wait for the closing handshake to complete. The setting of `SO_LINGER` and the specification of the wait time are given to `setsockopt()` using the **linger** structure:

```
struct linger {
    int  l_onoff;   // Nonzero to linger
    int  l_linger;  // Time (secs.) to linger
};
```

To use the linger behavior, set `l_onoff` to a nonzero value and specify the time to linger in `l_linger`. When `SO_LINGER` is set, `close()` blocks *until the closing handshake is completed* or until the specified amount of time passes. If the handshake does not complete in time, an error indication (`ETIMEDOUT`) is returned. Thus, if `SO_LINGER` is set and `close()` returns no error, the application is assured that everything it sent reached *RecvQ*.

The final subtlety of closing a TCP connection revolves around the need for the Time-Wait state. The TCP specification requires that when a connection terminates, at least one of the sockets persists in the Time-Wait state for a period of time after both closing handshakes complete. This requirement is motivated by the possibility of messages being delayed in the network. If both ends' underlying structures go away as soon as both closing handshakes complete, and a *new* connection is immediately established between the same pair of socket addresses, a message from the previous connection, which happened to be delayed in the network, could arrive just after the new connection is established. Because it would contain the same source and destination addresses, the old message could be mistaken for a message belonging to the new connection, and its data might (incorrectly) be delivered to the application.

Unlikely though this scenario may be, TCP employs multiple mechanisms to prevent it, including the Time-Wait state. The Time-Wait state ensures that every TCP connection ends with a quiet time, during which no data is sent. The quiet time is supposed to be equal to twice the maximum amount of time a packet can remain in the network. Thus, by the time a connection goes away completely (i.e., the socket structure leaves the Time-Wait state and is deallocated) and clears the way for a new connection between the same pair of addresses, no messages from the old instance can still be in the network. In practice, the length of the quiet time is implementation dependent, because there is no real mechanism that limits how long a packet can be delayed by the network. Values in use range from 4 minutes down to 30 seconds or even shorter.

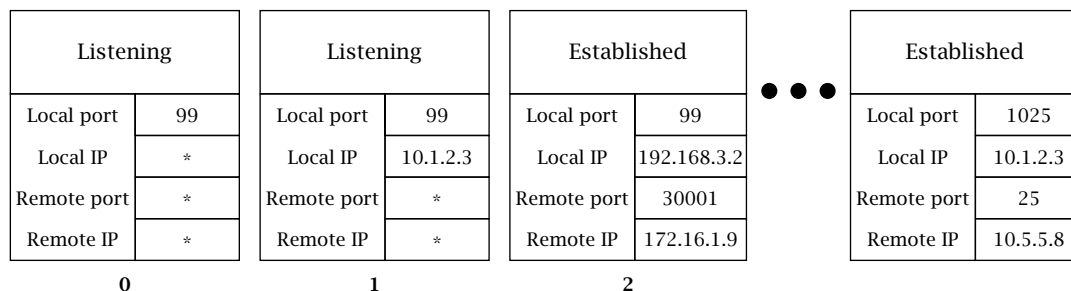
The most important consequence of Time-Wait is that as long as the underlying socket structure exists, no other socket is permitted to bind to the same local port. (More on this below.)

## 7.5 Demultiplexing Demystified

The fact that different sockets on the same machine can have the same local address and port number is implicit in the preceding discussions. For example, on a machine with only one IP address, every new socket accept()ed via a listening socket will have the same local address and port number as the listening socket. Clearly, the process of deciding to which socket an incoming packet should be delivered—that is, the *demultiplexing* process—involves looking at more than just the packet's destination address and port. Otherwise there could be ambiguity about which socket an incoming packet is intended for. The process of matching an incoming packet to a socket is actually the same for both TCP and UDP, and can be summarized by the following points:

- The local port in the socket structure *must* match the destination port number in the incoming packet.
- Any address fields in the socket structure that contain the wildcard value (\*) are considered to match *any* value in the corresponding field in the packet.





**Figure 7.12:** Demultiplexing with multiple matching sockets.

- If more than one socket structure matches an incoming packet for all four address fields, the one that matches using the fewest wildcards gets the packet.

For example, consider a host with two IP addresses, 10.1.2.3 and 192.168.3.2, and with a subset of its active TCP socket structures, as shown in Figure 7.12. The structure labeled 0 is associated with a listening socket and has port 99 with a wildcard local address. Socket structure 1 is also for a listening socket on the same port, but with the local IP address 10.1.2.3 specified (so it will only accept connection requests to that address). Structure 2 is for a connection that was accepted via structure 0's listening socket, and thus has the same local port number (99), but also has its local and remote Internet addresses filled in. Other sockets belong to other active connections. Now consider a packet with source IP address 172.16.1.10, source port 56789, destination IP address 10.1.2.3, and destination port 99. It will be delivered to the socket associated with structure 1, because that one matches with the fewest wildcards.

When a program attempts to `bind()` to a particular local port number, the existing sockets are checked to make sure that no socket is already using that local port. The call to `bind()` will fail and set `EADDRINUSE` if *any* socket matches the local port and local IP address (if any) specified in the argument to `bind()`. This can cause problems in the following scenario:

1. A server's listening socket is bound to some particular port *P*.
2. The server accepts a connection from a client, which enters the Established state.
3. The server terminates for some reason—say, because the programmer has created a new version and wants to test it. When the server program exits, the underlying system automatically (and virtually) calls `close()` on all of its existing sockets. The socket that was in the Established state immediately transitions to the Time-Wait state.
4. The programmer starts up a new instance of the server, which attempts to `bind()` to port *P*.

Unfortunately the new server's call to `bind()` will fail with `EADDRINUSE` because of the old socket in Time-Wait state.

As of this writing, there are two ways around this. One is to wait until the underlying structure leaves the Time-Wait state. The other is for the server to set the `SO_REUSEADDR` socket option *before* calling `bind()`. That lets `bind()` succeed in spite of the existence of any sockets representing earlier connections to the server's port. There is no danger of ambiguity, because the existing connections (whether still in the Established or Time-Wait state) have remote addresses filled in, while the socket being bound does not. In general, the `SO_REUSEADDR` option also enables a socket to bind to a local port to which another socket is already bound, provided that the IP address to which it is being bound (typically the wildcard `INADDR_ANY` address) is different from that of the existing socket. The default `bind()` behavior is to disallow such requests.

## Exercises

1. The TCP protocol is designed so that simultaneous connection attempts will succeed. That is, if an application using port P and Internet address W.X.Y.Z attempts to connect to address A.B.C.D, port Q, at the same time as an application using the same address and port tries to connect to W.X.Y.Z, port P, they will end up connected to each other. Can this be made to happen when the programs use the sockets API?
2. The first example of “buffer deadlock” in this chapter involves the programs on both ends of a connection trying to send large messages. However, this is not necessary for deadlock. How could the `TCPEchoClient` from earlier chapters be made to deadlock when it connects to the `TCPEchoServer` from that chapter?