4

Elementary TCP Sockets

4.1 Introduction

This chapter describes the elementary socket functions required to write a complete TCP client and server. We will first describe all the elementary socket functions that we will be using and then develop the client and server in the next chapter. We will work with this client and server throughout the text, enhancing it many times (Figures 1.12 and 1.13).

We will also describe concurrent servers, a common Unix technique for providing concurrency when numerous clients are connected to the same server at the same time. Each client connection causes the server to `fork` a new process just for that client. In this chapter, we consider only the one-*process*-per-client model using `fork`, but we will consider a different one-*thread*-per-client model when we describe threads in Chapter 26.

Figure 4.1 shows a timeline of the typical scenario that takes place between a TCP client and server. First, the server is started, then sometime later, a client is started that connects to the server. We assume that the client sends a request to the server, the server processes the request, and the server sends a reply back to the client. This continues until the client closes its end of the connection, which sends an end-of-file notification to the server. The server then closes its end of the connection and either terminates or waits for a new client connection.

4.2 `socket` Function

To perform network I/O, the first thing a process must do is call the `socket` function, specifying the type of communication protocol desired (TCP using IPv4, UDP using IPv6, Unix domain stream protocol, etc.).

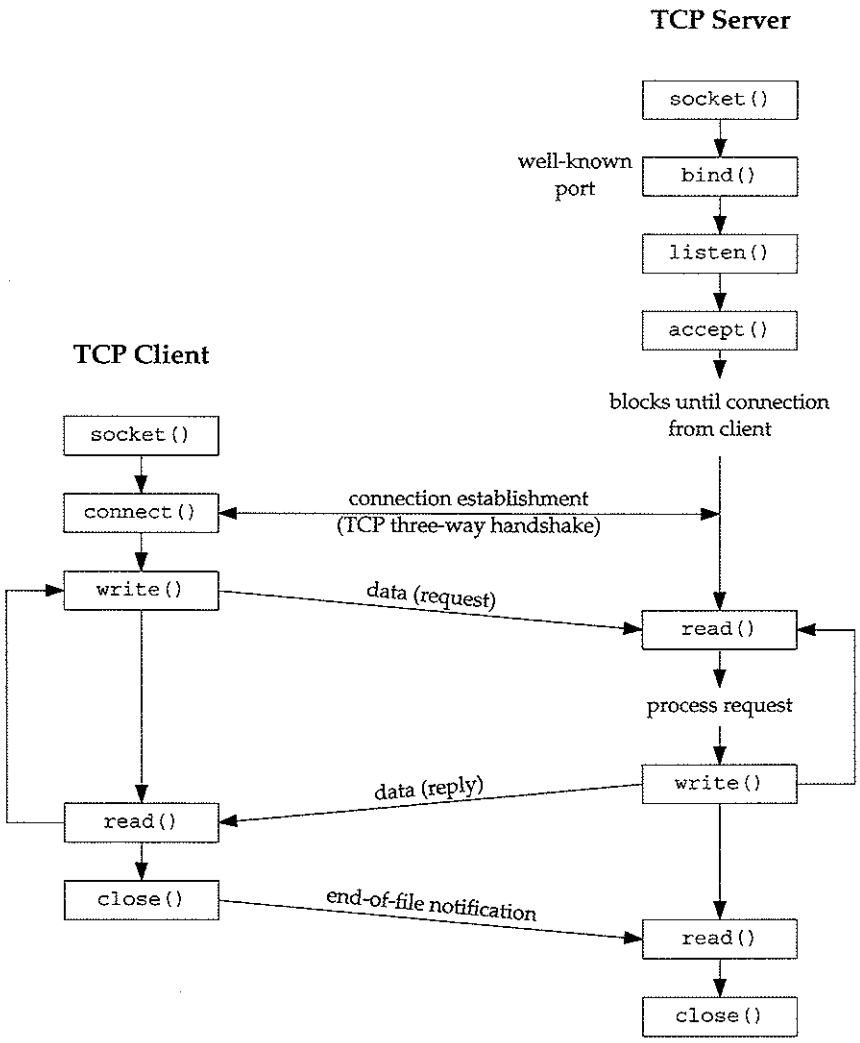

Figure 4.1 Socket functions for elementary TCP client/server.

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

int socket(int family, int type, int protocol);
```

Returns: non-negative descriptor if OK, –1 on error

family specifies the protocol family and is one of the constants shown in Figure 4.2. This argument is often referred to as *domain* instead of *family*. The socket *type* is one of the

constants shown in Figure 4.3. The *protocol* argument to the socket function should be set to the specific protocol type found in Figure 4.4, or 0 to select the system's default for the given combination of *family* and *type*.

Not all combinations of socket *family* and *type* are valid. Figure 4.5 shows the valid combinations, along with the actual protocols that are valid for each pair. The boxes marked "Yes" are valid but do not have handy acronyms. The blank boxes are not supported.

family	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols (Chapter 15)
AF_ROUTE	Routing sockets (Chapter 18)
AF_KEY	Key socket (Chapter 19)

Figure 4.2 Protocol *family* constants for socket function.

type	Description
SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_SEQPACKET	sequenced packet socket
SOCK_RAW	raw socket

Figure 4.3 *type* of socket for socket function.

Protocol	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

Figure 4.4 *protocol* of sockets for AF_INET or AF_INET6.

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP | SCTP	TCP | SCTP	Yes		
SOCK_DGRAM	UDP	UDP	Yes		
SOCK_SEQPACKET	SCTP	SCTP	Yes		
SOCK_RAW	IPv4	IPv6		Yes	Yes

Figure 4.5 Combinations of *family* and *type* for the socket function.

You may also encounter the corresponding PF_*xxx* constant as the first argument to socket. We will say more about this at the end of this section.

We note that you may encounter AF_UNIX (the historical Unix name) instead of AF_LOCAL (the POSIX name), and we will say more about this in Chapter 15.

There are other values for the *family* and *type* arguments. For example, 4.4BSD supports both AF_NS (the Xerox NS protocols, often called XNS) and AF_ISO (the OSI protocols). Similarly, the *type* of SOCK_SEQPACKET, a sequenced-packet socket, is implemented by both the Xerox NS protocols and the OSI protocols, and we will describe its use with SCTP in Section 9.2. But, TCP is a byte stream protocol, and supports only SOCK_STREAM sockets.

Linux supports a new socket type, SOCK_PACKET, that provides access to the datalink, similar to BPF and DLPI in Figure 2.1. We will say more about this in Chapter 29.

The key socket, AF_KEY, is newer than the others. It provides support for cryptographic security. Similar to the way that a routing socket (AF_ROUTE) is an interface to the kernel's routing table, the key socket is an interface into the kernel's key table. See Chapter 19 for details.

On success, the socket function returns a small non-negative integer value, similar to a file descriptor. We call this a *socket descriptor*, or a *sockfd*. To obtain this socket descriptor, all we have specified is a protocol family (IPv4, IPv6, or Unix) and the socket type (stream, datagram, or raw). We have not yet specified either the local protocol address or the foreign protocol address.

AF_*xxx* versus PF_*xxx*

The "AF_" prefix stands for "address family" and the "PF_" prefix stands for "protocol family." Historically, the intent was that a single protocol family might support multiple address families and that the PF_ value was used to create the socket and the AF_ value was used in socket address structures. But in actuality, a protocol family supporting multiple address families has never been supported and the <sys/socket.h> header defines the PF_ value for a given protocol to be equal to the AF_ value for that protocol. While there is no guarantee that this equality between the two will always be true, should anyone change this for existing protocols, lots of existing code would break. To conform to existing coding practice, we use only the AF_ constants in this text, although you may encounter the PF_ value, mainly in calls to socket.

Looking at 137 programs that call socket in the BSD/OS 2.1 release shows 143 calls that specify the AF_ value and only 8 that specify the PF_ value.

Historically, the reason for the similar sets of constants with the AF_ and PF_ prefixes goes back to 4.1cBSD [Lanciani 1996] and a version of the socket function that predates the one we are describing (which appeared with 4.2BSD). The 4.1cBSD version of socket took four arguments, one of which was a pointer to a sockproto structure. The first member of this structure was named sp_family and its value was one of the PF_ values. The second member, sp_protocol, was a protocol number, similar to the third argument to socket today. Specifying this structure was the only way to specify the protocol family. Therefore, in this early system, the PF_ values were used as structure tags to specify the protocol family in the sockproto structure, and the AF_ values were used as structure tags to specify the address family in the socket address structures. The sockproto structure is still in 4.4BSD (pp. 626–627 of TCPv2), but is only used internally by the kernel. The original definition had the comment "protocol family" for the sp_family member, but this has been changed to "address family" in the 4.4BSD source code.

To confuse this difference between the AF_ and PF_ constants even more, the Berkeley kernel data structure that contains the value that is compared to the first argument to socket (the dom_family member of the domain structure, p. 187 of TCPv2) has the comment that it contains an AF_ value. But, some of the domain structures within the kernel are initialized to the corresponding AF_ value (p. 192 of TCPv2) while others are initialized to the PF_ value (p. 646 of TCPv2 and p. 229 of TCPv3).

As another historical note, the 4.2BSD man page for socket, dated July 1983, calls its first argument *af* and lists the possible values as the AF_ constants.

Finally, we note that the POSIX standard specifies that the first argument to socket be a PF_ value, and the AF_ value be used for a socket address structure. But, it then defines only one family value in the addrinfo structure (Section 11.6), intended for use in either a call to socket or in a socket address structure!

4.3 connect Function

The connect function is used by a TCP client to establish a connection with a TCP server.

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
                                                    Returns: 0 if OK, –1 on error
```

sockfd is a socket descriptor returned by the socket function. The second and third arguments are a pointer to a socket address structure and its size, as described in Section 3.3. The socket address structure must contain the IP address and port number of the server. We saw an example of this function in Figure 1.5.

The client does not have to call bind (which we will describe in the next section) before calling connect: the kernel will choose both an ephemeral port and the source IP address if necessary.

In the case of a TCP socket, the connect function initiates TCP's three-way handshake (Section 2.6). The function returns only when the connection is established or an error occurs. There are several different error returns possible.

1. If the client TCP receives no response to its SYN segment, ETIMEDOUT is returned. 4.4BSD, for example, sends one SYN when connect is called, another 6 seconds later, and another 24 seconds later (p. 828 of TCPv2). If no response is received after a total of 75 seconds, the error is returned.

   Some systems provide administrative control over this timeout; see Appendix E of TCPv1.

2. If the server's response to the client's SYN is a reset (RST), this indicates that no process is waiting for connections on the server host at the port specified (i.e., the server process is probably not running). This is a *hard error* and the error ECONNREFUSED is returned to the client as soon as the RST is received.

An RST is a type of TCP segment that is sent by TCP when something is wrong. Three conditions that generate an RST are: when a SYN arrives for a port that has no listening server (what we just described), when TCP wants to abort an existing connection, and when TCP receives a segment for a connection that does not exist. (TCPv1 [pp. 246–250] contains additional information.)

3. If the client's SYN elicits an ICMP "destination unreachable" from some intermediate router, this is considered a *soft error*. The client kernel saves the message but keeps sending SYNs with the same time between each SYN as in the first scenario. If no response is received after some fixed amount of time (75 seconds for 4.4BSD), the saved ICMP error is returned to the process as either `EHOSTUNREACH` or `ENETUNREACH`. It is also possible that the remote system is not reachable by any route in the local system's forwarding table, or that the `connect` call returns without waiting at all.

   Many earlier systems, such as 4.2BSD, incorrectly aborted the connection establishment attempt when the ICMP "destination unreachable" was received. This is wrong because this ICMP error can indicate a transient condition. For example, it could be that the condition is caused by a routing problem that will be corrected.

   Notice that `ENETUNREACH` is not listed in Figure A.15, even when the error indicates that the destination network is unreachable. Network unreachables are considered obsolete, and applications should just treat `ENETUNREACH` and `EHOSTUNREACH` as the same error.

We can see these different error conditions with our simple client from Figure 1.5. We first specify the local host (127.0.0.1), which is running the daytime server, and see the output.

```
solaris % daytimetcpcli 127.0.0.1
Sun Jul 27 22:01:51 2003
```

To see a different format for the returned reply, we specify a different machine's IP address (in this example, the IP address of the HP-UX machine).

```
solaris % daytimetcpcli 192.6.38.100
Sun Jul 27 22:04:59 PDT 2003
```

Next, we specify an IP address that is on the local subnet (192.168.1/24) but the host ID (100) is nonexistent. That is, there is no host on the subnet with a host ID of 100, so when the client host sends out ARP requests (asking for that host to respond with its hardware address), it will never receive an ARP reply.

```
solaris % daytimetcpcli 192.168.1.100
connect error: Connection timed out
```

We only get the error after the `connect` times out (around four minutes with Solaris 9). Notice that our `err_sys` function prints the human-readable string associated with the `ETIMEDOUT` error.

Our next example is to specify a host (a local router) that is not running a daytime server.

```
solaris % daytimetcpcli 192.168.1.5
connect error: Connection refused
```

The server responds immediately with an RST.

Our final example specifies an IP address that is not reachable on the Internet. If we watch the packets with `tcpdump`, we see that a router six hops away returns an ICMP host unreachable error.

```
solaris % daytimetcpcli 192.3.4.5
connect error: No route to host
```

As with the `ETIMEDOUT` error, in this example, `connect` returns the `EHOSTUNREACH` error only after waiting its specified amount of time.

In terms of the TCP state transition diagram (Figure 2.4), `connect` moves from the CLOSED state (the state in which a socket begins when it is created by the `socket` function) to the SYN_SENT state, and then, on success, to the ESTABLISHED state. If `connect` fails, the socket is no longer usable and must be closed. We cannot call `connect` again on the socket. In Figure 11.10, we will see that when we call `connect` in a loop, trying each IP address for a given host until one works, each time `connect` fails, we must `close` the socket descriptor and call `socket` again.

4.4 `bind` Function

The `bind` function assigns a local protocol address to a socket. With the Internet protocols, the protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number.

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

int bind(int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
```

Returns: 0 if OK, –1 on error

Historically, the man page description of `bind` has said "`bind` assigns a name to an unnamed socket." The use of the term "name" is confusing and gives the connotation of domain names (Chapter 11) such as `foo.bar.com`. The `bind` function has nothing to do with names. `bind` assigns a protocol address to a socket, and what that protocol address means depends on the protocol.

The second argument is a pointer to a protocol-specific address, and the third argument is the size of this address structure. With TCP, calling `bind` lets us specify a port number, an IP address, both, or neither.

- Servers bind their well-known port when they start. We saw this in Figure 1.9. If a TCP client or server does not do this, the kernel chooses an ephemeral port for the socket when either `connect` or `listen` is called. It is normal for a TCP client to let the kernel choose an ephemeral port, unless the application requires a reserved port (Figure 2.10), but it is rare for a TCP server to let the kernel choose an ephemeral port, since servers are known by their well-known port.

Exceptions to this rule are Remote Procedure Call (RPC) servers. They normally let the kernel choose an ephemeral port for their listening socket since this port is then registered with the RPC port mapper. Clients have to contact the port mapper to obtain the ephemeral port before they can `connect` to the server. This also applies to RPC servers using UDP.

- A process can `bind` a specific IP address to its socket. The IP address must belong to an interface on the host. For a TCP client, this assigns the source IP address that will be used for IP datagrams sent on the socket. For a TCP server, this restricts the socket to receive incoming client connections destined only to that IP address.

  Normally, a TCP client does not `bind` an IP address to its socket. The kernel chooses the source IP address when the socket is connected, based on the outgoing interface that is used, which in turn is based on the route required to reach the server (p. 737 of TCPv2).

  If a TCP server does not bind an IP address to its socket, the kernel uses the destination IP address of the client's SYN as the server's source IP address (p. 943 of TCPv2).

As we said, calling `bind` lets us specify the IP address, the port, both, or neither. Figure 4.6 summarizes the values to which we set `sin_addr` and `sin_port`, or `sin6_addr` and `sin6_port`, depending on the desired result.

Process specifies		Result
IP address	port	
Wildcard	0	Kernel chooses IP address and port
Wildcard	nonzero	Kernel chooses IP address, process specifies port
Local IP address	0	Process specifies IP address, kernel chooses port
Local IP address	nonzero	Process specifies IP address and port

Figure 4.6 Result when specifying IP address and/or port number to `bind`.

If we specify a port number of 0, the kernel chooses an ephemeral port when `bind` is called. But if we specify a wildcard IP address, the kernel does not choose the local IP address until either the socket is connected (TCP) or a datagram is sent on the socket (UDP).

With IPv4, the *wildcard* address is specified by the constant `INADDR_ANY`, whose value is normally 0. This tells the kernel to choose the IP address. We saw the use of this in Figure 1.9 with the assignment

```
struct sockaddr_in   servaddr;

servaddr.sin_addr.s_addr = htonl(INADDR_ANY);    /* wildcard */
```

While this works with IPv4, where an IP address is a 32-bit value that can be represented as a simple numeric constant (0 in this case), we cannot use this technique with IPv6, since the 128-bit IPv6 address is stored in a structure. (In C we cannot represent a

constant structure on the right-hand side of an assignment.) To solve this problem, we write

```
struct sockaddr_in6    serv;
serv.sin6_addr = in6addr_any;    /* wildcard */
```

The system allocates and initializes the `in6addr_any` variable to the constant `IN6ADDR_ANY_INIT`. The `<netinet/in.h>` header contains the `extern` declaration for `in6addr_any`.

The value of `INADDR_ANY` (0) is the same in either network or host byte order, so the use of `htonl` is not really required. But, since all the `INADDR_` constants defined by the `<netinet/in.h>` header are defined in host byte order, we should use `htonl` with any of these constants.

If we tell the kernel to choose an ephemeral port number for our socket, notice that `bind` does not return the chosen value. Indeed, it cannot return this value since the second argument to `bind` has the `const` qualifier. To obtain the value of the ephemeral port assigned by the kernel, we must call `getsockname` to return the protocol address.

A common example of a process binding a non-wildcard IP address to a socket is a host that provides Web servers to multiple organizations (Section 14.2 of TCPv3). First, each organization has its own domain name, such as `www.`*organization*`.com`. Next, each organization's domain name maps into a different IP address, but typically on the same subnet. For example, if the subnet is 198.69.10, the first organization's IP address could be 198.69.10.128, the next 198.69.10.129, and so on. All these IP addresses are then *aliased* onto a single network interface (using the `alias` option of the `ifconfig` command on 4.4BSD, for example) so that the IP layer will accept incoming datagrams destined for any of the aliased addresses. Finally, one copy of the HTTP server is started for each organization and each copy `bind`s only the IP address for that organization.

> An alternative technique is to run a single server that binds the wildcard address. When a connection arrives, the server calls `getsockname` to obtain the destination IP address from the client, which in our discussion above could be 198.69.10.128, 198.69.10.129, and so on. The server then handles the client request based on the IP address to which the connection was issued.
>
> One advantage in binding a non-wildcard IP address is that the demultiplexing of a given destination IP address to a given server process is then done by the kernel.
>
> We must be careful to distinguish between the interface on which a packet arrives versus the destination IP address of that packet. In Section 8.8, we will talk about the weak end system model and the strong end system model. Most implementations employ the former, meaning it is okay for a packet to arrive with a destination IP address that identifies an interface other than the interface on which the packet arrives. (This assumes a multihomed host.) Binding a non-wildcard IP address restricts the datagrams that will be delivered to the socket based only on the destination IP address. It says nothing about the arriving interface, unless the host employs the strong end system model.

A common error from `bind` is `EADDRINUSE` ("Address already in use"). We will say more about this in Section 7.5 when we talk about the `SO_REUSEADDR` and `SO_REUSEPORT` socket options.

4.5 `listen` Function

The `listen` function is called only by a TCP server and it performs two actions:

1. When a socket is created by the `socket` function, it is assumed to be an active socket, that is, a client socket that will issue a `connect`. The `listen` function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket. In terms of the TCP state transition diagram (Figure 2.4), the call to `listen` moves the socket from the CLOSED state to the LISTEN state.
2. The second argument to this function specifies the maximum number of connections the kernel should queue for this socket.

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

int listen(int sockfd, int backlog);
```

Returns: 0 if OK, –1 on error

This function is normally called after both the `socket` and `bind` functions and must be called before calling the `accept` function.

To understand the *backlog* argument, we must realize that for a given listening socket, the kernel maintains two queues:

1. An *incomplete connection queue*, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way handshake. These sockets are in the SYN_RCVD state (Figure 2.4).
2. A *completed connection queue*, which contains an entry for each client with whom the TCP three-way handshake has completed. These sockets are in the ESTABLISHED state (Figure 2.4).

Figure 4.7 depicts these two queues for a given listening socket.

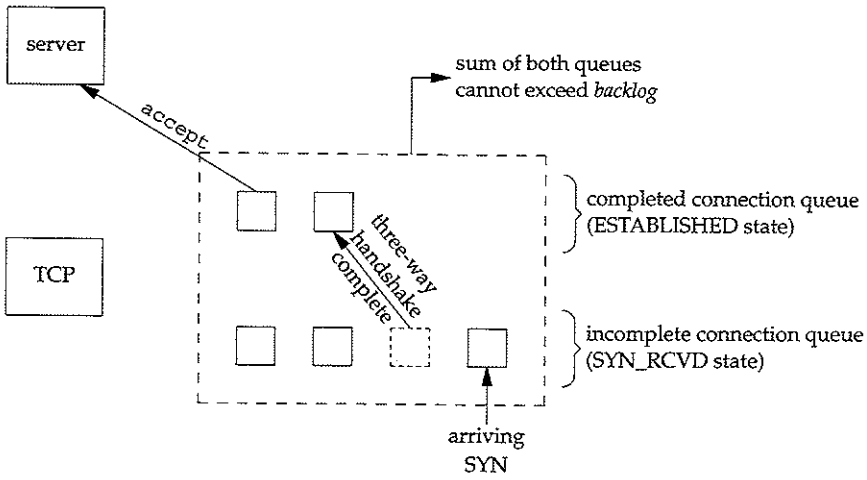

Figure 4.7 The two queues maintained by TCP for a listening socket.

When an entry is created on the incomplete queue, the parameters from the listen socket are copied over to the newly created connection. The connection creation mechanism is completely automatic; the server process is not involved. Figure 4.8 depicts the packets exchanged during the connection establishment with these two queues.

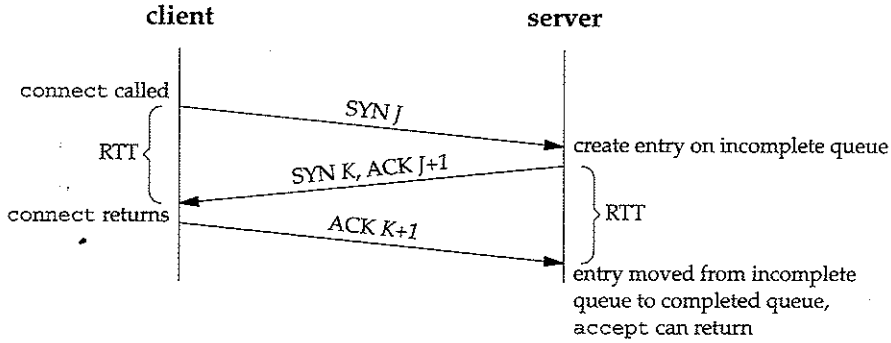

Figure 4.8 TCP three-way handshake and the two queues for a listening socket.

When a SYN arrives from a client, TCP creates a new entry on the incomplete queue and then responds with the second segment of the three-way handshake: the server's SYN with an ACK of the client's SYN (Section 2.6). This entry will remain on the incomplete queue until the third segment of the three-way handshake arrives (the client's ACK of the server's SYN), or until the entry times out. (Berkeley-derived implementations have a timeout of 75 seconds for these incomplete entries.) If the three-way handshake completes normally, the entry moves from the incomplete queue to the end of the completed queue. When the process calls `accept`, which we will describe in the next section, the first entry on the completed queue is returned to the process, or if the

queue is empty, the process is put to sleep until an entry is placed onto the completed queue.

There are several points to consider regarding the handling of these two queues.

- The *backlog* argument to the `listen` function has historically specified the maximum value for the sum of both queues.

  > There has never been a formal definition of what the *backlog* means. The 4.2BSD man page says that it "defines the maximum length the queue of pending connections may grow to." Many man pages and even the POSIX specification copy this definition verbatim, but this definition does not say whether a pending connection is one in the SYN_RCVD state, one in the ESTABLISHED state that has not yet been accepted, or either. The historical definition in this bullet is the Berkeley implementation, dating back to 4.2BSD, and copied by many others.

- Berkeley-derived implementations add a fudge factor to the *backlog*: It is multiplied by 1.5 (p. 257 of TCPv1 and p. 462 of TCPv2). For example, the commonly specified *backlog* of 5 really allows up to 8 queued entries on these systems, as we show in Figure 4.10.

  > The reason for adding this fudge factor appears lost to history [Joy 1994]. But if we consider the *backlog* as specifying the maximum number of completed connections that the kernel will queue for a socket ([Borman 1997b], as discussed shortly), then the reason for the fudge factor is to take into account incomplete connections on the queue.

- Do not specify a *backlog* of 0, as different implementations interpret this differently (Figure 4.10). If you do not want any clients connecting to your listening socket, close the listening socket.
- Assuming the three-way handshake completes normally (i.e., no lost segments and no retransmissions), an entry remains on the incomplete connection queue for one RTT, whatever that value happens to be between a particular client and server. Section 14.4 of TCPv3 shows that for one Web server, the median RTT between many clients and the server was 187 ms. (The median is often used for this statistic, since a few large values can noticeably skew the mean.)
- Historically, sample code always shows a *backlog* of 5, as that was the maximum value supported by 4.2BSD. This was adequate in the 1980s when busy servers would handle only a few hundred connections per day. But with the growth of the World Wide Web (WWW), where busy servers handle millions of connections per day, this small number is completely inadequate (pp. 187–192 of TCPv3). Busy HTTP servers must specify a much larger *backlog*, and newer kernels must support larger values.

  > Many current systems allow the administrator to modify the maximum value for the *backlog*.

- A problem is: What value should the application specify for the *backlog*, since 5 is often inadequate? There is no easy answer to this. HTTP servers now specify a larger value, but if the value specified is a constant in the source code, to increase the constant requires recompiling the server. Another method is to assume some default but allow a command-line option or an environment variable to override

the default. It is always acceptable to specify a value that is larger than supported by the kernel, as the kernel should silently truncate the value to the maximum value that it supports, without returning an error (p. 456 of TCPv2).

We can provide a simple solution to this problem by modifying our wrapper function for the `listen` function. Figure 4.9 shows the actual code. We allow the environment variable `LISTENQ` to override the value specified by the caller.

lib/wrapsock.c

```
137 void
138 Listen(int fd, int backlog)
139 {
140     char    *ptr;

141         /* can override 2nd argument with environment variable */
142     if ( (ptr = getenv("LISTENQ")) != NULL)
143         backlog = atoi(ptr);

144     if (listen(fd, backlog) < 0)
145         err_sys("listen error");
146 }
```

lib/wrapsock.c

Figure 4.9 Wrapper function for `listen` that allows an environment variable to specify *backlog*.

- Manuals and books have historically said that the reason for queuing a fixed number of connections is to handle the case of the server process being busy between successive calls to `accept`. This implies that of the two queues, the completed queue should normally have more entries than the incomplete queue. Again, busy Web servers have shown that this is false. The reason for specifying a large *backlog* is because the incomplete connection queue can grow as client SYNs arrive, waiting for completion of the three-way handshake.

- If the queues are full when a client SYN arrives, TCP ignores the arriving SYN (pp. 930–931 of TCPv2); it does not send an RST. This is because the condition is considered temporary, and the client TCP will retransmit its SYN, hopefully finding room on the queue in the near future. If the server TCP immediately responded with an RST, the client's `connect` would return an error, forcing the application to handle this condition instead of letting TCP's normal retransmission take over. Also, the client could not differentiate between an RST in response to a SYN meaning "there is no server at this port" versus "there is a server at this port but its queues are full."

  Some implementations do send an RST when the queue is full. This behavior is incorrect for the reasons stated above, and unless your client specifically needs to interact with such a server, it's best to ignore this possibility. Coding to handle this case reduces the robustness of the client and puts more load on the network in the normal RST case, where the port really has no server listening on it.

- Data that arrives after the three-way handshake completes, but before the server calls `accept`, should be queued by the server TCP, up to the size of the connected socket's receive buffer.

Figure 4.10 shows the actual number of queued connections provided for different values of the *backlog* argument for the various operating systems in Figure 1.16. For seven different operating systems there are five distinct columns, showing the variety of interpretations about what *backlog* means!

	Maximum actual number of queued connections				
backlog	MacOS 10.2.6 AIX 5.1	Linux 2.4.7	HP-UX 11.11	FreeBSD 4.8 FreeBSD 5.1	Solaris 2.9
0	1	3	1	1	1
1	2	4	1	2	2
2	4	5	3	3	4
3	5	6	4	4	5
4	7	7	6	5	6
5	8	8	7	6	8
6	10	9	9	7	10
7	11	10	10	8	11
8	13	11	12	9	13
9	14	12	13	10	14
10	16	13	15	11	16
11	17	14	16	12	17
12	19	15	18	13	19
13	20	16	19	14	20
14	22	17	21	15	22

Figure 4.10 Actual number of queued connections for values of *backlog*.

AIX and MacOS have the traditional Berkeley algorithm, and Solaris seems very close to that algorithm as well. FreeBSD just adds one to *backlog*.

The program to measure these values is shown in the solution for Exercise 15.4.

As we said, historically the *backlog* has specified the maximum value for the sum of both queues. During 1996, a new type of attack was launched on the Internet called *SYN flooding* [CERT 1996b]. The hacker writes a program to send SYNs at a high rate to the victim, filling the incomplete connection queue for one or more TCP ports. (We use the term *hacker* to mean the attacker, as described in [Cheswick, Bellovin, and Rubin 2003].) Additionally, the source IP address of each SYN is set to a random number (this is called *IP spoofing*) so that the server's SYN/ACK goes nowhere. This also prevents the server from knowing the real IP address of the hacker. By filling the incomplete queue with bogus SYNs, legitimate SYNs are not queued, providing a *denial of service* to legitimate clients. There are two commonly used methods of handling these attacks, summarized in [Borman 1997b]. But what is most interesting in this note is revisiting what the `listen` *backlog* really means. It should specify the maximum number of *completed* connections for a given socket that the kernel will queue. The purpose of having a limit on these completed connections is to stop the kernel from accepting new connection requests for a given socket when the application is not accepting them (for whatever reason). If a system implements this interpretation, as does BSD/OS 3.0, then the application need not specify huge *backlog* values just because the server handles lots of client requests (e.g., a busy

Web server) or to provide protection against SYN flooding. The kernel handles lots of incomplete connections, regardless of whether they are legitimate or from a hacker. But even with this interpretation, scenarios do occur where the traditional value of 5 is inadequate.

4.6 accept Function

accept is called by a TCP server to return the next completed connection from the front of the completed connection queue (Figure 4.7). If the completed connection queue is empty, the process is put to sleep (assuming the default of a blocking socket).

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

int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

Returns: non-negative descriptor if OK, –1 on error

The *cliaddr* and *addrlen* arguments are used to return the protocol address of the connected peer process (the client). *addrlen* is a value-result argument (Section 3.3): Before the call, we set the integer value referenced by **addrlen* to the size of the socket address structure pointed to by *cliaddr*; on return, this integer value contains the actual number of bytes stored by the kernel in the socket address structure.

If accept is successful, its return value is a brand-new descriptor automatically created by the kernel. This new descriptor refers to the TCP connection with the client. When discussing accept, we call the first argument to accept the *listening socket* (the descriptor created by socket and then used as the first argument to both bind and listen), and we call the return value from accept the *connected socket*. It is important to differentiate between these two sockets. A given server normally creates only one listening socket, which then exists for the lifetime of the server. The kernel creates one connected socket for each client connection that is accepted (i.e., for which the TCP three-way handshake completes). When the server is finished serving a given client, the connected socket is closed.

This function returns up to three values: an integer return code that is either a new socket descriptor or an error indication, the protocol address of the client process (through the *cliaddr* pointer), and the size of this address (through the *addrlen* pointer). If we are not interested in having the protocol address of the client returned, we set both *cliaddr* and *addrlen* to null pointers.

Figure 1.9 shows these points. The connected socket is closed each time through the loop, but the listening socket remains open for the life of the server. We also see that the second and third arguments to accept are null pointers, since we were not interested in the identity of the client.

Example: Value-Result Arguments

We will now show how to handle the value-result argument to accept by modifying the code from Figure 1.9 to print the IP address and port of the client. We show this in Figure 4.11.

intro/daytimetcpsrv1.c

```
 1 #include    "unp.h"
 2 #include    <time.h>

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     listenfd, connfd;
 7     socklen_t len;
 8     struct sockaddr_in servaddr, cliaddr;
 9     char    buff[MAXLINE];
10     time_t  ticks;

11     listenfd = Socket(AF_INET, SOCK_STREAM, 0);

12     bzero(&servaddr, sizeof(servaddr));
13     servaddr.sin_family = AF_INET;
14     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
15     servaddr.sin_port = htons(13);  /* daytime server */

16     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

17     Listen(listenfd, LISTENQ);

18     for ( ; ; ) {
19         len = sizeof(cliaddr);
20         connfd = Accept(listenfd, (SA *) &cliaddr, &len);
21         printf("connection from %s, port %d\n",
22                Inet_ntop(AF_INET, &cliaddr.sin_addr, buff, sizeof(buff)),
23                ntohs(cliaddr.sin_port));

24         ticks = time(NULL);
25         snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
26         Write(connfd, buff, strlen(buff));

27         Close(connfd);
28     }
29 }
```

intro/daytimetcpsrv1.c

Figure 4.11 Daytime server that prints client IP address and port.

New declarations

7–8 We define two new variables: `len`, which will be a value-result variable, and `cliaddr`, which will contain the client's protocol address.

Accept connection and print client's address

19–23 We initialize `len` to the size of the socket address structure and pass a pointer to the `cliaddr` structure and a pointer to `len` as the second and third arguments to `accept`. We call `inet_ntop` (Section 3.7) to convert the 32-bit IP address in the socket address structure to a dotted-decimal ASCII string and call `ntohs` (Section 3.4) to convert the 16-bit port number from network byte order to host byte order.

> Calling `sock_ntop` instead of `inet_ntop` would make our server more protocol-independent, but this server is already dependent on IPv4. We will show a protocol-independent version of this server in Figure 11.13.

If we run our new server and then run our client on the same host, connecting to our server twice in a row, we have the following output from the client:

```
solaris % daytimetcpcli 127.0.0.1
Thu Sep 11 12:44:00 2003
solaris % daytimetcpcli 192.168.1.20
Thu Sep 11 12:44:09 2003
```

We first specify the server's IP address as the loopback address (127.0.0.1) and then as its own IP address (192.168.1.20). Here is the corresponding server output:

```
solaris # daytimetcpsrv1
connection from 127.0.0.1, port 43388
connection from 192.168.1.20, port 43389
```

Notice what happens with the client's IP address. Since our daytime client (Figure 1.5) does not call `bind`, we said in Section 4.4 that the kernel chooses the source IP address based on the outgoing interface that is used. In the first case, the kernel sets the source IP address to the loopback address; in the second case, it sets the address to the IP address of the Ethernet interface. We can also see in this example that the ephemeral port chosen by the Solaris kernel is 43388, and then 43389 (recall Figure 2.10).

As a final point, our shell prompt for the server script changes to the pound sign (#), the commonly used prompt for the superuser. Our server must run with superuser privileges to `bind` the reserved port of 13. If we do not have superuser privileges, the call to `bind` will fail:

```
solaris % daytimetcpsrv1
bind error: Permission denied
```

4.7 fork and exec Functions

Before describing how to write a concurrent server in the next section, we must describe the Unix `fork` function. This function (including the variants of it provided by some systems) is the only way in Unix to create a new process.

```
#include <unistd.h>

pid_t fork(void);
```

Returns: 0 in child, process ID of child in parent, –1 on error

If you have never seen this function before, the hard part in understanding `fork` is that it is called *once* but it returns *twice*. It returns once in the calling process (called the parent) with a return value that is the process ID of the newly created process (the child). It also returns once in the child, with a return value of 0. Hence, the return value tells the process whether it is the parent or the child.

The reason `fork` returns 0 in the child, instead of the parent's process ID, is because a child has only one parent and it can always obtain the parent's process ID by calling `getppid`. A parent, on the other hand, can have any number of children, and there is