ELEC S212

Network Programming and Design

Unit 6

Readings

香港公開大學
THE OPEN UNIVERSITY OF HONG KONG

科技學院 School of Science and Technology

Readings

Copyright acknowledgements

Printed July 2018.
Scanned September 2018.

6.1 Stevens, W R, and Rago, S A (2005) *Advanced Programming in the UNIX Environment*, 2nd edn, Boston: Addison-Wesley, Sections 8.2 & 4.4.

8.2. Process Identifiers

Every process has a unique process ID, a non-negative integer. Because the process ID is the only well-known identifier of a process that is always unique, it is often used as a piece of other identifiers, to guarantee uniqueness. For example, applications sometimes include the process ID as part of a filename in an attempt to generate unique filenames.

Although unique, process IDs are reused. As processes terminate, their IDs become candidates for reuse. Most UNIX systems implement algorithms to delay reuse, however, so that newly created processes are assigned IDs different from those used by processes that terminated recently. This prevents a new process from being mistaken for the previous process to have used the same ID.

There are some special processes, but the details differ from implementation to implementation. Process ID 0 is usually the scheduler process and is often known as the *swapper*. No program on disk corresponds to this process, which is part of the kernel and is known as a system process. Process ID 1 is usually the `init` process and is invoked by the kernel at the end of the bootstrap procedure. The program file for this process was `/etc/init` in older versions of the UNIX System and is `/sbin/init` in newer versions. This process is responsible for bringing up a UNIX system after the kernel has been bootstrapped. `init` usually reads the system-dependent initialization filesthe `/etc/rc*` files or `/etc/inittab` and the files in `/etc/init.d`and brings the system to a certain state, such as multiuser. The `init` process never dies. It is a normal user process, not a system process within the kernel, like the swapper, although it does run with superuser privileges. Later in this chapter, we'll see how `init` becomes the parent process of any orphaned child process.

Each UNIX System implementation has its own set of kernel processes that provide operating system services. For example, on some virtual memory implementations of the UNIX System, process ID 2 is the *pagedaemon*. This process is responsible for supporting the paging of the virtual memory system.

In addition to the process ID, there are other identifiers for every process. The following functions return these identifiers.

```
#include <unistd.h>

pid_t getpid(void);
```

Returns: process ID of calling process

```
pid_t getppid(void);
```

Returns: parent process ID of calling process

```
uid_t getuid(void);
```

Returns: real user ID of calling process

```
uid_t geteuid(void);
```

Returns: effective user ID of calling process

```
gid_t getgid(void);
```

Returns: real group ID of calling process

```
gid_t getegid(void);
```

Returns: effective group ID of calling process

Note that none of these functions has an error return. We'll return to the parent process ID in the next section when we discuss the `fork` function. The real and effective user and group IDs were discussed in Section 4.4.

4.4. Set-User-ID and Set-Group-ID

Every process has six or more IDs associated with it. These are shown in Figure 4.5.

Figure 4.5. User IDs and group IDs associated with each process

real user ID real group ID	who we really are
effective user ID effective group ID supplementary group IDs	used for file access permission checks
saved set-user-ID saved set-group-ID	saved by `exec` functions

- The real user ID and real group ID identify who we really are. These two fields are taken from our entry in the password file when we log in. Normally, these values don't change during a login session, although there are ways for a superuser process to change them, which we describe in Section 8.11.
- The effective user ID, effective group ID, and supplementary group IDs determine our file access permissions, as we describe in the next section. (We defined supplementary group IDs in Section 1.8.)
- The saved set-user-ID and saved set-group-ID contain copies of the effective user ID and the effective group ID when a program is executed. We describe the function of these two saved values when we describe the `setuid` function in Section 8.11.

> The saved IDs are required with the 2001 version of POSIX.1. They used to be optional in older versions of POSIX. An application can test for the constant `_POSIX_SAVED_IDS` at compile time or can call `sysconf` with the `_SC_SAVED_IDS` argument at runtime, to see whether the implementation supports this feature.

Normally, the effective user ID equals the real user ID, and the effective group ID equals the real group ID.

Every file has an owner and a group owner. The owner is specified by the `st_uid` member of the `stat` structure; the group owner, by the `st_gid` member.

When we execute a program file, the effective user ID of the process is usually the real user ID, and the effective group ID is usually the real group ID. But the capability exists to set a special flag in the file's mode word (`st_mode`) that says "when this file is executed, set the effective user ID of the process to be the owner of the file (`st_uid`)." Similarly, another bit can be set in the file's mode word that causes the effective group ID to be the group owner of the file (`st_gid`). These two bits in the file's mode word are called the *set-user-ID* bit and the *set-group-ID* bit.

For example, if the owner of the file is the superuser and if the file's set-user-ID bit is set, then while that program file is running as a process, it has superuser privileges. This happens regardless of the real user ID of the process that executes the file. As an example, the UNIX System program that allows anyone to change his or her password, `passwd`(1), is a set-user-ID program. This is required so that the program can write the new password to the password file, typically either `/etc/passwd` or `/etc/shadow`, files that should be writable only by the superuser. Because a process that is running set-user-ID to some other user usually assumes extra permissions, it must be written carefully. We'll discuss these types of programs in more detail in Chapter 8.

Returning to the `stat` function, the set-user-ID bit and the set-group-ID bit are contained in the file's `st_mode` value. These two bits can be tested against the constants `S_ISUID` and `S_ISGID`.

6.2 Stevens, W R, Fenner, B, and Rudoff, A M (2003) *UNIX Network Programming, Volume 1: The Sockets Networking API*, 3rd edn, Boston: Addison-Wesley, Sections 4.1 and 8.1.

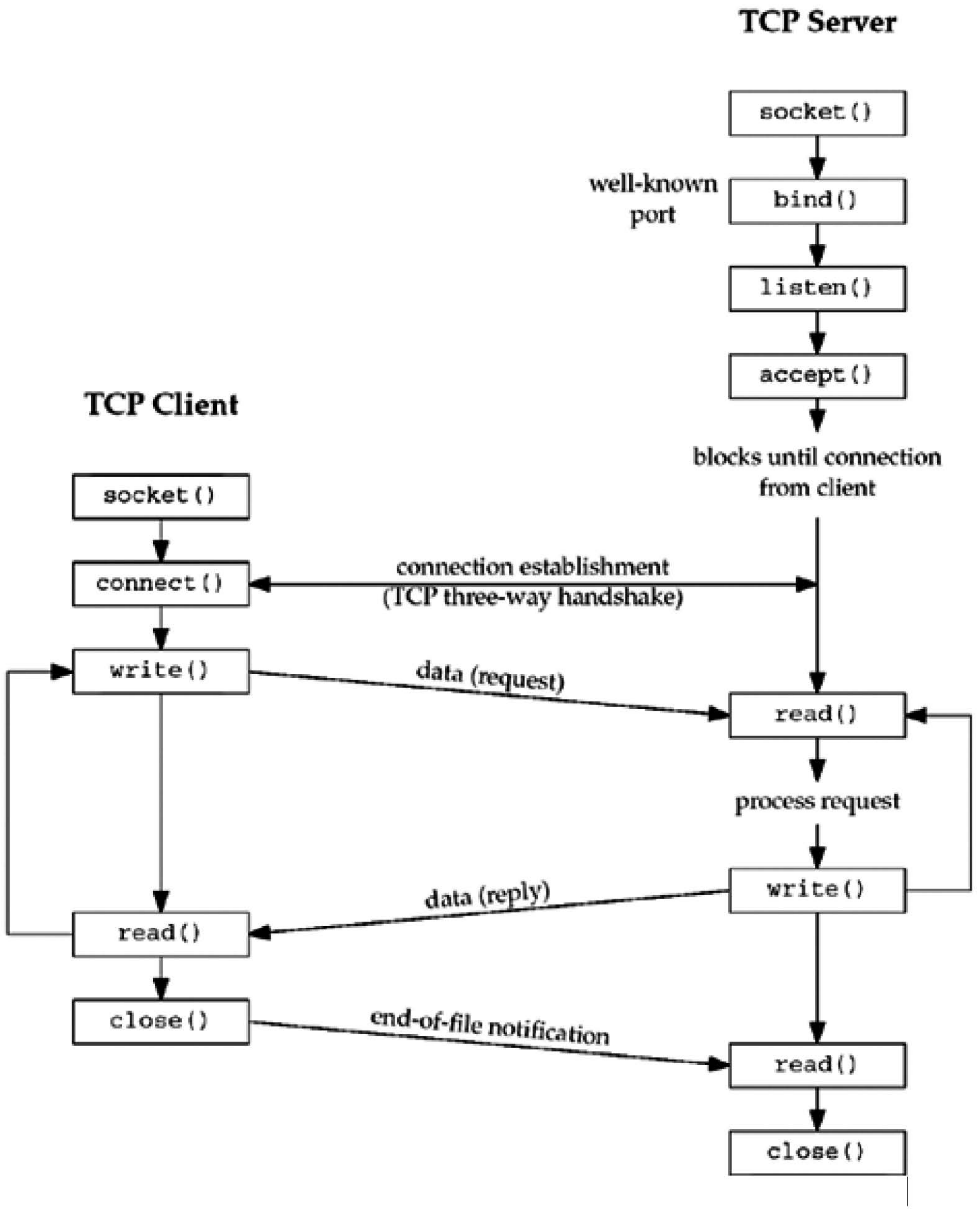
4.1 Introduction

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.

Figure 4.1. Socket functions for elementary TCP client/server.


TCP Server
socket()
well-known port
bind()
listen()
accept()
blocks until connection from client
TCP Client
socket()
connect()
connection establishment (TCP three-way handshake)
write()
data (request)
read()
process request
write()
data (reply)
read()
close()
end-of-file notification
read()
close()

8.1 Introduction

There are some fundamental differences between applications written using TCP versus those that use UDP. These are because of the differences in the two transport layers: UDP is a connectionless, unreliable, datagram protocol, quite unlike the connection-oriented, reliable byte stream provided by TCP. Nevertheless, there are instances when it makes sense to use UDP instead of TCP, and we will go over this design choice in Section 22.4. Some popular applications are built using UDP: DNS, NFS, and SNMP, for example.

Figure 8.1 shows the function calls for a typical UDP client/server. The client does not establish a connection with the server. Instead, the client just sends a datagram to the server using the `sendto` function (described in the next section), which requires the address of the destination (the server) as a parameter. Similarly, the server does not accept a connection from a client. Instead, the server just calls the `recvfrom` function, which waits until data arrives from some client. `recvfrom` returns the protocol address of the client, along with the datagram, so the server can send a response to the correct client.

Figure 8.1. Socket functions for UDP client/server.

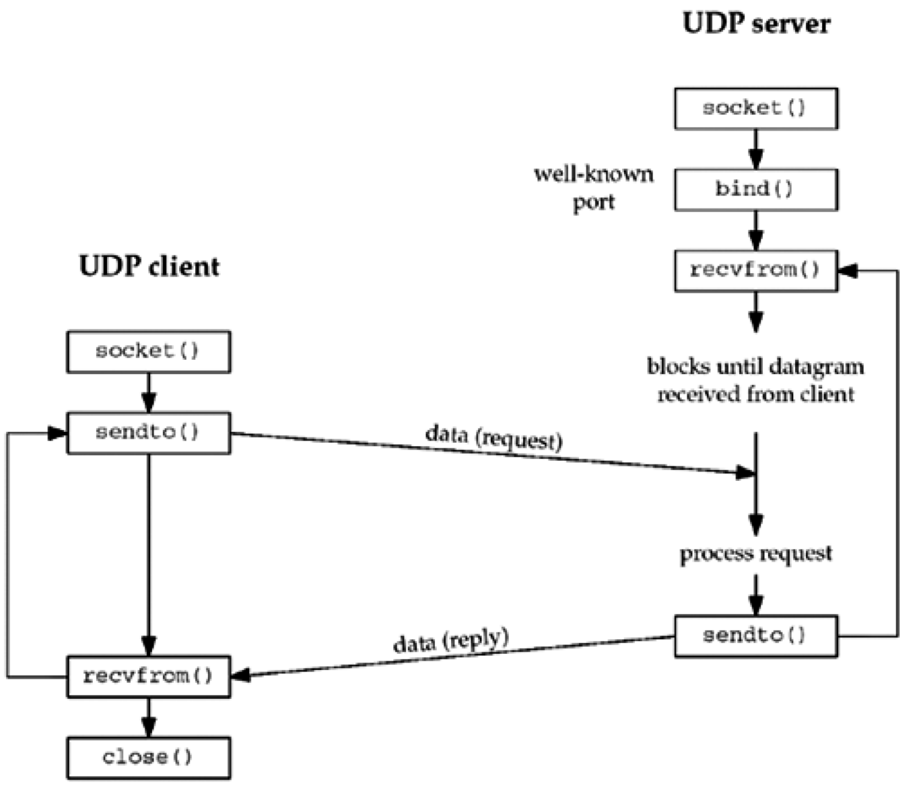


Figure 8.1 shows a timeline of the typical scenario that takes place for a UDP client/server exchange. We can compare this to the typical TCP exchange that

was shown in Figure 4.1.

In this chapter, we will describe the new functions that we use with UDP sockets, `recvfrom` and `sendto`, and redo our echo client/server to use UDP. We will also describe the use of the `connect` function with a UDP socket, and the concept of asynchronous errors.

6.3 Stevens, W R, Fenner, B, and Rudoff, A M (2003) *UNIX Network Programming, Volume 1: The Sockets Networking API*, 3rd edn, Boston: Addison-Wesley, Sections 3.1–3.6.

3.1 Introduction

This chapter begins the description of the sockets API. We begin with socket address structures, which will be found in almost every example in the text. These structures can be passed in two directions: from the process to the kernel, and from the kernel to the process. The latter case is an example of a value-result argument, and we will encounter other examples of these arguments throughout the text.

The address conversion functions convert between a text representation of an address and the binary value that goes into a socket address structure. Most existing IPv4 code uses `inet_addr` and `inet_ntoa`, but two new functions, `inet_pton` and `inet_ntop`, handle both IPv4 and IPv6.

One problem with these address conversion functions is that they are dependent on the type of address being converted: IPv4 or IPv6. We will develop a set of functions whose names begin with `sock_` that work with socket address structures in a protocol-independent fashion. We will use these throughout the text to make our code protocol-independent.

3.2 Socket Address Structures

Most socket functions require a pointer to a socket address structure as an argument. Each supported protocol suite defines its own socket address structure. The names of these structures begin with `sockaddr_` and end with a unique suffix for each protocol suite.

IPv4 Socket Address Structure

An IPv4 socket address structure, commonly called an "Internet socket address structure," is named `sockaddr_in` and is defined by including the `<netinet/in.h>` header. Figure 3.1 shows the POSIX definition.

Figure 3.1 The Internet (IPv4) socket address structure: `sockaddr_in`.

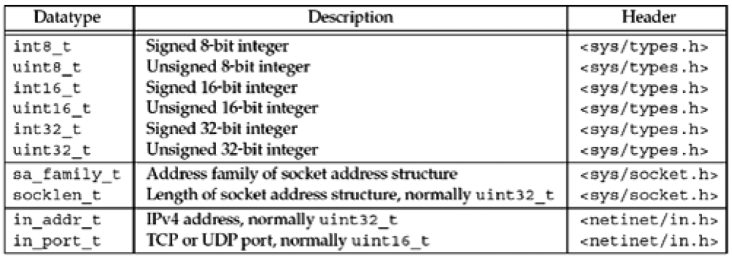
```
struct in_addr {
  in_addr_t   s_addr;           /* 32-bit IPv4 address */
                                /* network byte ordered */
};

struct sockaddr_in {
  uint8_t         sin_len;      /* length of structure (16) */
  sa_family_t     sin_family;   /* AF_INET */
  in_port_t       sin_port;     /* 16-bit TCP or UDP port number */
                                /* network byte ordered */
  struct in_addr  sin_addr;     /* 32-bit IPv4 address */
                                /* network byte ordered */
  char            sin_zero[8];  /* unused */
};
```

There are several points we need to make about socket address structures in general using this example:

- The length member, `sin_len`, was added with 4.3BSD-Reno, when support for the OSI protocols was added (Figure 1.15). Before this release, the first member was `sin_family`, which was historically an `unsigned short`. Not all vendors support a length field for socket address structures and the POSIX specification does not require this member. The datatype that we show, `uint8_t`, is typical, and POSIX-compliant systems provide datatypes of this form (Figure 3.2).

Figure 3.2. Datatypes required by the POSIX specification.



Datatype	Description	Header
`int8_t`	Signed 8-bit integer	`<sys/types.h>`
`uint8_t`	Unsigned 8-bit integer	`<sys/types.h>`
`int16_t`	Signed 16-bit integer	`<sys/types.h>`
`uint16_t`	Unsigned 16-bit integer	`<sys/types.h>`
`int32_t`	Signed 32-bit integer	`<sys/types.h>`
`uint32_t`	Unsigned 32-bit integer	`<sys/types.h>`
`sa_family_t`	Address family of socket address structure	`<sys/socket.h>`
`socklen_t`	Length of socket address structure, normally `uint32_t`	`<sys/socket.h>`
`in_addr_t`	IPv4 address, normally `uint32_t`	`<netinet/in.h>`
`in_port_t`	TCP or UDP port, normally `uint16_t`	`<netinet/in.h>`

  Having a length field simplifies the handling of variable-length socket address structures.

- Even if the length field is present, we need never set it and need never examine it, unless we are dealing with routing sockets (Chapter 18). It is used within the kernel by the routines that deal with socket address structures from various protocol families (e.g., the routing table code).

    The four socket functions that pass a socket address structure from the process to the kernel, `bind`, `connect`, `sendto`, and `sendmsg`, all go through the `sockargs` function in a Berkeley-derived implementation (p. 452 of TCPv2). This function copies the socket address structure from the process and explicitly sets its `sin_len` member to the size of the structure that was passed as an argument to these four functions. The five socket functions that pass a socket address structure from the kernel to the process, `accept`, `recvfrom`, `recvmsg`, `getpeername`, and `getsockname`, all set the `sin_len` member before returning to the process.

    Unfortunately, there is normally no simple compile-time test to determine whether an implementation defines a length field for its socket address structures. In our code, we test our own `HAVE_SOCKADDR_SA_LEN` constant (Figure D.2), but whether to define this constant or not requires trying to compile a simple test program that uses this optional structure member and seeing if the compilation succeeds or not. We will see in Figure 3.4 that IPv6 implementations are required to define `SIN6_LEN` if socket address structures have a length field. Some IPv4 implementations provide the length field of the socket address structure to the application based on a compile-time option (e.g., `_SOCKADDR_LEN`). This feature provides compatibility for older programs.

- The POSIX specification requires only three members in the structure: `sin_family`, `sin_addr`, and `sin_port`. It is acceptable for a POSIX-compliant implementation to define additional structure members, and this is normal for an Internet socket address structure. Almost all implementations add the `sin_zero` member so that all socket address structures are at least 16 bytes in size.

- We show the POSIX datatypes for the `s_addr`, `sin_family`, and `sin_port` members. The `in_addr_t` datatype must be an unsigned integer type of at least 32 bits, `in_port_t` must be an unsigned integer type of at least 16 bits, and `sa_family_t` can be any unsigned integer type. The latter is normally an 8-bit unsigned integer if the implementation supports the length field, or an unsigned 16-bit integer if the length field is not supported. Figure 3.2 lists these three POSIX-defined datatypes, along with some other POSIX datatypes that we will encounter.

- You will also encounter the datatypes `u_char`, `u_short`, `u_int`, and `u_long`, which are all unsigned. The POSIX specification defines these with a note that they are obsolete. They are provided for backward compatibility.

- Both the IPv4 address and the TCP or UDP port number are always stored in the structure in network byte order. We must be cognizant of this when using these members. We will say more about the difference between host byte order and network byte order in Section 3.4.

- The 32-bit IPv4 address can be accessed in two different ways. For example, if `serv` is defined as an Internet socket address structure, then `serv.sin_addr` references the 32-bit IPv4 address as an `in_addr` structure, while `serv.sin_addr.s_addr` references the same 32-bit IPv4 address as an `in_addr_t` (typically an unsigned 32-bit integer). We must be certain that we are referencing the IPv4 address correctly, especially when it is used as an argument to a function, because compilers often pass structures differently from integers.

  The reason the `sin_addr` member is a structure, and not just an `in_addr_t`, is historical. Earlier releases (4.2BSD) defined the `in_addr` structure as a `union` of various structures, to allow access to each of the 4 bytes and to both of the 16-bit values contained within the 32-bit IPv4 address. This was used with class A, B, and C addresses to fetch the appropriate bytes of the address. But with the advent of subnetting and then the disappearance of the various address classes with classless addressing (Section A.4), the need for the `union` disappeared. Most systems today have done away with the `union` and just define `in_addr` as a structure with a single `in_addr_t` member.

- The `sin_zero` member is unused, but we *always* set it to 0 when filling in one of these structures. By convention, we always set the entire structure to 0 before filling it in, not just the `sin_zero` member.

  Although most uses of the structure do not require that this member be 0, when binding a non-wildcard IPv4 address, this member must be 0 (pp. 731–732 of TCPv2).

- Socket address structures are used only on a given host: The structure itself is not communicated between different hosts, although certain fields (e.g., the IP address and port) are used for communication.

Generic Socket Address Structure

A socket address structures is *always* passed by reference when passed as an argument to any socket functions. But any socket function that takes one of these pointers as an argument must deal with socket address structures from *any* of the supported protocol families.

A problem arises in how to declare the type of pointer that is passed. With ANSI C, the solution is simple: `void *` is the generic pointer type. But, the socket functions predate ANSI C and the solution chosen in 1982 was to define a *generic* socket address structure in the `<sys/socket.h>` header, which we show in Figure 3.3.

Figure 3.3 The generic socket address structure: `sockaddr`.

```
struct sockaddr {
  uint8_t      sa_len;
  sa_family_t  sa_family;    /* address family: AF_xxx value */
  char         sa_data[14];  /* protocol-specific address */
};
```

The socket functions are then defined as taking a pointer to the generic socket address structure, as shown here in the ANSI C function prototype for the `bind` function:

```
int bind(int, struct sockaddr *, socklen_t);
```

This requires that any calls to these functions must cast the pointer to the protocol-specific socket address structure to be a pointer to a generic socket address structure. For example,

```
struct sockaddr_in  serv;      /* IPv4 socket address structure */

/* fill in serv{} */

bind(sockfd, (struct sockaddr *) &serv, sizeof(serv));
```

If we omit the cast "(`struct sockaddr *`)," the C compiler generates a warning of the form "warning: passing arg 2 of 'bind' from incompatible pointer type," assuming the system's headers have an ANSI C prototype for the `bind` function.

From an application programmer's point of view, the *only* use of these generic socket address structures is to cast pointers to protocol-specific structures.

> Recall in Section 1.2 that in our `unp.h` header, we define `SA` to be the string "`struct sockaddr`" just to shorten the code that we must write to cast these pointers.
>
> From the kernel's perspective, another reason for using pointers to generic socket address structures as arguments is that the kernel must take the caller's pointer, cast it to a `struct sockaddr *`, and then look at the value of `sa_family` to determine the type of the structure. But from an application programmer's perspective, it would be simpler if the pointer type was `void *`, omitting the need for the explicit cast.

IPv6 Socket Address Structure

The IPv6 socket address is defined by including the `<netinet/in.h>` header, and we show it in Figure 3.4.

Figure 3.4 IPv6 socket address structure: `sockaddr_in6`.

```
struct in6_addr {
  uint8_t  s6_addr[16];          /* 128-bit IPv6 address */
                                 /* network byte ordered */
};

#define SIN6_LEN      /* required for compile-time tests */
```

```
struct sockaddr_in6 {
  uint8_t         sin6_len;      /* length of this struct (28) */
  sa_family_t     sin6_family;   /* AF_INET6 */
  in_port_t       sin6_port;     /* transport layer port# */
                                 /* network byte ordered */
  uint32_t        sin6_flowinfo; /* flow information, undefined */
  struct in6_addr sin6_addr;     /* IPv6 address */
                                 /* network byte ordered */
  uint32_t        sin6_scope_id; /* set of interfaces for a scope */
};
```

The extensions to the sockets API for IPv6 are defined in RFC 3493 [Gilligan et al. 2003].

Note the following points about Figure 3.4:

- The `SIN6_LEN` constant must be defined if the system supports the length member for socket address structures.
- The IPv6 family is `AF_INET6`, whereas the IPv4 family is `AF_INET`.
- The members in this structure are ordered so that if the `sockaddr_in6` structure is 64-bit aligned, so is the 128-bit `sin6_addr` member. On some 64-bit processors, data accesses of 64-bit values are optimized if stored on a 64-bit boundary.
- The `sin6_flowinfo` member is divided into two fields:
  - The low-order 20 bits are the flow label

  The flow label field is described with Figure A.2. The use of the flow label field is still a research topic.
- The `sin6_scope_id` identifies the scope zone in which a scoped address is meaningful, most commonly an interface index for a link-local address (Section A.5).

New Generic Socket Address Structure

A new generic socket address structure was defined as part of the IPv6 sockets API, to overcome some of the shortcomings of the existing `struct sockaddr`. Unlike the `struct sockaddr`, the new `struct sockaddr_storage` is large enough to hold any socket address type supported by the system. The `sockaddr_storage` structure is defined by including the `<netinet/in.h>` header, which we show in Figure 3.5.

Figure 3.5 The storage socket address structure: `sockaddr_storage`.

```
struct sockaddr_storage {
  uint8_t      ss_len;       /* length of this struct (implementation dependent) */
  sa_family_t  ss_family;    /* address family: AF_xxx value */
  /* implementation-dependent elements to provide:
   * a) alignment sufficient to fulfill the alignment requirements of
   *    all socket address types that the system supports.
   * b) enough storage to hold any type of socket address that the
   *    system supports.
   */
};
```

The `sockaddr_storage` type provides a generic socket address structure that is different from `struct sockaddr` in two ways:

a. If any socket address structures that the system supports have alignment requirements, the `sockaddr_storage` provides the strictest alignment requirement.

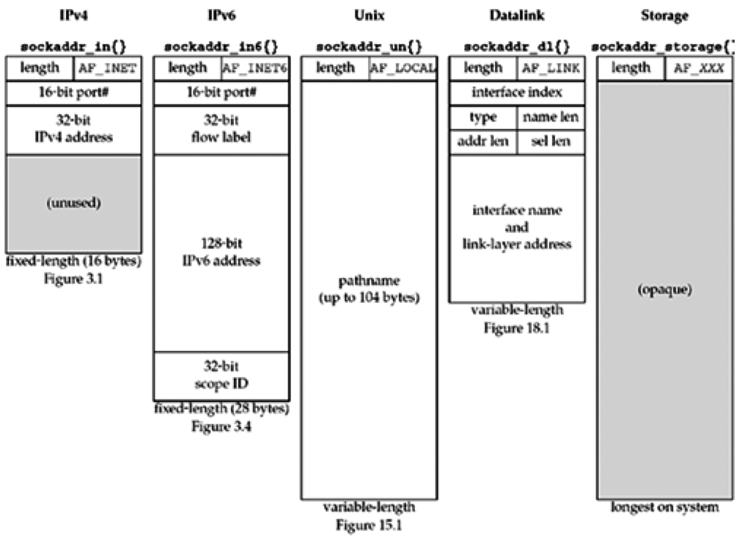
b. The `sockaddr_storage` is large enough to contain any socket address structure that the system supports.

Note that the fields of the `sockaddr_storage` structure are opaque to the user, except for `ss_family` and `ss_len` (if present). The `sockaddr_storage` must be cast or copied to the appropriate socket address structure for the address given in `ss_family` to access any other fields.

Comparison of Socket Address Structures

Figure 3.6 shows a comparison of the five socket address structures that we will encounter in this text: IPv4, IPv6, Unix domain (Figure 15.1), datalink (Figure 18.1), and storage. In this figure, we assume that the socket address structures all contain a one-byte length field, that the family field also occupies one byte, and that any field that must be at least some number of bits is exactly that number of bits.

Figure 3.6. Comparison of various socket address structures.



Two of the socket address structures are fixed-length, while the Unix domain structure and the datalink structure are variable-length. To handle variable-length structures, whenever we pass a pointer to a socket address structure as an argument to one of the socket functions, we pass its length as another argument. We show the size in bytes (for the 4.4BSD implementation) of the fixed-length structures beneath each structure.

The `sockaddr_un` structure itself is not variable-length (Figure 15.1), but the amount of information—the pathname within the structure—is variable-length. When passing pointers to these structures, we must be careful how we handle the length field, both the length field in the socket address structure itself (if supported by the implementation) and the length to and from the kernel.

This figure shows the style that we follow throughout the text: structure names are always shown in a bolder font, followed by braces, as in **`sockaddr_in{}`**.

We noted earlier that the length field was added to all the socket address structures with the 4.3BSD Reno release. Had the length field been present with the original release of sockets, there would be no need for the length argument to all the socket functions: the third argument to `bind` and `connect`, for example. Instead, the size of the structure could be contained in the length field of the structure.

3.3 Value-Result Arguments

We mentioned that when a socket address structure is passed to any socket function, it is always passed by reference. That is, a pointer to the structure is passed. The length of the structure is also passed as an argument. But the way in which the length is passed depends on which direction the structure is being passed: from the process to the kernel, or vice versa.

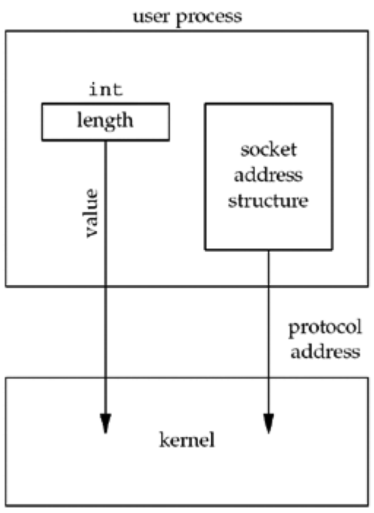
1. Three functions, `bind`, `connect`, and `sendto`, pass a socket address structure from the process to the kernel. One argument to these three functions is the pointer to the socket address structure and another argument is the integer size of the structure, as in

```
struct sockaddr_in serv;

/* fill in serv{} */
connect (sockfd, (SA *) &serv, sizeof(serv));
```

   Since the kernel is passed both the pointer and the size of what the pointer points to, it knows exactly how much data to copy from the process into the kernel. Figure 3.7 shows this scenario.

Figure 3.7. Socket address structure passed from process to kernel.



   We will see in the next chapter that the datatype for the size of a socket address structure is actually `socklen_t` and not `int`, but the POSIX specification recommends that `socklen_t` be defined as `uint32_t`.

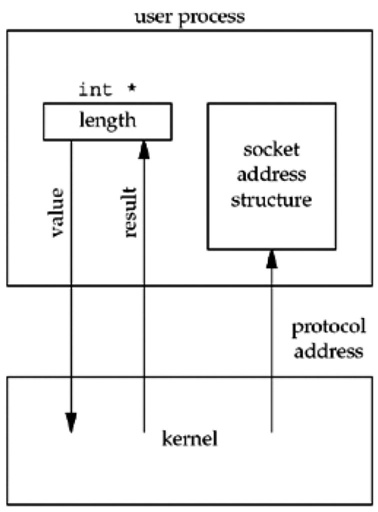
2. Four functions, `accept`, `recvfrom`, `getsockname`, and `getpeername`, pass a socket address structure from the kernel to the process, the reverse direction from the previous scenario. Two of the arguments to these four functions are the pointer to the socket address structure along with a pointer to an integer containing the size of the structure, as in

```
struct sockaddr_un  cli;   /* Unix domain */
socklen_t  len;

len = sizeof(cli);         /* len is a value */
getpeername(unixfd, (SA *) &cli, &len);
/* len may have changed */
```

   The reason that the size changes from an integer to be a pointer to an integer is because the size is both a *value* when the function is called (it tells the kernel the size of the structure so that the kernel does not write past the end of the structure when filling it in) and a *result* when the function returns (it tells the process how much information the kernel actually stored in the structure). This type of argument is called a *value-result* argument. Figure 3.8 shows this scenario.

Figure 3.8. Socket address structure passed from kernel to process.



We will see an example of value-result arguments in Figure 4.11.

We have been talking about socket address structures being passed between the process and the kernel. For an implementation such as 4.4BSD, where all the socket functions are system calls within the kernel, this is correct. But in some implementations, notably System V, socket functions are just library functions that execute as part of a normal user process. How these functions interface with the protocol stack in the kernel is an implementation detail that normally does not affect us. Nevertheless, for simplicity, we will continue to talk about these structures as being passed between the process and the kernel by functions such as `bind` and `connect`. (We will see in Section C.1 that System V implementations do indeed pass socket address structures between processes and the kernel, but as part of `STREAMS` messages.)

Two other functions pass socket address structures: `recvmsg` and `sendmsg` (Section 14.5). But, we will see that the length field is not a function argument but a structure member.

When using value-result arguments for the length of socket address structures, if the socket address structure is fixed-length (Figure 3.6), the value returned by the kernel will always be that fixed size: 16 for an IPv4 `sockaddr_in` and 28 for an IPv6 `sockaddr_in6`, for example. But with a variable-length socket address structure (e.g., a Unix domain `sockaddr_un`), the value returned can be less than the maximum size of the structure (as we will see with Figure 15.2).

With network programming, the most common example of a value-result argument is the length of a returned socket address structure. But, we will encounter other value-result arguments in this text:

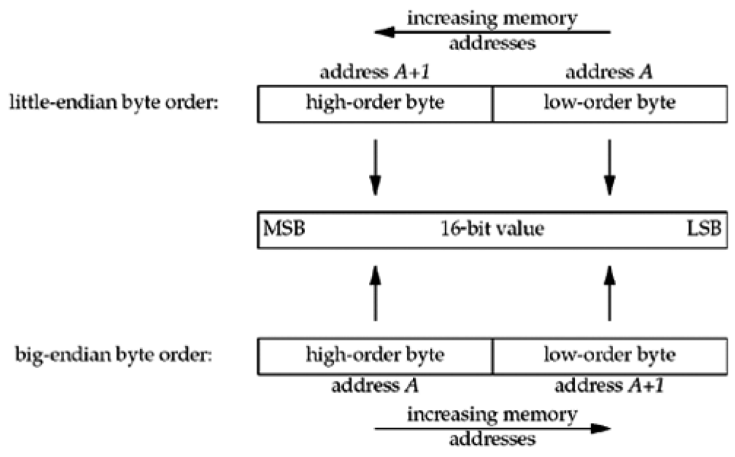
- The middle three arguments for the `select` function (Section 6.3)
- The length argument for the `getsockopt` function (Section 7.2)
- The `msg_namelen` and `msg_controllen` members of the `msghdr` structure, when used with `recvmsg` (Section 14.5)

- The `ifc_len` member of the `ifconf` structure (Figure 17.2)
- The first of the two length arguments for the `sysctl` function (Section 18.4)

3.4 Byte Ordering Functions

Consider a 16-bit integer that is made up of 2 bytes. There are two ways to store the two bytes in memory: with the low-order byte at the starting address, known as *little-endian* byte order, or with the high-order byte at the starting address, known as *big-endian* byte order. We show these two formats in Figure 3.9.

Figure 3.9. Little-endian byte order and big-endian byte order for a 16-bit integer.



In this figure, we show increasing memory addresses going from right to left in the top, and from left to right in the bottom. We also show the most significant bit (MSB) as the leftmost bit of the 16-bit value and the least significant bit (LSB) as the rightmost bit.

> The terms "little-endian" and "big-endian" indicate which end of the multibyte value, the little end or the big end, is stored at the starting address of the value.

Unfortunately, there is no standard between these two byte orderings and we encounter systems that use both formats. We refer to the byte ordering used by a given system as the *host byte order*. The program shown in Figure 3.10 prints the host byte order.

Figure 3.10 Program to determine host byte order.

intro/byteorder.c

```
#include     "unp.h"

int
main(int argc, char **argv)
{
    union {
        short   s;
        char    c[sizeof(short)];
    } un;

    un.s = 0x0102;
    printf("%s: ", CPU_VENDOR_OS);
    if (sizeof(short) == 2) {
        if (un.c[0] == 1 && un.c[1] == 2)
            printf("big-endian\n");
        else if (un.c[0] == 2 && un.c[1] == 1)
            printf("little-endian\n");
        else
            printf("unknown\n");
    } else
        printf("sizeof(short) = %d\n", sizeof(short));

    exit(0);
}
```

We store the two-byte value `0x0102` in the short integer and then look at the two consecutive bytes, `c[0]` (the address *A* in Figure 3.9) and `c[1]` (the address *A+1* in Figure 3.9), to determine the byte order.

The string `CPU_VENDOR_OS` is determined by the GNU `autoconf` program when the software in this book is configured, and it identifies the CPU type, vendor, and OS release. We show some examples here in the output from this program when run on the various systems in Figure 1.16.

```
freebsd4 % byteorder
i386-unknown-freebsd4.8: little-endian

macosx % byteorder
powerpc-apple-darwin6.6: big-endian

freebsd5 % byteorder
sparc64-unknown-freebsd5.1: big-endian

aix % byteorder
powerpc-ibm-aix5.1.0.0: big-endian

hpux % byteorder
hppa1.1-hp-hpux11.11: big-endian

linux % byteorder
i586-pc-linux-gnu: little-endian

solaris % byteorder
sparc-sun-solaris2.9: big-endian
```

We have talked about the byte ordering of a 16-bit integer; obviously, the same discussion applies to a 32-bit integer.

> There are currently a variety of systems that can change between little-endian and big-endian byte ordering, sometimes at system reset, sometimes at run-time.

We must deal with these byte ordering differences as network programmers because networking protocols must specify a *network byte order*. For example, in a TCP segment, there is a 16-bit port number and a 32-bit IPv4 address. The sending protocol stack and the receiving protocol stack must agree on the order in which the bytes of these multibyte fields will be transmitted. The Internet protocols use big-endian byte ordering for these multibyte integers.

In theory, an implementation could store the fields in a socket address structure in host byte order and then convert to and from the network byte order when moving the fields to and from the protocol headers, saving us from having to worry about this detail. But, both history and the POSIX specification say that certain fields in the socket address structures must be maintained in network byte order. Our concern is therefore converting between host byte order and network byte order. We use the following four functions to convert between these two byte orders.

```
#include <netinet/in.h>

uint16_t htons(uint16_t host16bitvalue) ;
```

```
uint32_t htonl(uint32_t host32bitvalue) ;
                                        Both return: value in network byte order
uint16_t ntohs(uint16_t net16bitvalue) ;
uint32_t ntohl(uint32_t net32bitvalue) ;
                                           Both return: value in host byte order
```

In the names of these functions, `h` stands for *host*, `n` stands for *network*, `s` stands for *short*, and `l` stands for *long*. The terms "short" and "long" are historical artifacts from the Digital VAX implementation of 4.2BSD. We should instead think of `s` as a 16-bit value (such as a TCP or UDP port number) and `l` as a 32-bit value (such as an IPv4 address). Indeed, on the 64-bit Digital Alpha, a long integer occupies 64 bits, yet the `htonl` and `ntohl` functions operate on 32-bit values.

When using these functions, we do not care about the actual values (big-endian or little-endian) for the host byte order and the network byte order. What we must do is call the appropriate function to convert a given value between the host and network byte order. On those systems that have the same byte ordering as the Internet protocols (big-endian), these four functions are usually defined as null macros.

We will talk more about the byte ordering problem, with respect to the data contained in a network packet as opposed to the fields in the protocol headers, in Section 5.18 and Exercise 5.8.

We have not yet defined the term "byte." We use the term to mean an 8-bit quantity since almost all current computer systems use 8-bit bytes. Most Internet standards use the term *octet* instead of byte to mean an 8-bit quantity. This started in the early days of TCP/IP because much of the early work was done on systems such as the DEC-10, which did not use 8-bit bytes.

Another important convention in Internet standards is bit ordering. In many Internet standards, you will see "pictures" of packets that look similar to the following (this is the first 32 bits of the IPv4 header from RFC 791):

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

This represents four bytes in the order in which they appear on the wire; the leftmost bit is the most significant. However, the numbering starts with zero assigned to the most significant bit. This is a notation that you should become familiar with to make it easier to read protocol definitions in RFCs.

> A common network programming error in the 1980s was to develop code on Sun workstations (big-endian Motorola 68000s) and forget to call any of these four functions. The code worked fine on these workstations, but would not work when ported to little-endian machines (such as VAXes).

3.5 Byte Manipulation Functions

There are two groups of functions that operate on multibyte fields, without interpreting the data, and without assuming that the data is a null-terminated C string. We need these types of functions when dealing with socket address structures because we need to manipulate fields such as IP addresses, which can contain bytes of 0, but are not C character strings. The functions beginning with `str` (for string), defined by including the `<string.h>` header, deal with null-terminated C character strings.

The first group of functions, whose names begin with `b` (for byte), are from 4.2BSD and are still provided by almost any system that supports the socket functions. The second group of functions, whose names begin with `mem` (for memory), are from the ANSI C standard and are provided with any system that supports an ANSI C library.

We first show the Berkeley-derived functions, although the only one we use in this text is `bzero`. (We use it because it has only two arguments and is easier to remember than the three-argument `memset` function, as explained on p. 8.) You may encounter the other two functions, `bcopy` and `bcmp`, in existing applications.

```
#include <strings.h>

void bzero(void *dest, size_t nbytes);

void bcopy(const void *src, void *dest, size_t nbytes);

int bcmp(const void *ptr1, const void *ptr2, size_t nbytes);

                                   Returns: 0 if equal, nonzero if unequal
```

This is our first encounter with the ANSI C `const` qualifier. In the three uses here, it indicates that what is pointed to by the pointer with this qualification, *src, ptr1*, and *ptr2*, is not modified by the function. Worded another way, the memory pointed to by the `const` pointer is read but not modified by the function.

`bzero` sets the specified number of bytes to 0 in the destination. We often use this function to initialize a socket address structure to 0. `bcopy` moves the specified number of bytes from the source to the destination. `bcmp` compares two arbitrary byte strings. The return value is zero if the two byte strings are identical; otherwise, it is nonzero.

The following functions are the ANSI C functions:

```
#include <string.h>

void *memset(void *dest, int c, size_t len);

void *memcpy(void *dest, const void *src, size_t nbytes);

int memcmp(const void *ptr1, const void *ptr2, size_t nbytes);

                         Returns: 0 if equal, <0 or >0 if unequal (see text)
```

`memset` sets the specified number of bytes to the value *c* in the destination. `memcpy` is similar to `bcopy`, but the order of the two pointer arguments is swapped. `bcopy` correctly handles overlapping fields, while the behavior of `memcpy` is undefined if the source and destination overlap. The ANSI C `memmove` function must be used when the fields overlap.

One way to remember the order of the two pointers for `memcpy` is to remember that they are written in the same left-to-right order as an assignment statement in C:

dest = src;

One way to remember the order of the final two arguments to `memset` is to realize that all of the ANSI C `memXXX` functions require a length argument, and it is always the final argument.

`memcmp` compares two arbitrary byte strings and returns 0 if they are identical. If not identical, the return value is either greater than 0 or less than 0, depending on whether the first unequal byte pointed to by *ptr1* is greater than or less than the corresponding byte pointed to by *ptr2*. The comparison is done assuming the two unequal bytes are `unsigned chars`.

3.6 `inet_aton`, `inet_addr`, and `inet_ntoa` Functions

We will describe two groups of address conversion functions in this section and the next. They convert Internet addresses between ASCII strings (what humans prefer to use) and network byte ordered binary values (values that are stored in socket address structures).

1. `inet_aton`, `inet_ntoa`, and `inet_addr` convert an IPv4 address from a dotted-decimal string (e.g., "`206.168.112.96`") to its 32-bit network byte ordered binary value. You will probably encounter these functions in lots of existing code.
2. The newer functions, `inet_pton` and `inet_ntop`, handle both IPv4 and IPv6 addresses. We describe these two functions in the next section and use them throughout the text.

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

int inet_aton(const char *strptr, struct in_addr *addrptr);
                                        Returns: 1 if string was valid, 0 on error

in_addr_t inet_addr(const char *strptr);
        Returns: 32-bit binary network byte ordered IPv4 address; INADDR_NONE if error

char *inet_ntoa(struct in_addr inaddr);
                                        Returns: pointer to dotted-decimal string
```

The first of these, `inet_aton`, converts the C character string pointed to by *strptr* into its 32-bit binary network byte ordered value, which is stored through the pointer *addrptr*. If successful, 1 is returned; otherwise, 0 is returned.

An undocumented feature of `inet_aton` is that if *addrptr* is a null pointer, the function still performs its validation of the input string but does not store any result.

`inet_addr` does the same conversion, returning the 32-bit binary network byte ordered value as the return value. The problem with this function is that all 2^{32} possible binary values are valid IP addresses (0.0.0.0 through 255.255.255.255), but the function returns the constant `INADDR_NONE` (typically 32 one-bits) on an error. This means the dotted-decimal string 255.255.255.255 (the IPv4 limited broadcast address, Section 20.2) cannot be handled by this function since its binary value appears to indicate failure of the function.

A potential problem with `inet_addr` is that some man pages state that it returns –1 on an error, instead of `INADDR_NONE`. This can lead to problems, depending on the C compiler, when comparing the return value of the function (an unsigned value) to a negative constant.

Today, `inet_addr` is deprecated and any new code should use `inet_aton` instead. Better still is to use the newer functions described in the next section, which handle both IPv4 and IPv6.

The `inet_ntoa` function converts a 32-bit binary network byte ordered IPv4 address into its corresponding dotted-decimal string. The string pointed to by the return value of the function resides in static memory. This means the function is not reentrant, which we will discuss in Section 11.18. Finally, notice that this function takes a structure as its argument, not a pointer to a structure.

Functions that take actual structures as arguments are rare. It is more common to pass a pointer to the structure.

6.4 Stevens, W R, Fenner, B, and Rudoff, A M (2003) *UNIX Network Programming, Volume 1: The Sockets Networking API*, 3rd edn, Boston: Addison-Wesley, Sections 4.1–4.6 and 4.9.

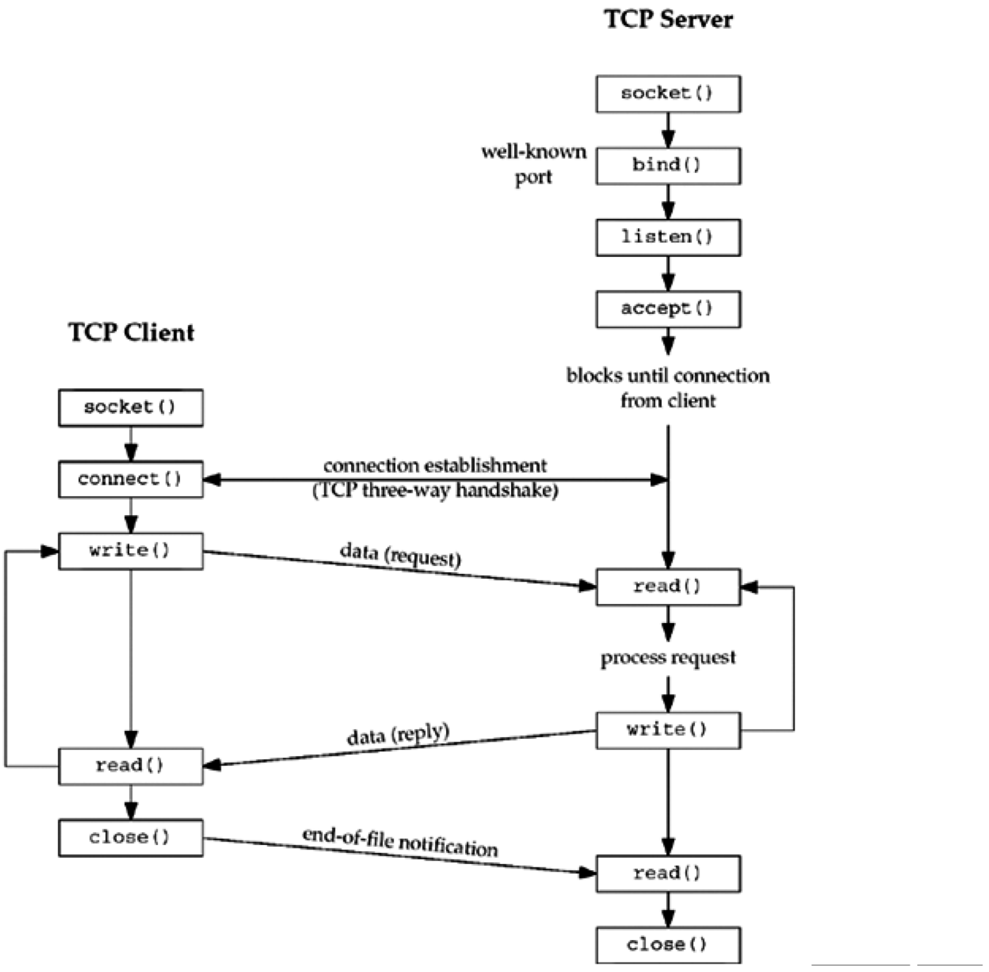
4.1 Introduction

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.

Figure 4.1. Socket functions for elementary TCP client/server.



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.).

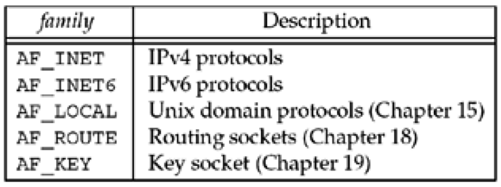
```
#include <sys/socket.h>

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

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

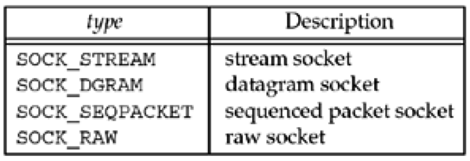
family specifies the protocol family and is one of the constants shown in Figure 4.2. This argument is often referred to as *domain* instead of *family*. The socket *type* is one of the constants shown in Figure 4.3. The *protocol* argument to the `socket` function should be set to the specific protocol type found in Figure 4.4, or 0 to select the system's default for the given combination of *family* and *type*.

Figure 4.2. Protocol *family* constants for `socket` function.



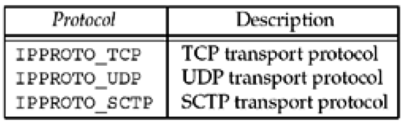
family	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols (Chapter 15)
AF_ROUTE	Routing sockets (Chapter 18)
AF_KEY	Key socket (Chapter 19)

Figure 4.3. *type* of socket for `socket` function.



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

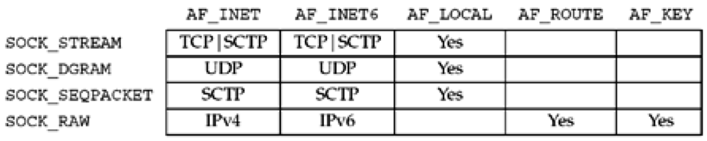
Figure 4.4. *protocol* of sockets for `AF_INET` or `AF_INET6`.



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

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.

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



	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

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.

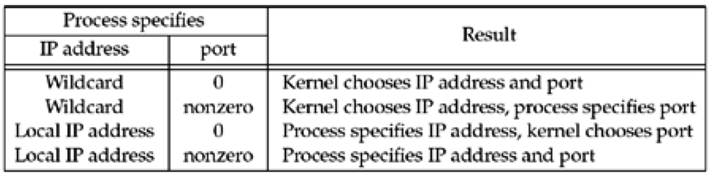
- A process can `bind` a specific IP address to its socket. The IP address must belong to an interface on the host. For a TCP client, this assigns the source IP address that will be used for IP datagrams sent on the socket. For a TCP server, this restricts the socket to receive incoming client connections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

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



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

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 `binds` 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.

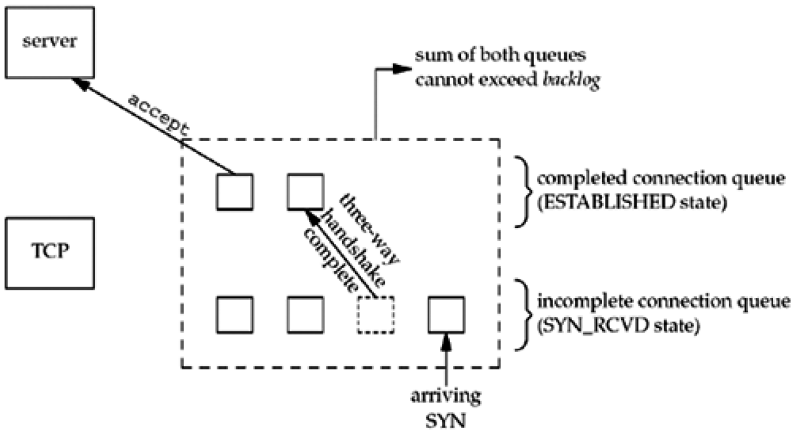
To understand the *backlog* argument, we must realize that for a given listening socket, the kernel maintains two queues:

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).

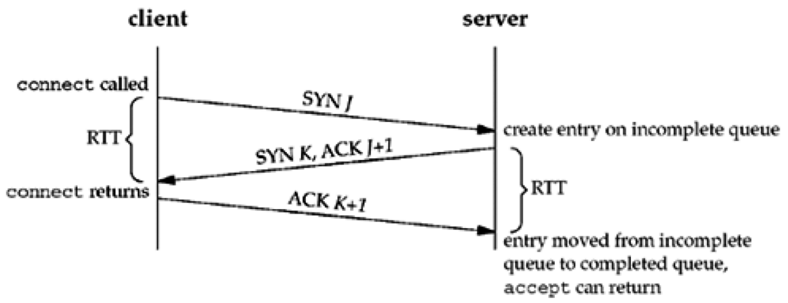
Figure 4.7 depicts these two queues for a given listening socket.

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


server
TCP
accept
sum of both queues cannot exceed *backlog*
three-way handshake complete
completed connection queue (ESTABLISHED state)
incomplete connection queue (SYN_RCVD state)
arriving SYN

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.


client
server
connect called
RTT
connect returns
SYN J
SYN K, ACK J+1
ACK K+1
create entry on incomplete queue
RTT
entry moved from incomplete queue to completed queue, accept can return

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.

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

lib/wrapsock.c

```
void
Listen (int fd, int backlog)
{
    char    *ptr;

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

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

- 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!

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

	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

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 `accept`ed (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.

Figure 4.11 Daytime server that prints client IP address and port

intro/daytimetcpsrv1.c

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

int
main(int argc, char **argv)
{
    int     listenfd, connfd;
    socklen_t len;
    struct sockaddr_in servaddr, cliaddr;
    char    buff[MAXLINE];
    time_t  ticks;

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    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 }
```

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
```

[Team LiB] ◂ PREVIOUS NEXT ▸
[Team LiB] ◂ PREVIOUS NEXT ▸

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 no way to obtain the process IDs of its children. If a parent wants to keep track of the process IDs of all its children, it must record the return values from `fork`.

All descriptors open in the parent before the call to `fork` are shared with the child after `fork` returns. We will see this feature used by network servers: The parent calls `accept` and then calls `fork`. The connected socket is then shared between the parent and child. Normally, the child then reads and writes the connected socket and the parent closes the connected socket.

There are two typical uses of `fork`:

1. A process makes a copy of itself so that one copy can handle one operation while the other copy does another task. This is typical for network servers. We will see many examples of this later in the text.
2. A process wants to execute another program. Since the only way to create a new process is by calling `fork`, the process first calls `fork` to make a copy of itself, and then one of the copies (typically the child process) calls `exec` (described next) to replace itself with the new program. This is typical for programs such as shells.

6.5 Stevens, W R, Fenner, B, and Rudoff, A M (2003) *UNIX Network Programming, Volume 1: The Sockets Networking API*, 3rd edn, Boston: Addison-Wesley, Section 8.2.

8.2 `recvfrom` and `sendto` Functions

These two functions are similar to the standard `read` and `write` functions, but three additional arguments are required.

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

ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags, struct
sockaddr *from, socklen_t *addrlen);

ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags,
const struct sockaddr *to, socklen_t addrlen);

            Both return: number of bytes read or written if OK, –1 on error
```

The first three arguments, *sockfd, buff*, and *nbytes*, are identical to the first three arguments for `read` and `write`: descriptor, pointer to buffer to read into or write from, and number of bytes to read or write.

We will describe the *flags* argument in Chapter 14 when we discuss the `recv`, `send`, `recvmsg`, and `sendmsg` functions, since we do not need them with our simple UDP client/server example in this chapter. For now, we will always set the *flags* to 0.

The *to* argument for `sendto` is a socket address structure containing the protocol address (e.g., IP address and port number) of where the data is to be sent. The size of this socket address structure is specified by *addrlen*. The `recvfrom` function fills in the socket address structure pointed to by *from* with the protocol address of who sent the datagram. The number of bytes stored in this socket address structure is also returned to the caller in the integer pointed to by *addrlen*. Note that the final argument to `sendto` is an integer value, while the final argument to `recvfrom` is a pointer to an integer value (a value-result argument).

The final two arguments to `recvfrom` are similar to the final two arguments to `accept`: The contents of the socket address structure upon return tell us who sent the datagram (in the case of UDP) or who initiated the connection (in the case of TCP). The final two arguments to `sendto` are similar to the final two arguments to `connect`: We fill in the socket address structure with the protocol address of where to send the datagram (in the case of UDP) or with whom to establish a connection (in the case of TCP).

Both functions return the length of the data that was read or written as the value of the function. In the typical use of `recvfrom`, with a datagram protocol, the return value is the amount of user data in the datagram received.

Writing a datagram of length 0 is acceptable. In the case of UDP, this results in an IP datagram containing an IP header (normally 20 bytes for IPv4 and 40 bytes for IPv6), an 8-byte UDP header, and no data. This also means that a return value of 0 from `recvfrom` is acceptable for a datagram protocol: It does not mean that the peer has closed the connection, as does a return value of 0 from `read` on a TCP socket. Since UDP is connectionless, there is no such thing as closing a UDP connection.

If the *from* argument to `recvfrom` is a null pointer, then the corresponding length argument (*addrlen*) must also be a null pointer, and this indicates that we are not interested in knowing the protocol address of who sent us data.

Both `recvfrom` and `sendto` can be used with TCP, although there is normally no reason to do so.